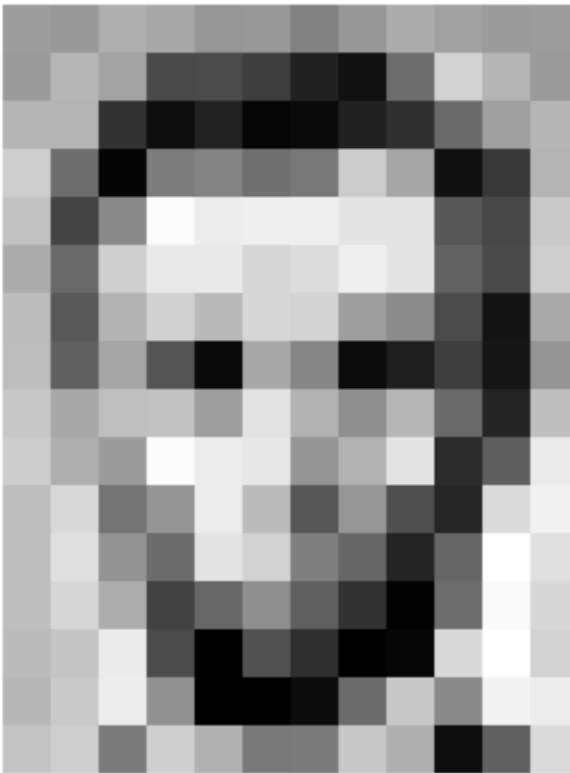


# **Tema 3. Introducción a las redes neuronales convolucionales**

Ana Jiménez Pastor  
[anjipas@gmail.com](mailto:anjipas@gmail.com)

# Introducción



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	105	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	105	95	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	105	95	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Las imágenes se representan como **matrices** donde cada celda corresponde a la intensidad de un píxel [0-255].

**Tamaño: filas x columnas x número canales.**

Ejemplo: 512 x 512 x 1 para una imagen en escala de grises  
512 x 512 x 3 para una imagen RGB (con color)

# Introducción

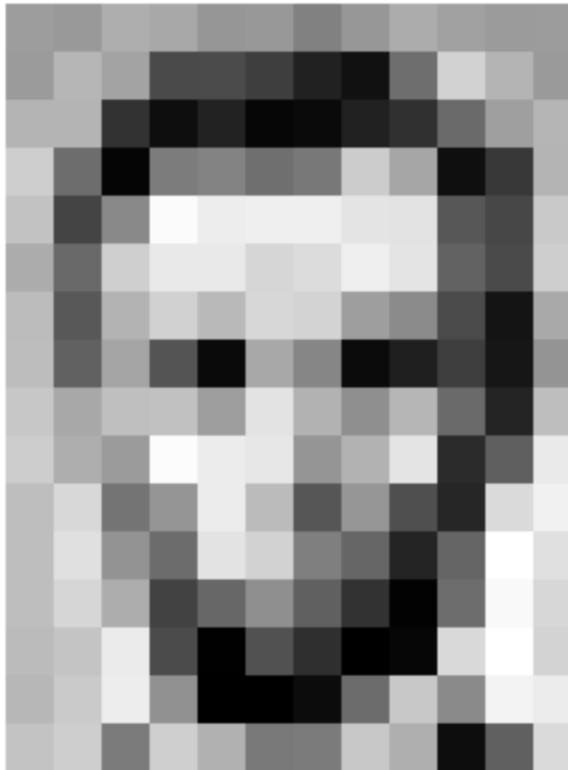


Imagen de entrada

Clasificación

Clases	Probabilidad
Lincoln	0.8
Washington	0.1
Jefferson	0.05
Obama	0.05

Salida

# Introducción

**Debemos identificar aquellas características que diferencian una clase de otra.**



Ojos  
Nariz  
Boca



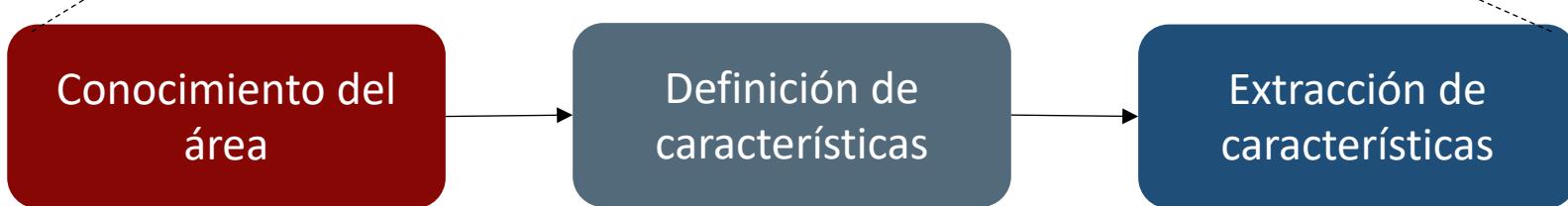
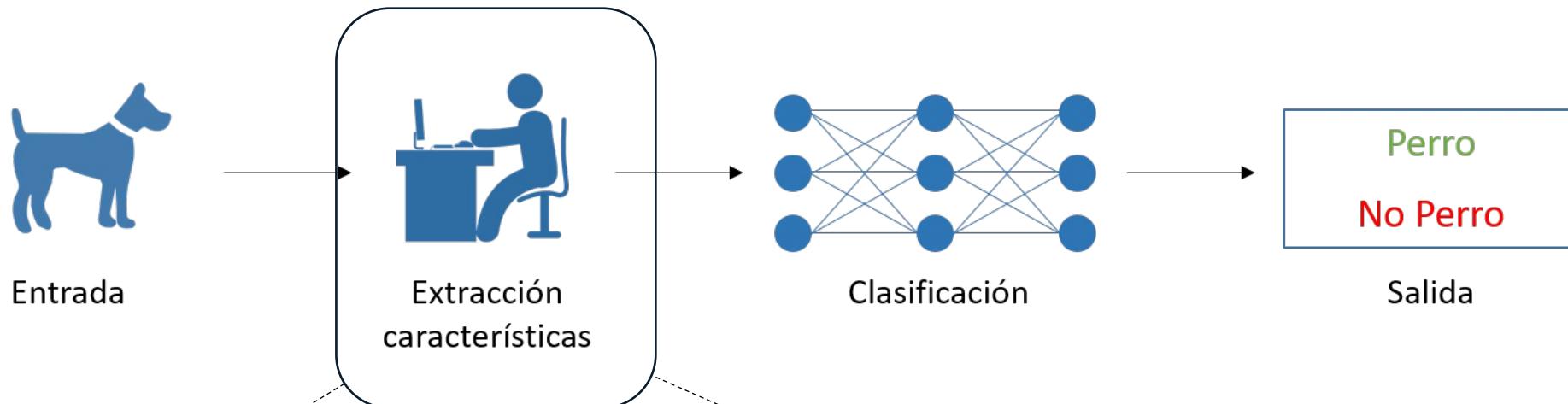
Ruedas  
Matrícula  
Faros  
Tamaño



Puerta  
Ventanas  
Tejado

# Introducción

## Extracción de características manual



¿Es esto una  
tarea sencilla?

# Introducción

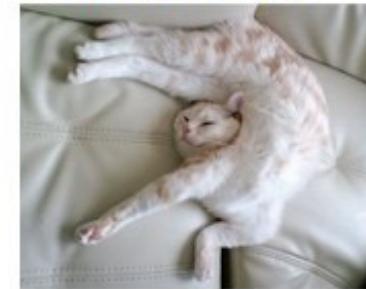
## Extracción de características manual



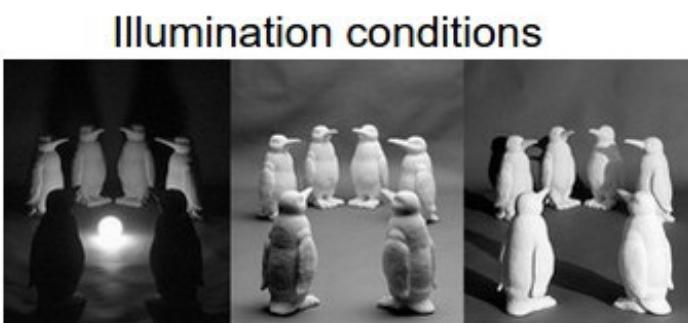
Scale variation



Deformation



Occlusion



Background clutter



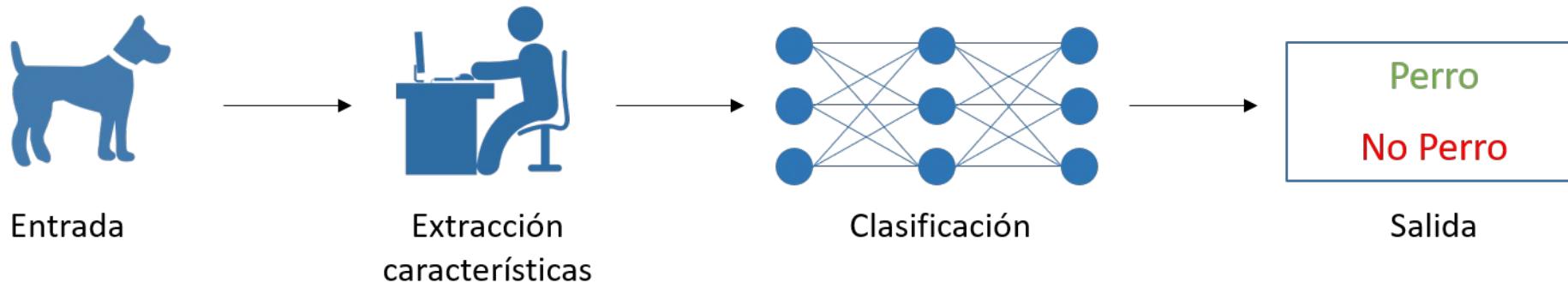
Intra-class variation



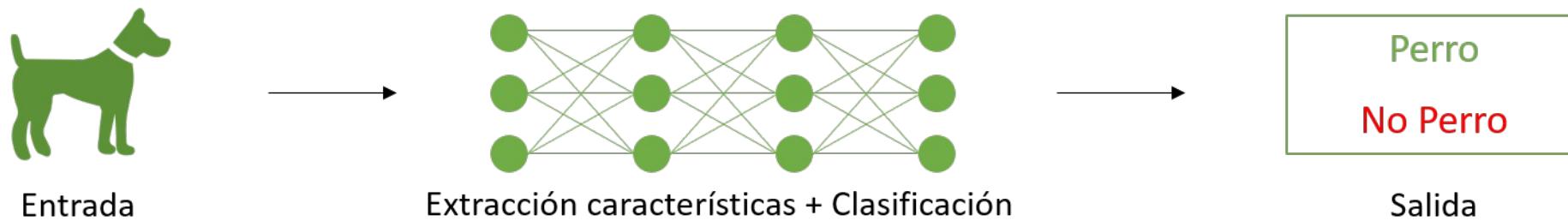
Fuente imagen: <https://anhvnn.wordpress.com/>

# Introducción

## Machine Learning tradicional



## Deep Learning

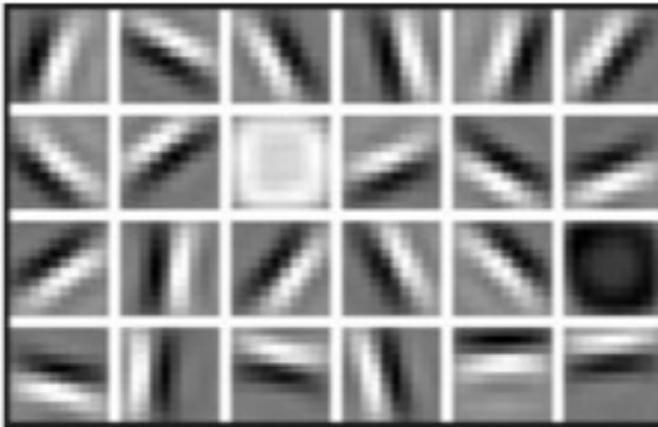


# Introducción

¿Podemos aprender **jerarquía de características** directamente de los datos en lugar de tener que extraerlas manualmente?

Las CNN extraen características a **distintos niveles** y las combinan para realizar la predicción.

Características bajo nivel



Ejes, contornos

Características nivel intermedio



Ojos, orejas, nariz, boca

Características alto nivel

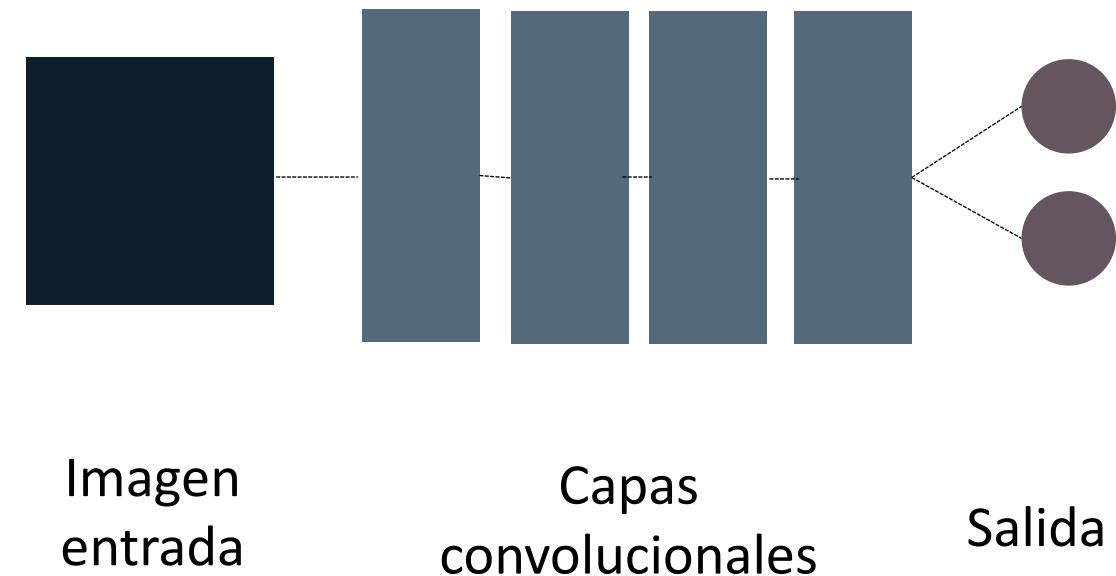


Caras

# Introducción

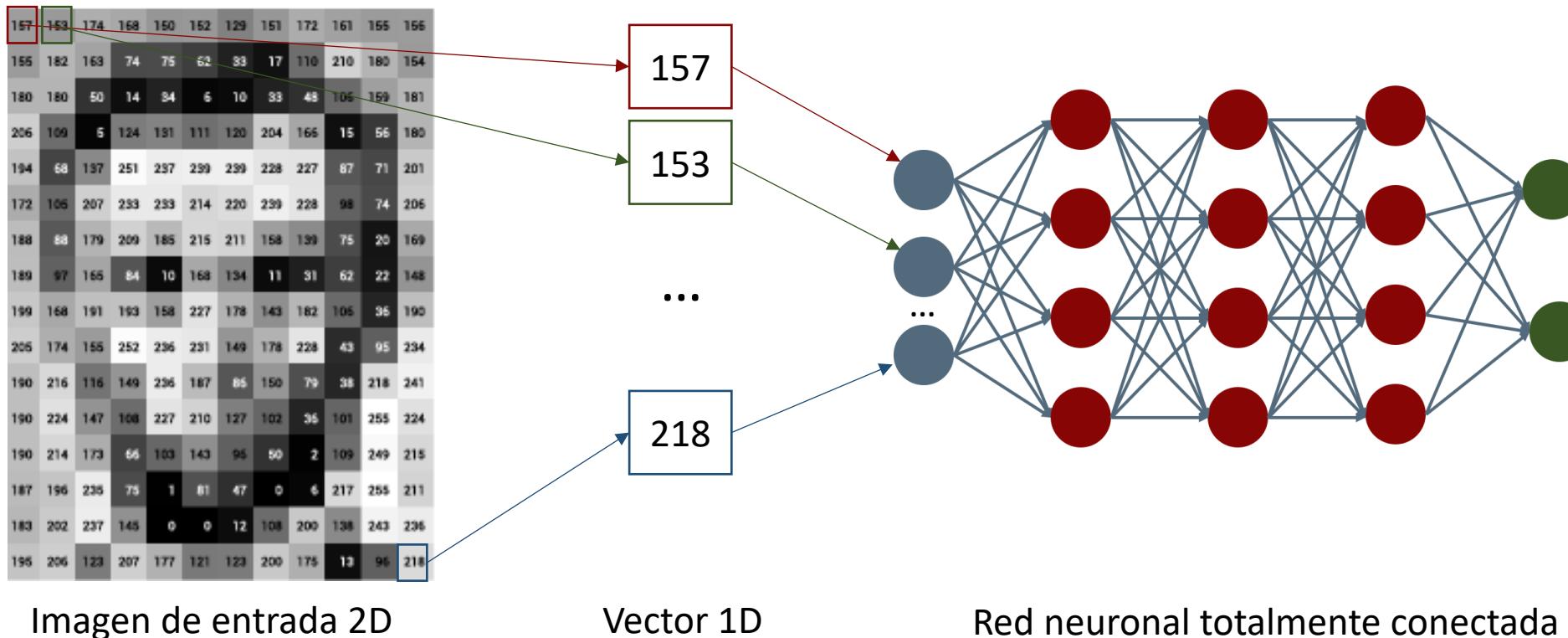
## Redes neuronales convolucionales (CNN, *Convolutional Neuronal Network*)

- Modelo diseñado para trabajar con **imágenes**.
- Además de los pesos de las conexiones, durante el proceso de entrenamiento, los **filtros se adaptan** para extraer características apropiadas para la resolución de nuestro problema.



**¿Cómo se aprenden estas  
características?**

# ¿Cómo se aprenden estas características?



- **Se pierde información espacial.**
- **1 neurona por píxel → Mucho parámetros**

# ¿Cómo se aprenden estas características?

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	84	6	10	33	48	105	159	181
206	109	6	124	191	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	186	215	241	158	199	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	176	143	182	105	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	85	150	79	38	218	241
190	224	147	108	227	210	127	102	35	101	255	224
190	214	173	66	103	143	95	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

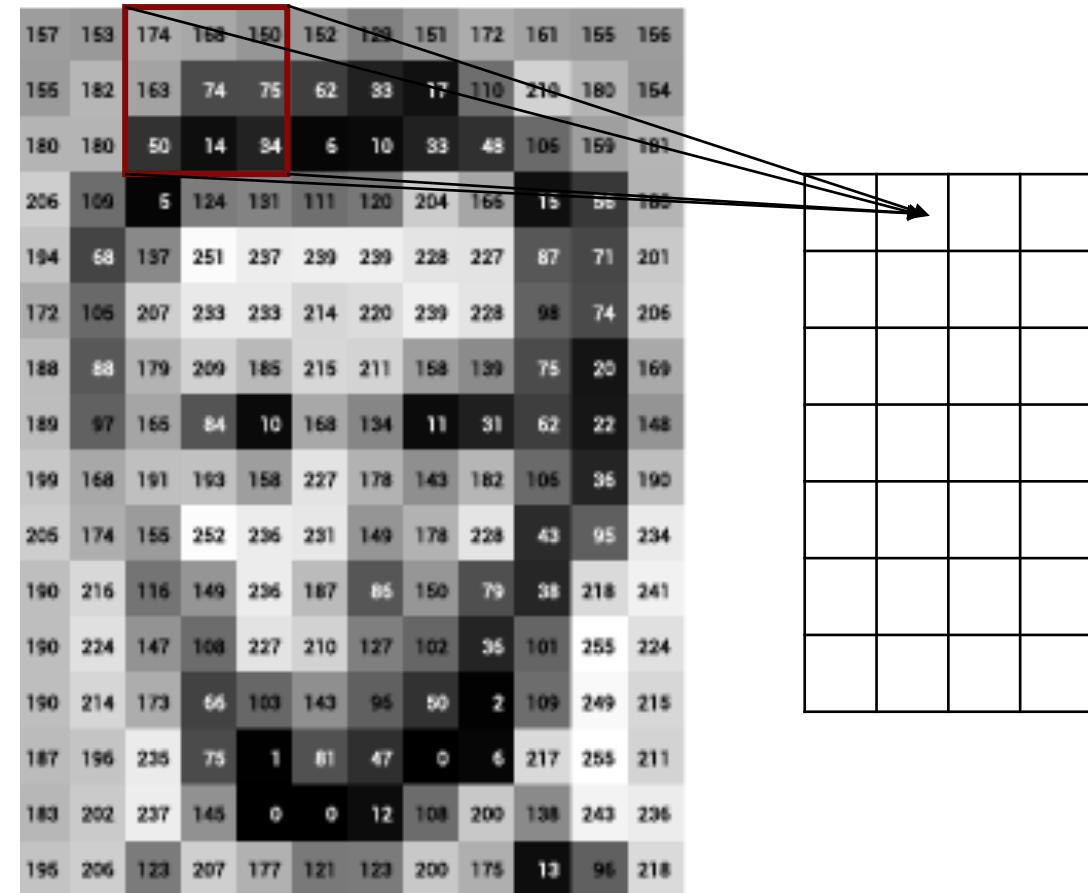
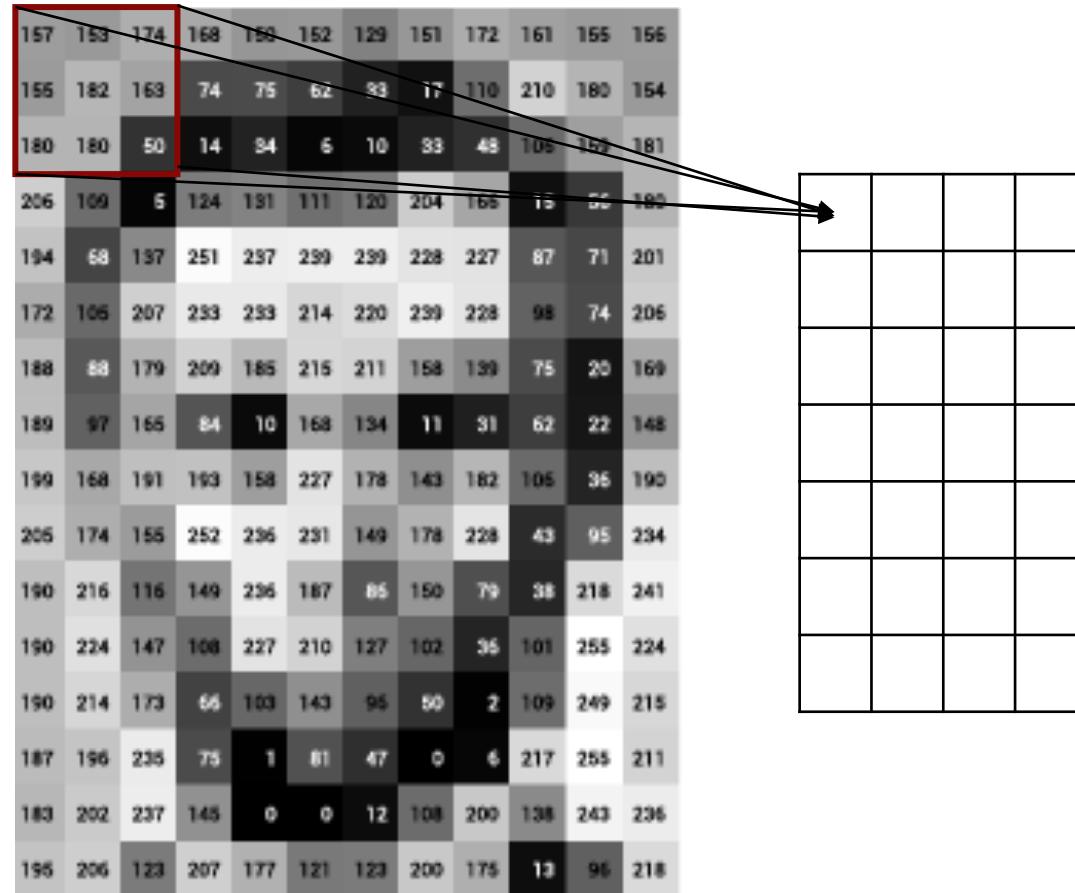
Imagen de entrada 2D

## Base de las CNN:

Se conecta una región (*patch*) de la imagen de entrada con una neurona de la capa oculta siguiente. Cada neurona únicamente “ve” esta región



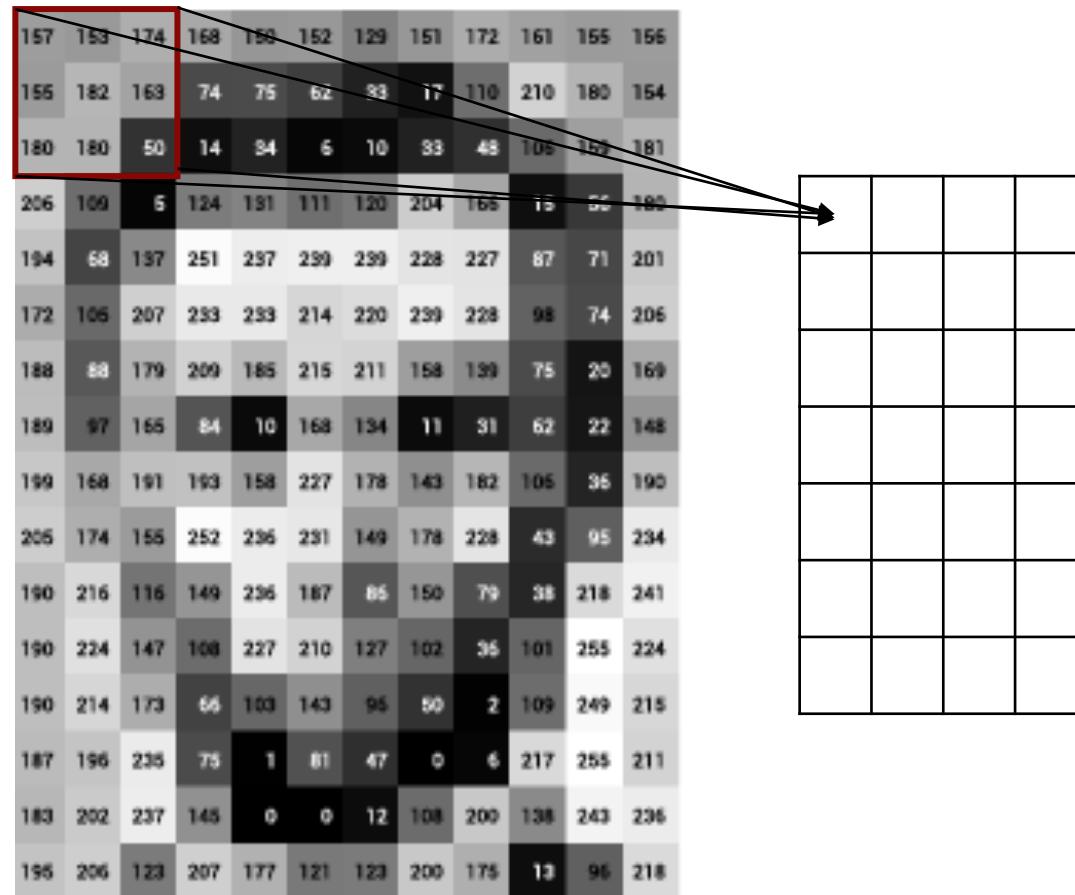
# ¿Cómo se aprenden estas características?



Se van analizando regiones de manera sucesiva mediante una técnica de **ventana deslizante** y se conecta cada una con una neurona de la capa siguiente.

Los **pesos** de estas conexiones definen las características que se extraen.

# ¿Cómo se aprenden estas características?



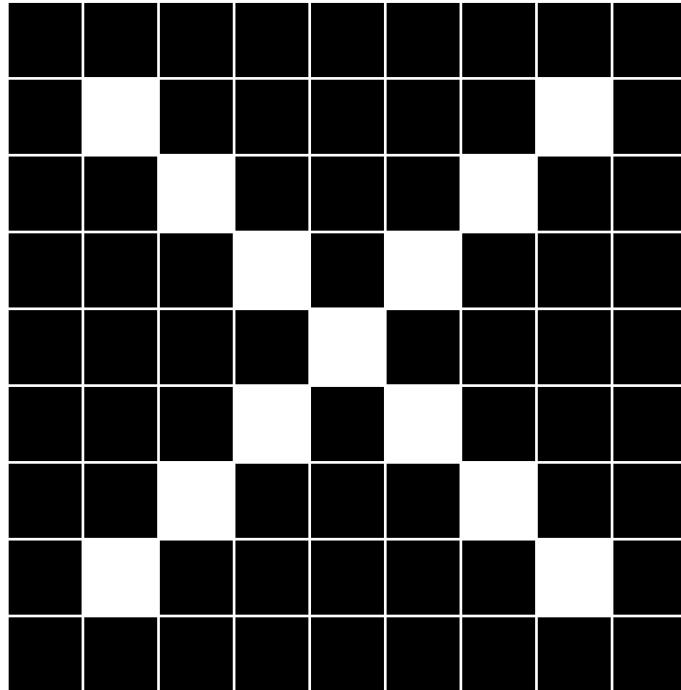
1. Se aplican una serie de pesos (filtro) para extraer características locales.
2. Se usan múltiples filtros para extraer diferentes características.

## Filtro convolucional

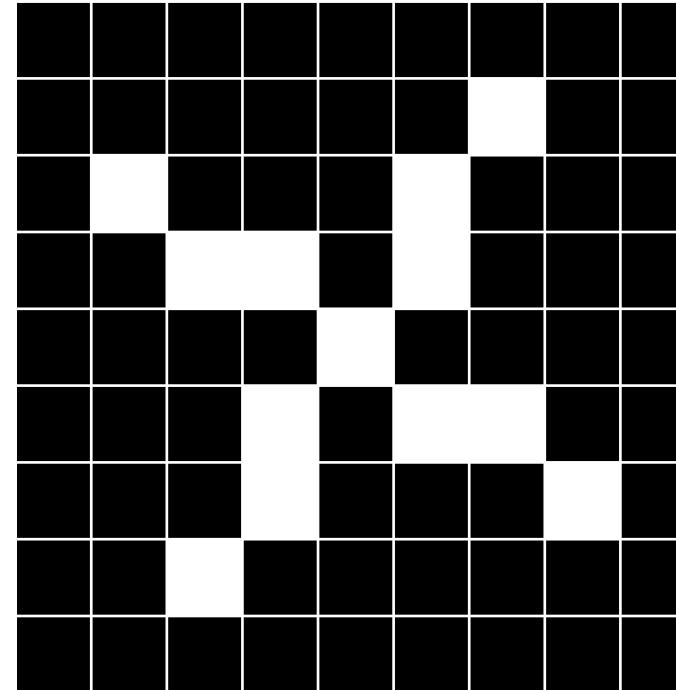
# Filtro convolucional

# Filtro convolucional

Queremos clasificar una imagen como X o no:



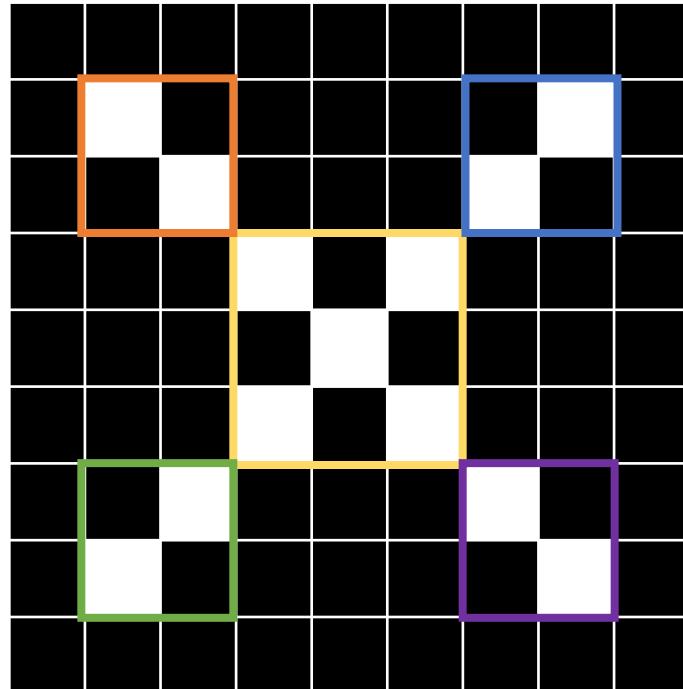
?



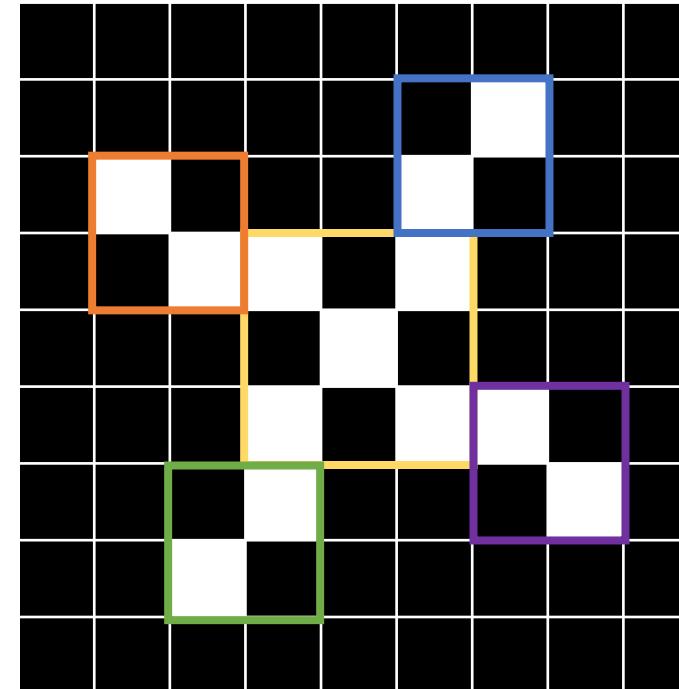
Debemos de ser capaces de clasificar la imagen independientemente de si la X se encuentra desplazada, rotada, deformada...

# Filtro convolucional

## Características de la X



=



Mismas características → Misma clase

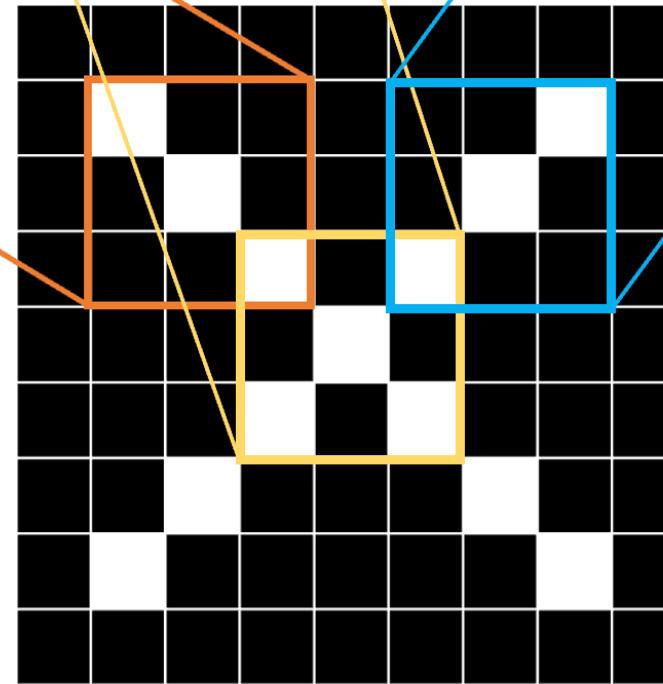
# Filtro convolucional

Filtros para detectar las características de X:

$$\begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & -1 \end{bmatrix}$$



# Filtro convolucional

1	-1	-1
-1	1	-1
-1	-1	1

Producto  
escalar

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

## Operación convolucional

1.  
Multiplicación  
componente a  
componente

O

1	1	1
1	1	1
1	1	1

2.  
Suma de  
todas las  
componentes

= 9

# Filtro convolucional

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input image

Operación convolucional

\*

1	0	-1
1	0	-1
1	0	-1

=


Kernel

Output image

# Filtro convolucional

Vamos a visualizar los filtros ajustados por una CNN ya existente, así como el mapa de características (imagen filtrada) generada por dichos filtros.

Para ello, vamos a emplear una arquitectura que se diseñó hace unos años (VGG16 – 2014) y que fue entrenada sobre la base de datos ImageNet.

Por lo tanto, lo primero que vamos a hacer es importar dicha arquitectura que se encuentra dentro de la propia API de Tensorflow.

```
from tensorflow.keras.applications.vgg16 import VGG16  
  
model = VGG16(include_top=False, weights='imagenet')
```

# Filtro convolucional

```
model.summary()
```

Model: "vgg16"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080

block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000
=====		
Total params: 138,357,544		
Trainable params: 138,357,544		
Non-trainable params: 0		

# Filtro convolucional

En primer lugar vamos a coger las capas de la CNN importada y a filtrar las capas convolucionales

```
for l, layer in enumerate(model.layers):
    if 'conv' not in layer.name:
        continue
    filters, biases = layer.get_weights()
    print('Capa', l, ':', layer.name, filters.shape)
```

Capa 1 : block1\_conv1 (3, 3, 3, 64)  
Capa 2 : block1\_conv2 (3, 3, 64, 64)  
Capa 4 : block2\_conv1 (3, 3, 64, 128)  
Capa 5 : block2\_conv2 (3, 3, 128, 128)  
Capa 7 : block3\_conv1 (3, 3, 128, 256)  
Capa 8 : block3\_conv2 (3, 3, 256, 256)  
Capa 9 : block3\_conv3 (3, 3, 256, 256)  
Capa 11 : block4\_conv1 (3, 3, 256, 512)  
Capa 12 : block4\_conv2 (3, 3, 512, 512)  
Capa 13 : block4\_conv3 (3, 3, 512, 512)  
Capa 15 : block5\_conv1 (3, 3, 512, 512)  
Capa 16 : block5\_conv2 (3, 3, 512, 512)  
Capa 17 : block5\_conv3 (3, 3, 512, 512)

# Filtro convolucional

A continuación, vamos a representar algunos de los filtros de la primera capa

```
# Cogemos los pesos aprendidos (filtros y sesgo) de la primera capa
filters, biases = model.layers[1].get_weights()

# Normalizamos los valores de los filtros para que estén contenidos entre 0 y 1 y se puedan visualizar adecuadamente
f_min, f_max = filters.min(), filters.max()
filters = (filters - f_min) / (f_max - f_min)

# Vamos a visualizar 10 filtros
n_filters, ix = 10, 1

plt.figure(figsize=(10,20))

for i in range(n_filters):

    f = filters[:, :, :, i]

    # Para cada filtro visualizamos 3 canales
    for j in range(3):
        ax = plt.subplot(n_filters, 3, ix)
        ax.set_xticks([])
        ax.set_yticks([])

        plt.imshow(f[:, :, j], cmap='gray')
        ix += 1

plt.show()
```

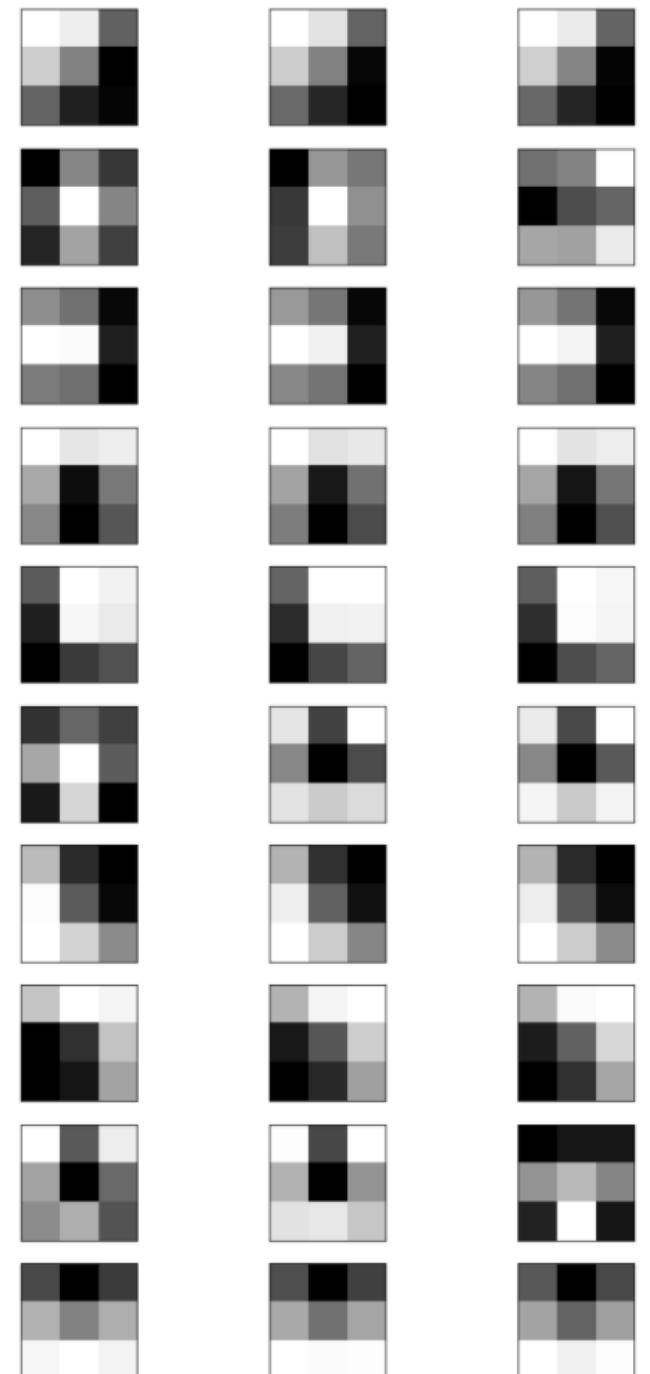
# Filtro convolucional

En la imagen tenemos los filtros aprendidos por la CNN en la primera capa convolucional. En esta capa había un total de 64 filtros, nosotros hemos representado los primeros 10. Además, para cada filtro tenemos, en columnas, el filtro aplicado a cada canal de la imagen de entrada (3 canales).

Podemos ver que en algunos casos, encontramos el mismo filtro para los 3 canales, sin embargo, en ocasiones son diferentes.

Los píxeles representados en negro son aquellos donde los pesos toman valores más bajos, mientras que los blancos son aquellos con más peso.

En la primera fila, por ejemplo, encontramos un filtro que resalta gradiente de cambio de intensidad de superior a inferior (bordes). Mientras que, por ejemplo, en las filas 5, 7, 8 vemos filtros capaces de resaltar esquinas.



# Filtro convolucional

Finalmente, vamos a ver los mapas de características (imagen filtrada) que generan estos filtros. Para esto tenemos que tomar una imagen de referencia.

## 1. Cargamos y preparamos la imagen

```
from tensorflow.keras.applications.vgg16 import preprocess_input
from PIL import Image
import numpy as np
from tensorflow.keras.models import Model

# Cargamos imagen y la procesamos
image = Image.open('imagen2.jpeg')
image = np.array(image)
plt.imshow(image)
# Preparamos la imagen para introducirla en el modelo para que
# tenga el formato [n casos, n filas, n columnas, n canales]
image = np.expand_dims(image, axis=0)

# Función para preprocessar la imagen de igual forma que se hizo
# para el desarrollo de VGG16
image = preprocess_input(image)
```

## 2. Generamos nuevo modelo y calculamos mapa de características

```
# Seleccionamos capa de la que queremos generar el mapa
# de características
layer = model.layers[2]
# Creamos modelo auxiliar en el que a la salida
# tengamos la salida de la capa deseada
model_aux = Model(inputs=[model.input],
                    outputs=[layer.output])

# Generamos mapa de características
feature_maps = model_aux.predict(image)
```



Imagen  
referencia

# Filtro convolucional



# **Redes Neuronales Convolucionales**

# Redes Neuronales Convolucionales

## Capas principales en una CNN

Capa  
convolucional

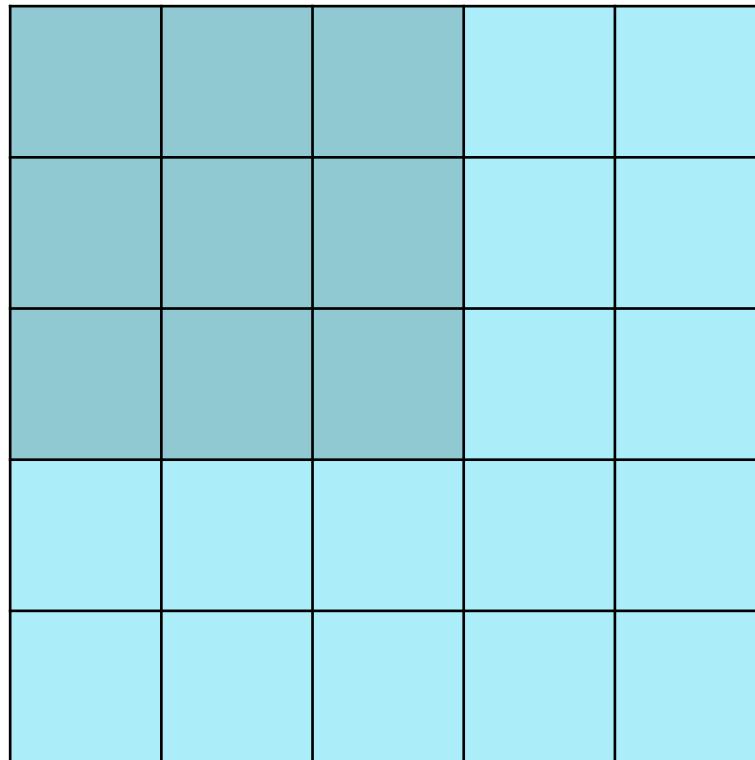
Función de  
activación

Capa  
pooling

1. **Filtro convolucional:** se aplican filtros para generar mapas de características.
2. **Función de activación:** se introducen no linealidades. Generalmente ReLU.
3. **Capa pooling:** se reduce el tamaño de la imagen.

# Redes Neuronales Convolucionales

## Capa convolucional

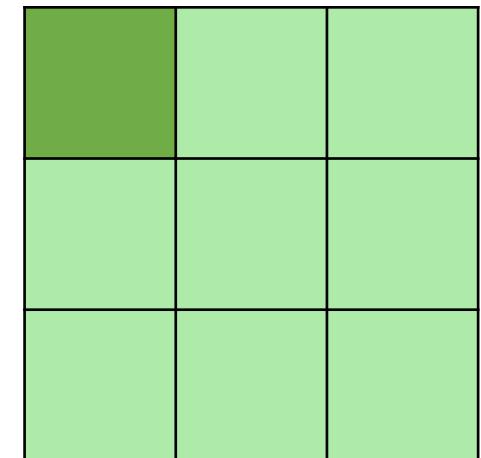


\*

$w_{11}$	$w_{21}$	$w_{31}$
$w_{12}$	$w_{22}$	$w_{32}$
$w_{13}$	$w_{23}$	$w_{33}$

Imagen entrada

Filtro



Mapa características

$$\text{Activación neurona } (p, q) = \sum_{i=1}^3 \sum_{j=1}^3 w_{ij} x_{i+p, j+q} + b$$

Cada neurona del mapa de características "ve" una pequeña región de la imagen → Información local

# Redes Neuronales Convolucionales

## Capa convolucional. *Stride*

Número de píxeles que definen el salto entre ventanas. Ejemplo stride = 2:

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

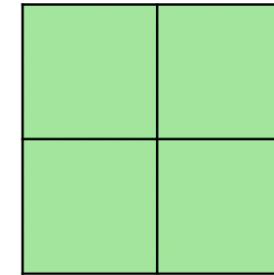
Input image

\*

1	0	-1
1	0	-1
1	0	-1

Kernel

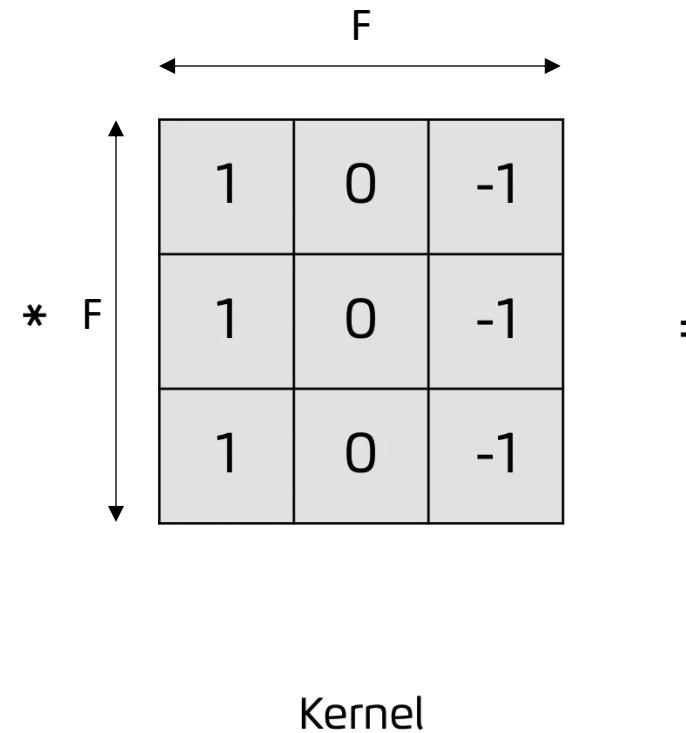
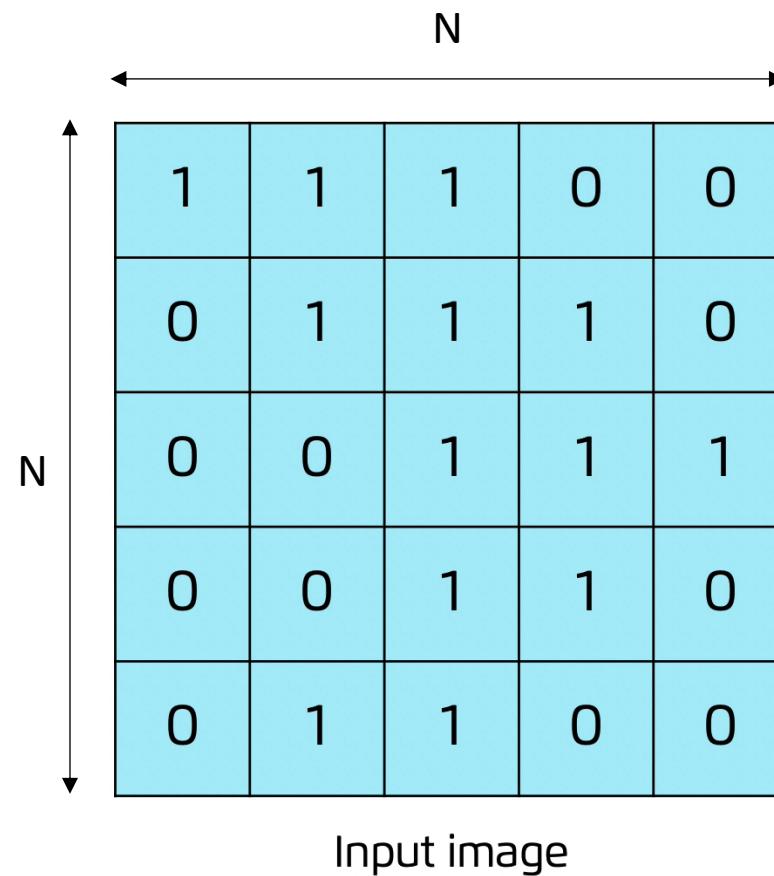
=



Output image

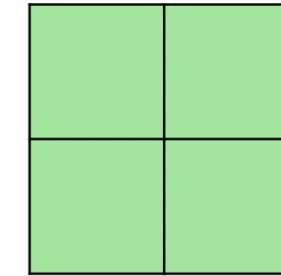
# Redes Neuronales Convolucionales

## Capa convolucional. *Stride*



Tamaño imagen salida:  
 $(N - F)/\text{stride} + 1$

e.g.  $N = 7, F = 3$   
 $\text{stride} = 1 \rightarrow (7-3)/1 + 1 = 5$   
 $\text{stride} = 2 \rightarrow (7-3)/2 + 1 = 3$   
 **$\text{stride} = 3 \rightarrow (7-3)/3 + 1 = 2,33$**



Output image

# Redes Neuronales Convolucionales

## Capa convolucional. *Padding*

Margen que se añade a la imagen de entrada para **controlar el tamaño** del mapa de características que se genera. Hay que evitar reducir el tamaño de la imagen muy rápido.  
Necesario para capturar información de los **bordes de la imagen**.

0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0
0	0	1	1	1	0	0	0
0	0	0	1	1	1	1	0
0	0	0	1	1	0	0	0
0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0

Input image

\*

1	0	-1
1	0	-1
1	0	-1

Kernel

=

-2				

Output image

# Redes Neuronales Convolucionales

## Capa convolucional. *Padding*

En Keras encontramos 2 valores de padding principales:

- "valid" → No se aplica padding
- "same" → Se ajusta el padding de forma que la imagen de salida tenga el mismo tamaño que la imagen de entrada.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0



0	0	0	0	0	0	0
0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	0	0	0	0	0	0

↑ *Padding=1* → *Padding same*  
 $(F-1)/2$

# Redes Neuronales Convolucionales

## Capa convolucional

Parámetros a definir al implementar un filtro convolucional:

- Tamaño del filtro (F).
- Número de filtros (K).
- Stride (S).
- Padding (P).

**Tamaño imagen salida: ( $N_s$ ,  $N_s$ , K)**

$$N_s = \left\lfloor \frac{N + 2P - F}{S} \right\rfloor + 1$$

**Nº parámetros a entrenar:**  
 $(F \cdot F \cdot C + 1) \cdot K$

Donde C es el número de canales de la imagen de entrada.

**Configuraciones comunes:**

K → múltiplos de 2

F=3, S=1, P=1

F=5, S=1, P=2

F=5, S=2, P=? (pad. same)

F=1, S=1, P=0

```
conv = Conv2D(filters=64, kernel_size=(3, 3), strides=(1,1), padding='same')
```

Número de  
filtros

Tamaño del  
filtro

Stride

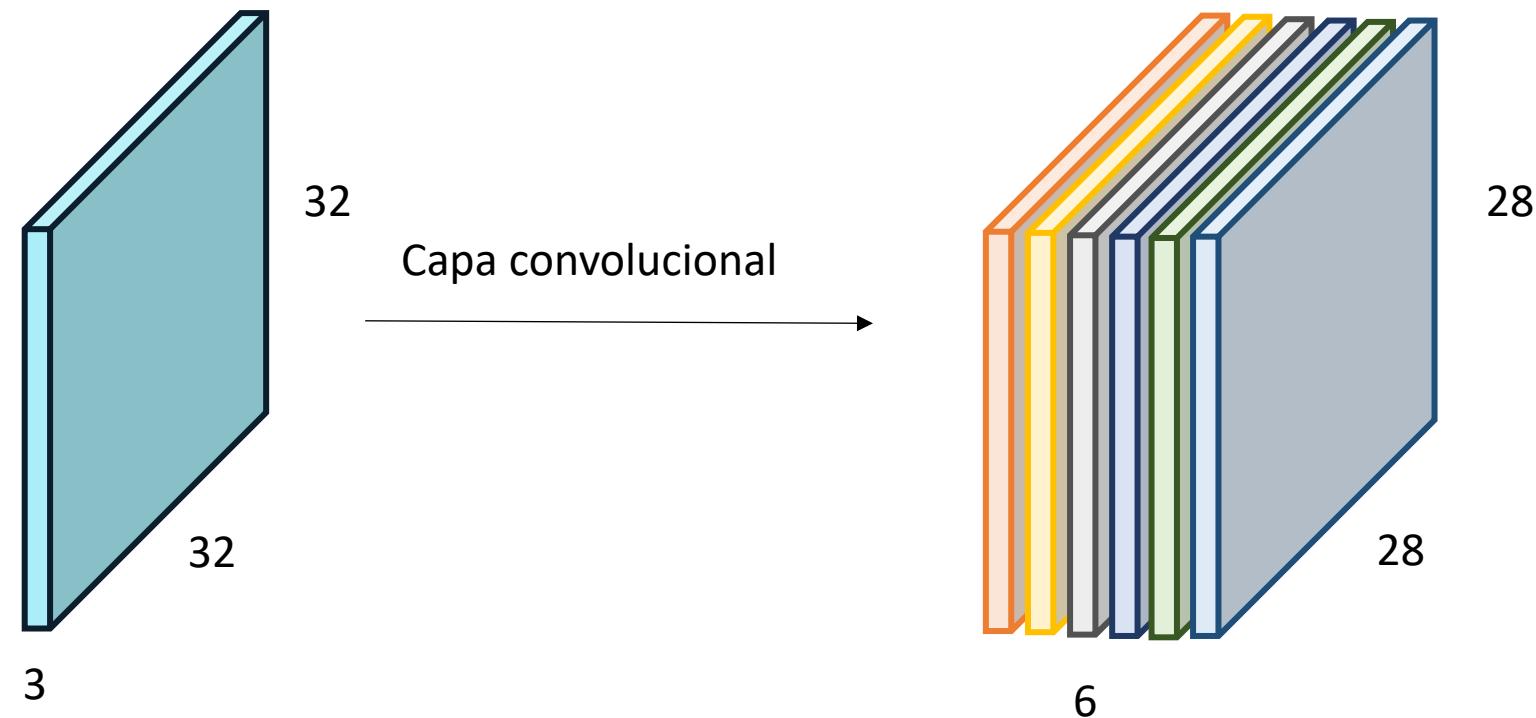
Padding



# Redes Neuronales Convolucionales

## Capa convolucional

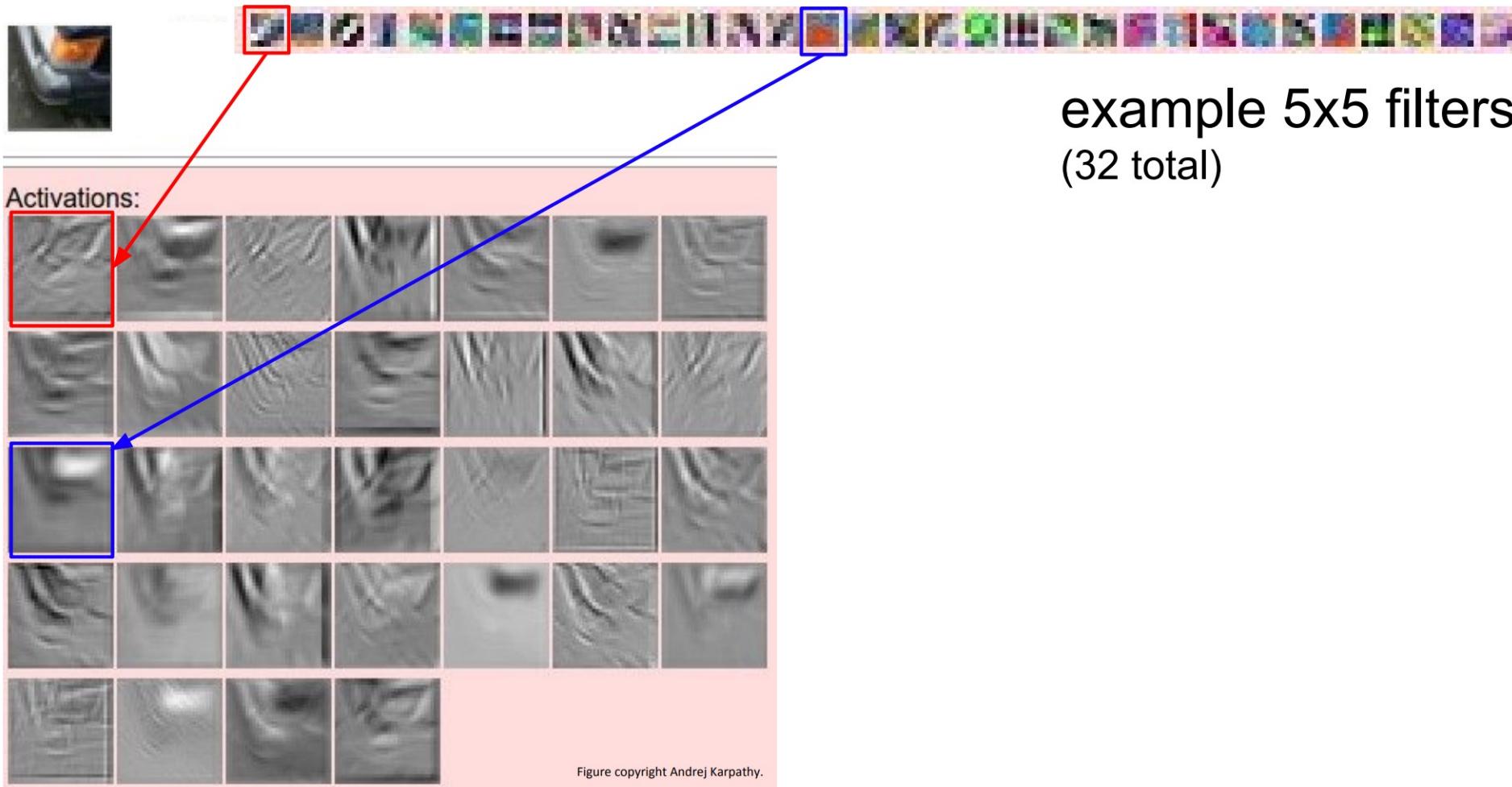
En una misma capa convolucional se emplean **múltiples filtros**. Si, por ejemplo, se emplean 6 filtros de  $5 \times 5$ , se generan 6 mapas de características.



Se **apilan** los 6 mapas de características para generar una imagen de salida de  $28 \times 28 \times 6$ .

# Redes Neuronales Convolucionales

## Capa convolucional



# Redes Neuronales Convolucionales

Vamos a ver como implementar una capa/filtro convolucional en Tensorflow/Keras y como afecta al tamaño de imagen de salida.

En primer lugar tomamos una imagen de referencia:

```
img = Image.open('imagen2.jpeg')
img = np.array(img)

# Añadimos primera dimensión para construir matriz tal que tengamos [n imágenes, n filas, n columnas,
# n canales]
img = np.expand_dims(img, axis=0)

print('Tamaño de imagen original: ', img.shape)
```

**Salida:**

Tamaño de imagen original: (1, 1000, 1500, 3)

# Redes Neuronales Convolucionales

A continuación, vamos a implementar un modelo que tenga únicamente la capa convolucional.

```
from tensorflow.keras.models import Model  
from tensorflow.keras import layers  
  
# Capa de entrada donde especificamos el tamaño de la imagen de entrada (n filas, n columnas, n canales)  
input_layer = layers.Input(shape=img.shape[1:])  
# Capa convolucional con 8 filtros de tamaño 3x3  
layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3))(input_layer)  
  
model = Model(inputs=[input_layer], outputs=[layer_conv])  
  
model.summary()
```

En una capa intermedia especificamos cuál es la capa anterior

Especificamos capa(s) de entrada y de salida

**Salida:**

Model: "model\_1"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 1000, 1500, 3)]	0
=====		
conv2d_1 (Conv2D)	(None, 500, 750, 8)	224
=====		
Total params:	224	
Trainable params:	224	
Non-trainable params:	0	

En la capa convolucional no hemos especificado ni el *stride* ni el *padding*. Si miramos en la [documentación de Keras](#), si no se especifica nada, por defecto se toma un valor de *strides*=(1,1) y *padding*=“valid”. Si queremos cambiar su valor deberemos de especificarlo.

# Redes Neuronales Convolucionales

A continuación, vamos a implementar un modelo que tenga únicamente la capa convolucional.

```
input_layer = layers.Input(shape=img.shape[1:])

layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2))(input_layer)

model = Model(inputs=[input_layer], outputs=[layer_conv])

model.summary()
```

## Salida:

Model: "model\_1"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[None, 1000, 1500, 3]	0
conv2d_1 (Conv2D)	(None, 499, 749, 8)	224

Total params: 224

Trainable params: 224

Non-trainable params: 0

¿Tamaño de la imagen de salida?

$$N_s = \left\lfloor \frac{N + 2P - F}{S} \right\rfloor + 1$$

$$N_f = \left\lfloor \frac{1000 + 2 * 0 - 3}{2} \right\rfloor + 1 = 498,5 \rightarrow 499$$

$$N_c = \left\lfloor \frac{1500 + 2 * 0 - 3}{2} \right\rfloor + 1 = 748,5 \rightarrow 749$$

# Redes Neuronales Convolucionales

A continuación, vamos a implementar un modelo que tenga únicamente la capa convolucional.

```
input_layer = layers.Input(shape=img.shape[1:])

layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2))(input_layer)

model = Model(inputs=[input_layer], outputs=[layer_conv])

model.summary()
```

## Salida:

Model: "model\_1"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 1000, 1500, 3)]	0
=====		
conv2d_1 (Conv2D)	(None, 499, 749, 8)	224
=====		
Total params: 224		
Trainable params: 224		
Non-trainable params: 0		

¿Número de parámetros a entrenar?

$$(F \cdot F \cdot C + 1) \cdot K$$
$$(3 \cdot 3 \cdot 3 + 1) \cdot 8 = 224$$

# Redes Neuronales Convolucionales

A continuación, vamos a implementar un modelo que tenga únicamente la capa convolucional.

```
input_layer = layers.Input(shape=img.shape[1:])
layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2), padding="same")(input_layer)
model = Model(inputs=[input_layer], outputs=[layer_conv])
model.summary()
```

## Salida:

Model: "model\_1"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 1000, 1500, 3)]	0
conv2d_1 (Conv2D)	(None, 500, 750, 8)	224
Total params:	224	
Trainable params:	224	
Non-trainable params:	0	

¿Porqué se ha reducido el tamaño de la imagen a la mitad si hemos puesto *padding="same"*?

# Redes Neuronales Convolucionales

Vemos que tamaño de imagen obtenemos al pasar la imagen por el modelo que acabamos de diseñar:

```
img_model = model.predict(img)  
print('Tamaño imagen filtrada: ', layer_conv.shape)
```

**Salida:**

Tamaño imagen filtrada: (None, 500, 750, 8)

¿Por qué la imagen de salida tiene 8 canales?

# Redes Neuronales Convolucionales

## Ejercicio 1

Implementa una capa convolucional con la siguientes propiedades:

- Tamaño de filtro: 5x5
- Número de filtros: 16
- Stride: 1x1
- Padding: Aquel que genere una imagen de salida de igual tamaño que la imagen de entrada

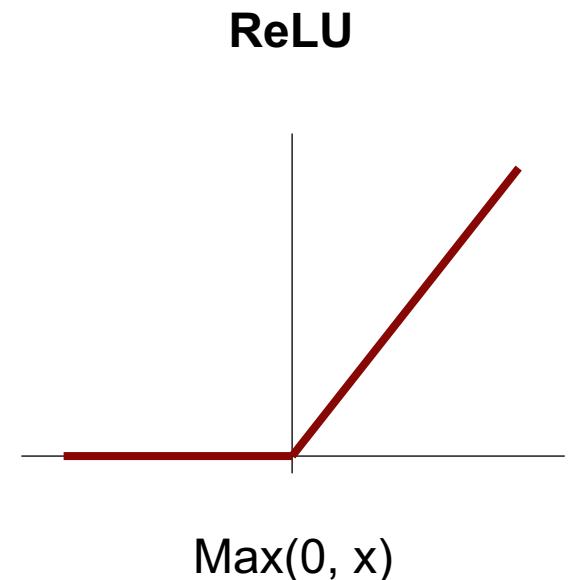
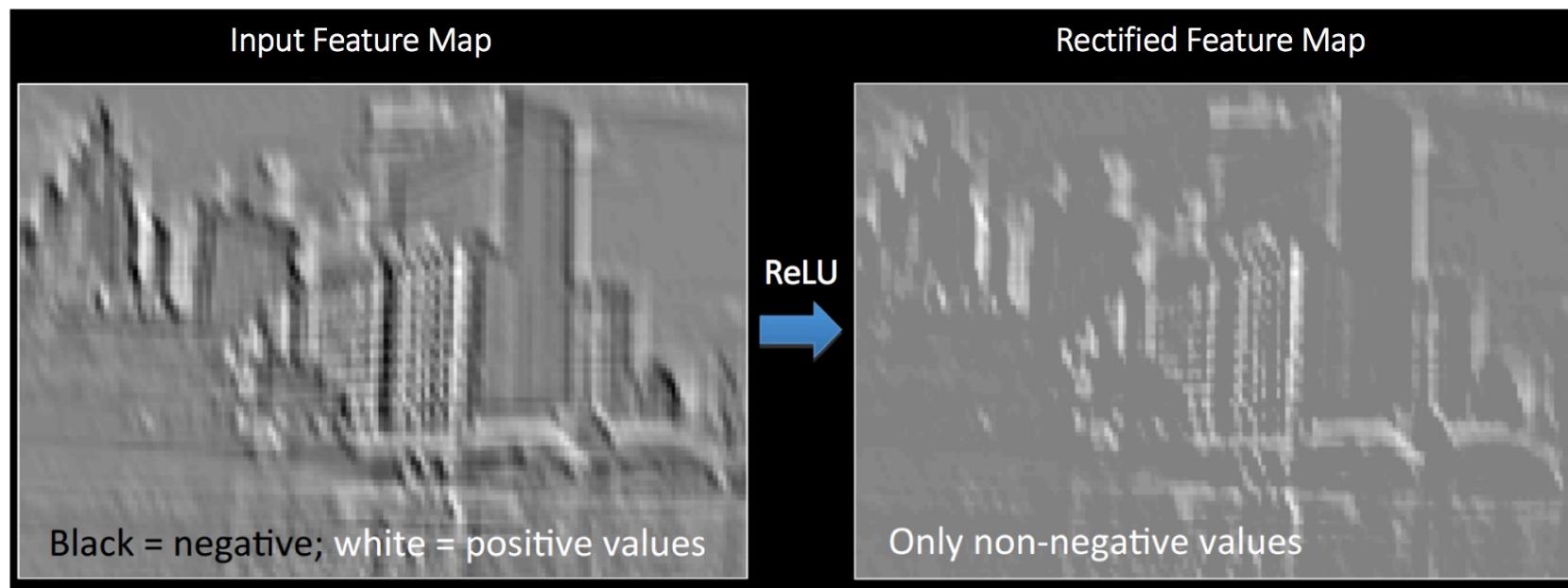
Preguntas:

- Calcula el número de parámetros que va a tener que ajustar la capa que hemos implementado, ¿coincide con los especificado por Keras?
- ¿Cuál es el tamaño de la imagen de salida?
- ¿Qué padding es el apropiado para conseguir nuestro objetivos?

# Redes Neuronales Convolucionales

## Función de activación

- Se introducen no linealidades.
- Se aplica tras cada filtro convolucional.
- La más común es **ReLU** (*Rectified Linear Unit*), reemplaza todos los píxeles negativos por cero.



# Redes Neuronales Convolucionales

## Función de activación

Para implementar en Keras la función de activación solo deberemos añadir un argumento más a nuestra capa convolucional.

```
# Como argumento dentro de la capa convolucional
input_layer = layers.Input(shape=img.shape[1:])
layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2), padding="same",
activation="relu")(input_layer)
```

Esta es la forma más sencilla de incorporar la función de activación tras el filtro convolucional. También la podemos implementar como una capa adicional (esto nos será útil cuando la función de activación sea parametrizable, e.g., LeakyRelu).

```
input_layer = layers.Input(shape=img.shape[1:])
layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2), padding="same")(input_layer)
act = layers.Activation("relu")(layer_conv)

model = Model(inputs=[input_layer], outputs=[act])

model.summary()
```

# Redes Neuronales Convolucionales

## Función de activación

**NOTA:** Cuando la función de activación se introduzca como argumento dentro de la capa convolucional no la veremos reflejada en el “summary”, en cambio, cuando la incorporemos como capa adicional si aparecerá. Ambas aproximaciones dan el mismo resultado.

### Función de activación como argumento

Model: "model\_2"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 1000, 1500, 3)]	0

conv2d_1 (Conv2D)	(None, 500, 750, 8)	224
=====		

Total params: 224

Trainable params: 224

Non-trainable params: 0

### Función de activación como capa independiente

Model: "model\_3"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 1000, 1500, 3)]	0

conv2d_1 (Conv2D)	(None, 500, 750, 8)	224
=====		

activation (Activation)	(None, 500, 750, 8).	0
=====		

Total params: 224

Trainable params: 224

Non-trainable params: 0

# Redes Neuronales Convolucionales

## Función de activación

**NOTA:** No vamos a experimentar más con las funciones de activación ya que lo veréis en la parte de datos estructurados, donde se emplean las mismas funciones de activación que en CNNs.

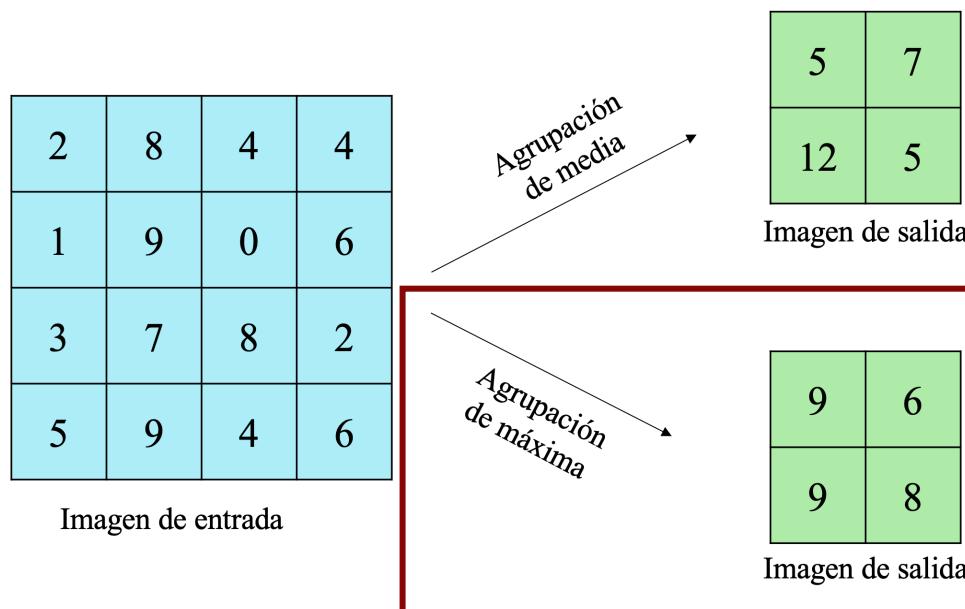
Únicamente nos vamos a quedar con que para filtros convolucionales en la gran mayoría de los casos vamos a emplear **función de activación ReLU** (para casos particulares lo iremos viendo conforme nos los vayamos encontrando).

# Redes Neuronales Convolucionales

## Pooling o agrupación

Capa para **reducir el tamaño** de los mapas de características preservando la información más importante.

- Una de las limitaciones de las capas convolucionales es que capturan la **posición exacta** de las características de la imagen de entrada.
- Reduciendo el tamaño de la imagen, aún se mantienen **elementos estructurales importantes**, sin contener el detalle fino que puede que no sea útil para la clasificación.



# Redes Neuronales Convolucionales

## Pooling

```
input_layer = layers.Input(shape=img.shape[1:])
layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2), padding="same",
activation="relu")(input_layer)
# Capa de Max pooling para dividir la imagen a la mitad
pool = layers.MaxPool2D(pool_size=(2, 2))(layer_conv)
model = Model(inputs=[input_layer], outputs=[pool])
model.summary()
```

En Keras hay otra capa llamada MaxPooling2D →  
Son lo mismo

# Redes Neuronales Convolucionales

## Pooling

Salida:

Model: "model\_23"

Layer (type)	Output Shape	Param #
input_25 (InputLayer)	[(None, 1000, 1500, 3)]	0
conv2d_30 (Conv2D)	(None, 500, 750, 8)	224
max_pooling2d_1 (MaxPooling2D)	(None, 250, 375, 8)	0
Total params: 224		
Trainable params: 224		
Non-trainable params: 0		

Las capas de MaxPooling NO introducen parámetros a ajustar durante el entrenamiento.

Reducción del tamaño de la imagen a la mitad.

# Redes Neuronales Convolucionales

## Ejercicio 2

Implementa un modelo que tenga la siguientes características:

- Capa de entrada de tamaño 256x256x3.
- Dos capas convolucionales (una a continuación de la otra) con las siguientes propiedades:
  - 1<sup>a</sup> capa: 16 filtros de 3x3, con stride (1, 1) y padding tal que mantenga el tamaño de la imagen.
  - 2<sup>a</sup> capa: 32 filtros de 5x5, con stride (2,2) y sin padding.
- Las dos capas convolucionales emplearán una función de activación ReLU.
- Capa de Max Pooling de forma que se reduzca el tamaño de la imagen a la mitad.

Preguntas:

- ¿Qué tamaño de imagen de salida habéis obtenido?
- ¿Cuántos parámetros debe ajustar el modelo en total?

# Redes Neuronales Convolucionales

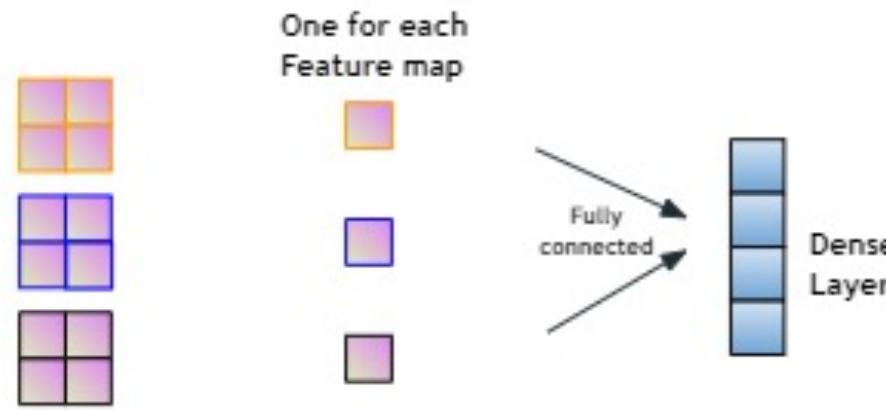
## Otras capas populares en una CNN



- **Capa de sobremuestro y capa de deconvolución:** ambas se emplean para generar representaciones más grandes de la imagen de entrada. La principal diferencia es que la capa de deconvolución "aprende" como generar esta representación (las encontraremos en modelos de segmentación, autocodificadores y GANs).
- **Normalización de lote:** permite normalizar la entrada a una capa para minimizar las diferencias entre lotes.
- **Flatten, Global Average pooling:** en una CNN de clasificación/regresión, la salida del último mapa de características se debe introducir a las capas totalmente conectadas (Dense), para ello, debemos convertir una matriz en vector empleando una de estas dos capas.

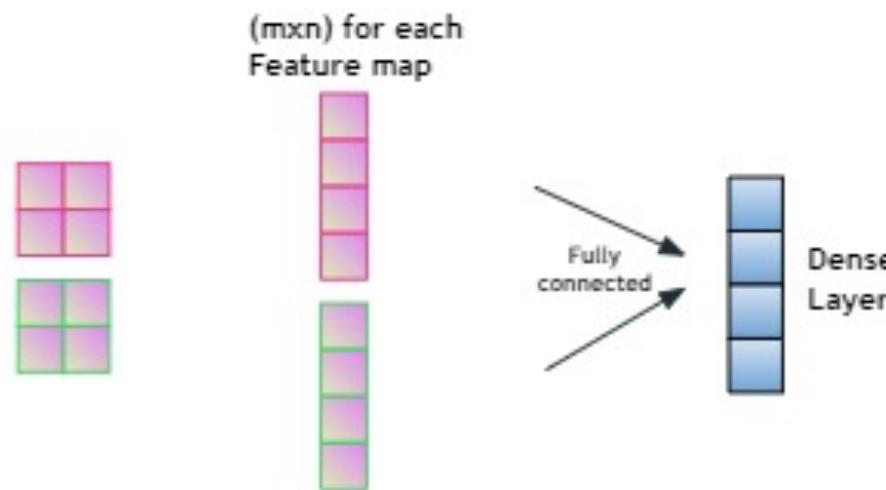
# Redes Neuronales Convolucionales

**Global Average Pooling**



De cada mapa de característica se mantiene una única característica igual a la media de los píxeles que componen dicho mapa. Menos nº de características introducidas en la capa totalmente conectada.

**Flatten**



Las matrices que componen los capas de características se convierten en vector y todos los píxeles de estos se introducen en la capa totalmente conectada. Más nº de características introducidas.

# Redes Neuronales Convolucionales

## Flatten/Global Average Pooling

```
input_layer = layers.Input(shape=img.shape[1:])
layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2), padding="same",
activation="relu")(input_layer)
pool = layers.MaxPool2D(pool_size=(2, 2))(layer_conv)

# Capa de flatten
flatten = layers.Flatten()(pool)

model = Model(inputs=[input_layer], outputs=[flatten])

model.summary()
```

```
input_layer = layers.Input(shape=img.shape[1:])
layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2), padding="same",
activation="relu")(input_layer)
pool = layers.MaxPool2D(pool_size=(2, 2))(layer_conv)

# Capa de GlobalAveragePooling
global_avg = layers.GlobalAveragePooling2D()(pool)

model = Model(inputs=[input_layer], outputs=[global_avg])

model.summary()
```

# Redes Neuronales Convolucionales

## Flatten/Global Average Pooling

Flatten

Model: "model\_5"

Layer (type)	Output Shape	Param #
input_6 (InputLayer)	[(None, 1000, 1500, 3)]	0
conv2d_5 (Conv2D)	(None, 500, 750, 8)	224
max_pooling2d_1 (MaxPooling2D)	(None, 250, 375, 8)	0
flatten (Flatten)	(None, 750000)	0

Total params: 224

Trainable params: 224

Non-trainable params: 0

Salida flatten:

$$250 * 375 * 8 = 75000$$

Global Average Pooling

Model: "model\_6"

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 1000, 1500, 3)]	0
conv2d_6 (Conv2D)	(None, 500, 750, 8)	224
max_pooling2d_2 (MaxPooling2D)	(None, 250, 375, 8)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 8)	0

Total params: 224

Trainable params: 224

Non-trainable params: 0

Salida Global Average Pooling:

Valor medio de cada canal de la salida de la capa anterior. Salida anterior (salida max polling) → 8 canales

# Redes Neuronales Convolucionales

## Capa totalmente conectada (Dense)

Al igual que para datos estructurados, vamos a emplear las capas totalmente conectadas para generar el modelo de clasificación. Por lo tanto, emplearemos capas totalmente conectadas con dos objetivos:

- **Capas ocultas**: tantas como queramos con el número de neuronas que queramos.
- **Capa de salida**: una capa con tantas neuronas como clases a predecir.

```
input_layer = layers.Input(shape=img.shape[1:])
layer_conv = layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(2, 2), padding="same",
activation="relu")(input_layer)
pool = layers.MaxPool2D(pool_size=(2, 2))(layer_conv)

global_avg = layers.GlobalAveragePooling2D()(pool)

# Capa oculta
dense_hidden = layers.Dense(32, activation='relu')(global_avg)
# Capa de salida: suponemos problema de clasificación multiclas con 4 salidas
dense_output = layers.Dense(4, activation='softmax')(dense_hidden)

model = Model(inputs=[input_layer], outputs=[dense_output])

model.summary()
```

Capa oculta con 32 neuronas.  
Función de activación ReLU.

Capa de salida con tantas neuronas como clases a predecir (e.g., 4) y función de activación Softmax (clasificación multiclas).

# Redes Neuronales Convolucionales

## Capa totalmente conectada (Dense)

Model: "model\_7"

Layer (type)	Output Shape	Param #
input_8 (InputLayer)	[(None, 1000, 1500, 3)]	0
conv2d_7 (Conv2D)	(None, 500, 750, 8)	224
max_pooling2d_3 (MaxPooling2	(None, 250, 375, 8)	0
global_average_pooling2d_1(	(None, 8)	0
dense (Dense)	(None, 32)	288
dense_1 (Dense)	(None, 4)	132

Total params: 644

Trainable params: 644

Non-trainable params: 0

¿Cuántos parámetros tendríamos aquí si hubiéramos empleado Flatten en lugar de GlobalAveragePooling?

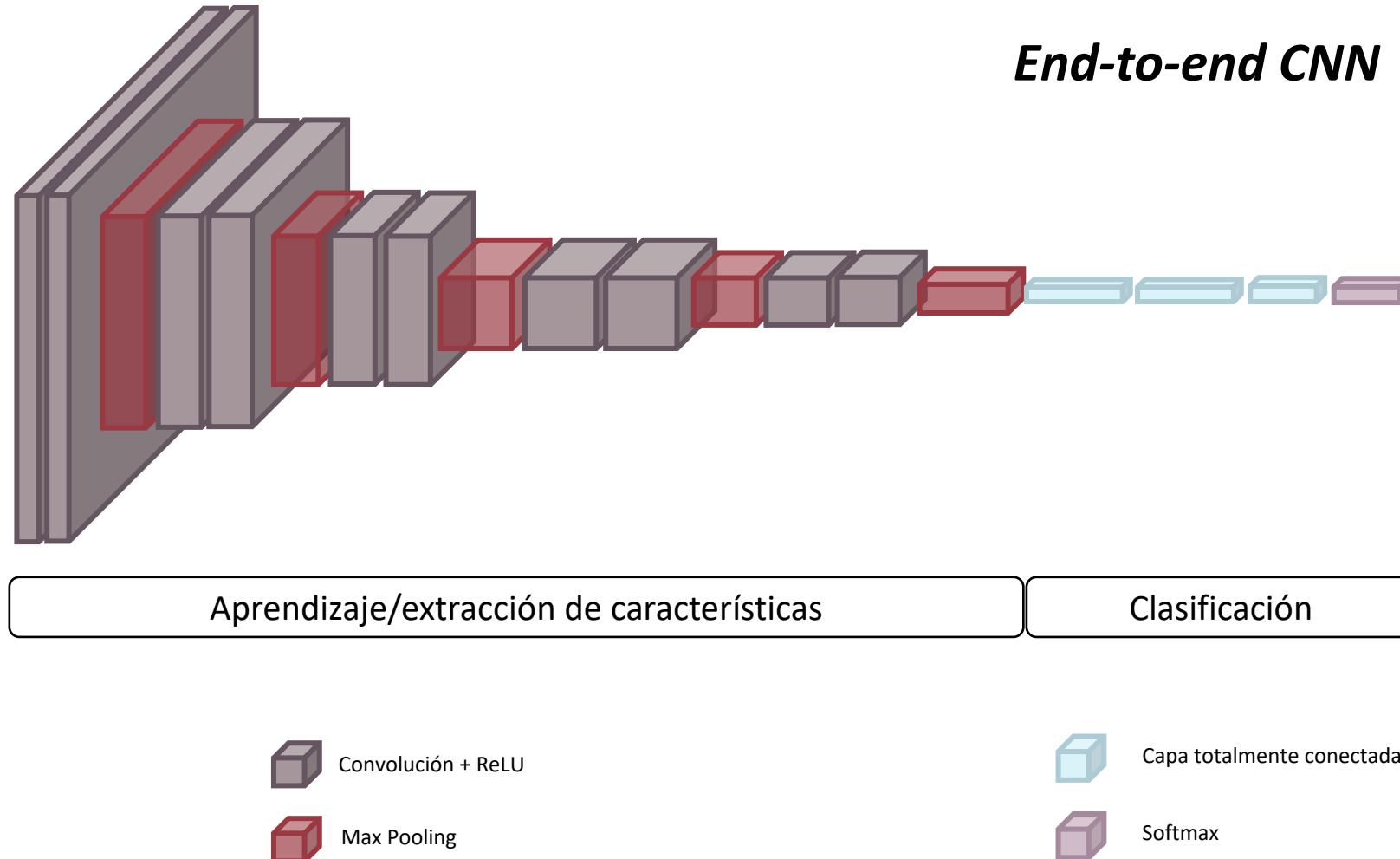
$$(32 * 750000) + (1 * 32) = 24000032$$

288



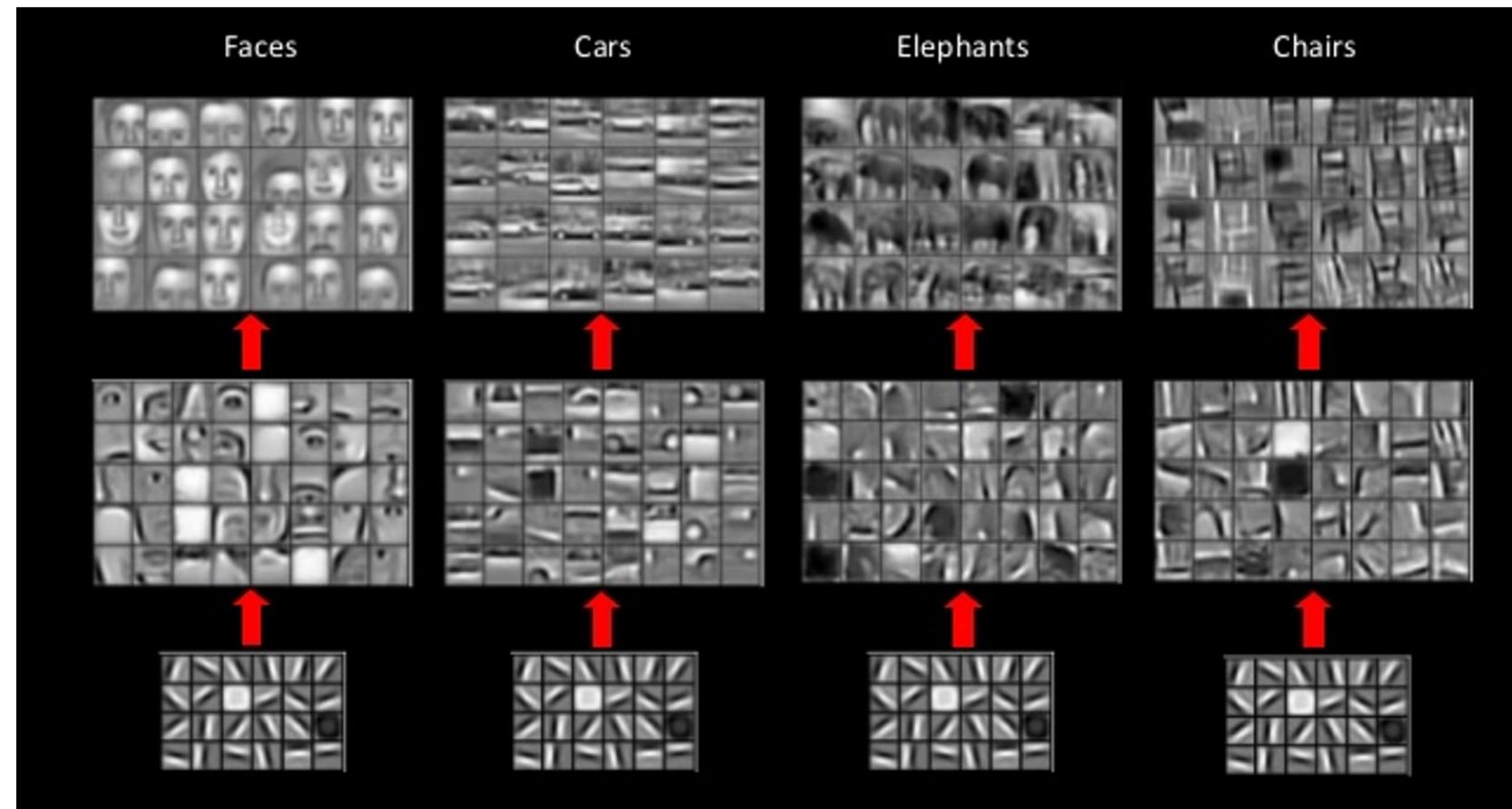
# Redes Neuronales Convolucionales

Las redes neuronales convolucionales están formadas por una primera parte de extracción de características formada por la **concatenación de varios bloques convolucionales**, seguidos de, en el caso de la clasificación o regresión, **redes neuronales totalmente conectadas**.



# Redes Neuronales Convolucionales

A mayor profundidad en las capas de la red, los mapas de características van a ser de más alto nivel.



<https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html>

# Redes Neuronales Convolucionales

## Ejercicio 3

Implementa la misma arquitectura que en el Ejercicio 2 y añade las siguientes capas:

- Capa Flatten.
- Dos capas oculta densas con 16 y 32 neuronas respectivamente, ambas con función de activación ReLU.
- Una capa de salida con 1 neurona y función de activación sigmoide (“sigmoid”).

Preguntas:

- ¿Cuántos parámetros debe ajustar el modelo en total?

# Redes Neuronales Convolucionales

## Ejercicio 4

Implementa la misma arquitectura que en el Ejercicio 3 pero emplea una capa GlobalAveragePooling2D en lugar de Flatten.

Preguntas:

- ¿Cuántos parámetros debe ajustar el modelo en total?

# Cargar y preparar los datos

# Cargar y preparar los datos

Existen dos formas principales con las que podemos cargar los datos para trabajar con modelos de CNN:

1. Cargar todas las imágenes en **memoria** previo al entrenamiento
  - **Ventaja:** no se depende de métodos compatibles con únicamente determinados formatos de imagen.
  - **Inconveniente:** requiere más memoria y dependiendo de la máquina que tengamos y la base de datos con la que trabajemos puede que no tengamos memoria suficiente para cargar todas las imágenes en memoria.
2. Cargar las imágenes en **lotes** durante el proceso de entrenamiento (en cada iteración se cargan las imágenes que se van a emplear).
  - **Ventaja:** requiere menos memoria (únicamente se carga un lote de imágenes simultáneamente) y nos permite trabajar con bases de datos más grandes.
  - **Inconveniente:** existen métodos para trabajar únicamente con determinados formatos de imagen, si queremos trabajar con otros tenemos que implementar/adaptar el método correspondiente para el *framework* con el que trabajemos (Tensorflow/Keras, Pytorch, etc.)

# Cargar y preparar los datos

Existen dos formas principales con las que podemos cargar los datos para trabajar con modelos de CNN:

- ~~1. Cargar todas las imágenes en memoria previo al entrenamiento~~
  - ~~• Ventaja: no se depende de métodos compatibles con únicamente determinados formatos de imagen.~~
  - ~~• Inconveniente: requiere más memoria y dependiendo de la máquina que tengamos y la base de datos con la que trabajemos puede que no tengamos memoria suficiente para cargar todas las imágenes en memoria.~~
2. Cargar las imágenes en **lotes** durante el proceso de entrenamiento (en cada iteración se cargan las imágenes que se van a emplear).
  - Ventaja:** requiere menos memoria (únicamente se carga un lote de imágenes simultáneamente) y nos permite trabajar con bases de datos más grandes.
  - Inconveniente:** existen métodos para trabajar únicamente con determinados formatos de imagen, si queremos trabajar con otros tenemos que implementar/adaptar el método correspondiente para el *framework* con el que trabajemos (Tensorflow/Keras, Pytorch, etc.)

# Cargar y preparar los datos

En primer lugar, vamos a hacer uso de Tensorflow para descargar un dataset público de una URL:

```
import tensorflow as tf  
  
_URL = 'https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'  
path_to_zip = tf.keras.utils.get_file('cats_and_dogs_filtered.zip', origin=_URL, extract=True)
```

Especificamos que además de descargar queremos descomprimir la carpeta

Vemos que hemos descargado:

```
from pathlib import Path  
  
path_to_zip = Path(path_to_zip)  
path_datasets = path_to_zip.parent  
print(list(path_datasets.iterdir()))  
path_dataset = path_datasets / 'cats_and_dogs_filtered'  
print('Contenido carpeta descargada: ', list(path_dataset.iterdir()))
```

Salida:

```
['cats_and_dogs_filtered', 'cats_and_dogs_filtered.zip']
```

```
Contenido carpeta descargada: ['vectorize.py', 'train', 'validation']
```

Vemos que esta base de datos contiene dos carpetas: 'train' y 'validation' correspondiente a las imágenes de entrenamiento y validación

# Cargar y preparar los datos

Vemos que contienen las carpetas de entrenamiento y validación:

```
path_train = path_dataset / 'train'  
path_val = path_dataset / 'validation'  
  
print('Contenido carpeta entrenamiento: ')  
print(list(path_train.iterdir()))  
print('Contenido carpeta validación: ')  
print(list(path_val.iterdir()))
```

**Salida:**

Contenido carpeta entrenamiento: ['dogs', 'cats']

Contenido carpeta validación: ['dogs', 'cats']

Dentro de las carpetas de entrenamiento y validación tenemos dos carpetas adicionales: 'dogs' y 'cats' correspondientes a las dos clases que queremos clasificar

Finalmente comprobamos que contienen las carpetas. Por ejemplo, miramos la de 'dogs' de entrenamiento:

```
path_dogs_train = path_train / 'dogs'  
list(path_dogs_train.iterdir())
```

**Salida:**

['dog.775.jpg', 'dog.761.jpg', 'dog.991.jpg', 'dog.749.jpg',  
'dog.985.jpg', 'dog.952.jpg', 'dog.946.jpg', 'dog.211.jpg', ...]

Finalmente tenemos las imágenes pertenecientes, en este caso, a las imágenes de entrenamiento de la clase perro.

# Cargar y preparar los datos

Por lo tanto, esta es la estructura de carpetas que tenemos:



# Cargar y preparar los datos

## Cargado de las imágenes en lotes

Para implementar esta aproximación vamos a hacer uso del método que ofrece Tensorflow/Keras [image dataset from directory](#) (NOTA: Esto solo se puede emplear para problemas de clasificación, en siguientes temas veremos maneras más genéricas de cargar un dataset).

Este método facilita la creación de datasets legibles por Tensorflow. Solo necesitamos tener las imágenes en nuestro sistema de archivos ordenadas de manera apropiada. Esto es, las imágenes pertenecientes a cada clase en una carpeta diferente. No es necesario tener carpeta de entrenamiento y validación, podemos crear posteriormente las particiones si todas las imágenes se encuentran en una única carpeta. De este modo, las etiquetas son inferidas, es decir, genera lotes de imágenes de los subdirectorios dentro de la carpeta, asignándoles etiquetas diferentes. Si no se especifica de otra forma, la asignación de etiquetas se realiza en orden alfabético (cats -> 0, dogs -> 1 en nuestro caso).

La base de datos que nos hemos descargado ya se encuentra en este formato.

# Cargar y preparar los datos

## Cargado de las imágenes en lotes

```
from tensorflow.keras.preprocessing import image_dataset_from_directory

# A diferencia del método anterior, aquí ya tenemos que especificar el tamaño del lote que se va
# a emplear durante el entrenamiento
BATCH_SIZE = 32

# El argumento "shuffle" a True hace que las imágenes se carguen y se reordenen de forma aleatoria
# Por lo que este paso ya no hará falta que lo hagamos manualmente como antes
train_dataset = image_dataset_from_directory(
    path_train,
    shuffle=True,
    batch_size=BATCH_SIZE,
    image_size=(n_rows, n_cols)
)
```

Definimos el tamaño de la imagen, así, de manera automática redimensionará la imagen a este tamaño cuando la cargue durante el entrenamiento, no será necesario que lo implementemos nosotros mismos.

```
validation_dataset = image_dataset_from_directory(
    path_val,
    shuffle=True,
    batch_size=BATCH_SIZE,
    image_size=(n_rows, n_cols)
)
```

```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
```

# Cargar y preparar los datos

## Cargado de las imágenes en lotes

Podemos explorar qué etiquetas ha sido capaz de identificar el método:

```
class_names = train_dataset.class_names  
print('Etiquetas encontradas: ', class_names)
```

**Salida:**

Etiquetas encontradas: ['cats', 'dogs']

# Cargar y preparar los datos

## Cargado de las imágenes en lotes

Visualizamos algunas imágenes:

```
plt.figure(figsize=(10, 10))
# En un dataset creado de la forma anterior podemos emplear el
# método "take" para coger n número de lotes. En nuestro caso
# vamos a coger 1
for images, labels in train_dataset.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        if labels[i] == 0:
            plt.title('Gato')
        elif labels[i] == 1:
            plt.title('Perro')
        plt.axis("off")
```



# Cargar y preparar los datos

## Cargado de las imágenes en lotes

Con los anterior hemos cargado las imágenes, las hemos redimensionado al tamaño deseado y las hemos reordenado aleatoriamente. Ahora vamos generar las particiones de entrenamiento, validación y test.

```
# Obtenemos cuantos lotes tiene la partición original de validación
val_batches = tf.data.experimental.cardinality(validation_dataset)
print('Número de lotes de validación original: ', val_batches.numpy())

# Asignamos la mitad a cada partición
test_dataset = validation_dataset.take(val_batches // 2) # Cogemos la mitad
validation_dataset = validation_dataset.skip(val_batches // 2) # Eliminamos la mitad
print('Número de lotes de validación: ', tf.data.experimental.cardinality(validation_dataset).numpy())
print('Número de lotes de test: ', tf.data.experimental.cardinality(test_dataset).numpy())

# Obtenemos numero de lotes que forman la partición de entrenamiento
train_batches = tf.data.experimental.cardinality(train_dataset)
print('Número de lotes de entrenamiento: ', train_batches.numpy())
```

### Salida:

Número de lotes de validación original: 32

Número de lotes de validación: 16

Número de lotes de test: 16

Número de lotes de entrenamiento: 63

# Cargar y preparar los datos

## Cargado de las imágenes en lotes

Cuando cargamos las imágenes durante el proceso de entrenamiento (en cada iteración se carga el número de imágenes definido) corremos el peligro de bloquear la E/S. Para que esto no pase y, así, mejorar el rendimiento en la carga de imágenes se emplea lo siguiente. **Esto lo ejecutaremos siempre que empleemos una aproximación de carga de datos como esta.**

```
AUTOTUNE = tf.data.AUTOTUNE

train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)
```

# Cargar y preparar los datos

## Cargado de las imágenes en lotes

Finalmente vamos a realizar el escalado de las imágenes. En este punto nos encontramos con un inconveniente, y es que las aproximaciones *featurewise* no las podemos aplicar en este caso (a no ser que calculemos la media y desviación estándar de manera independiente), ya que en memoria únicamente vamos a tener cargado 1 lote simultáneamente. Sin embargo, el resto de escalados si podemos aplicarlos (normalización y centrado y estandarización *samplewise*).

Vamos a ver el ejemplo de normalización:

```
# Vamos a hacer uso de la función "map" que aplica a cada registro dentro del dataset la transformación
# que nosotros definamos, en nuestro caso "scaling". A la función que llamemos con "map" siempre le van a
# entrar los mismos dos argumentos: "image" y "label".
def scaling_normalization(image, label):
    image = tf.cast(image/255., tf.float32)
    return image, label

train_dataset = train_dataset.map(scaling_normalization)
validation_dataset = validation_dataset.map(scaling_normalization)
test_dataset = test_dataset.map(scaling_normalization)
```

# Definición de la arquitectura

# Definición de la arquitectura

Una vez tenemos los datos listos y previo al entrenamiento del modelo debemos de definir la arquitectura. Para ello tenemos que especificar 3 aspectos fundamentales:

- **Conjunto de capas que forman la arquitectura** → Como hemos visto anteriormente, ante un problema de clasificación estará formado por dos partes principales:
  - Extracción de características, con capas convolucionales y de agrupamiento principalmente.
  - Clasificación, con capas totalmente conectadas o densas.
- **Tasa de aprendizaje** → Regula la magnitud de actualización de los pesos en cada iteración.
- **Función de coste** → Función a minimizar durante el proceso de entrenamiento.
- **Métricas** → Valores que muestran como de bien se comporta nuestro modelo. Estas se calculan en cada época sobre el conjunto de datos de entrenamiento y, en su caso, de validación.

Como estamos ante un problema de clasificación, las funciones de coste y métricas son las mismas que las que habéis visto en la parte de datos estructurados.

# Definición de la arquitectura

## Ejercicio 5

Implementa una arquitectura que se ajuste al siguiente “summary” donde el tamaño de los filtros convolucionales es de 3x3 y se emplea una función de activación de tipo ReLU en dichos filtros.

En la capa densa oculta también se emplea una función de activación ReLU.

Finalmente, en la capa densa de salida, se hace uso de la función de activación sigmoide (‘sigmoid’).

Model: "model_1"		
Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[None, 160, 160, 3]	0
conv2d_3 (Conv2D)	(None, 160, 160, 16)	448
max_pooling2d_2 (MaxPooling 2D)	(None, 80, 80, 16)	0
conv2d_4 (Conv2D)	(None, 80, 80, 32)	4640
max_pooling2d_3 (MaxPooling 2D)	(None, 40, 40, 32)	0
conv2d_5 (Conv2D)	(None, 40, 40, 64)	18496
flatten_1 (Flatten)	(None, 102400)	0
dense_2 (Dense)	(None, 32)	3276832
dense_3 (Dense)	(None, 1)	33
<hr/>		
Total params: 3,300,449		
Trainable params: 3,300,449		
Non-trainable params: 0		

# Definición de la arquitectura

## Compilación

Ya tenemos la arquitectura definida, para poder inicializar el entrenamiento solo nos queda definir la tasa de aprendizaje, la función de coste y las métricas.

La tasa de aprendizaje la definiremos a través de argumento "optimizer", la función de coste a través del argumento "loss" y las métricas irán definidas dentro de una lista y se definirán a través del argumento "metrics".

```
model_base.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

# Definición de la arquitectura

## Entrenamiento

Finalmente, vamos a hacer el entrenamiento del modelo, para ello vamos a hacer uso del método “[fit](#)” al cual le deberemos de especificar los siguientes argumentos (hay otros que no vamos a especificar y que emplearemos sus valores por defecto).

Además, durante el entrenamiento, vamos a hacer uso de un *callback* de Keras: ModelCheckpoint. Este permite almacenar el modelo que mejores resultados nos de sobre el conjunto de datos de validación de entre todas las épocas. Este *callback* es recomendable emplearlo **siempre**.

# Definición de la arquitectura

## Entrenamiento

```
# Creamos directorio donde almacenar resultados
path_models = Path('models')
path_experiment = path_models / 'Train1'
path_experiment.mkdir(exist_ok=True, parents=True)

# Inicializamos el callback
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=path_experiment / 'dogs_and_cats.h5',
    monitor='val_accuracy',
    mode='max',
    save_best_only=True,
    verbose=1)
```

Métrica a monitorizar: exactitud validación, nos quedamos con el valor máximo, y únicamente almacenamos el modelo que ofrece mejores resultados

#

### Entrenamos

```
history_cnn_base = model_base.fit(train_dataset,
                                    epochs=10,
                                    batch_size=32,
                                    validation_data=validation_dataset,
                                    callbacks=[model_checkpoint_callback],
                                    verbose=1)
```

Añadimos callback

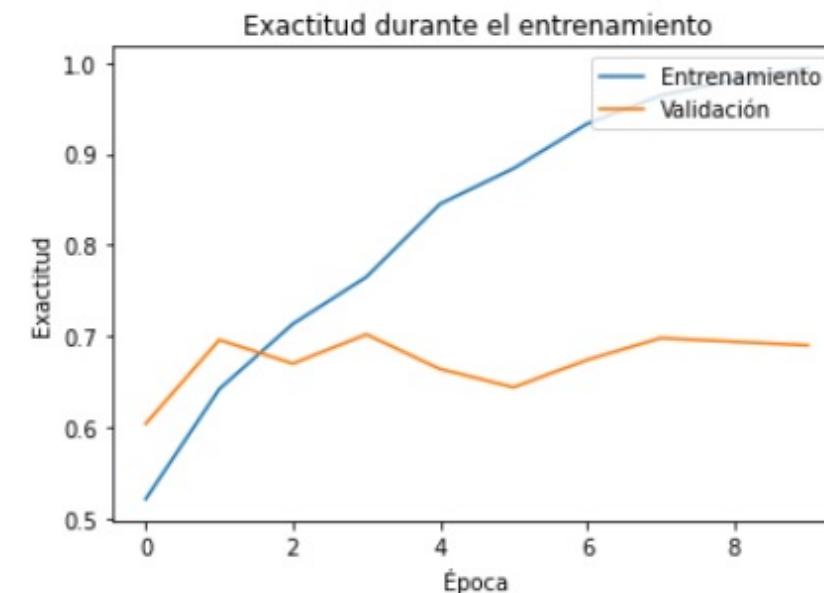
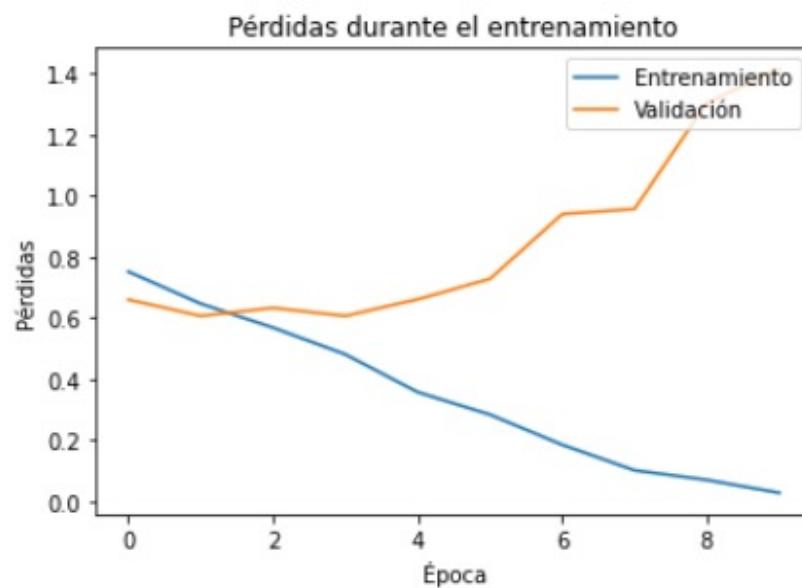
```
np.save(path_experiment / 'history.npy', history_cnn_base.history)
```

Almacenamos historia entrenamiento en disco

# Definición de la arquitectura

## Entrenamiento

Una vez finalizado el entrenamiento debemos visualizar como ha ido el mismo. Para ello deberemos ver las pérdidas y/o métricas sobre los datos de entrenamiento y validación.



En las gráficas anteriores vemos que llega un momento (aprox. época 2) en el que las pérdidas en entrenamiento continúan disminuyendo pero las de validación empiezan a aumentar → Sobreajuste.

# Definición de la arquitectura

## Evaluación

Finalmente, empleamos la partición de test para evaluar el rendimiento del modelo entrenado en un conjunto de datos independiente.

Debemos evaluar las métricas sobre el modelo óptimo. Si empleamos directamente en que esta en memoria, estaremos empleando el correspondiente a la última época. Para hacer uso de aquel que mejores resultados nos ha dado en validación, debemos cargar el modelo previamente.

```
# Cargamos el modelo
model_base = tf.keras.models.load_model(path_experiment / 'dogs_and_cats.h5')

# Extraemos métricas test
_, acc_base = model_base.evaluate(X_test, y_test, verbose=0)
print('Exactitud test: ', acc_base)
```

**Salida:**

Exactitud test: 0.6660000085830688

# Ejercicios 6-11

# **Tema 3. Introducción a las redes neuronales convolucionales**

Ana Jiménez Pastor  
[anjipas@gmail.com](mailto:anjipas@gmail.com)