

# Redes convolucionales

Programa ejecutivo de Inteligencia Artificial

Año de realización: 2019-2020

**Profesor:**  
**Alberto Torres Barrán**



# Introducción

# Repaso de Deep Learning

- Modelo principal: red profunda (**feed-forward**): composición de **proyecciones lineales** y **no-linealidades**

$$\begin{aligned} h^{(i+1)} &= Wz^{(i)} + b \\ z^{(i+1)} &= \sigma(h^{(i+1)}) \end{aligned}$$

- Al final: añadir coste apropiado para regresión o clasificación.
- Las redes neuronales se conocen desde mediados del siglo XX, pero su fuerte resurgimiento no fue hasta esta década:
  - Paralelización en tarjetas gráficas (**GPUs**).
  - Librerías de **diferenciación automática**.

# Diferenciación Automática (AD) (1)

- ¿Cómo calcular el gradiente en una red profunda?
- **A mano:** no escala a nuevas arquitecturas, propenso a errores.
- **Diferenciación numérica:** acumulación de errores y elevado coste computacional.

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon} + O(\epsilon^2)$$

- **Diferenciación simbólica:** manipulación exacta de expresiones (mediante tablas de derivadas), pero explosión en la cantidad de términos:

$n$	$l_n$	$\frac{d}{dx} l_n$
1	$x$	1
2	$4x(1-x)$	$4(1-x) - 4x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2 - 16x(1-2x)^2 - 64x(1-x)(1-2x)$
4	$64x(1-x)(1-2x)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2$

# Diferenciación Automática (AD) (2)

- Surge la **diferenciación automática o algorítmica**: aplica diferenciación simbólica pero solo a expresiones simples, y al componerlas, actualiza los resultados numéricos parciales (que serán **exactos**)
- Ejemplo para calcular la derivada de  $y = f(x_1, x_2) = \log(x_1) + x_1x_2 - \sin(x_2)$  en  $(x_1, x_2) = (2, 5)$ :

Forward Primal Trace			Reverse Adjoint (Derivative) Trace		
$v_{-1} = x_1$	= 2		$\bar{x}_1 = \bar{v}_{-1}$	= 5.5	
$v_0 = x_2$	= 5		$\bar{x}_2 = \bar{v}_0$	= 1.716	
$v_1 = \ln v_{-1}$	= $\ln 2$		$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	= $\bar{v}_{-1} + \bar{v}_1/v_{-1} = 5.5$	
$v_2 = v_{-1} \times v_0$	= $2 \times 5$		$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	= $\bar{v}_0 + \bar{v}_2 \times v_{-1} = 1.716$	
$v_3 = \sin v_0$	= $\sin 5$		$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	= $\bar{v}_2 \times v_0$	= 5
$v_4 = v_1 + v_2$	= $0.693 + 10$		$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	= $\bar{v}_3 \times \cos v_0$	= -0.284
$v_5 = v_4 - v_3$	= $10.693 + 0.959$		$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	= $\bar{v}_4 \times 1$	= 1
$y = v_5$	= 11.652		$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	= $\bar{v}_4 \times 1$	= 1
			$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	= $\bar{v}_5 \times (-1)$	= -1
			$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	= $\bar{v}_5 \times 1$	= 1
			$\bar{v}_5 = \bar{y}$	= 1	

# Diferenciación Automática (AD) (3)

- ¿Por qué **backpropagación**?
- Ejemplo: considera una serie de funciones  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ ,  $g : \mathbb{R}^m \rightarrow \mathbb{R}^m$  y  $h : \mathbb{R}^m \rightarrow \mathbb{R}$ . Queremos obtener la derivada de su composición,  $\frac{\partial(h \circ g \circ f)}{\partial x}$
- Queda que

$$\frac{\partial(h \circ g \circ f)}{\partial x} = \frac{\partial h}{\partial g} \frac{\partial g}{\partial f} \frac{\partial f}{\partial x}$$

- Asociando  $\frac{\partial h}{\partial g} \left( \frac{\partial g}{\partial f} \frac{\partial f}{\partial x} \right)$ , queda un producto matriz-matriz y otro producto vector-matriz.
- Asociando  $\left( \frac{\partial h}{\partial g} \frac{\partial g}{\partial f} \right) \frac{\partial f}{\partial x}$ , ¡solo hay que hacer productos vector-matriz!: mucho más eficiente.
- En ML es habitual optimizar funciones de tipo  $\mathbb{R}^d \rightarrow \mathbb{R}$ , por tanto es más eficiente propagar los gradientes hacia atrás (**backpropagation**) que hacia adelante (**forward propagation**).

# Optimizando mediante SGD o Adam.

- Una vez hemos calculado el gradiente en un punto mediante AD, las opciones actuales más populares son
- **Descenso por el gradiente estocástico (SGD)**: ya visto, estimación usando mini-batches.
- **Adam** (2014: <https://arxiv.org/abs/1412.6980>) : corrige el gradiente estimando una ventana móvil de su media y de su varianza.

```
while  $\theta_t$  not converged do
     $t \leftarrow t + 1$ 
     $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )
     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)
     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)
     $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)
     $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)
     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)
end while
```

# Volviendo a Deep Learning

- Ya tenemos todos los ingredientes:
  - Datasets enormes.
  - Redes neuronales como **aproximadores universales**.
  - Librerías para **diferenciación automática**: tensorflow, keras, pytorch...
  - Potentes CPUs o GPUs para **paralelizar a lo largo de cada ejemplo del mini-batch**.
- ¿Qué falta en muchas ocasiones?
  - Solo teóricamente las redes neuronales son totalmente expresivas.
  - Conviene añadir un **sesgo inductivo (inductive bias)** para ayudar al aprendizaje:
    - Imágenes, señales: invarianza a traslaciones, escala: **redes convolucionales**.
    - Texto, secuencias: sensibilidad al orden de los símbolos: **redes recurrentes**.

# Redes convolucionales

# Introducción

- Tipo de red neuronal para datos con topología similar a una rejilla
  1. 1D, series temporales, audio
  2. 2D, imágenes, datos espaciales
  3. 3D, video, datos espacio-temporales, meteorología
- Red convolucional: en al menos una capa se usan convoluciones en lugar de operaciones con matrices

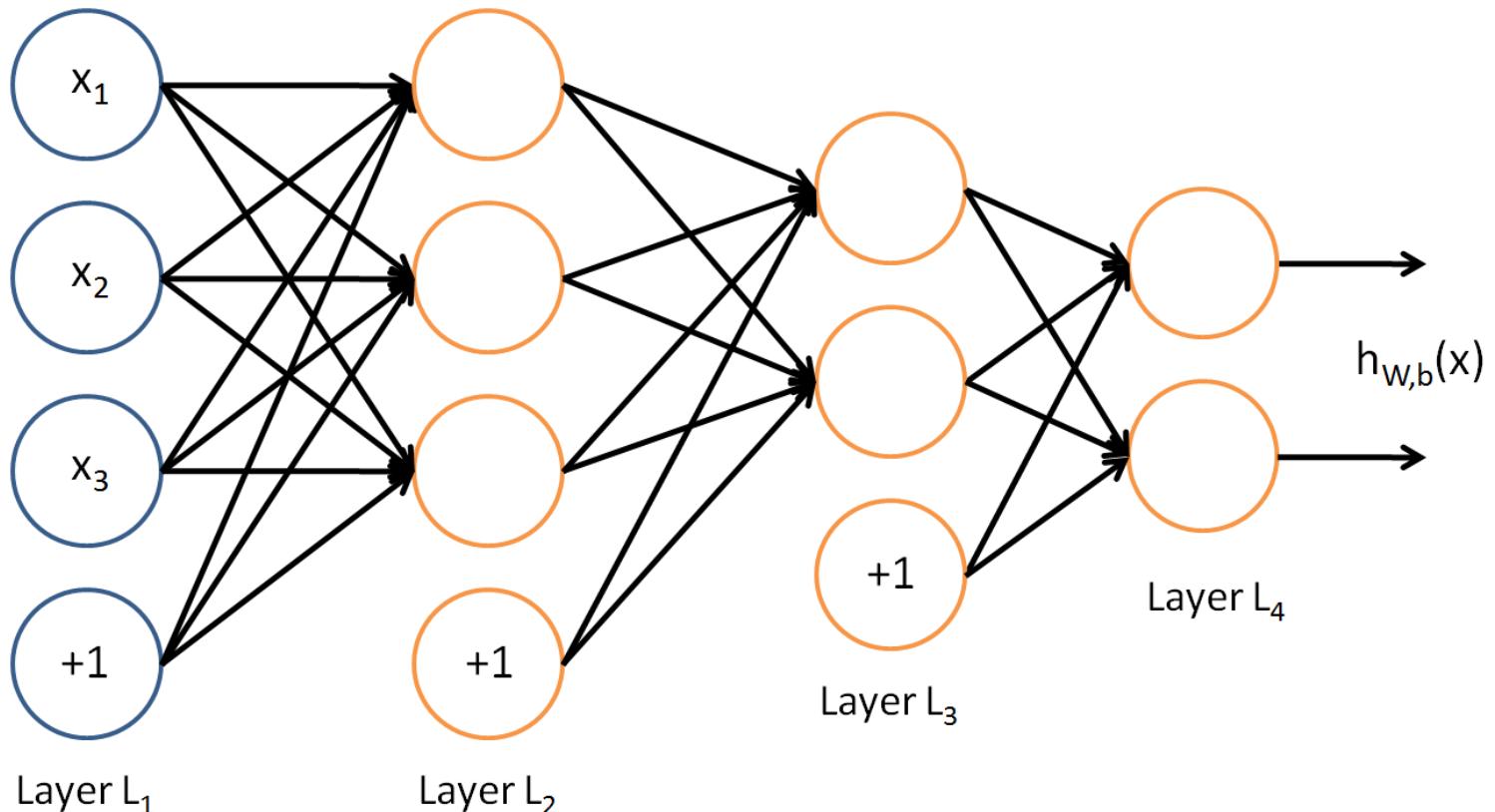
# Competición ImageNet

- Más de 14 millones de imágenes anotadas a mano
- Más de 20,000 categorías
- Desde 2010, competición anual de clasificación automática (ILSVRC)
  - únicamente 1000 categorías
  - en 2011, el mejor error era de aprox. 25%
  - en 2017, 29/38 equipos tenían un error menor del 5%

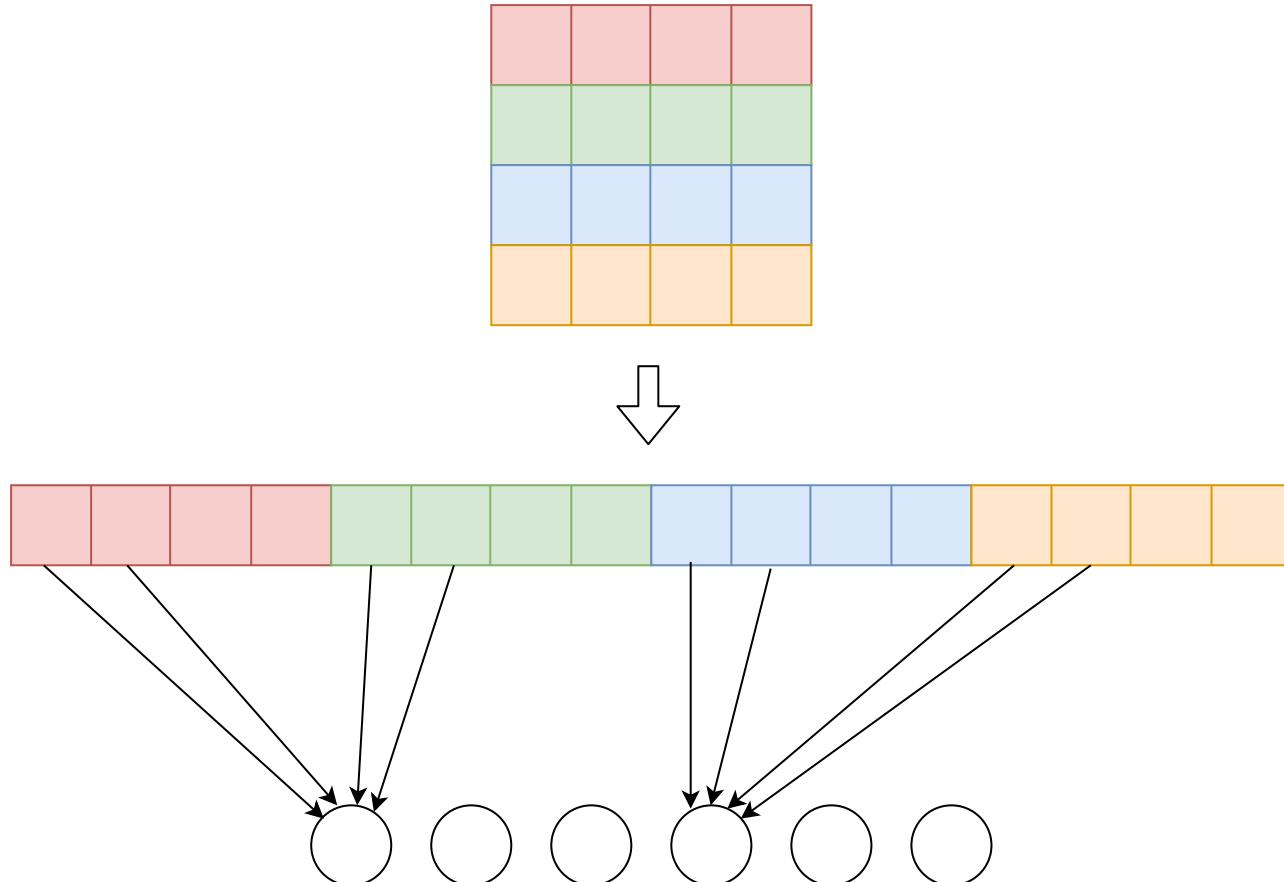
# Historia

1. En 1990, **Lecun et al.** usa una CNN para leer dígitos de códigos postales
  - una de las primeras aplicaciones reales de una red neuronal
  - más del 90% de tasa de acierto
2. En 2012, **Krizhevsky et al.** usan una CNN para ganar la competición ILSVRC2012
  - tasa de acierto (top 5), 15.3%
  - segundo mejor modelo, 26.2%
3. A partir de 2012 múltiples arquitecturas más complejas siguen reduciendo el error:
  - 2014: VGG-16 (7.3%), GoogleNet (6.7%)
  - 2015: Microsoft ResNet (3.57%)

# Conexiones densas (*fully connected*)



# Conexiones sparse (*locally connected*)



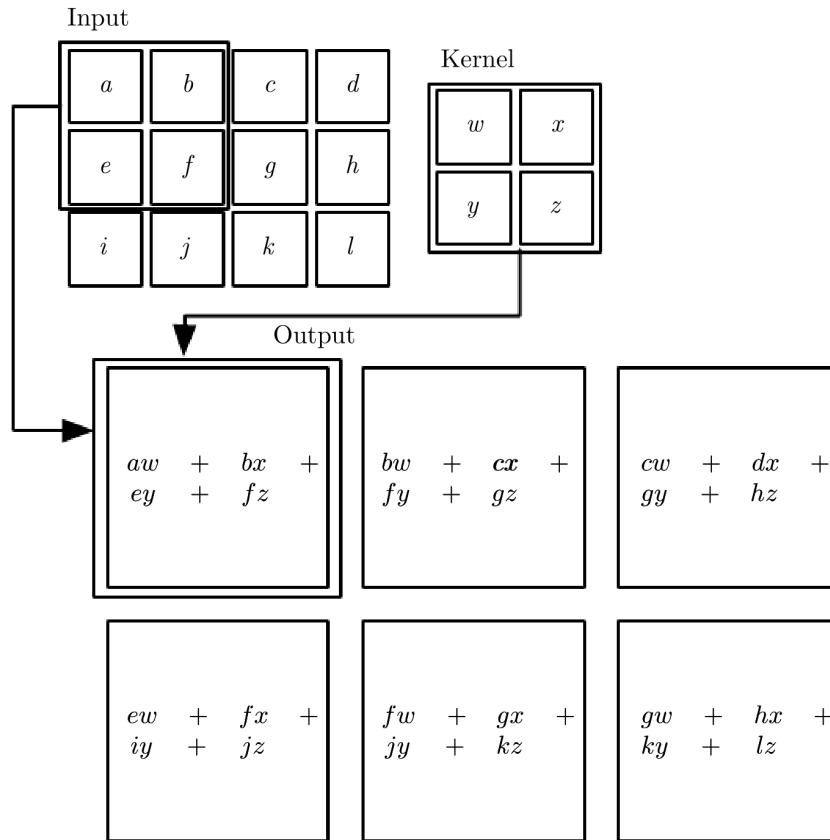
# Convolución en 2D

- $I$  es la matriz de entrada (2D)
- $K$  es el kernel (2D)

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

- La convolución o filtro se aplica a toda la imagen con los mismos pesos
- Se define con 4 parámetros:
  - *stride* o paso de la convolución
  - tamaño del kernel, generalmente cuadrado
  - *depth*, número de filtros o convoluciones distintas a aplicar
  - *padding*

# Ejemplo



Goodfellow et al. Deep Learning (2016)

# Motivación

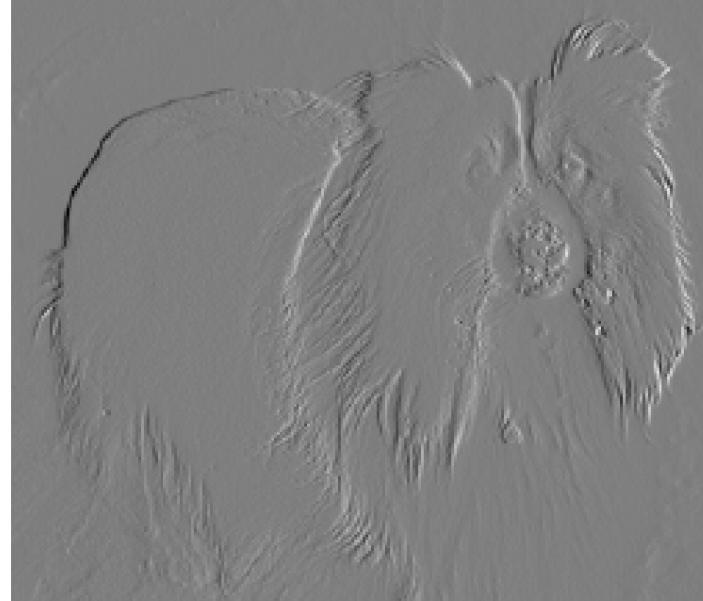
## 1. conexiones dispersas

- explotar estructura espacial
- detectar características locales (aristas, etc.)

## 2. compartición de pesos

- invariante frente a traslaciones
- reduce la cantidad de memoria necesaria

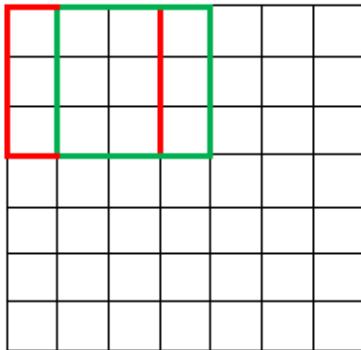
# Ejemplo características locales



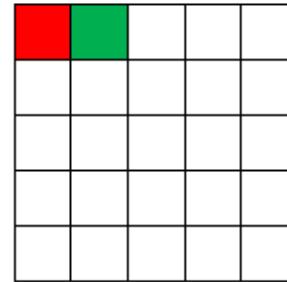
- Imagen de la derecha: restar a cada píxel su vecino por la izquierda
- Esta operación se puede representar de forma muy eficiente con una convolución

# *Stride (paso)*

7 x 7 Input Volume

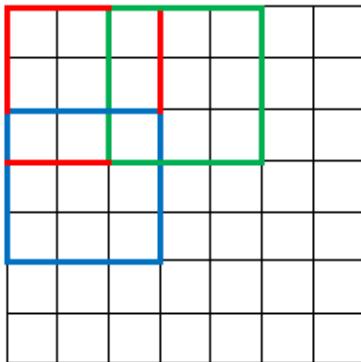


5 x 5 Output Volume

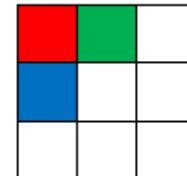


**stride = 1**

7 x 7 Input Volume



3 x 3 Output Volume

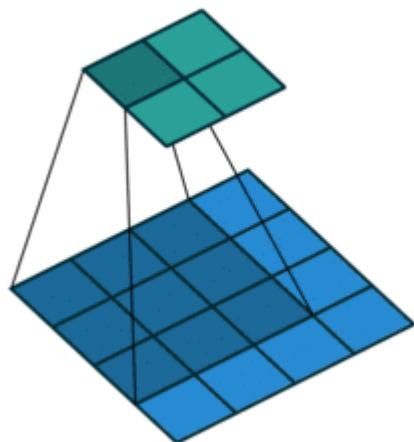


**stride = 2**

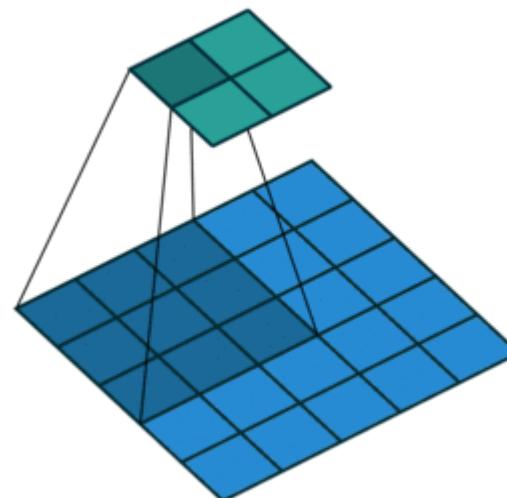
Fuente

# Ejemplos

- Entrada  $4 \times 4$
- Kernel  $3 \times 3$
- Stride 1
- Salida  $2 \times 2$

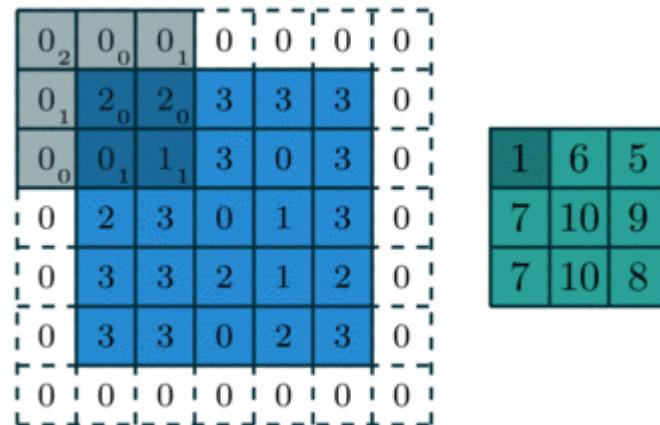


- Entrada  $5 \times 5$
- Kernel  $3 \times 3$
- Stride 2
- Salida  $2 \times 2$

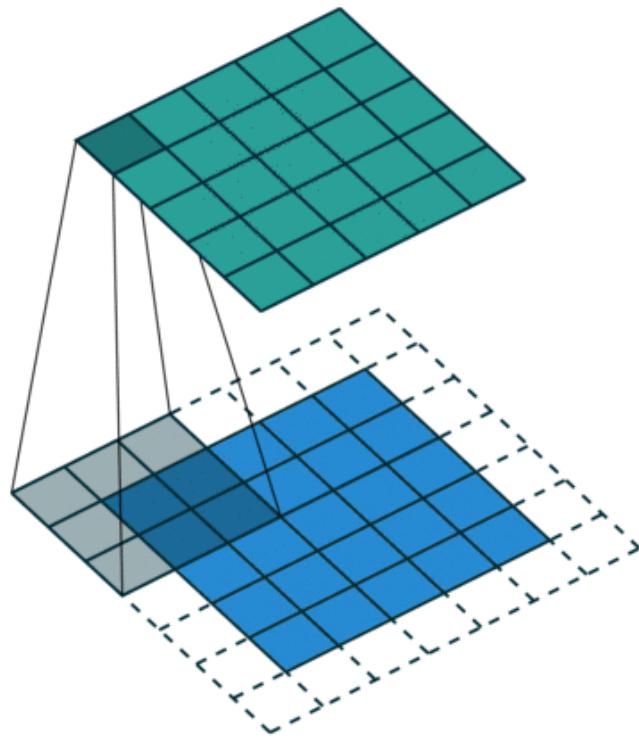


# Padding

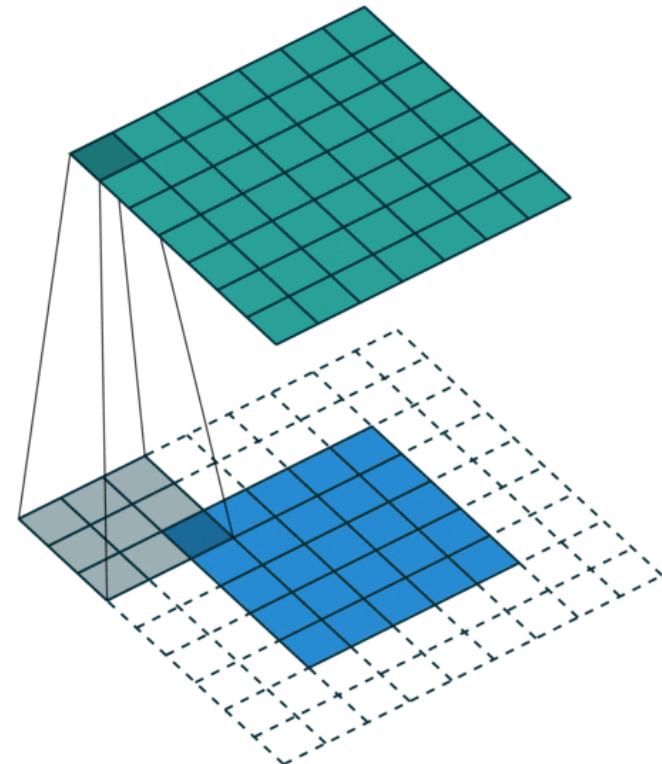
- En ocasiones se añade un *padding* de 0 al borde de la imagen:
  1. preservar el tamaño de la entrada
  2. cuando es necesario por la combinación de tamaño de entrada, tamaño de kernel y stride
- Ejemplo: entrada  $5 \times 5$ , kernel  $3 \times 3$  y *stride* 2



- Generalmente la salida tiene menor tamaño que la entrada
- Aplicando padding podemos hacer que tenga el mismo o incluso mayor
- Entrada  $5 \times 5$ , stride 1, kernel  $3 \times 3$

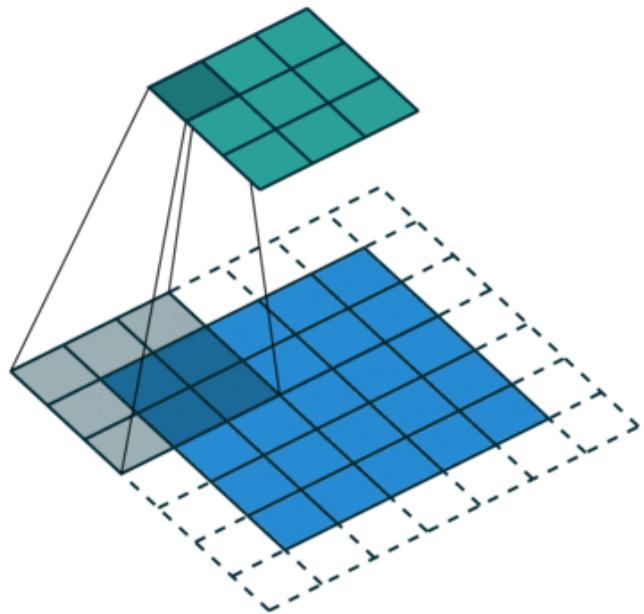


Salida  $5 \times 5$

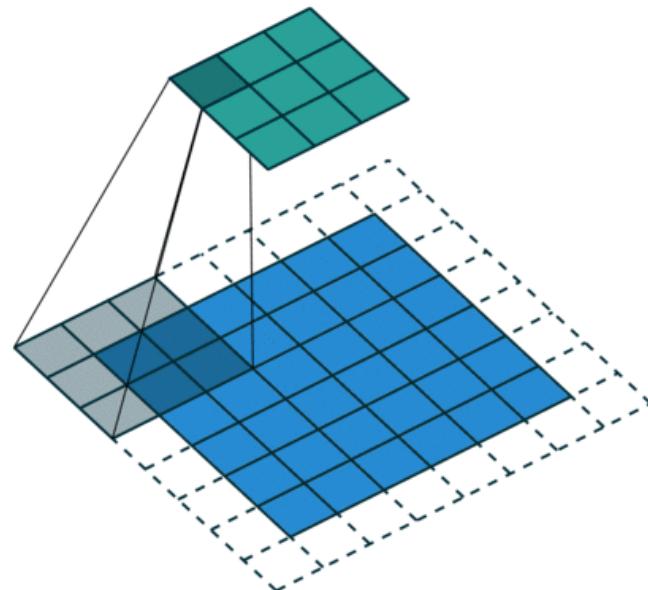


Salida  $7 \times 7$

- A veces el padding es necesario para poder aplicar el kernel
- Ejemplo: kernel  $3 \times 4$ , stride 2
- La salida tiene el mismo tamaño en ambos!



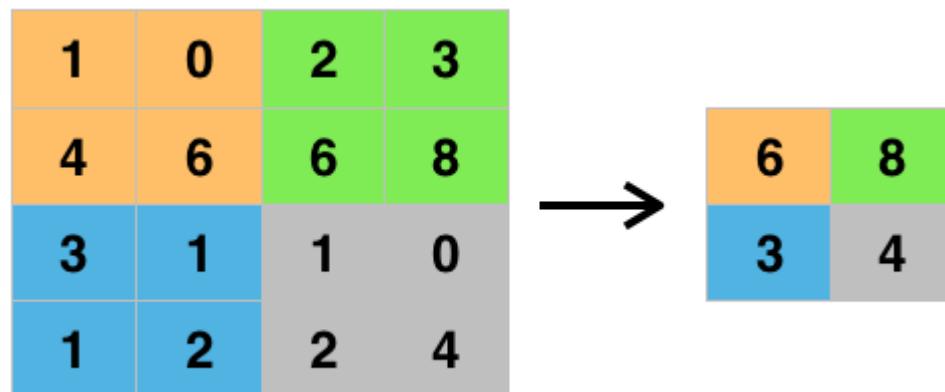
Entrada  $5 \times 5$



Entrada  $6 \times 6$

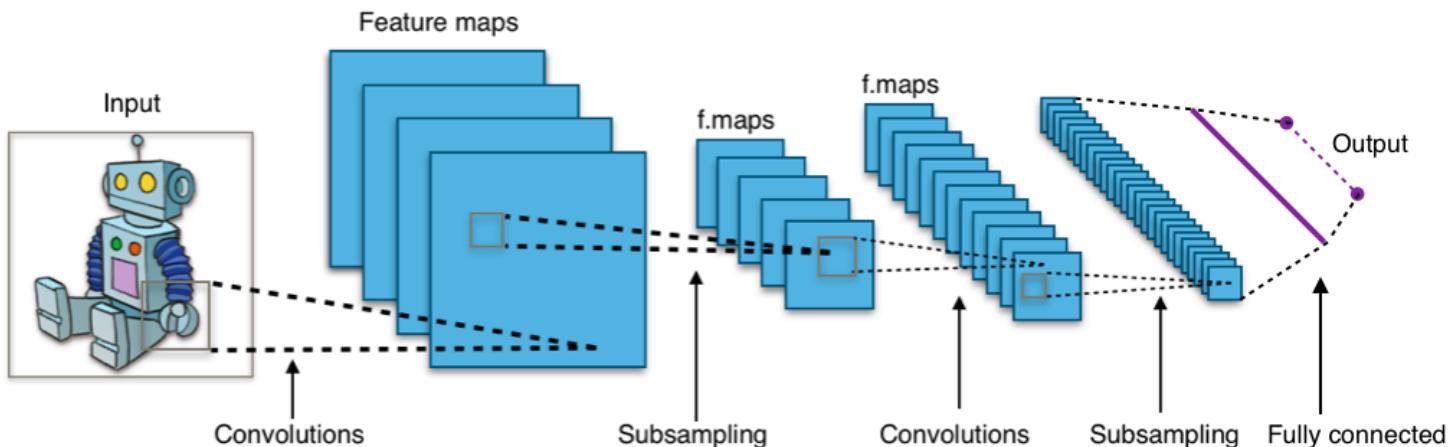
# Pooling

- Capa de submuestreo no lineal
- Previene sobreajuste, reduciendo número de parámetros
- Ayuda con la invarianza frente a traslaciones
- Útil cuando interesa conocer si una característica está o no, pero no su localización exacta  $\Rightarrow$  clasificación imágenes
- Más habitual: **max pooling**

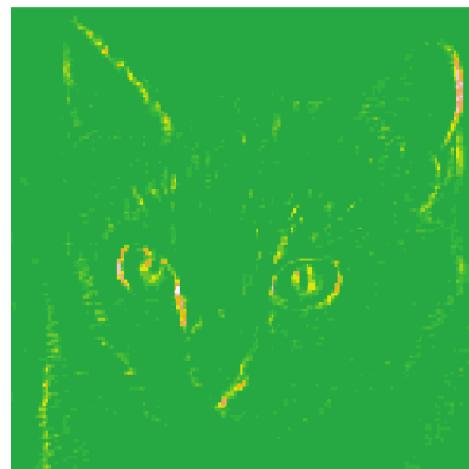
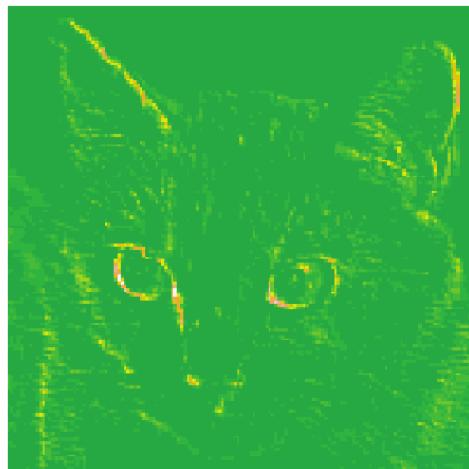


# Arquitectura típica

- Parámetros: número de *feature maps*, tamaño del kernel, stride
- Subsampling: max pooling
- Antes de las capas *fully connected*, hay que aplanar (*flatten*) la salida



# Visualizando activaciones



# Primeras capas

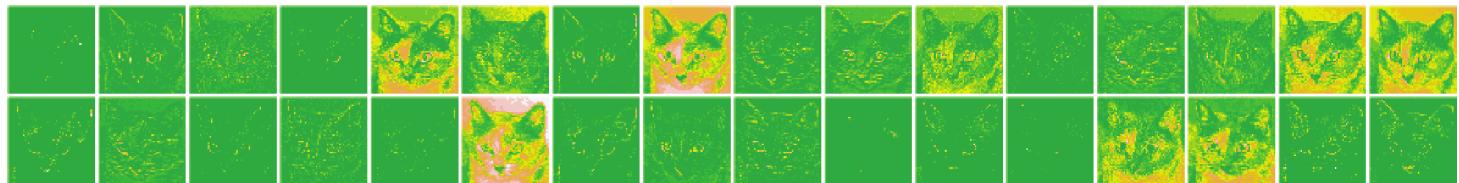


Figure 5.20 conv2d\_5

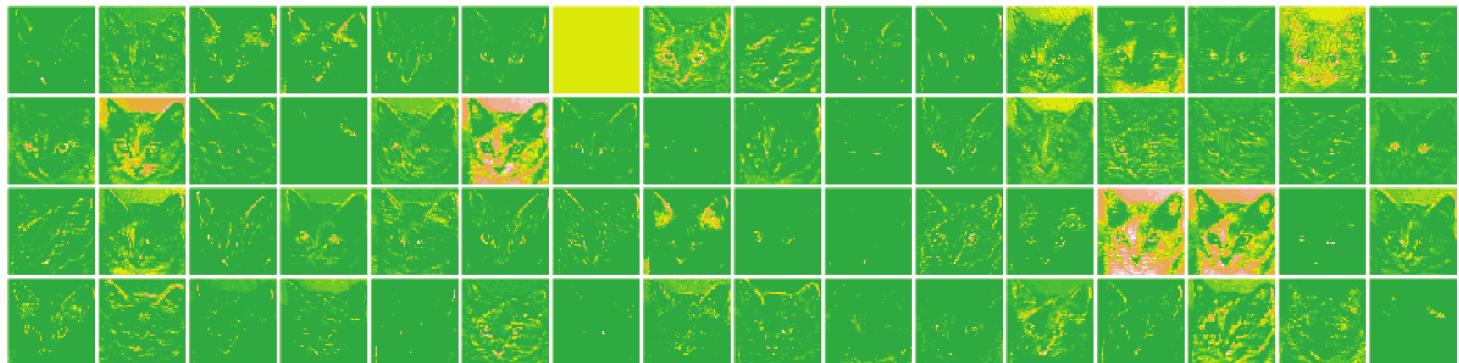


Figure 5.21 conv2d\_6

# Últimas capas

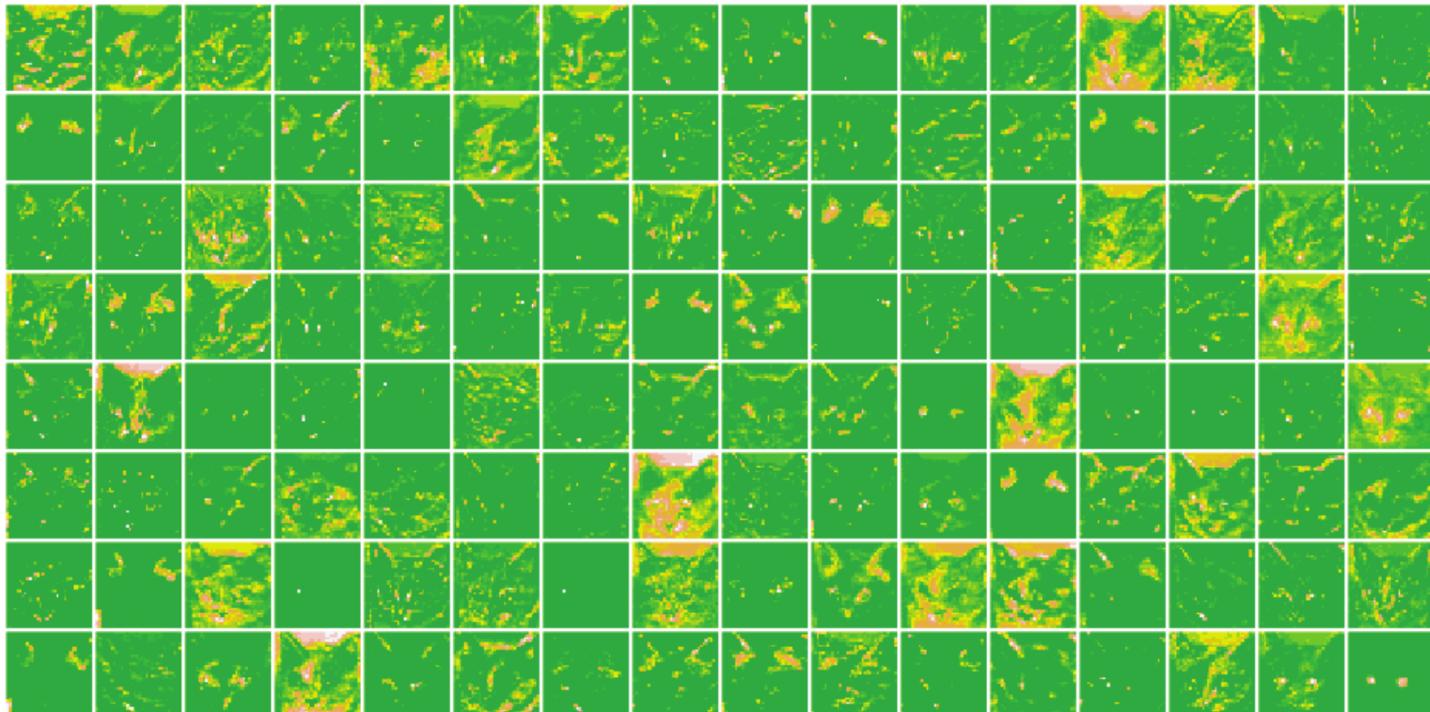


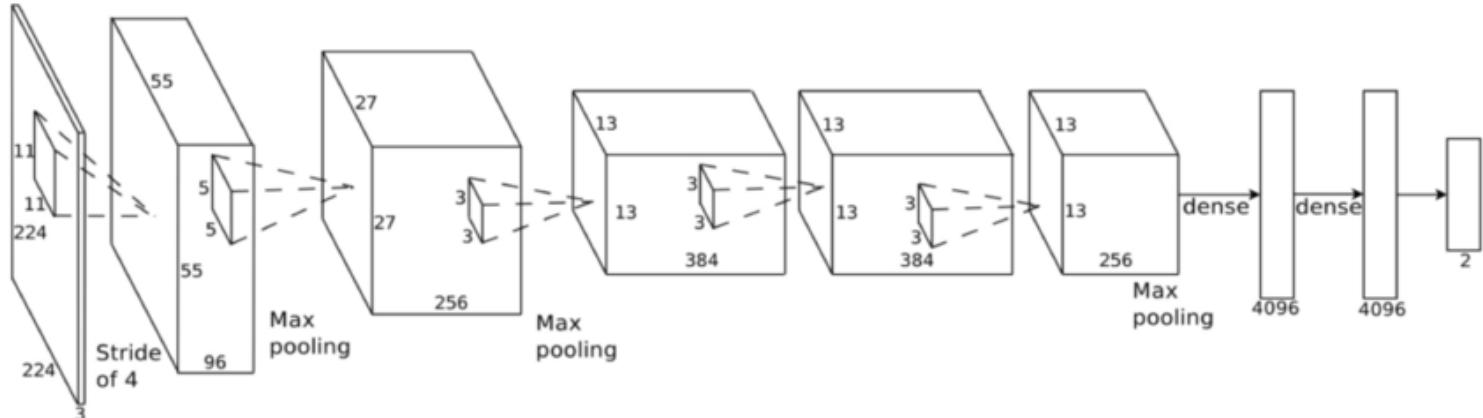
Figure 5.22 conv2d\_7

# CAM (class activation map)



Figure 5.32 Superimposing the class-activation heatmap on the original picture

# Ejemplo arquitectura: AlexNet



```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 96,
                kernel_size = c(11, 11),
                activation = 'relu',
                input_shape = c(224, 224, 3),
                strides = c(4, 4),
                padding = 'valid') %>%
  layer_max_pooling_2d(pool_size = c(2, 2),
                      strides = c(2, 2),
                      padding = 'valid') %>%
  ...
```

# Recursos adicionales

# Enlaces de interés

- <https://reddit.com/r/LearnMachineLearning>: nivel introductorio/medio
- <https://reddit.com/r/machinelearning>: discusiones sobre artículos y temas de actualidad
- <https://medium.com/topic/machine-learning>: artículos hacia audiencia general