



Tema 6:

Deep Learning

Departamento de Tecnologías de la Información

Área de Ciencias de la Computación
e Inteligencia Artificial



Instituto Andaluz
Interuniversitario en
Data Science and
Computational Intelligence



Tema 6: Deep Learning

Índice:

1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

Tema 6: Deep Learning

Índice:

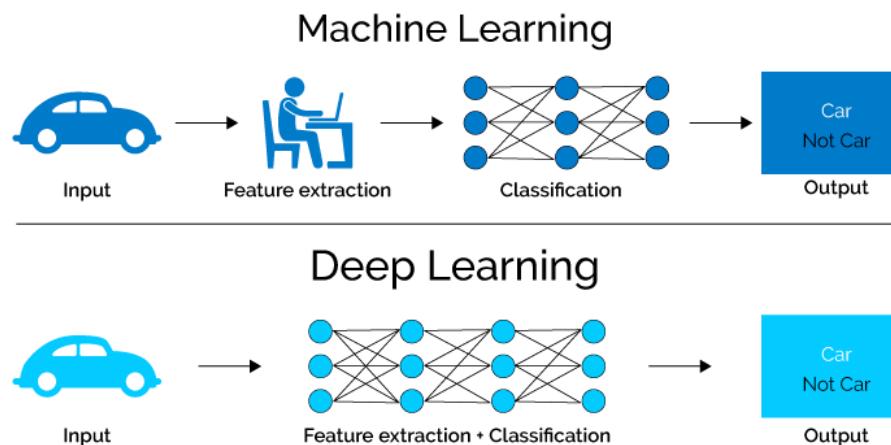
1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

Estamos en "la era" del:

- **Big Data**
 - Explotación de los datos

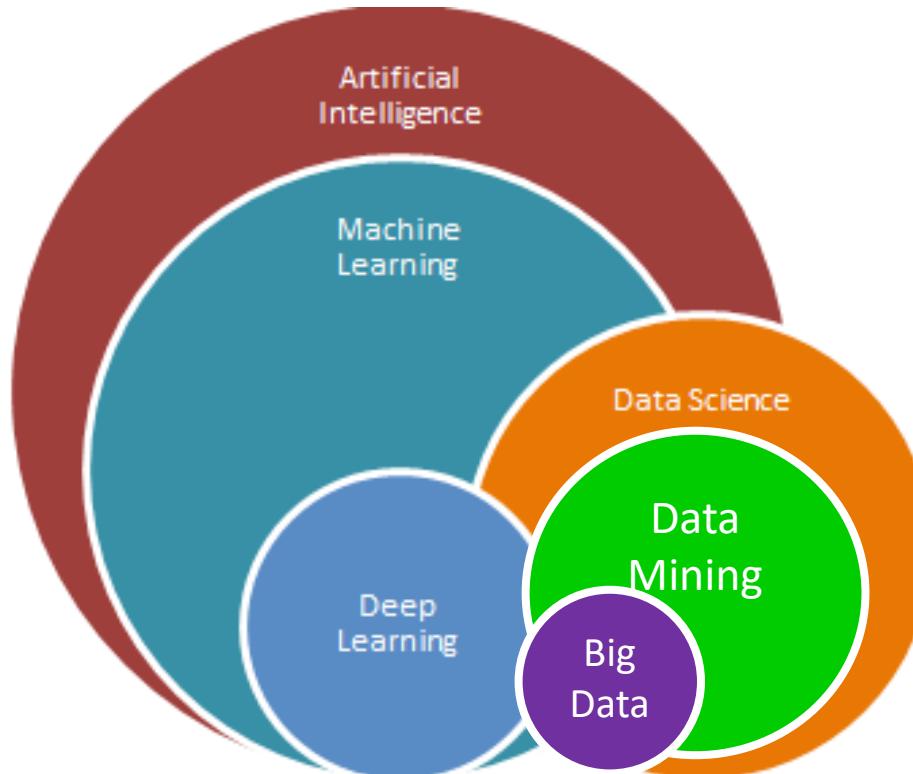


- **Deep Learning**
 - Más allá del *Machine Learning*

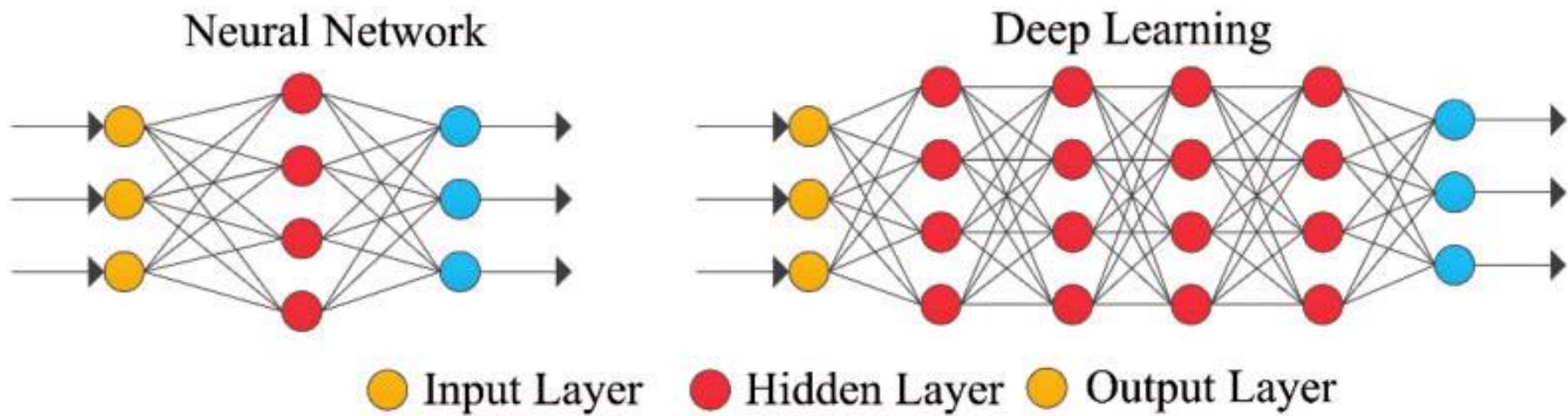


- **Aplicaciones:** comercialización de la IA

... quién es quién:

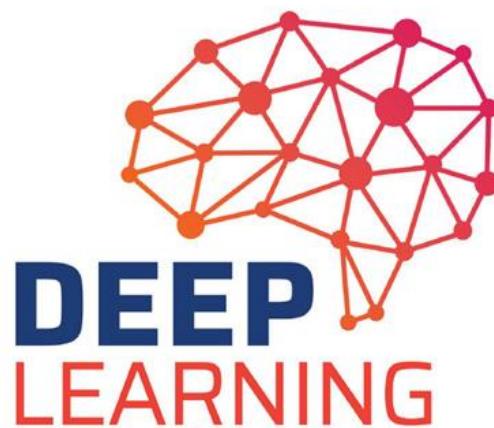


Redes Neuronales Artificiales vs Deep Learning



La Irrupción del *Deep Learning*:

- Hace unos pocos años surge **un nuevo protagonista** en el ámbito de IA:



- Decenas de aplicaciones empiezan a invadir nuestra vida, hasta el punto de convertirse en uno de los **mayores exponentes actuales de la Inteligencia Artificial**

La Irrupción del *Deep Learning*:

- En 2010 nace una empresa, que en 2014 es vendida a Google por 400 millones de euros:

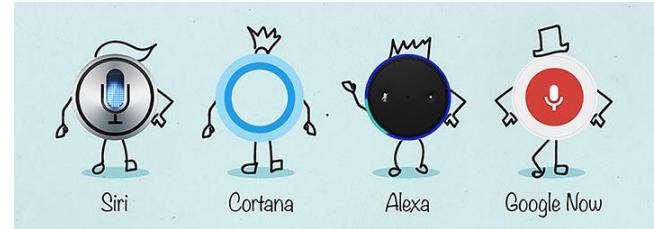


- Aplicaciones que usamos y lo tienen en su base:
 - Organiza galería de fotos de forma automática: *Google Photos*
 - Coloreo de Imágenes (de BN a color)
 - Traducción automática de textos: *Google Translate*, *YouTube* (voz a texto y traducción automática)
 - Reconocimiento facial: *Facebook*
 - Asistentes personales...



La Irrupción del *Deep Learning (DL)*:

- Aplicaciones en general:
 - Añadir audio al video automáticamente
 - Reconocimiento de voz
 - Interpretación semántica
 - Visión computacional
 - Detección de amenazas: armas, ...
 - Robots, drones, conducción autónoma, ...
 - *Health-care*
 - Asistentes personales
 - Descripción de imágenes con texto
 - Reproducción de partituras musicales como los humanos
 - Generación de escritura manuscrita
 - Predicción del precio de la energía, predicción agrícola, ...
 - ... y un muy largo etcétera... que es realidad hoy en día.



- Observamos pues, que **de forma casi repentina**, desde hace sólo unos pocos años, surge en el ámbito de la IA, y copa muchas aplicaciones

Tecnología Disruptiva

- En realidad, **no es algo totalmente nuevo** o que fuese desconocido: es una evolución de las Redes Neuronales Artificiales (RNs)
- Las RNs ya eran empleadas desde hacia varias décadas, con sus capacidades, dónde eran útiles y dónde no funcionaban correctamente ... pero entonces...

¿Qué ha ocurrido para que, de repente, esto cambiase?

Motores del DL {

- ✓ Supercomputación
- ✓ Grandes cantidades de datos (*Big Data*)

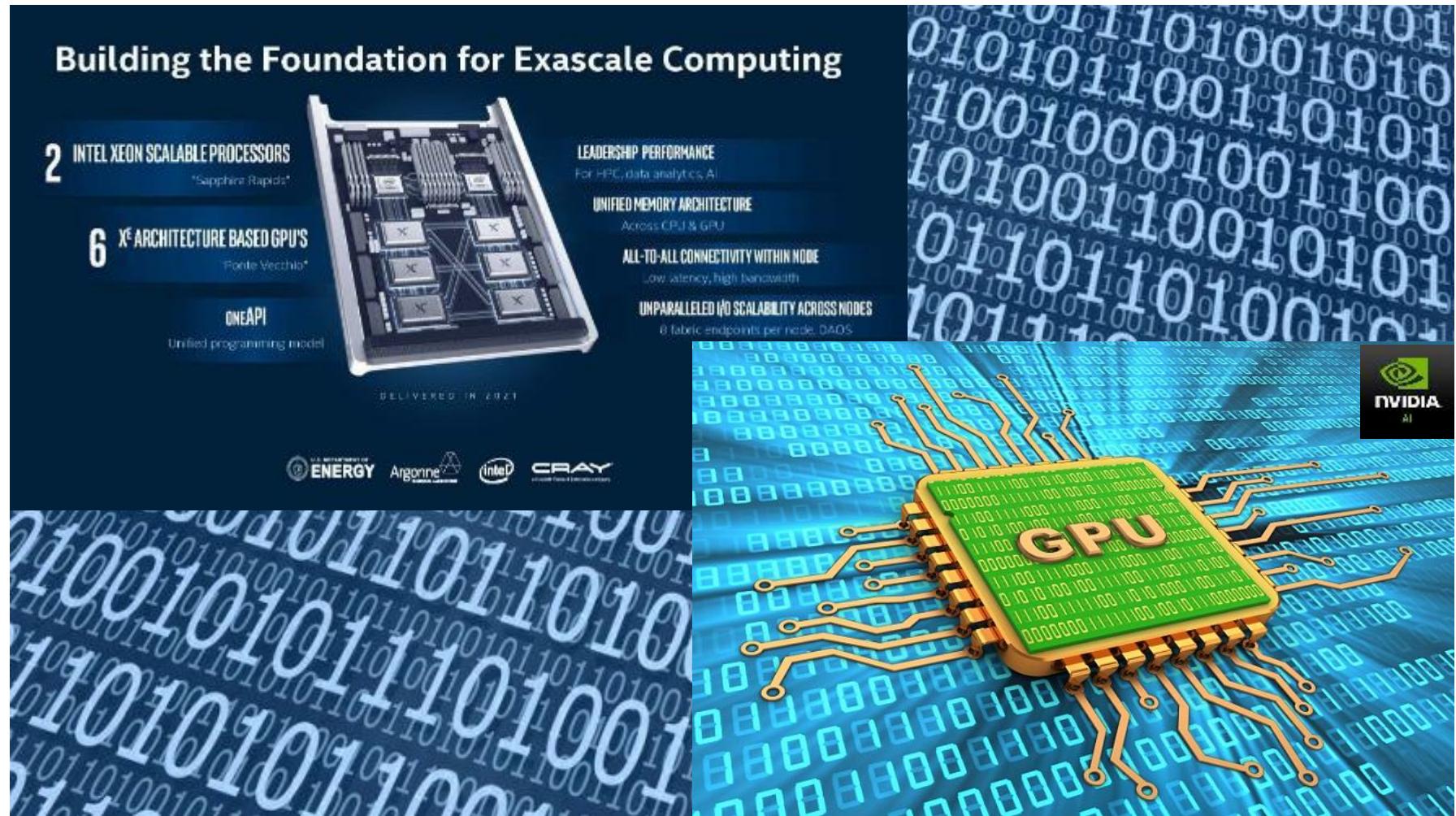


- El desarrollo del **Big Data** permite la gestión de grandes cantidades de datos (almacenamiento y procesamiento), los cuales están disponibles y son empleados para el entrenamiento con grandes *datasets* (anteriormente imposible), sobre arquitecturas computacionales varias veces más potentes



Deep Learning:
Intelligence from Big Data

Se ha multiplicado la potencia computacional disponible



- ...además, es posible disponer de estos recursos a través del:



- Existen diversas plataformas para desarrollo de Deep Learning

Caffe



DL4J

Deeplearning4j

MINERVA

mxnet



MatConvNet



theano



Tema 6: Deep Learning

Índice:

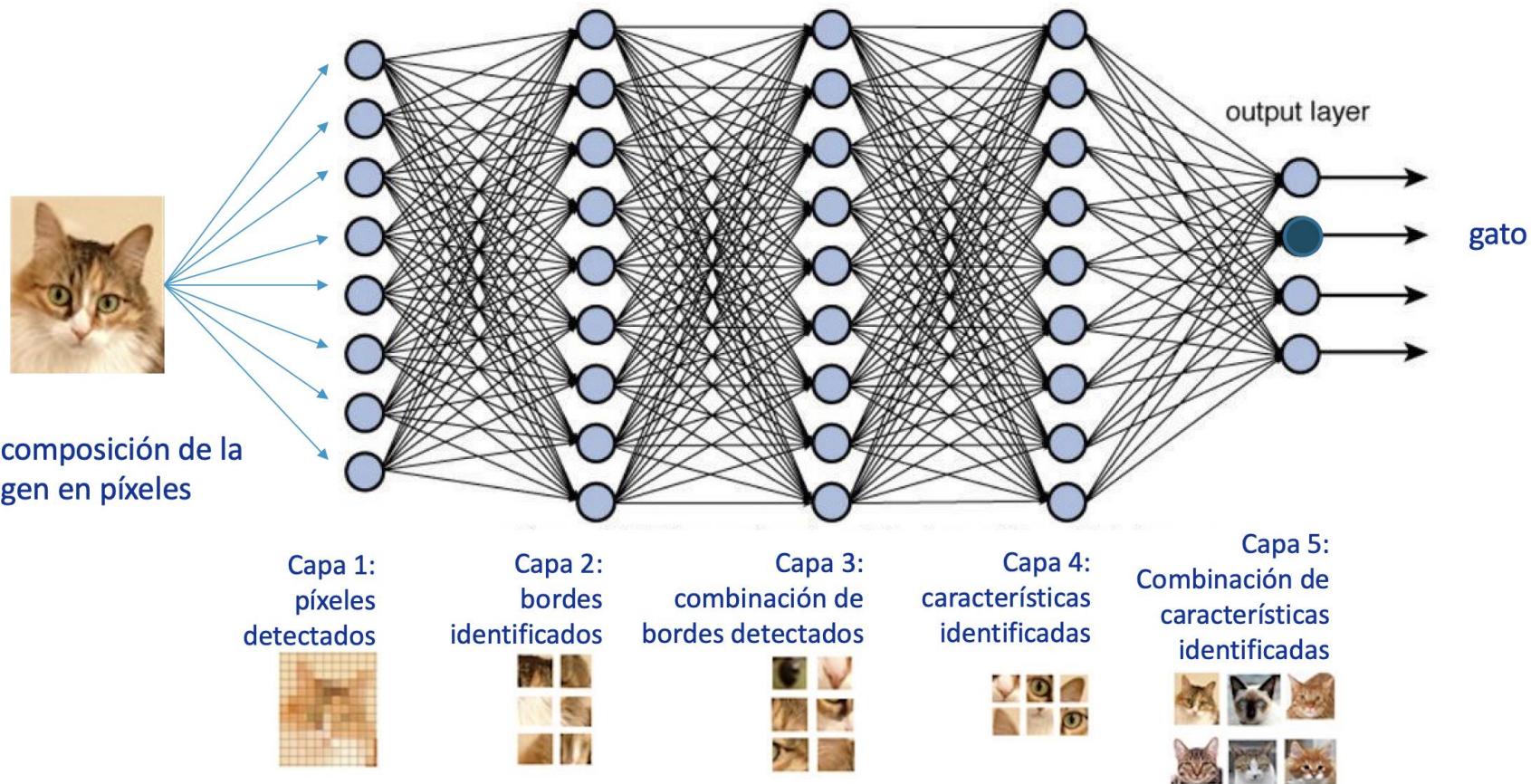
1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

2. Definiciones y Fundamentos

Es un sub-área dentro del Machine Learning basado en algoritmos para el aprendizaje de múltiples niveles de representación para modelar relaciones complejas entre datos. Las características y conceptos de mayor nivel son por tanto definidos en términos de los de inferior nivel, y esa jerarquía de características se llama "arquitectura profunda" (deep architecture). La mayoría de estos modelos están basados en aprendizaje no supervisado de representaciones. (Wikipedia)

Otras definiciones en L. Deng and D. Yu. Deep Learning methods and applications. Foundations and Trends in Signal Processing Vol. 7, Issues 3-4, 2014.

2. Definiciones y Fundamentos



- El DL es heredero del área de investigación de las RNA, de ahí que muchas veces se escuche denominarlo como “*la nueva generación de RNA*”, o “*RNA profundas*” (*Deep Neural Networks (DNNs)*), si bien, **no son los únicos modelos con arquitectura profunda.**
- El **procesamiento necesario para manejar los sistemas de información humanos** tales como la audición, el habla, o la visión, **sugieren mecanismos con arquitecturas profundas y complejas** a partir de los datos obtenidos desde los diferentes sensores.

Revisión de conceptos de *Machine Learning* utilizados (I):

- *Feature*: característica, atributo o variable, generalmente de entrada.
- *Label* (etiqueta): atributo a predecir
- *Model* (modelo): relación entre atributos (de entrada) y etiquetas
- Fase de *training* o *entrenamiento*: aprendizaje del modelo
- Fase de *test* o prueba: *inferencia* o *predicción*, uso del modelo
- El modelo puede expresarse como: (*se trata pues de aprender esta expresión*)

$$y = \omega x + b$$

donde y es la etiqueta,

x es el atributo

ω es el peso (a aprender) (*weight*)

b es el sesgo (a aprender) (*bias*)

Revisión de conceptos de *Machine Learning* utilizados (y II):

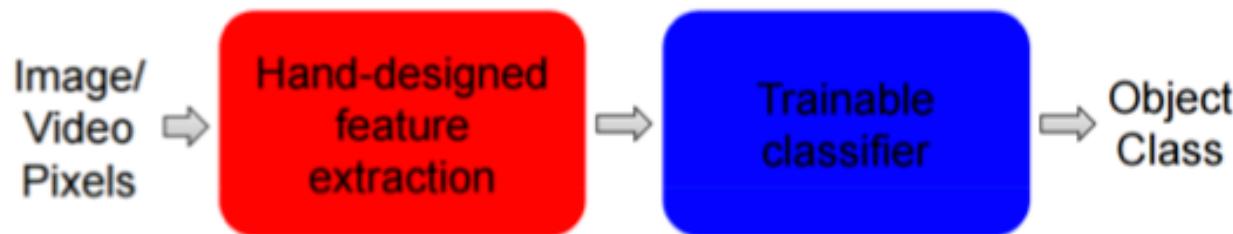
- *Con varias variables, tendríamos:*

Por ejemplo, con tres: $y = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3 + b$

- En fase de aprendizaje se aprenden por tanto los ω_i y el b
- La diferencia entre los valores de los ω_i y el b aprendidos y los ideales definen el error, pérdida o **loss** (es un valor a minimizar). Este concepto así expresado agrega los errores individuales de los ω_i y el b . En DL, esta diferencia representará la **penalización** de una mala predicción.
- Por último, el sobreajuste (*overfitting*) se produce cuando un modelo en fase de aprendizaje se adapta demasiado a los ejemplos de entrenamiento y pierde capacidad de generalizar.

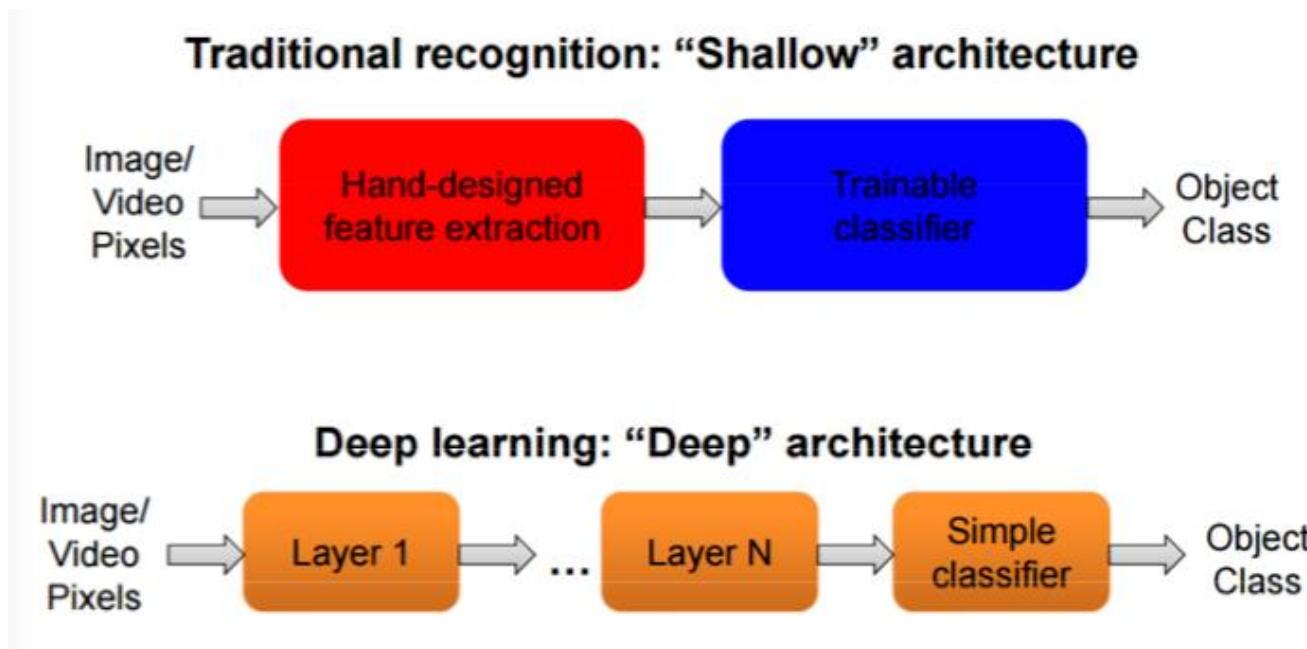
Ingeniería de Características (I de VI):

- Las aproximaciones tradicionales para reconocimiento de imágenes o videos trabajan **sin aprender las características**, de esta forma:



Ingeniería de Características (II de VI):

- En contraposición (a las “*superficiales*”), las arquitecturas “*profundas*”:

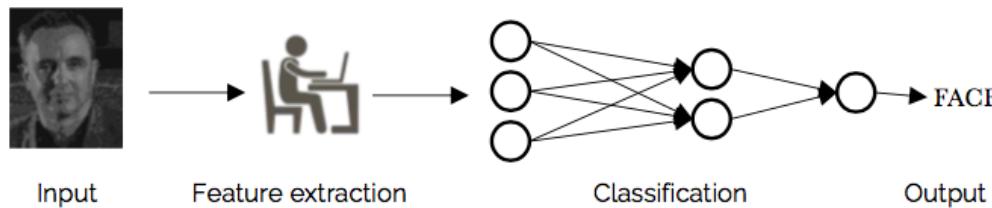


... el número de capas estará relacionado con la complejidad de las características que se deban aprender en ese problema

Ingeniería de Características (III de VI):

- **En Machine Learning (ML) tradicional:**

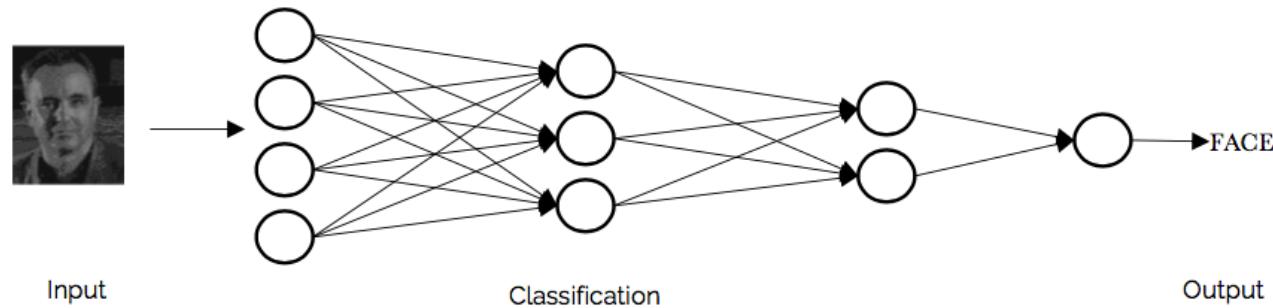
- Es necesario:
 - reducir la complejidad de los datos, y
 - hacer más visibles los patrones a los algoritmos de aprendizaje.
- Por tanto, deben crearse **extractores** de determinadas características basándonos en el conocimiento del problema concreto que tenemos de antemano (por lo cual, será necesario un experto que nos oriente sobre cuales son esas características (texturas, posición, orientación, contornos...)).
- El rendimiento de los algoritmos de ML dependerá fuertemente de:
 - La **identificación** realizada de las características
 - La **precisión** con la que sean extraídas de los datos originales



Ingeniería de Características (IV de VI):

- **En Deep Learning:**

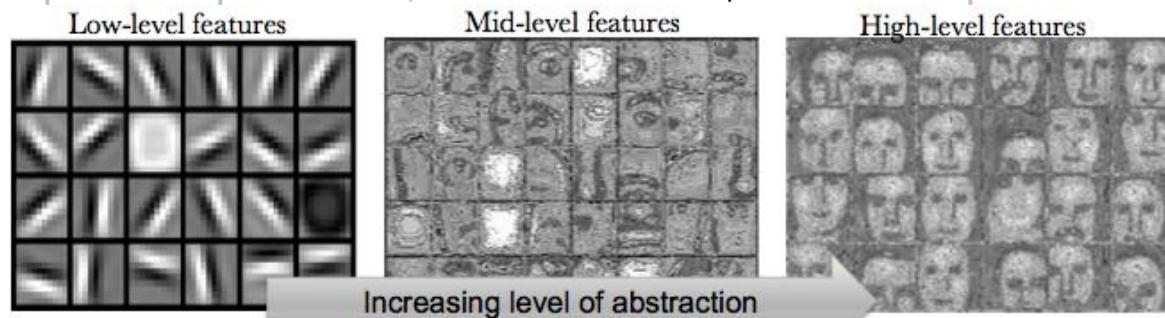
- Los algoritmos tratan de aprender por sí mismos las características de alto nivel a partir de los datos: *Descubrimiento Automático de Características*
- Por tanto:
 - Ahorran la fase de identificar e implementar esa extracción de características para cada problema
 - Las Redes Neuronales Convolucionales (RNC), que son un modelo de RNA profundas, tratarán de aprender las **características de bajo nivel**, tales como ejes o líneas en los niveles iniciales y posteriormente en capas más profundas, **las de alto nivel**, tales como partes de la cara.



Ingeniería de Características (V de VI):

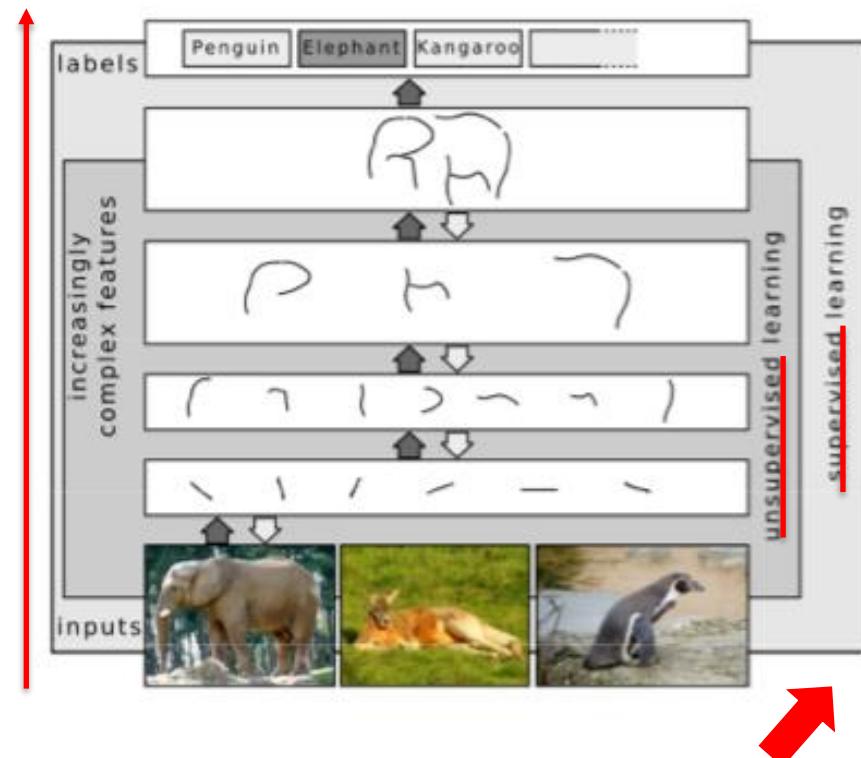
- **En Deep Learning:**

- Los algoritmos tratan de aprender por sí mismos las características de alto nivel a partir de los datos. *Descubrimiento Automático de Características*
- Por tanto:
 - Ahorran la fase de identificar e implementar la extracción de características para cada problema
 - Las Redes Neuronales Convolucionales (RNC), que son un modelo de RNA profundas, tratarán de aprender las **características de bajo nivel**, tales como ejes o líneas en los niveles iniciales y posteriormente en capas más profundas, **las de alto nivel**, tales como partes de la cara.

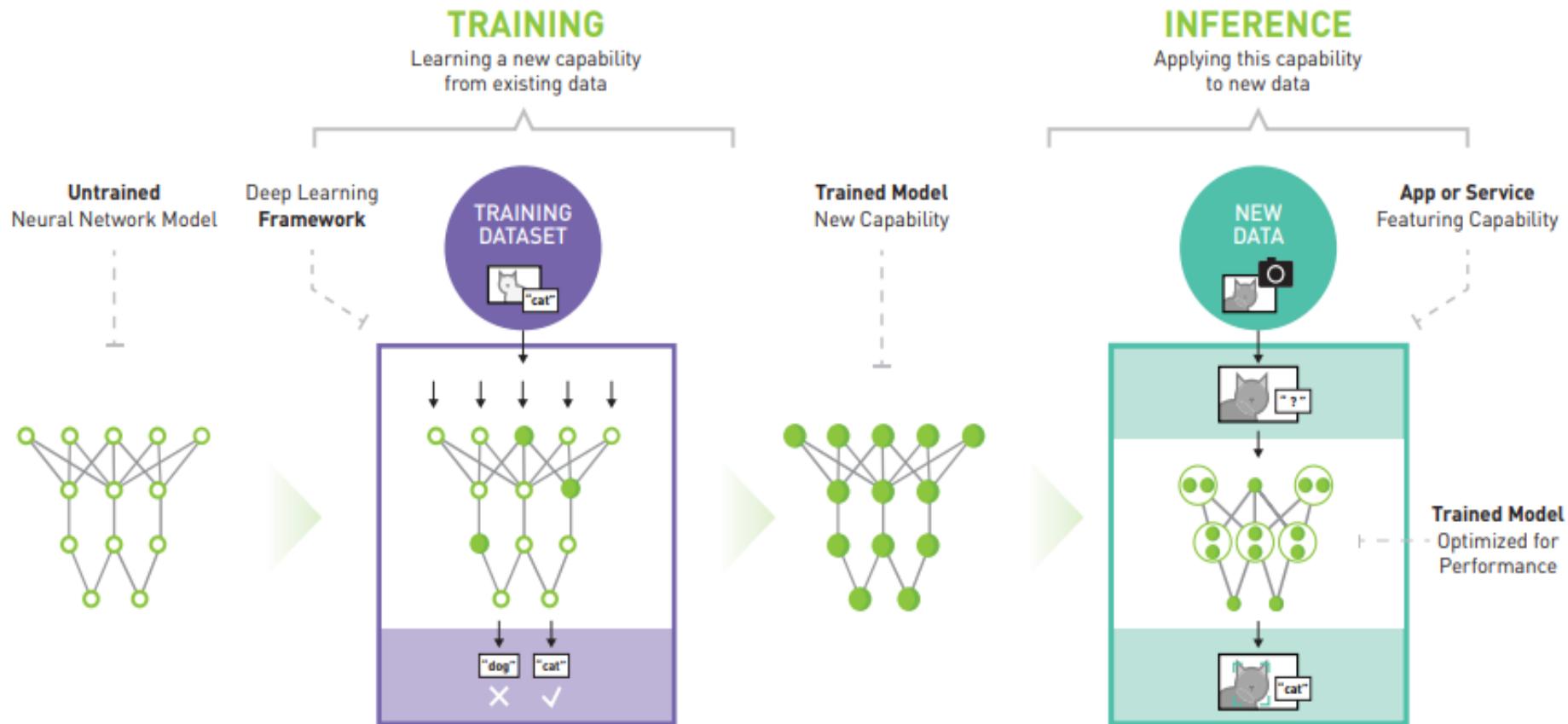


Ingeniería de Características (y VI):

- Por este motivo, el DL también es denominado *Aprendizaje Jerárquico de Características*:
 - Alcanza complejidad progresando de forma natural desde estructuras de bajo nivel a las de alto nivel
 - Más fácil de monitorizar el qué está siendo aprendido y orientar a mejores subespacios
 - Normalmente funciona mejor cuando el espacio de entrada está localmente estructurado espacial o temporalmente: imágenes, lenguaje, etc. vs. características de entrada completamente arbitrarias

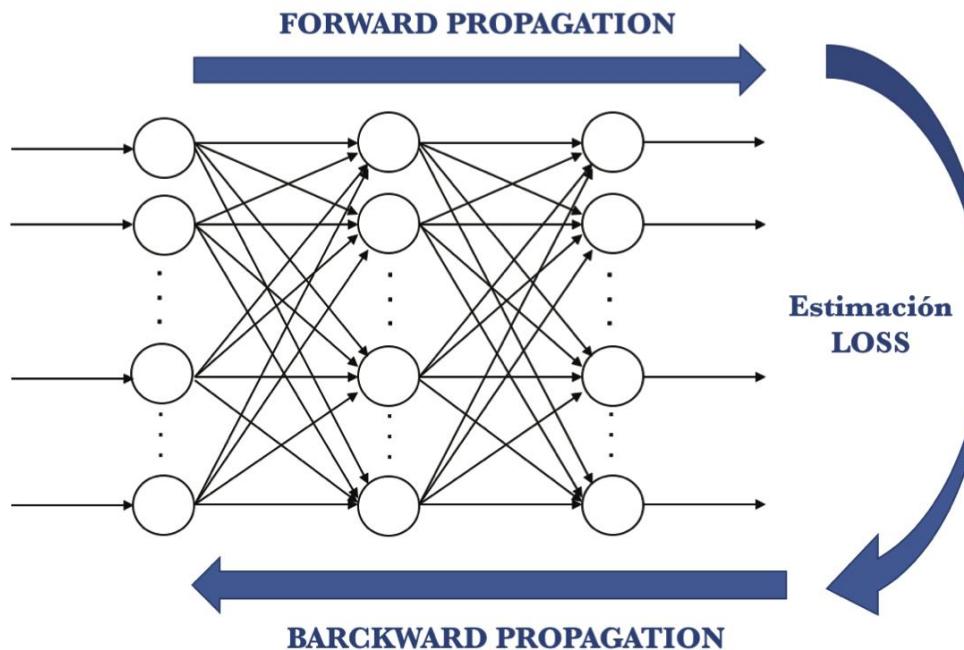


- **Deep Learning Workflow: Entrenamiento / Inferencia**



- Como se introdujo en el tema anterior, las RNs, en fase de aprendizaje, utilizan un mecanismo de *forward / backward propagation* para minimizar el error de sus salidas
- Esto se lleva a cabo ajustando los pesos entre la capa j y k , w_{jk} , en dos fases, para cada ejemplo:
 - *Forward*
 - *Backpropagation* de los errores

- **Forwardpropagation:** cuando se le presenta a la red un nuevo ejemplo, y esta calcula su salida
- **Backpropagation:** cuando la red reajusta los pesos usando el error de predicción (estimación *loss*) con cada ejemplo que anteriormente se ha propagado hacia adelante (forward)



- The Evolution of a Neural Learning System: A Novel Architecture Combining the Strengths of NTs, CNNs, and ELMs. N. Martínez, C. Micheloni. IEEE SMC Magazine 2015.

- Actualmente, se pueden desarrollar aplicaciones de Deep Learning empleando recursos *Open Source*
 - Diversos *frameworks* que requieren sólo conocimientos Python, para la creación y entrenamiento de modelos por ejemplo:
 - *TensorFlow* (by Google Brain: dominante)
 - *Keras*: con una API de alto nivel sobre RNA, idónea por ello para iniciarse. Se trabaja en Python, y se ejecuta sobre el motor de ejecución de TensorFlow entre otros (se puede cambiar por otros dentro de los soportados, sin tocar el código Keras).
 - *PyTorch*: Entorno de ML en Python (implementados en C, usando OpenMP y CUDA) para arquitecturas altamente paralelas, desarrollado por Facebook.

- Aunque hay muchos otros...
 - *Caffe: Universidad de Berkeley*
 - *Caffe2: Facebook*
 - *CNTK: Microsoft*
 - *DIGITS, Deeplearning4j, Chainer: Nvidia*
 - *Theano: Montreal Institute of Learning Algorithms*
 - ...

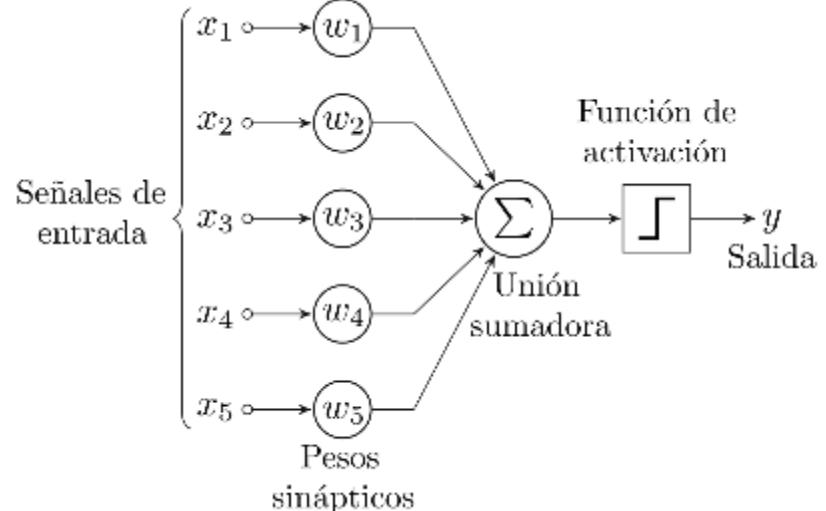
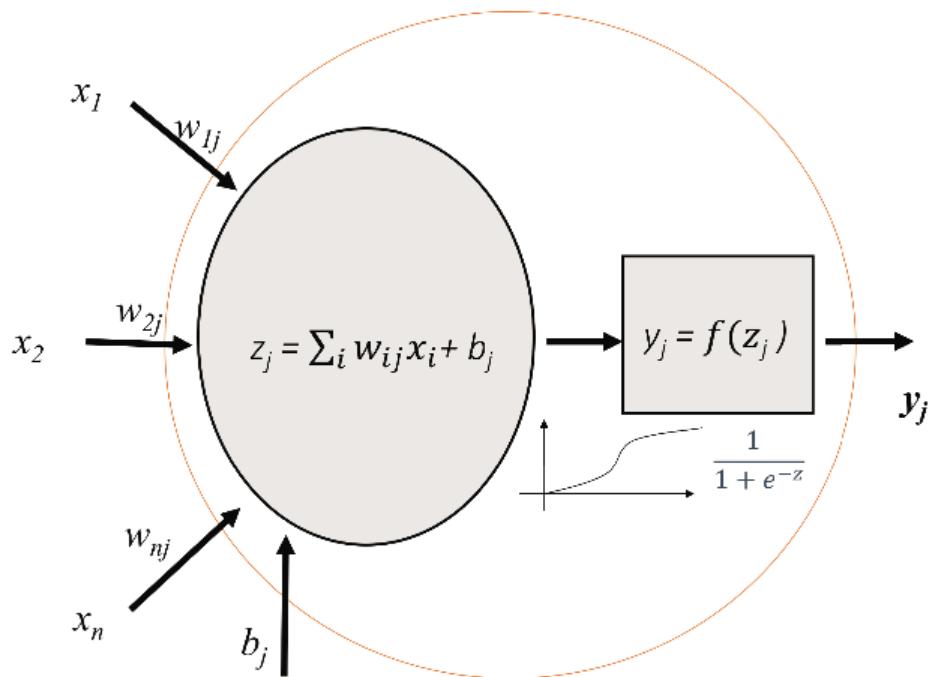


Tema 6: Deep Learning

Índice:

1. Introducción
2. Definiciones y fundamentos
- 3. RNs Densamente conectadas**
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

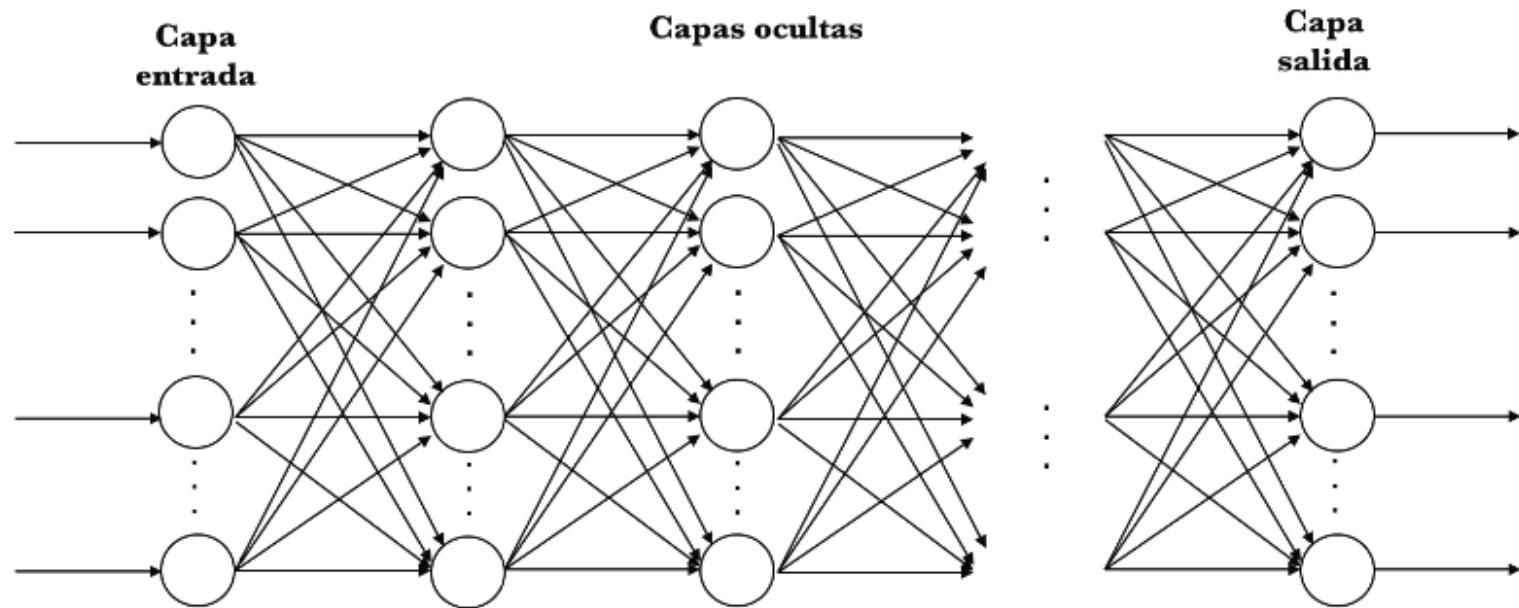
Perceptrón (RN más simple, con una sola neurona):



Deep Learning, Introducción práctica con Keras, Jordi Torres.
Lulu Press, Inc.

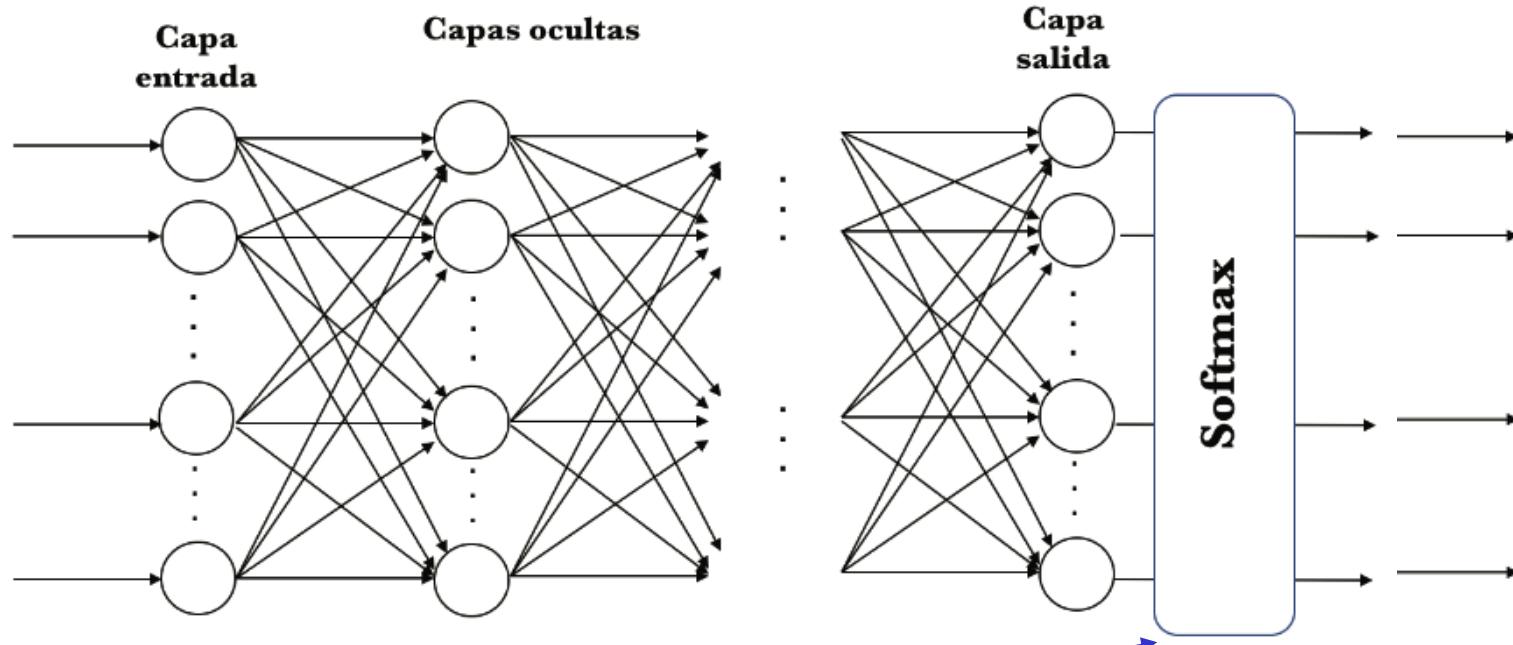
Wikipedia: Alejandro Cartas - Trabajo propio, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=41534843>

Perceptrón Multicapa:



- Cada neurona es como la del perceptrón, (en las capas ocultas, las entradas conectan con las salidas de la capa previa).
- Las funciones de activación son diferentes entre unas y otras.

Perceptrón Multicapa:



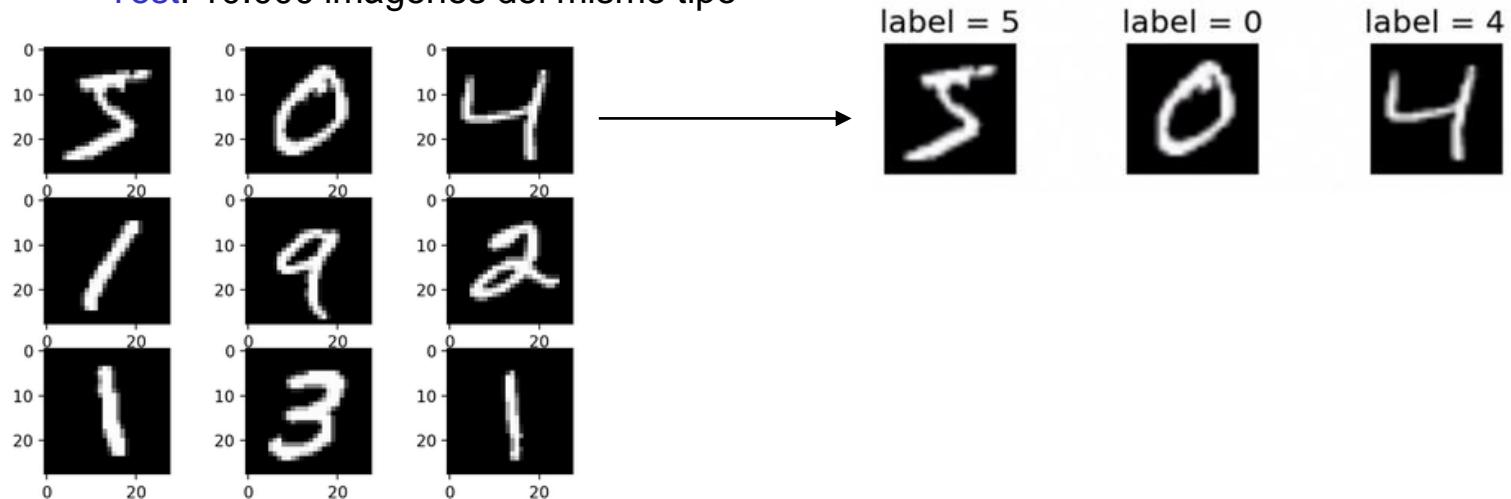
Probabilidad de pertenencia a la clase i :

$$\text{Softmax}_i = \frac{e^{\text{evidencia}_i}}{\sum_j e^{\text{evidencia}_j}}$$

La función de activación de tipo **softmax** en la capa de salida es apropiada para labores de clasificación, ya que tiene en cuenta que debe clasificar por una sola clase y considerar todas las salidas de las neuronas de la capa de salida para quedarse sólo con una.

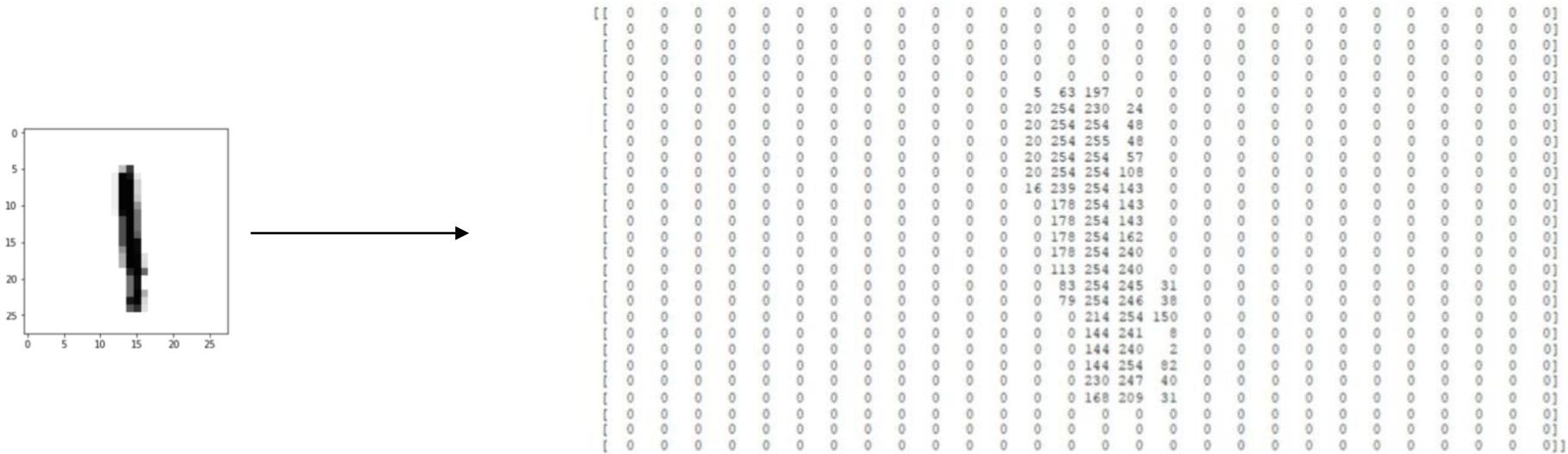
Poniendo en marcha un ejemplo muy sencillo (I de XXIV):

- Clasificador de dígitos (0 a 9) escritos a mano, con un perceptron multicapa
 - Emplearemos la base de datos clásica MNIST:
 - **Entrenamiento:** 60.000 imágenes de 255 tonos de gris, de imágenes de 28x28 pixeles, de los 10 dígitos (del 0 al 9)
 - **Test:** 10.000 imágenes del mismo tipo



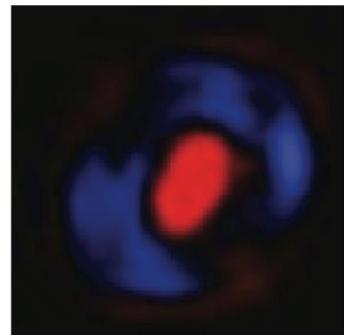
Poniendo en marcha un ejemplo muy sencillo (II de XXIV):

- Características (*Features*): Matriz de 28x28 pixeles con valores de 0 a 255, pues nuestras imágenes digitales son matrices de valores (pixeles) con información sobre su brillo (imágenes en grises, con 0: negro completo, y 255: blanco completo) o tonalidad por canal RGB.
 - Etiquetas (*Labels*) o clases de salida: valores entre 0 y 9



Poniendo en marcha un ejemplo muy sencillo (III de XXIV):

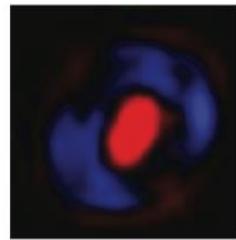
- Usaremos una función de activación de tipo **softmax**, ya que parece apropiada al problema de clasificación en cuestión
- **Softmax** trabaja primero **recogiendo evidencias** de que una imagen pertenece a una clase, y luego, **convirtiendo las evidencias en probabilidades** para cada una de las clases posibles. Veamos:
- Evidencia de un 0:



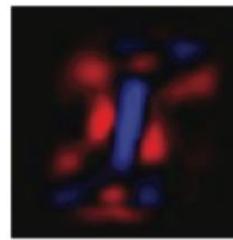
La **zona roja** es la de los pesos negativos aprendidos por la red, es decir, la zona donde si hay trazo, no es un 0; mientras que la **zona azul** representa que si hay trazo ahí (pesos positivos), es una evidencia de que es un 0. En **negro** donde no hay evidencia ni positiva ni negativa: neutrales.

Poniendo en marcha un ejemplo muy sencillo (IV de XXIV):

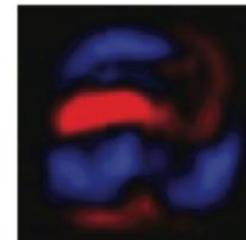
- Evidencias de los diferentes dígitos:



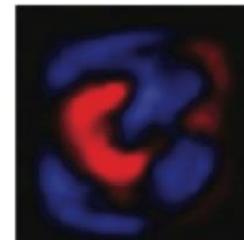
0



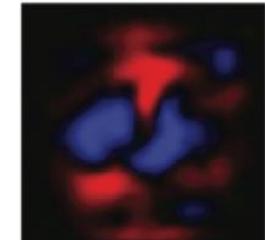
1



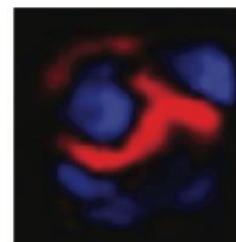
2



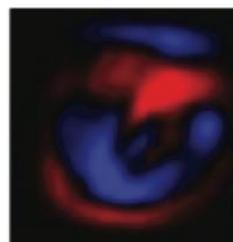
3



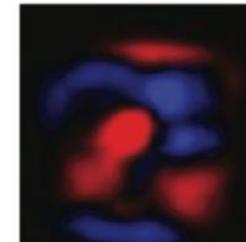
4



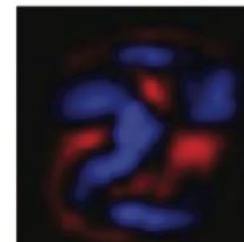
5



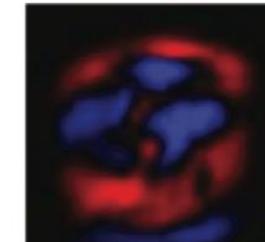
6



7



8



9

Poniendo en marcha un ejemplo muy sencillo (V de XXIV):

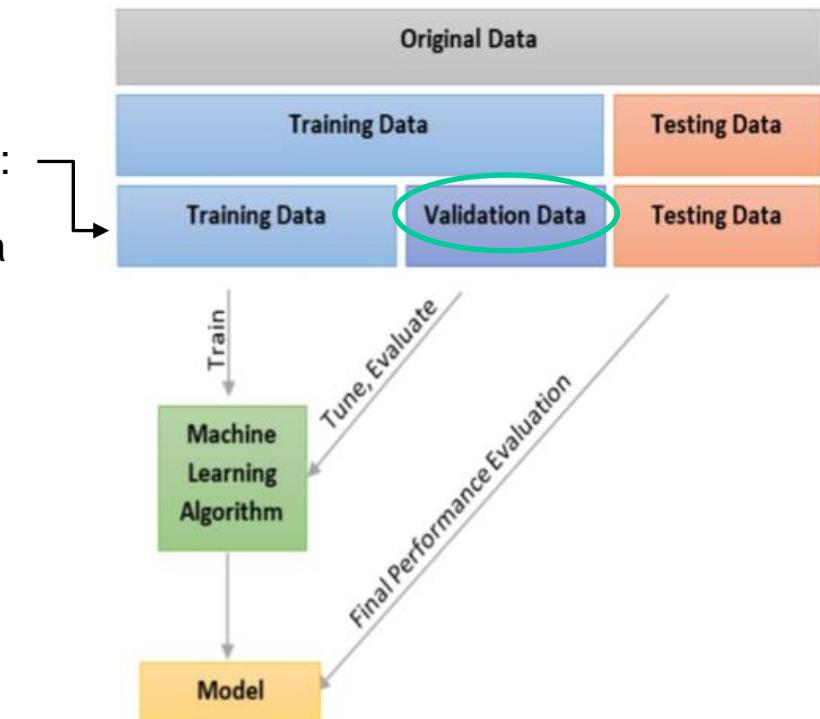
- En segundo lugar, **softmax** convierte las evidencias en probabilidades, usando una exponencial (*para que el efecto de la que más evidencia tiene sea multiplicador*) normalizada para que sumen 1, es decir, se forma esta distribución de probabilidad de pertenencia a la clase i :

$$\text{Softmax}_i = \frac{e^{\text{evidencia}_i}}{\sum_j e^{\text{evidencia}_j}} \quad i = 0..9, \text{ representa a las clases}$$

- Esta función tiene la ventaja de conseguir una sola salida cercana a probabilidad 1 en el vector de salida, mientras que las demás estarán todas cercanas entre sí y próximas a 0, por lo que escoger la etiqueta será sencillo.

Poniendo en marcha un ejemplo muy sencillo (VI de XXIV):

- Los datos disponibles los organizaremos basándonos en la forma común de **entrenamiento y prueba**; idealmente en este ámbito, se trabaja así, con **una separación extra**:
 - **Entrenamiento (*training*)**: Orientado a que la red **aprenda sus pesos**.
 - **Validación (*validation*)**: Para ajustar los **hiperparámetros**, en aquellos casos en los que haya problemas de aprendizaje en la fase de entrenamiento (por ejemplo, sobreajuste)
 - **Prueba (*test*)**: prueba final del rendimiento



JORDI TORRES

Poniendo en marcha un ejemplo muy sencillo (VII de XXIV):

- Pasos a seguir en la creación de un modelo usando **Keras**:

1. Cargar (*load*) los datos
2. Definir el modelo (*model*)
3. Compilar el modelo y configurar el aprendizaje
4. Entrenar o ajustar (*fit*) el modelo
5. Evaluar el modelo
6. Usar el modelo

Poniendo en marcha un ejemplo muy sencillo (VIII de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

1. Cargar (*load*) los datos

```
import keras

from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

- Código que carga las imágenes en 4 arrays Numpy con sus etiquetas (*labels*, de 0 a 9):
 - **x_train e y_train**: conjunto de datos de entrenamiento (en x_train están las imágenes, mientras que en y_train están las etiquetas)
 - **x_test e y_test**: conjunto de datos de test (imágenes y etiquetas igualmente)

Poniendo en marcha un ejemplo muy sencillo (IX de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:
 1. Cargar (load) los datos
 - Específicamente, ha cargado la estructura básica de datos **tensor**, que consiste en un **array multidimensional**, parecido a los de Numpy, donde sus 3 atributos principales son:
 - **Numero de ejes** (Rank o ndim del **tensor**):
 - » Si Rank o **ndim** = 1, significa 0-dimensional (i.e. un valor solamente),
 - » Si Rank o **ndim** = 2, significa 1-dimensional (i.e. una lista de valores),
 - » Si Rank o **ndim** = 3, significa 2-dimensional (i.e. una imagen plana),
 - » Si Rank o **ndim** = 4, significa 3-dimensional ... etc., ...
 - **Forma** (shape): tupla de enteros que dice las **dimensiones** que tiene el **tensor** en cada eje
 - **Tipo de datos** (data type o dtype): **tipo de datos del tensor**: uint8, ... float32, float64... char...

Poniendo en marcha un ejemplo muy sencillo (X de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

1. Cargar (load) los datos

Tras la carga, podemos ver los atributos del tensor poniendo:

```
print(x_train.ndim)
```

Mostrará un 3, es decir, tenemos un array de imágenes planas (2D)

```
print(x_train.shape)
```

mostrará (60000, 28, 28) debido a que son 60 mil imágenes de 28x28 pixeles

```
print(x_train.dtype)
```

lo cual mostrará uint8 es decir, utiliza 256 niveles de gris para cada pixel, por tanto, tenemos en este tensor x_train, 60 arrays 2D con valores de tipo uint8

Poniendo en marcha un ejemplo muy sencillo (X de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

1. Cargar (load) los datos

Si queremos visualizar una imagen, pondríamos, por ejemplo:

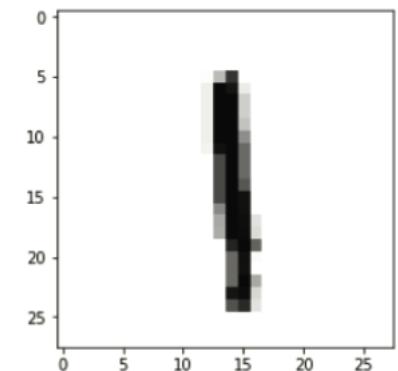
```
import matplotlib.pyplot as plt  
plt.imshow(x_train[8], cmap=plt.cm.binary)
```

Lo cual previsualiza la imagen 8 del dataset cargado:

... y para ver la etiqueta asociada, se pondría:

```
print(y_train[8])
```

lo cual mostraría, lógicamente, un: 1



Poniendo en marcha un ejemplo muy sencillo (XII de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

1. Cargar (load) los datos

Dependiendo del tipo de red, puede ser necesario **normalizar** a determinados rangos diferentes a los de entrada, o formatear previamente los datos (imágenes en este caso) de entrada. Por ejemplo, así (mediante `astype`) se escalarían los datos a tipo `float32`, y posteriormente trasladamos los valores al intervalo [0,1]:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

```
x_train = x_train / 255
```

Poniendo en marcha un ejemplo muy sencillo (XIII de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

1. Cargar (load) los datos

En muchas ocasiones, se debe hacer también una transformación (mediante `reshape`) del tensor (imagen) de dos dimensiones a un vector de una dimensión: (convertir la matriz de 28x28 números en una matriz de 784 números, concatenando cada fila una tras otra, de modo que la primera dimensión indexa la imagen, y la segunda el pixel de la imagen (valor ya de 0 a 1) (en capas convolucionales no suele ser necesario))

```
x_train = x_train.reshape(60000, 784)  
x_test = x_test.reshape(10000, 784)
```

Para comprobar el resultados pondríamos de nuevo:

```
print(x_train.shape)  
print(x_test.shape)
```

y ahora obtendríamos:

(60000, 784)

(10000, 784)

Poniendo en marcha un ejemplo muy sencillo (XIV de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

1. Cargar (load) los datos

```
from keras.utils import to_categorical
```

Nos ocupamos ahora de las etiquetas (vector y): Si para ellas necesitamos usar *one-hot encoding*, con objeto de emplear la información de las etiquetas en forma de vector binario con tantos ceros como etiquetas distintas puedan darse, y con un 1 en el índice que corresponde con la etiqueta que tiene, importamos de `keras.utils` la función `to_categorical`

Para ver el estado actual (`shape`) de las etiquetas, pondríamos:

```
print(y_test[0])  
7  
print(y_train[0])  
5  
print(y_train.shape)  
(60000,)
```

valores que queremos pasar a “categóricos”

... es el conjunto vacío cuando es un solo valor

Poniendo en marcha un ejemplo muy sencillo (XV de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

Para trasladarlo a *one-hot encoding*, usando `to_categorical`

```
y_train = to_categorical(y_train, num_classes=10)  
y_test = to_categorical(y_test, num_classes=10)  
  
print(y_test[0])
```

[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.] ← ... convertidos

```
print(y_train[0])
```

[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]

Para chequear el shape ahora del vector de etiquetas:

```
print(y_train.shape)
```

(60000, 10)

```
print(y_test.shape)
```

(10000, 10)

... ahora *shape* es 10

Poniendo en marcha un ejemplo muy sencillo (XVI de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

2. Definir el modelo (*model*)

- Una RN básica en Keras se representa por la clase `Sequential`, así llamada, por ser una RN secuencial (entrenamiento secuencial). Se crea así:

```
from keras.models import Sequential  
model = Sequential()
```

- Pero este modelo no está aún completamente definido, por lo que le agregaremos las distintas capas que necesitemos empleando el método `add` con los detalles de las mismas

Poniendo en marcha un ejemplo muy sencillo (XVII de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

2. Definir el modelo (model)

- Así por ejemplo se completaría un modelo básico:

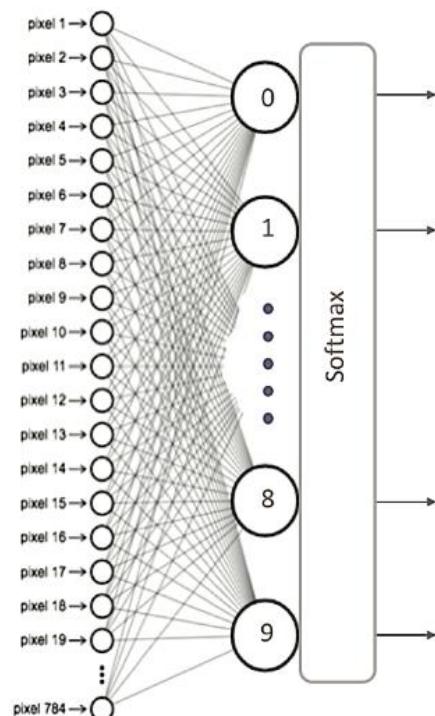
```
from keras.models import Sequential
from keras.layers.core import Dense, Activation

model = Sequential()
model.add(Dense(10, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

Estamos definiendo una red densamente conectada

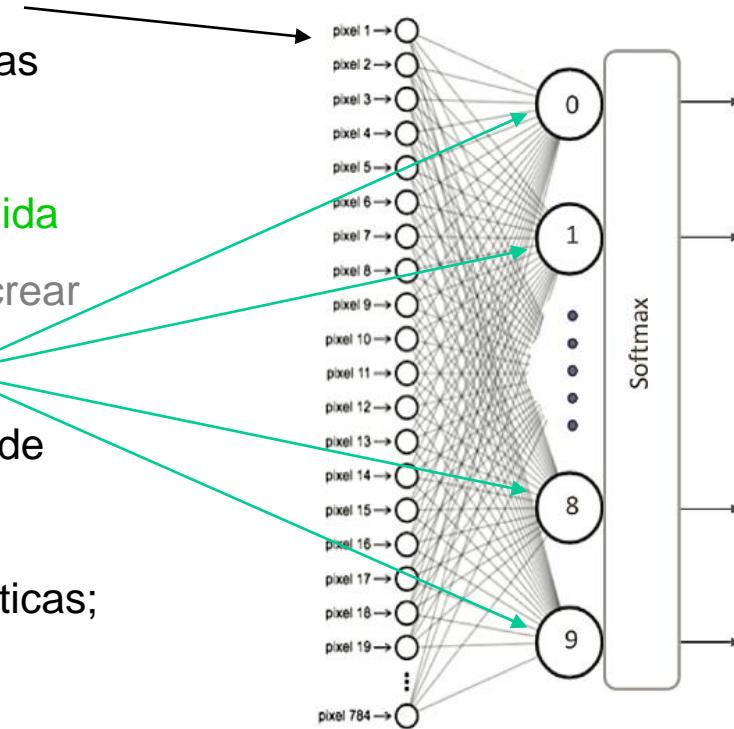
Añadimos dos capas de neuronas:

1. Capa de 10 neuronas densamente conectadas con las 784 neuronas (*features*, o píxeles de entrada), con activación sigmoide
2. Capa de salida de 10 neuronas softmax. Tensorflow deduce en esta nueva capa el `input_shape` (porque conecta con la anterior).



NOTA:

- La imagen entra en la RN a través de una capa de entrada en la que hay una neurona para cada pixel, esto es, como tenemos $28 \times 28 = 784$ píxeles, hay una **capa de entrada de 784 neuronas**.
- Cuando estamos añadiendo capas mediante (**add**) al modelo, son realmente **capas ocultas y de salida** (no la de entrada que estaba al crear el modelo): primera capa oculta, segunda (y última en este caso) de salida, son las que estamos especificando con sus características; cuando añadimos una capa, expresamos cómo se conecta con su **capa anterior**, etc.



Poniendo en marcha un ejemplo muy sencillo (XVIII de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:
 2. Definir el modelo (model)
 - Existe una forma de comprobar la arquitectura que es mediante el método `summary()`:

```
model.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 10)	7850
dense_2 (Dense)	(None, 10)	110
Total params:	7,960	
Trainable params:	7,960	
Non-trainable params:	0	

La primera capa oculta tiene 10 neuronas (i de 0 a 9), y cada una tiene 784 entradas y por tanto, 784 pesos ω_{ij} , $10 \times 784 = 7840$ parámetros, más 10 parámetros para los sesgos b_j .

La segunda capa añadida tiene 10 neuronas softmax (porque es nuestra capa de salida), conectando sus 10 neuronas con las 10 neuronas de la anterior: $10 \times 10 = 100$, más sus 10 sesgos = 110 parámetros

Poniendo en marcha un ejemplo muy sencillo (XIX de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

3. Compilar el modelo y configurar el aprendizaje

- Compilar el modelo es definir cómo será su proceso de aprendizaje.

```
model.compile(loss="categorical_crossentropy",
              optimizer="sgd",
              metrics = ['accuracy'])
```

- Los argumentos de `compile` son los que nos permiten hacerlo:
 - `loss`: función de evaluación del grado de error entre las salidas de la RN y las salidas deseadas según los datos de entrenamiento
 - `optimizer`: algoritmo de cálculo de los pesos con la función `loss` y las entradas dadas (`sgd: stochastic gradient descent`)
 - `metrics`: métrica para monitorizar el aprendizaje de la RN (como la `accuracy`, que mide la fracción de imágenes correctamente clasificadas)
- Otras opciones y argumentos se pueden consultar en manuales avanzados de Keras.

Poniendo en marcha un ejemplo muy sencillo (XX de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

4. Entrenar o ajustar (fit) el modelo

```
model.fit(x_train, y_train, batch_size=100, epochs=5)
```

- Estamos indicando que el modelo debe entrenarse con los datos de los arrays Numpy de entrenamiento `x_train` e `y_train`
- Mediante `batch_size` se está indicando el número de datos (ejemplos) empleados para cada actualización de los parámetros del modelo: observará el loss con esos 100 datos, tras ello actualiza el modelo (recalcula pesos), para posteriormente, coger los siguientes 100 datos, etc.
- El argumento `epochs` (“épocas”) indica el número de veces que se usarán todos los datos (los 60 mil del conjunto en este ejemplo) en el proceso de aprendizaje

Poniendo en marcha un ejemplo muy sencillo (XXI de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

4. Entrenar o ajustar (fit) el modelo

```
model.fit(x_train, y_train, batch_size=100, epochs=5)
```

- Este proceso, como es sabido, es el que requiere tiempo y esfuerzo computacional importantes. Su evolución, tanto en tiempo como en error y precisión se visualiza en unos mensajes similares a estos:

```
Epoch 1/5
60000/60000 [=====] - 1s 15us/step - loss: 2.1822 - acc: 0.2916
Epoch 2/5
60000/60000 [=====] - 1s 12us/step - loss: 1.9180 - acc: 0.5283
Epoch 3/5
60000/60000 [=====] - 1s 13us/step - loss: 1.6978 - acc: 0.5937
Epoch 4/5
60000/60000 [=====] - 1s 14us/step - loss: 1.5102 - acc: 0.6537
Epoch 5/5
60000/60000 [=====] - 1s 13us/step - loss: 1.3526 - acc: 0.7034
10000/10000 [=====] - 0s 22us/step
```

Poniendo en marcha un ejemplo muy sencillo (XXII de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

5. Evaluar el modelo

- Para evaluar el modelo se ejecuta el método `evaluate`:

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

- Se le han introducido para ello los datos `x_test` e `y_test` que la RN no ha empleado en fase de entrenamiento.
- Se puede visualizar el resultado de la precisión obtenida:

```
print('Test accuracy:', test_acc)
```

Obteniéndose una salida del tipo:

Test accuracy: 0.9018

que significaría que obtiene una precisión levemente superior al 90%
(también es posible obtener la *matriz de confusión*)

Poniendo en marcha un ejemplo muy sencillo (XXIII de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

5. Evaluar el modelo

- Matriz de confusión:

		Predicción	
		Positivos	Negativos
Observación	Positivos	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

$$\text{Accuracy} = (\text{VP} + \text{VN}) / (\text{VP} + \text{FP} + \text{VN} + \text{TN})$$

- Sabemos que el *accuracy* es una medida a veces engañosa, por lo que también disponemos de la *sensibilidad* (o *recall*)

$$\text{Sensitivity} = \text{VP} / \text{P} = \text{VP} / (\text{VP} + \text{FN})$$

```
loss, accuracy, f1_score, precision, recall = model.evaluate(x_test, y_test)
```

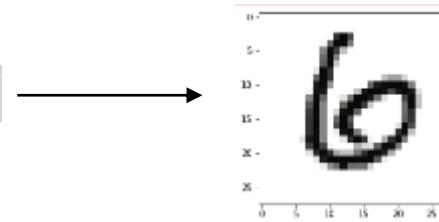
Poniendo en marcha un ejemplo muy sencillo (XXIV de XXIV):

- Pasos a seguir en la creación de un modelo usando Keras:

6. Usar el modelo

- El modelo estaría listo para utilizarlo (inferir) después de todos los pasos anteriores. Keras dispone del método `predict` para ello. Hemos elegido el elemento 11 del dataset `x_test`, el cual podemos previsualizar así:

```
plt.imshow(x_test[11], cmap=plt.cm.binary)
```



```
predictions = model.predict(x_test)
```

calcula el vector resultado de la predicción para todo el conjunto de datos `x_test`.

Ahora podemos ver el valor más alto (mayor probabilidad) para el elemento 11:

```
np.argmax(predictions[11])
```

→ 6

También podemos ver el array de probabilidades:

```
print(predictions[11])
```

```
[0.06 0.01 0.17 0.01 0.05 0.04 0.54 0. 0.11 0.02]
```

Parámetros e Hiperparámetros de una RN:

- Los **hiperparámetros** de una RN son valores de configuración externos al modelo, es decir, que no dependen de los datos (los **parámetros** en cambio serían aquellos valores que se pueden estimar desde los datos): el número de capas, el tipo de cada capa, el número de neuronas por capa, el tipo de función de activación en cada capa, el número de épocas, ... son *hiperparámetros*.
- El ajuste de los **hiperparámetros** se hace fundamentalmente en base a la **experiencia** (o bien, a la **consulta** de documentos en los que se detallen *hiperparámetros* apropiados para **problemas similares**).
- El cambio de algunos, no supone **cambios en el tiempo** de entrenamiento (o suponen poco), por ejemplo, cambiar la función de activación en una capa... y podría mejorar o empeorar la precisión de la RN; sin embargo, otros cambios suelen tener impacto grande directo en los tiempos requeridos (por ejemplo, un cambio en el número de neuronas de una capa, o la mayoría de los otros cambios...).

- Tipos de *hiperparámetros*:
 - Los que tienen relación con la **topología** de la RN: por ejemplo, el número de capas, el tipo de cada capa, el número de neuronas por capa, el tipo de función de activación en cada capa
 - Los que tienen relación con el **algoritmo de aprendizaje**, por ejemplo, el número de épocas, el *learning rate*, el tamaño del *batch*, etc.
- Influencia de algunos *hiperparámetros*:
 - **Número de épocas (epochs)**: número de veces que pasan los datos por la RN. El aumento del número de épocas puede ser un recurso para mejorar, hasta que aparezca sobreaprendizaje, sobreadaptación, o *overfitting*.
 - **Batch size**: es el tamaño de los lotes que usa el método *fit* en cada iteración del entrenamiento para actualizar el gradiente. El tamaño que podamos usar está en relación con el tamaño de la memoria del servidor.

- **Learning rate**: Factor que multiplica la magnitud del gradiente, en los algoritmos de gradiente descendente. Valores más altos aceleran la búsqueda pero propician mínimos locales; valores mínimos enlentecen el proceso. Si no funciona nuestra red, este valor se podría bajar.
- **Learning rate decay**: Cuando se usa un *learning rate* variable, este valor define cómo se va reduciendo a medida que el modelo mejora.
- **Momentum**: valor entre 0 y 1 relacionado con el reinicio para escapar de óptimos locales (gradientes 0)
- **Inicialización de los pesos**: suele ser la mejor política que sean aleatorios.

Tema 6: Deep Learning

Índice:

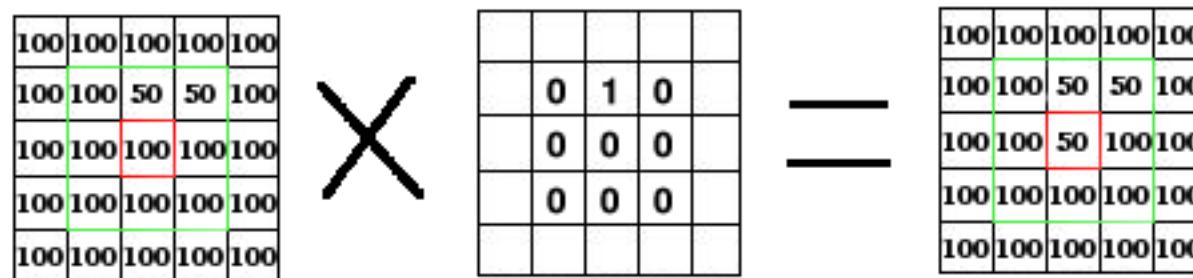
1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
- 4. RNs Convolucionales**
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

4. Redes Neuronales Convolucionales:

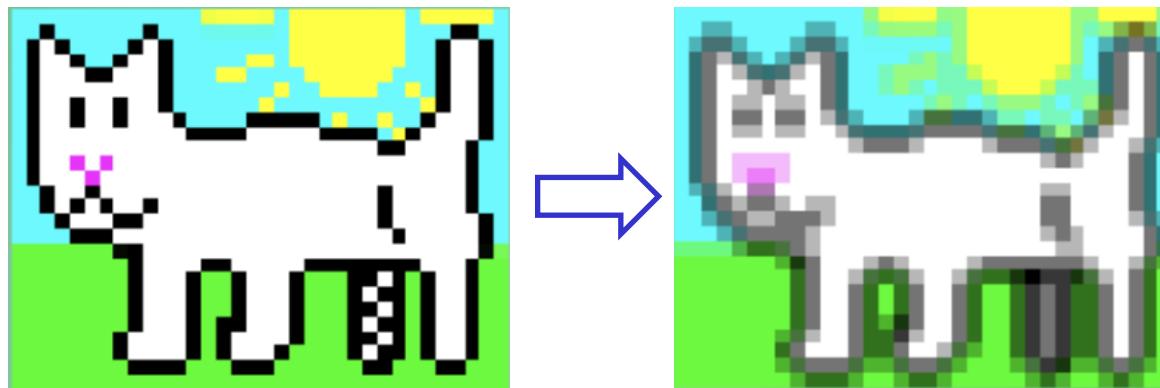
- Redes Neuronales Convolucionales o **Convolutional Neural Networks** (CNNs) es un modelo de *Deep Learning*, basado en la organización de las neuronas del **cortex** (parte de la mente que procesa la visión), donde la particularidad es que no todas las neuronas se conectan con todas las entradas del campo visual, sino que el citado campo visual está parcelado en zonas con grupos de neuronas (**campos receptivos**) que se superponen unos con otros.
- Las CNNs procesan sus entradas en bloques superpuestos de ellas, empleando **operadores matemáticos de convolución**, los cuales aproximan de modo similar a como funciona un **campo perceptivo**.
- Esta organización se debe a cómo los humanos distinguimos si algo es o no, por ejemplo, una cara: lo es si tiene ojos, nariz, boca, orejas... todo esto son elementos concretos.

La convolución es una especie de “media móvil”

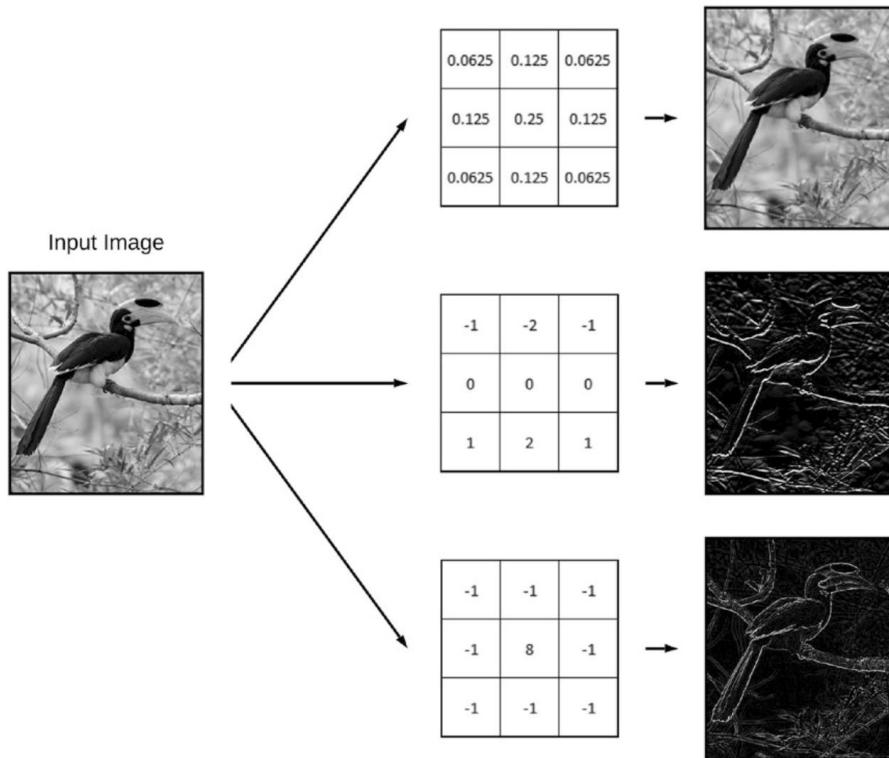
- Filtros de convolución: Técnica tradicional muy empleada en procesamiento de imágenes



...dependiendo de la matriz de convolución empleada, se consiguen distintos efectos tales como suavizados de la imagen, difuminado, detección de fronteras, etc.

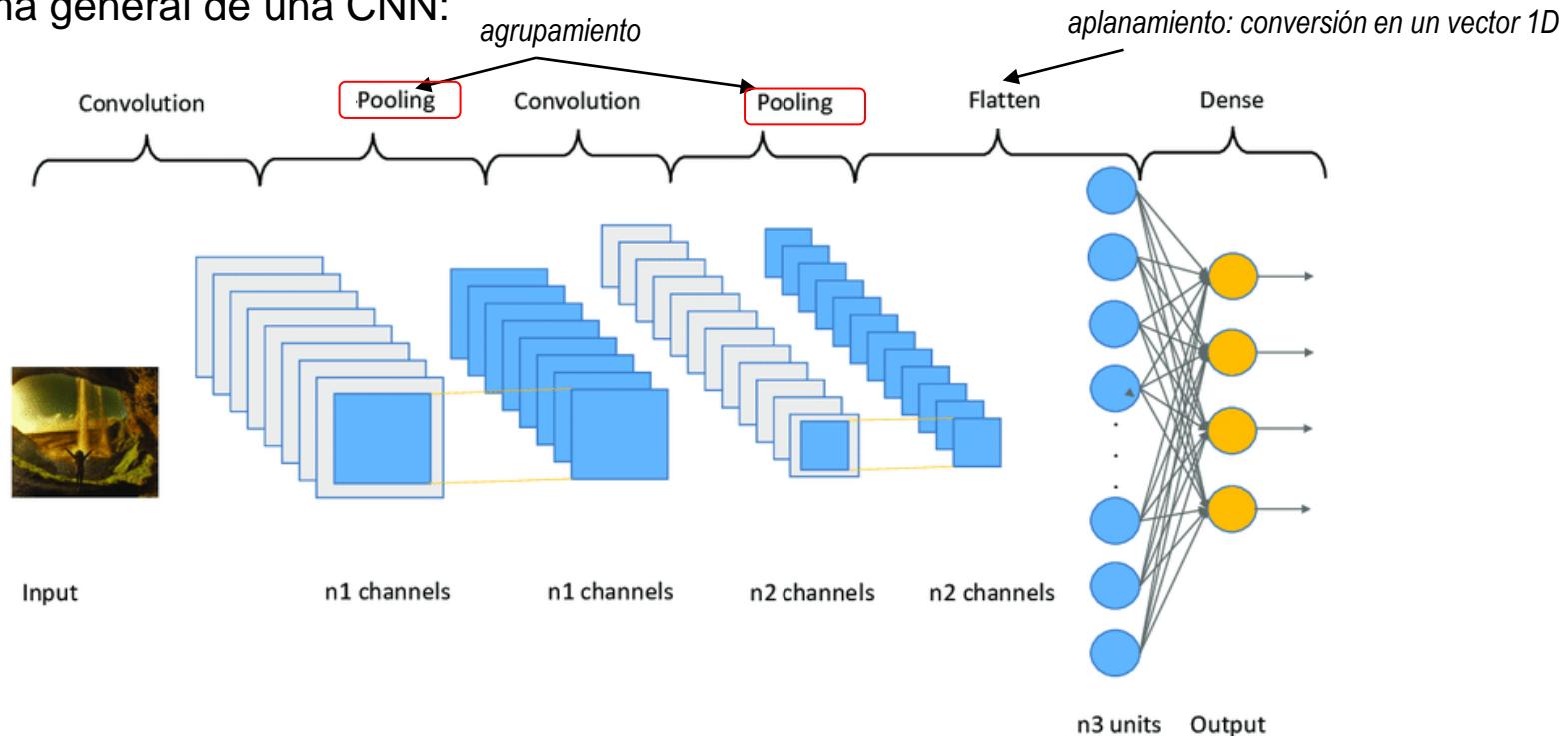


- Ejemplos del efecto de distintos filtros de convolución sobre una misma imagen



Deep Learning on Windows, Thimira Amaratunga. Apress.

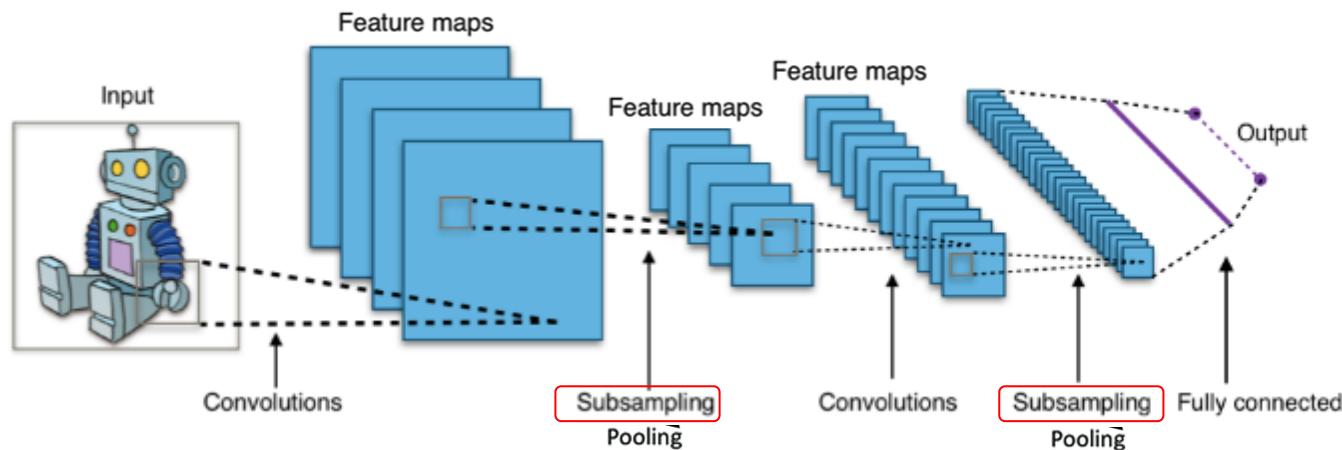
- Esquema general de una CNN:



- Hay dos tipos de capas en las CNNs: **convolucionales** y de **pooling** (o agrupamiento)
- La primera **capa de neuronas de convolución** utiliza un conjunto de filtros de convolución para **identificar un conjunto de características de bajo nivel** de la imagen.
- Estas características son **agrupadas (pooled)** en la siguiente capa de **pooling** ...

... que a su vez es la **entrada de la siguiente capa de convolución**, que de nuevo usa una serie de filtros convolucionales para **identificar un conjunto de características de más alto nivel** sobre las características de bajo nivel de la capa anterior.

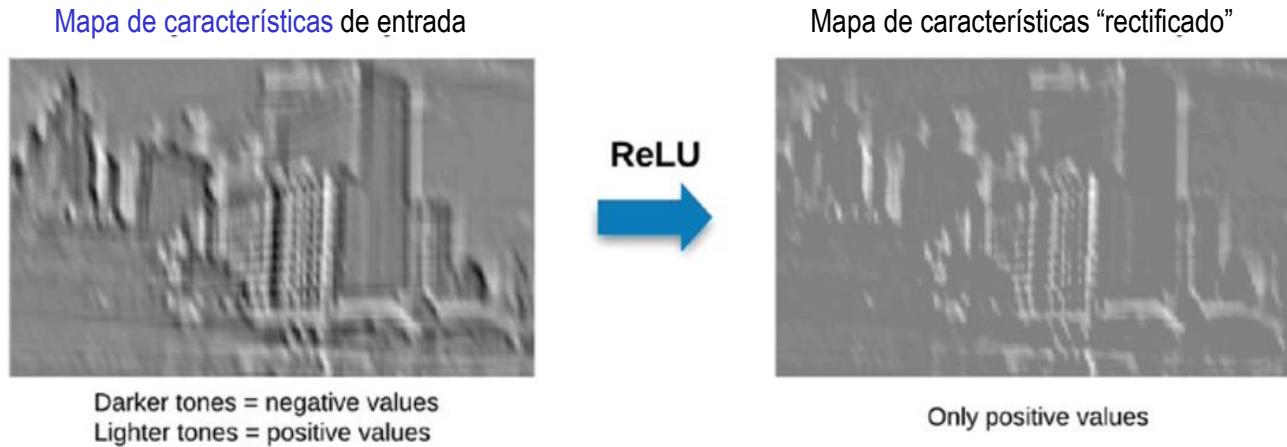
- Finalmente, **la salida de la última capa de convolución** se conecta a una capa de **neuronas densamente conectadas** para la tarea de clasificación final.
- Cada paso de **convolución + pooling** consigue, progresivamente, construir una **estructura jerárquica de características**, lo cual permite aprender patrones complejos construidos gradualmente de otros más simples (i.e. aumentando el nivel de abstracción)



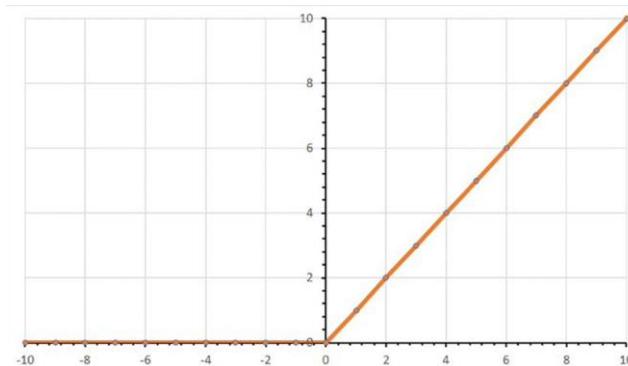
- Cuando utilizamos una CNN **no especificamos los valores del conjunto de filtros de convolución** que obtienen características de la imagen de entrada, sino **sólo cuántos filtros y qué tamaño tienen**. El **proceso de aprendizaje es quien determina los tipos de filtros usados**.
- La **capa final densamente conectada** de las CNN es, frente a las **capas convolucionales que son operadores lineales**, una **operación no lineal**. Esta operación final no lineal es necesaria, ya que los datos del mundo real no son lineales.
- Las capas convolucionales típicamente se implementan por ejemplo con la **función ReLU** (que da buenos resultados cuando se usa *backpropagation* en el aprendizaje), sigmoides o tanh, pero éstas suelen dar peores resultados particularmente en redes más profundas.
- Cuánta **profundidad deba tener una CNN** (pares de capas de convolución-*pooling*), **dependerá de la complejidad del modelo** que se esté aprendiendo.
- Para algunos especialistas en el área, los modelos profundos sólo son aquellos que tienen una estructura jerárquica, es decir, que construyen esa jerarquía de patrones, y por tanto, no sería una RNA profunda una que por muchas capas que utilizase, todas ellas fuesen densamente conectadas (pero una con unas pocas convolucionales sí).

Capa Convolucional

- Ejemplo de resultado de la función de transferencia ReLU, aplicado a mapa de caract.:.



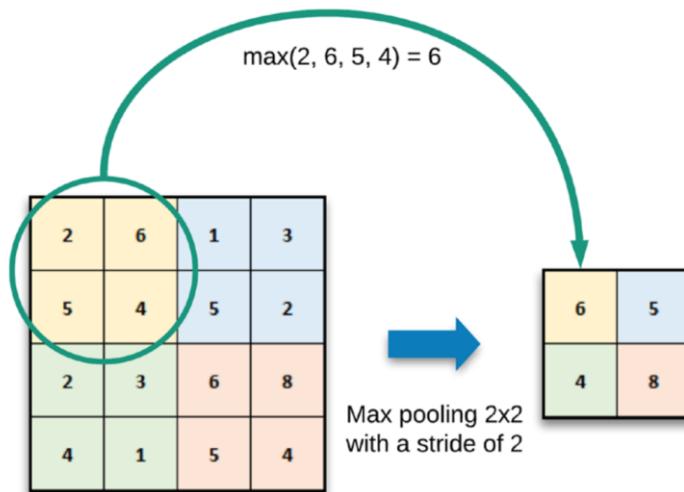
La función ReLU, que se aplica a cada pixel, retiene cada valor positivo mientras que pone a 0 los valores negativos:



$$\text{Salida} = \max(0, \text{Entrada})$$

Deep Learning on Windows, Thimira Amaratunga. Apress.

- El **pooling** (agrupación o submuestreo (*subsampling*)) que sigue a la capa convolucional reduce la dimensionalidad de cada **mapa de características**, ya que su objetivo es retener sólo la información relevante.
- Hay tres tipos de *pooling*:
 - *Max pooling*: se queda con el mayor valor de los píxeles de la zona; suele dar los mejores resultados de los tres tipos de *pooling*
 - *Average pooling*: toma el promedio de los píxeles
 - *Sum pooling*: se queda con la suma

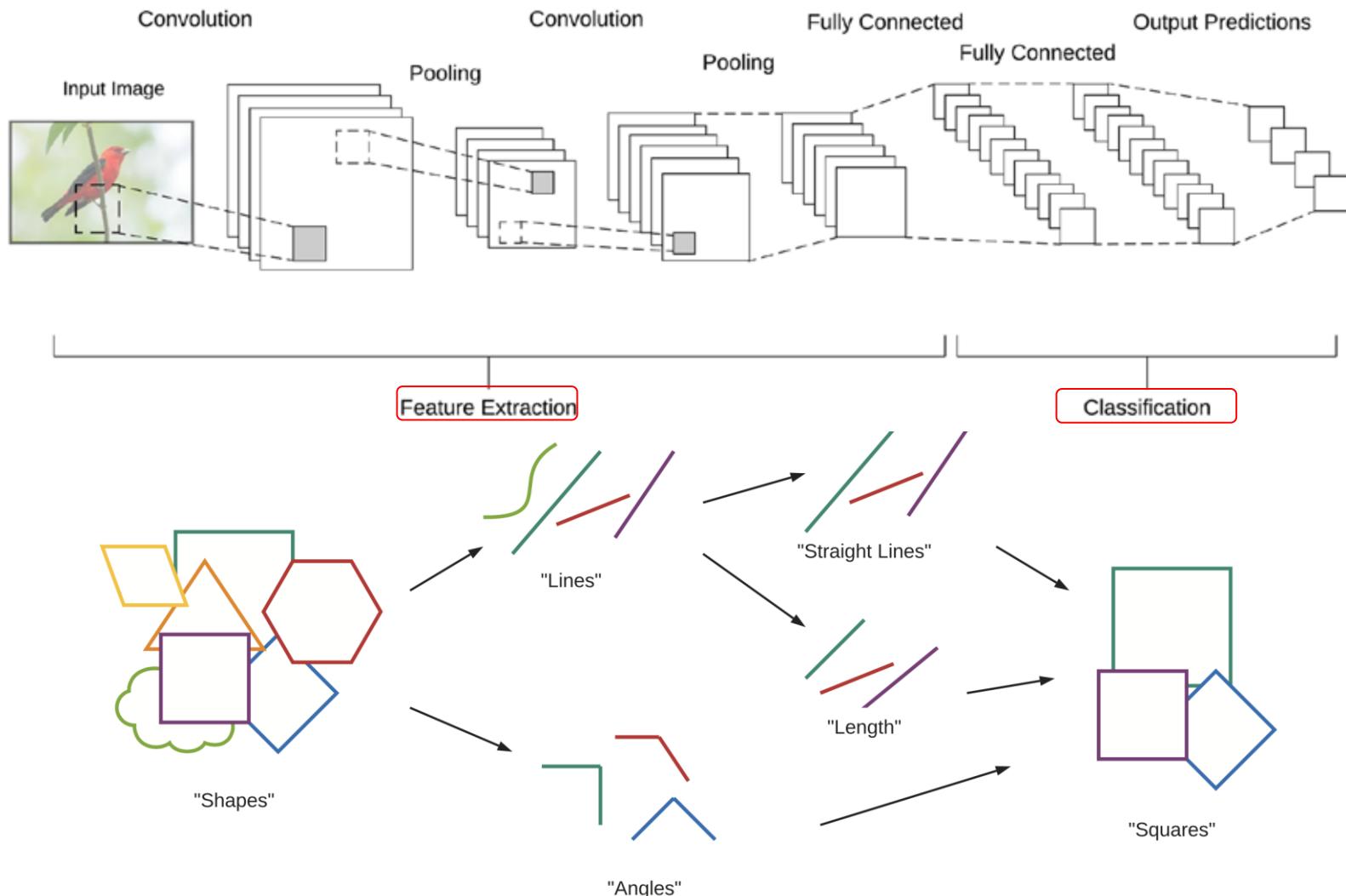


Ejemplo de *Max pooling* de paso 2,
es decir, ventana de 2x2

Deep Learning on Windows, Thimira Amaratunga. Apress.

- El ***pooling*** por tanto:
 - Reduce la dimensionalidad de las características y las hace más manejables
 - Reduce el *overfitting* (a través de la reducción de parámetros disponibles)
 - Hace a la red más insensible a pequeños cambios y ruido: mejor generalización
 - Permite detectar objetos en cualquier parte de la imagen de entrada
- La siguiente (y también las sucesivas) doble capa *convolution-pooling*, toma como entrada un mapa de características... y realiza la misma operación: de esta forma, se propicia el citado efecto de aprender características jerárquicamente.
- La capa completamente conectada (o densa) de salida, como se ha dicho, clasifica igual que un perceptrón multicapa tradicional; emplea, como se mostró previamente, ***softmax*** como función de activación fundamentalmente en **problemas multiclase**, mientras que sería preferible una de tipo ***sigmode*** para los casos de **clasificación binaria**.
- La acción combinada es una secuencia de **extracción de características (*convolution*)** y **reducción de las necesarias (*pooling*)**

Ilustración sobre la extracción de características y clasificación de una RN Convolucional



Conceptos y reflexiones importantes

- La diferencia entre una **capa convolucional** y una capa densamente conectada es que la capa densa **aprende patrones globales en su espacio global de entrada**, mientras que las capas convolucionales **aprenden patrones locales en pequeñas ventanas de dos dimensiones**: esto es precisamente porque **no** todas las neuronas están conectadas con todas sino sólo con algunas.
- Gracias a ello, la capa convolucional es capaz de detectar características concretas en las imágenes.
- Una **capa convolucional**, **una vez que aprende una característica en una zona de la imagen, puede detectarla en cualquier otra zona de ella**.
- Una **capa densa** no puede hacer eso, y para detectar ese patrón, debe aprender de nuevo el patrón en su nueva localización.
- La otra característica de las capas convolucionales es su capacidad para aprender jerarquías de patrones: conceptos visuales cada vez más complejos.
- Las capas convolucionales actúan sobre tensores 3D (los llamados *feature maps*) con dos ejes espaciales: altura y anchura (*height* y *width*), y un eje profundidad (*depth*) (1 si es en niveles de grises, y 3 si es en color (RGB))

Conexión entre un filtro de convolución clásico y lo que hace una capa convolucional de una RN

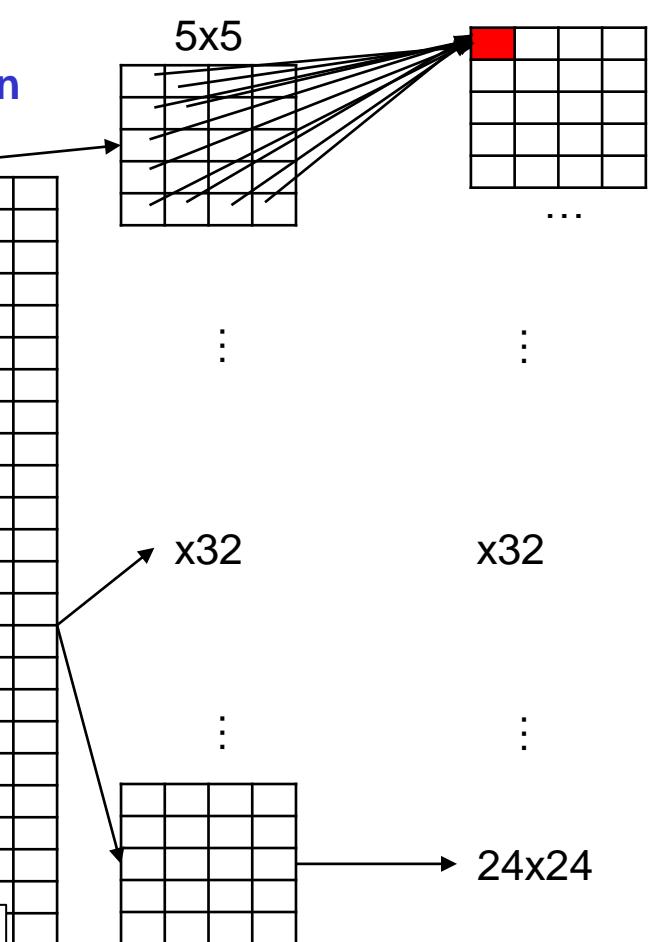
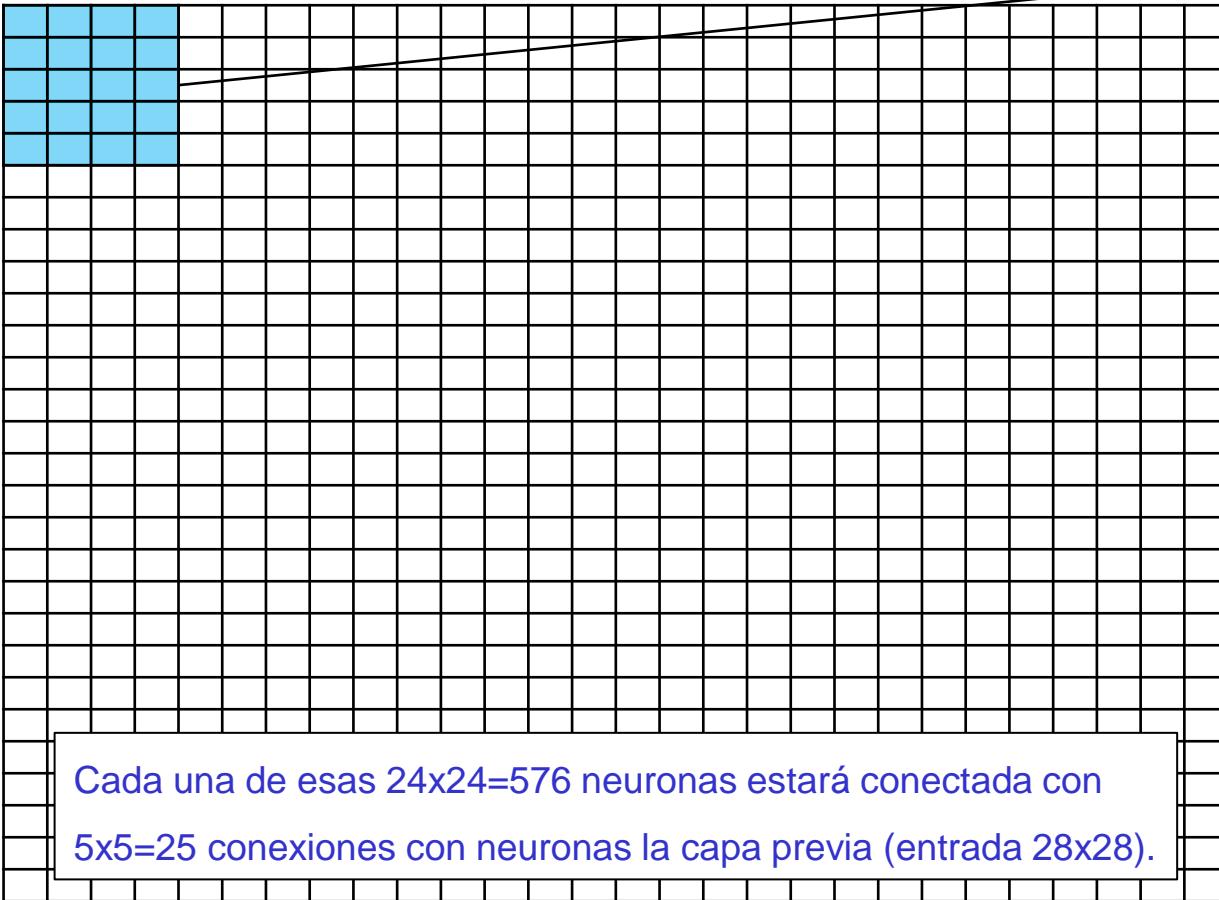
- Hemos comentado que la convolución es un filtro que se aplica **deslizando** una matriz pequeña de un tamaño determinado (3×3 , 4×4 , 5×5 ...) sobre los píxeles de la imagen, calculando así *una especie de media*. ¿Cómo hace esto la capa convolucional de una RN?
 - Cada neurona de la capa oculta se conecta con una región pequeña de la capa de **entrada** (capa de entrada que tiene una neurona por pixel) de forma que intuitivamente, **se puede considerar que es como una ventana deslizante que se mueve por la imagen** (por ejemplo, la de 28×28 pixeles) o capa de 28×28 neuronas: para cada posición de la ventana de deslizante (en realidad, cada grupo de neuronas de entrada tiene un pequeño grupo de neuronas al que está conectado en la capa oculta).

Tamaño del filtro de convolución, efecto de reducción

- Si movemos un filtro de convolución de 3x3 sobre una imagen de 28x28, en realidad, de salida tenemos una imagen de 26x26 pixeles (al colocar el filtro sobre la esquina superior izquierda para aplicarlo, se “moverá” centrado en un subgrupo de 26x26 pixeles, no por los 28x28 originales (i.e. perdemos uno por cada lado)).
- Si en vez de ser de 3x3 fuese de 5x5, entonces tendríamos 24x24 como salida (perdemos dos por cada lado: dos es la mitad truncada del tamaño del filtro).
- Trasladado a las RNCs, si la capa de entrada tiene 28x28 neuronas (una por pixel), un filtro de 5x5 nos dejaría una primera capa oculta de 24x24 neuronas; por tanto, cuando especificamos cuántas neuronas hay en la capa oculta, estamos implícitamente indicando el tamaño de nuestro filtro de convolución.
- Cada una de esas $24 \times 24 = 576$ neuronas estará conectada con $5 \times 5 = 25$ conexiones con neuronas la capa previa (entrada).

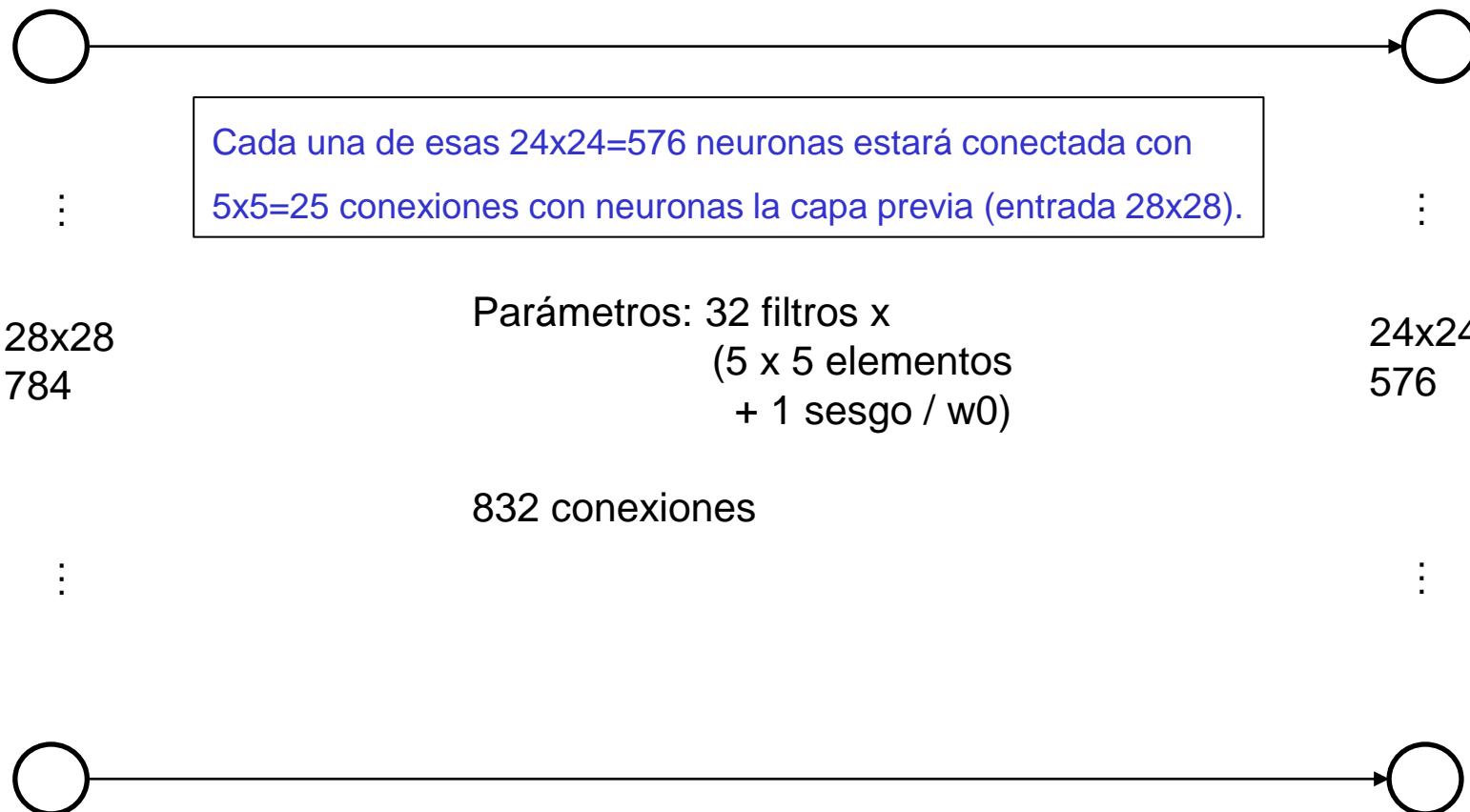
Tamaño del filtro de convolución, efecto de reducción

28x28

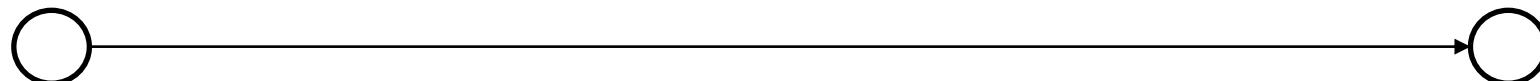


Parámetros: 32 filtros x
(5×5 elementos
+ 1 sesgo / w_0)

Tamaño del filtro de convolución, efecto de reducción



Tamaño del filtro de convolución, efecto de reducción



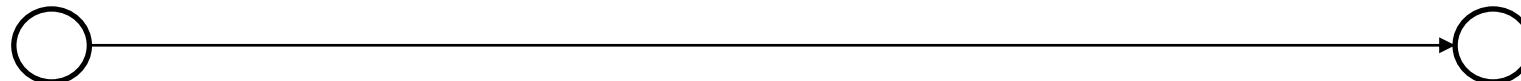
Cada una de esas $24 \times 24 = 576$ neuronas estará conectada con
 $5 \times 5 = 25$ conexiones con neuronas la capa previa (entrada 28×28).

$28 \times 28 \times 1$
784

Parámetros: 18.432 neuronas
 \times (25 conexiones
+ 1 sesgo)

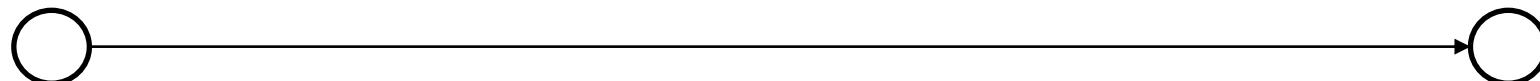
$24 \times 24 \times 32$
18.432

479.232 conexiones



Si consideramos cada neurona por separado

Tamaño del filtro de convolución, efecto de reducción



Cada una de esas $24 \times 24 = 576$ neuronas estará conectada con
 $5 \times 5 = 25$ conexiones con neuronas la capa previa (entrada 28×28).
⋮ ⋮

$28 \times 28 \times 1$
784

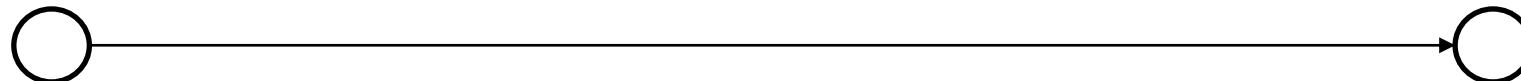
Parámetros: 18.432 neuronas
 \times 784 neuronas

$24 \times 24 \times 32$
18.432

14.450.688 conexiones

⋮ ⋮

Si consideramos que están densamente conectadas

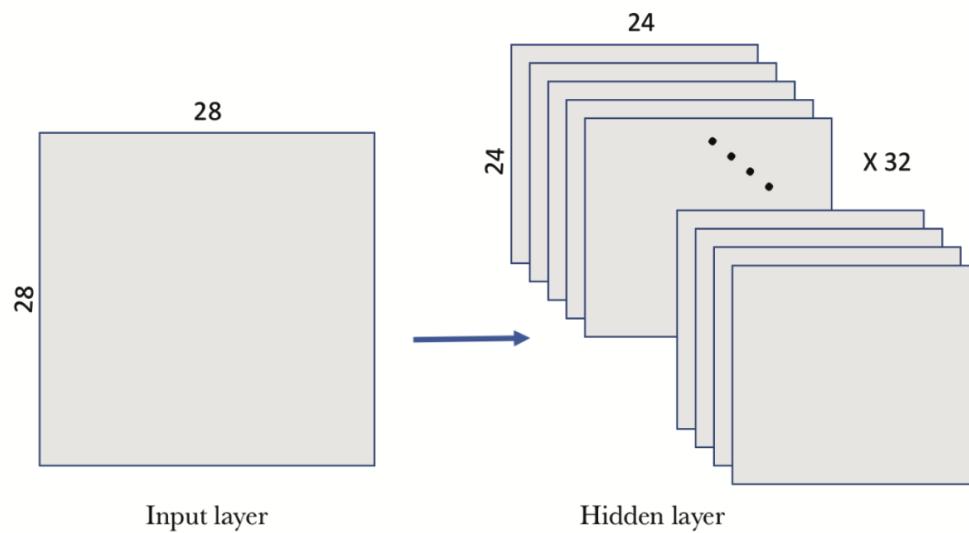


Número de parámetros en RNCs

- Esos $5 \times 5 = 25$ conexiones **son** los parámetros o elementos del filtro o matriz de convolución.
- Un filtro de convolución de 3×3 , debe aprender 9 parámetros... uno de 5×5 debe aprender 25 parámetros... etc.
- Las RNCs usan el mismo filtro (pesos y sesgo) para todas las neuronas de una misma capa oculta, en este caso, para las 24×24 neuronas, lo cual significa que hay muchos menos parámetros que aprender en la práctica que si fuese, por ejemplo, la capa de una RN densamente conectada.
- RNC con un filtro de 5×5 (por tanto, con 576 neuronas ocultas), tendrá 25 parámetros del filtro + sesgos.

Varios filtros convolucionales en una capa convolucional

- Como **cada filtro convolucional aprende una característica (*feature*)** (por ejemplo, uno puede aprender líneas verticales), **una capa convolucional debe emplear varios filtros convolucionales a la vez** (así puede aprender líneas verticales, líneas horizontales, oblicuas de derecha a izquierda, oblicuas de izquierda a derecha, arcos ... etc.).
- Por ejemplo, podríamos decidir tener 32 filtros convolucionales, de tal forma que:

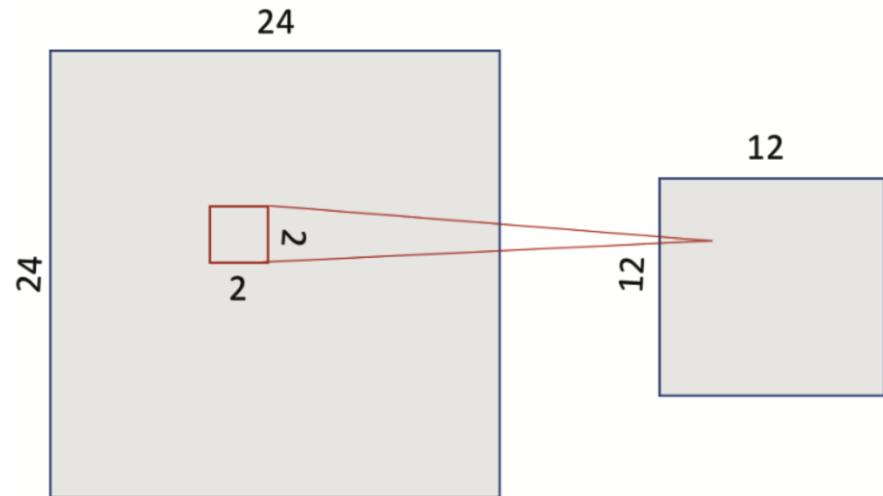


El tensor de entrada de la capa convolucional es de $(28, 28, 1)$, y como salida genera un tensor de $(24, 24, 32)$. Cada filtro tiene sus parámetros, y genera su salida (*feature map*) diferente.

Capas de *Pooling*: Simplificación (I)

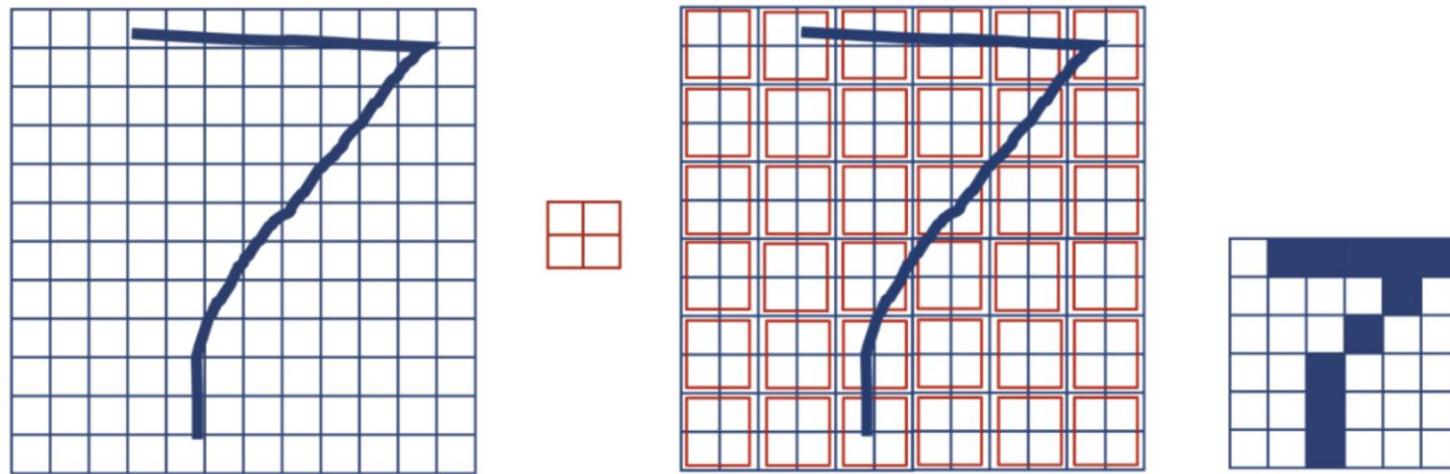
- Como se ha visto, las capas de *pooling* hacen una simplificación de la información detectada por la capa convolucional que le precede.
- Esta simplificación, como se comentó previamente, se puede hacer mediante distintos modelos, por ejemplo, *max-pooling*, que opta por el valor máximo de la ventana de entrada utilizada

Si aplicamos, por ejemplo, una ventana de *pooling* de 2×2 a la entrada de 24×24 (proveniente de la convolución), obtenemos una salida de 12×12 (que es 4 veces menor que su entrada) ($24 \times 24 / 4 = 144 / 4 = 12 \times 12$)



Capas de *Pooling*: Simplificación (II)

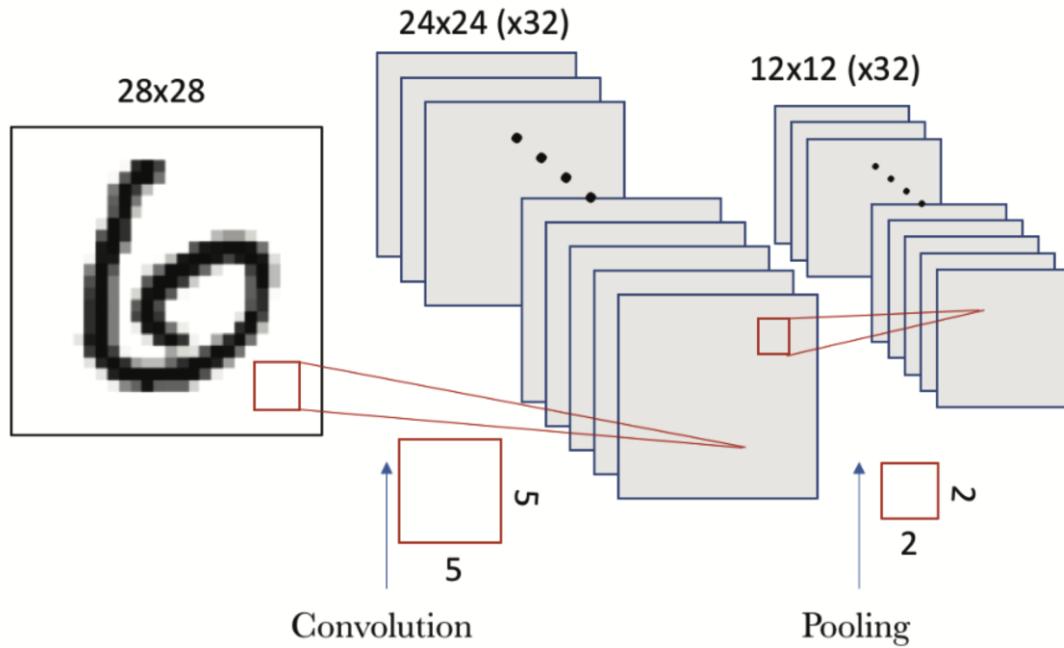
- Ejemplo con una matriz de entrada de 12x12, y una ventana de 2x2 (salida de 6x6) *max-pooling*:



- Como se puede apreciar, la ventana de *pooling*, a diferencia de la convolucional no es deslizante (con solapamiento), sino que simplemente **divide** la matriz de entrada
- **Las capas de *pooling* no tienen parámetros para ser aprendidos:** son cálculos

Capas de *Pooling*: Simplificación (III)

- Tendremos tantos filtros de *pooling* como filtros convolucionales; pero sabemos que tenemos múltiples filtros convolucionales...



La convolución se hizo con 24x24 neuronas en cada filtro, y hemos empleado en el *pooling* 12x12 “neuronas” (por filtro)

Implementación con Keras de una RNC (I)

- Vamos a implementar el modelo que se ha descrito previamente, para el mismo problema que empleamos para ilustrar la RN densamente conectada.

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (5,5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))

model.summary()
```

32 filtros convolucionales de 5x5

Tensor de entrada (28x28,1)

Ventana de pooling de 2x2

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 24, 24, 32)	832
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 32)	0
Total params:	832	
Trainable params:	832	
Non-trainable params:	0	

$(1_{(\text{imagen})} \times 25_{(5 \times 5)} + 1_{(\text{sesgo})} \times 32_{(\text{filtros})}) = 832$

No tiene porque es una operación matemática de calcular un máximo

Implementación con Keras de una RNC (II)

- Añadimos más capas:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (5, 5), activation='relu')) 64 filtros convolucionales de 5x5: porque hemos querido que sea así  
(normalmente, se duplica el número de filtros de la capa anterior)
model.add(layers.MaxPooling2D((2, 2))) Esta vez no ponemos el  
input_shape porque lo deduce  
Keras automáticamente
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 24, 24, 32)	832
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 32)	0
conv2d_2 (Conv2D)	(None, 8, 8, 64)	51264
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 64)	0
Total params: 52,096		
Trainable params: 52,096		
Non-trainable params: 0	<i>Al pasar un filtro de 5x5 sobre una entrada de 12x12, perdemos dos a cada lado de la matriz</i>	<i>(32_(imagen) x 25_(5x5) + 1_(sesgo)) x 64_(filtros) = 51.264</i>

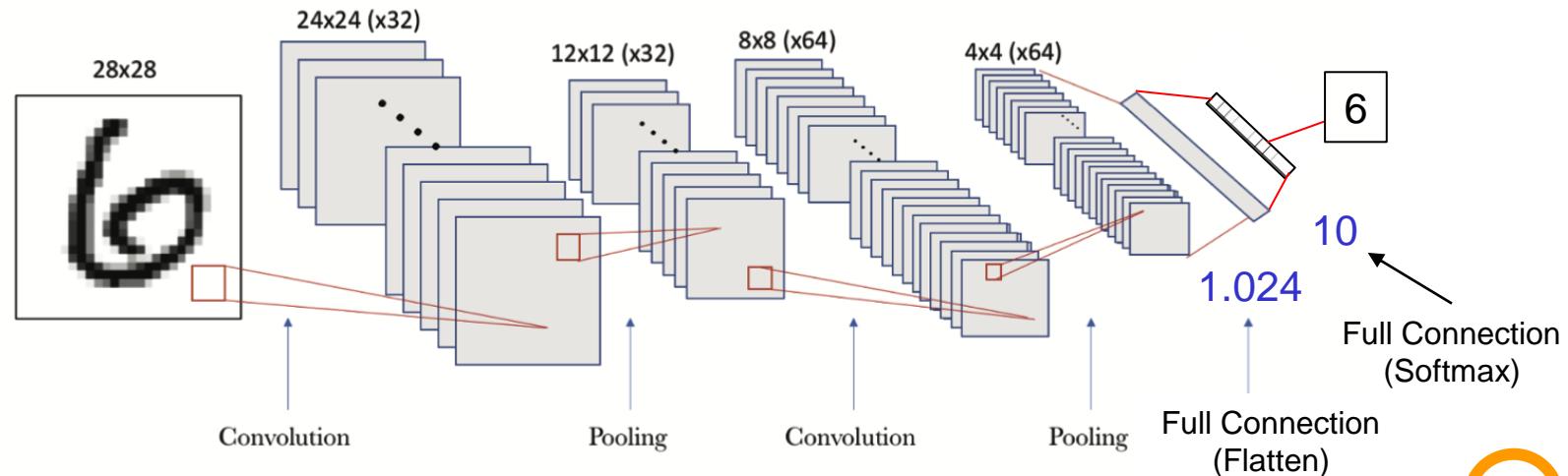
Implementación con Keras de una RNC (III)

- La salida de lo anterior es un tensor 3D, pues tiene (4,4,64) (*height, width, channels*)
- Ahora se añadiría una capa densamente conectada que utilice *softmax* para la clasificación final.
- Sin embargo, como la salida que tenemos de la última capa es un tensor 3D, hay que aplatarlo (*Flatten*) previamente (1024 en 1D) ($4 \times 4 \times 64 = 1024$), por tanto, todo el proceso sería:

```
model = models.Sequential()  
  
model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(28, 28, 1)))  
model.add(layers.MaxPooling2D((2, 2)))  
  
model.add(layers.Conv2D(64, (5, 5), activation='relu'))  
model.add(layers.MaxPooling2D((2, 2)))  
  
model.add(layers.Flatten())  
  
model.add(layers.Dense(10, activation='softmax'))
```

Implementación con Keras de una RNC (IV)

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_1 (Conv2D)	(None, 24, 24, 32)	832
<hr/>		
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 32)	0
<hr/>		
conv2d_2 (Conv2D)	(None, 8, 8, 64)	51264
<hr/>		
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 64)	0
<hr/>		
flatten_1 (Flatten)	(None, 1024)	0
<hr/>		
dense_1 (Dense)	(None, 10)	10250
<hr/>		
Total params: 62,346		
Trainable params: 62,346		
Non-trainable params: 0		



Implementación con Keras de una RNC (V)

- Carga de datos, compilación, entrenamiento, evaluación:

```
from keras.datasets import mnist
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])

model.fit(train_images, train_labels, batch_size=100, epochs=5, verbose=1)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)
```

Dará: Test accuracy: 0.9704 es mejor resultado que el obtenido por la RN densamente contactada

Configuraciones de RNCs estándar

- Escoger una configuración para una RNC es algo que se hace, bien en base a la experiencia previa, o bien buscando modelos similares. En ese sentido, existen algunas RNCs muy conocidas ya que resuelven problemas clásicos, y se les conoce con el nombre que les dieron sus autores, y lo mejor: **están ya construidas**, es decir, no sólo podemos consultar su arquitectura y describirlas, sino que **están en la biblioteca de Keras**, y podemos importarlas directamente, **incluso con sus pesos de entrenamiento para algunos datasets**.
- Ejemplos:
 - LeNet (años 90, Yann LeCun): La que acabamos de utilizar para caracteres, con 32x32
 - AlexNet (2012, Alex Krizhevsky), ganó el concurso ImageNet de ese año
 - GoogleLeNet, mejora AlexNet
 - VGGNet, etc.
- Importando VGGNet, y **cargando los parámetros de su entrenamiento con *imagenet***

```
from keras.applications import VGG16
```

```
model = VGG16(weights='imagenet')
```

importamos sus más de 18 millones de parámetros entrenados

Configuraciones de RNCs estándar

- ... es decir: Keras incorpora muchos modelos pre-entrenados:
 - VGG16
 - VGG19
 - Xception
 - ResNet50
 - InceptionV3
 - InceptionResNetV2
 - MobileNet
 - DenseNet
 - NASNet

Tema 6: Deep Learning

Índice:

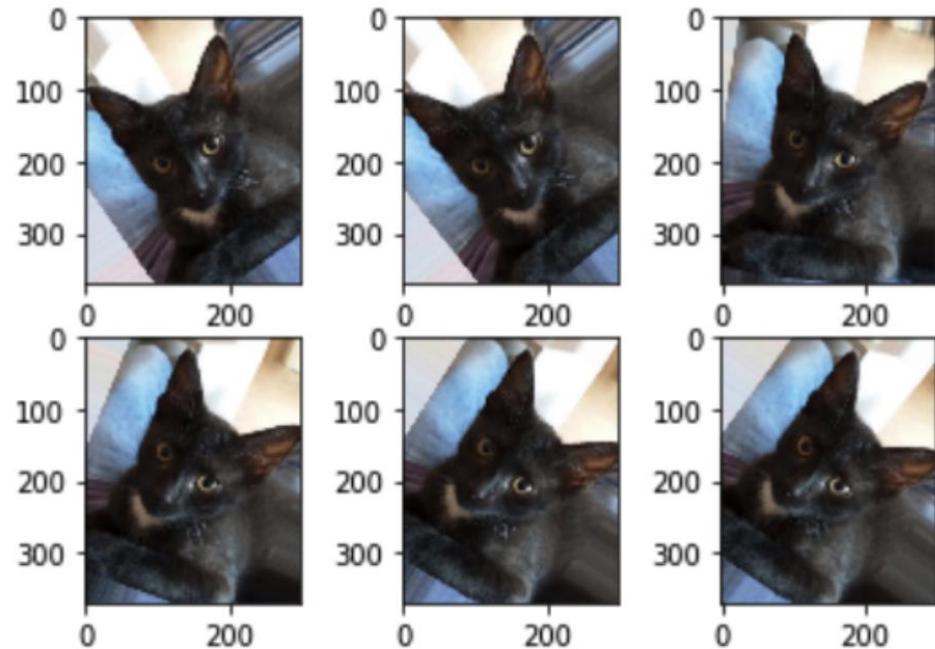
1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

5. Falta de datos: Data Augmentation & Transfer Learning

- La falta de datos (i.e. conjuntos pequeños para aprendizaje) para poder entrenar estos modelos de RNs profundas (que necesitan grandes volúmenes), es un problema que conduce al sobreajuste. Hay dos formas fundamentales de combatirlo:
 - *Data Augmentation*: Derivar más datos sintéticamente de los que ya tenemos
 - *Transfer Learning*: Utilizar modelos pre-entrenados, para ajustarlos con pocos datos
- ***Data Augmentation***: Consiste en **utilizar transformaciones aleatorias de la imagen** para producir **nuevas imágenes**, para que el modelo que estamos creando **no trabaje dos veces con la misma imagen en las diferentes épocas**.
- Las transformaciones consiste en rotaciones, giros, volteos, etc., para producir nuevas imágenes que podrían pertenecer perfectamente al conjunto original, pero que la RN las ve como imágenes nuevas y diferentes.

- Las transformaciones en Keras se pueden hacer mediante `ImageDataGenerator`.

```
train_datagen = ImageDataGenerator(  
    rescale=1./255,  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```



- **Transfer Learning**

- Hay una **cualidad relevante** de las RNs que permite combatir la falta de datos, o la falta de gran capacidad de cálculo para entrenamiento, que es **utilizar modelos pre-entrenados** (Keras los proporciona).
- Esta característica de algunos modelos, se basa en que no todo nuevo modelo necesita ser entrenado *from-scratch*, (con muchos datos, y quizás una importante potencia computacional) para cada aplicación, dado que es posible reutilizar una red que ha sido previamente entrenada, es decir, con sus **miles o millones de parámetros ya ajustados**. (Por ejemplo, VGG16 tiene 138 millones de parámetros).
- Esta red pre-entrenada, sólo necesita un **conjunto pequeño de datos para adaptarse y servir para la nueva aplicación**.
- ¿Por qué es posible esto? – El entrenamiento previo para otra aplicación, contiene las **características aprendidas en la jerarquía**, las cuales son reaprovechadas, dentro de **aplicaciones parecidas relativamente**, como pueden ser por ejemplo, reconocimiento de objetos en imágenes.

- Las características más genéricas en la jerarquía son las que se reutilizan, por ejemplo, la detección de bordes o líneas básicas (rectas en distintas posiciones, arcos, etc.) que puedan ser empleados para reconocer coches y camiones (originalmente), y se cree una RN para reconocer aviones y trenes.
- Hay **dos variantes** de Transfer Learning:
 - **Extracción de Características:** Se trata de aprovechar las características aprendidas en las capas de convolución/*pooling*, y aprender sólo el clasificador final: **se borran o somete a re-entrenamiento sólo los pesos del clasificador** (capa densa), y se aprende con los nuevos datos. Keras permite cargar un modelo e indicarle qué capas deben entrenarse (**entrenables**), y qué capas no (**congeladas**).
 - **Ajuste fino:** Cuando se considera que alguna/s capas convolucionales deben actualizarse... por ejemplo, si tenemos 3 ó 4 niveles, quizás el 4 nivel es necesario para una nueva aplicación... (basado en la experiencia, y prueba error).

Tema 6: Deep Learning

Índice:

1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

6. Otros modelos de RNs Profundas

- Además de los modelos de RNA Profundas de tipo Convolucional, existen muchos otros modelos, es decir, arquitecturas conocidas para aplicaciones específicas, inclusive con capas especiales para determinado tipo de objetivo. Así, se puede hablar entre otras de:
 - **Recurrent Neural Networks (RNN) (1997): Long Short Term Memory (LSTM)**
 - Conexiones hacia atrás (*loops*) de las capas
 - Emplean celdas de memoria para recordar valores en periodos que pueden ser cortos o largos de tiempo
 - Aptas para procesamiento de series temporales
 - **Deep Belief Networks (DBN) (2006): Inicio del Deep Learning**
 - Plantearon el pre-entrenamiento no supervisado de la red como método de inicialización inteligente, para posteriormente, en la aplicación definitiva, entrenar con normalidad de forma supervisada
 - **Generative Adversarial Networks (GAN) (2014)**
 - Dos redes, una generando imágenes y la otra tratando de ver si son verdaderas o no, mejoran progresivamente hasta que la que trata de ver si son artificiales o reales, no es capaz de distinguirlas... y en ese momento, las imágenes ficticias que se producen son de gran realismo.

- **Autoencoders (Codificadores Automáticos)**

- Son modelos de aprendizaje no supervisado en los que la entrada y la salida de la red, representan lo mismo
- Actúan *codificando* la imagen de entrada, es decir, la transforman en un código con una dimensión considerablemente menor, y posteriormente, realizan el proceso inverso, *descodificando* ese código en una imagen de la dimensión original
- Aprenden un método de compresión y descompresión
- Se usan para eliminar ruido, imputar datos perdidos, generar nuevos datos, identificar anomalías, segmentación semántica...

- **Transformers**

- Es un modelo de red que ha revolucionado y transformado el mundo del Procesamiento del Lenguaje Natural (PLN), y son la base de los famosos modelos de ChatGPT (*Generative Pre-trained Transformers*), etc.
- Se trata de una RN que aprende contexto, y por tanto, significado mediante el seguimiento de relaciones en datos secuenciales (Rick Merritt, NVIDIA 2022)

- **Etc.**



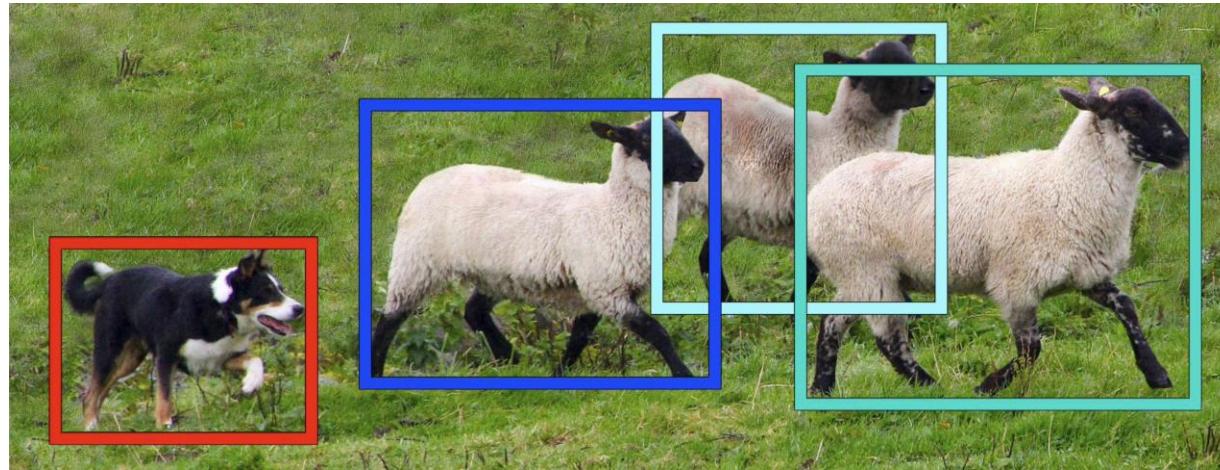
- Para el caso específico del **tratamiento de imágenes**, vamos a plantear los posibles modelos desde el punto de vista de lo que pueden hacer, que sería:
 - **Modelos de Visión para Clasificación:** Se trata de modelos que disciernen clases entre los objetos que "ven", por ejemplo, perros y gatos, o entre animales en general, etc. **No son los que más requisitos computacionales requieren (nivel de CPU)**, por tanto, son muy veloces.
 - **Modelos de Visión para Localización:** Se trata de obtener las coordenadas y las clases de uno o varios objetos en la imagen (hacen rectángulo (*bounding box*) que contienen el objeto y la etiqueta de clase). **Suelen ser modelos de un peso computacional intermedio**, es decir, mejor utilizarlos en GPU, aunque algunos pueden utilizarse en CPU. Dos tipos:
 - Un solo objeto: Localización propiamente, para ubicar ese objeto en la imagen.
 - Múltiples objetos: **Detección de objetos**. Ejemplo: en una calle, detectar los coches, autobuses, señales, peatones, semáforos... etc.
 - **Modelos de Visión para Segmentación:** Se trata de clasificar a nivel de pixel, es decir, nos van a marcar la silueta de cada objeto para distinguirlo de otros o del fondo. **Son los modelos computacionalmente más pesados (nivel de GPU), por tanto, son lentos.**

- **Modelos de Visión para Clasificación:**

- Se trata de modelos que disciernen clases entre los objetos que "ven", por ejemplo, perros y gatos, o entre animales en general, etc.
- No son los que más requisitos computacionales requieren (nivel de CPU), por tanto, son muy veloces.
- Modelos de RNAs profundas para clasificación por excelencia son las Convolucionales (RNCs), si bien, no toda RNC es una arquitectura creada para clasificación, pues muchos modelos de RNCs tienen como objetivo, por ejemplo, segmentación (U-Net y Res-Net).
- Ejemplos:
 - AlexNet, es uno de los primeros modelos empleados razonablemente competitivo
 - VGG16, con arquitectura simple y simétrica
 - ResNet, apropiada para redes muy profundas, dado que combate el problema del desvanecimiento del gradiente
 - Etc.

- **Modelos de Visión para Localización:**

- Se trata de obtener las coordenadas y las clases de uno o varios objetos en la imagen (hacen rectángulo que contienen el objeto y la etiqueta de clase).
- Suelen ser modelos de un peso computacional intermedio, es decir, mejor utilizarlos en GPU, aunque algunos pueden utilizarse en CPU. Dos tipos:
 - Un solo objeto: Localización propiamente
 - Múltiples objetos: **Detección de objetos.** Ejemplo: en una calle, detectar los coches, autobuses, señales, peatones, semáforos... etc.



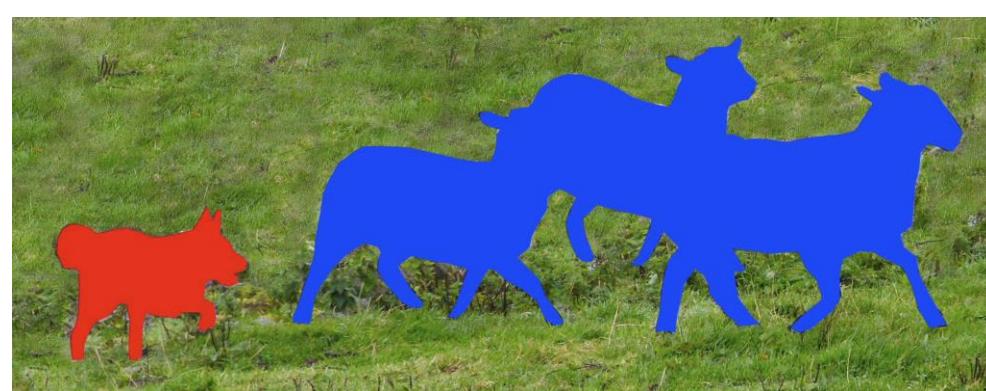
- **Modelos de Visión para Localización:**

- **Localización y Detección de objetos:** Ejemplos:

- RCNN, para trabajar sobre zonas concretas de una imagen
 - Faster RCNN, versión rápida de la RCNN, que utiliza una RNC, y trabaja sobre imágenes completas en lugar de zonas restringidas como RCNN, en un paso
 - SSD, muy rápida y precisa, a base de una RNC única (una etapa), empleada en aplicaciones de tiempo real
 - YOLO: Arquitectura rápida y precisa, con una sola RNC para detección y localización en tiempo real
 - RetinaNet, usando en este caso un enfoque de dos etapas para detectar objetos
 - Etc.

- **Modelos de Visión para Segmentación:**

- Se trata de clasificar a nivel de pixel, es decir, nos van a marcar la silueta de cada objeto para distinguirlo de otros o del fondo. Son los modelos computacionalmente más pesados (nivel de GPU), por tanto, son lentos.
- Nos permiten identificar la forma de los objetos (u objetos de interés).
- Se utilizan, por ejemplo, para poder medir o cuantificar los objetos.
- Distinguir entre:
 - **Segmentación semántica:** Cada pixel de la imagen se clasifica con una etiqueta; por tanto, para toda la imagen se usa un número de segmentos o etiquetas (árboles, coches, peatones, carretera, plantas...). **No ve objetos; sólo píxeles.**



Fuente: Robotic Manipulation: Perception, Planning and Control. Russ Tedrake 2022

Fuente: Object Detection and Instance Segmentation: A detailed overview. Shaunak Halbe, 2020

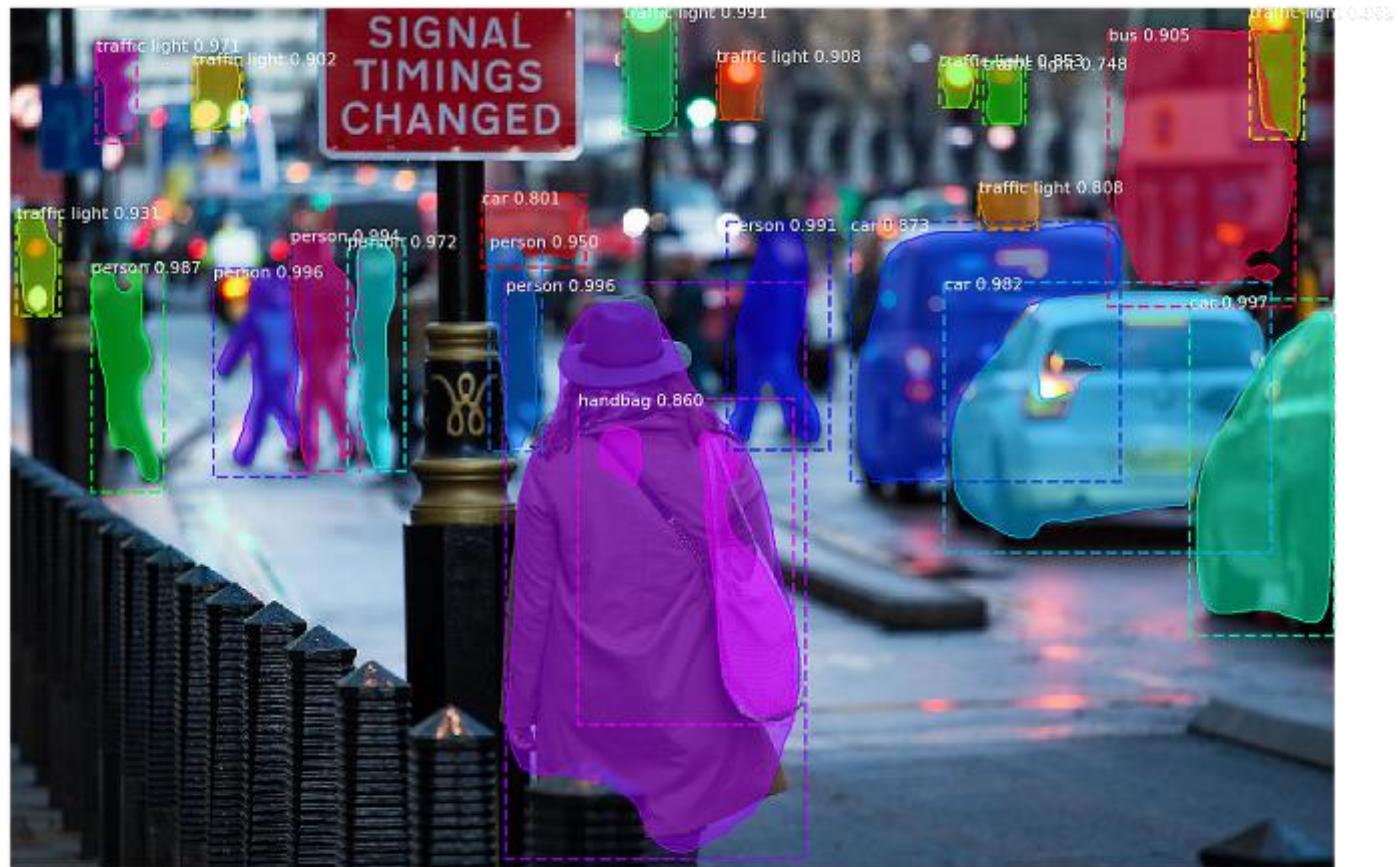
- **Modelos de Visión para Segmentación:**

- **Segmentación semántica:** Ejemplos:

- FCN, basada en la arquitectura de RNCs
 - U-Net, muy popular para imágenes médicas, llamada así por la forma de U de su arquitectura
 - SegNet, también basada en RNCs, con mecanismos de pooling inverso
 - Etc.

- **Modelos de Visión para Segmentación:**

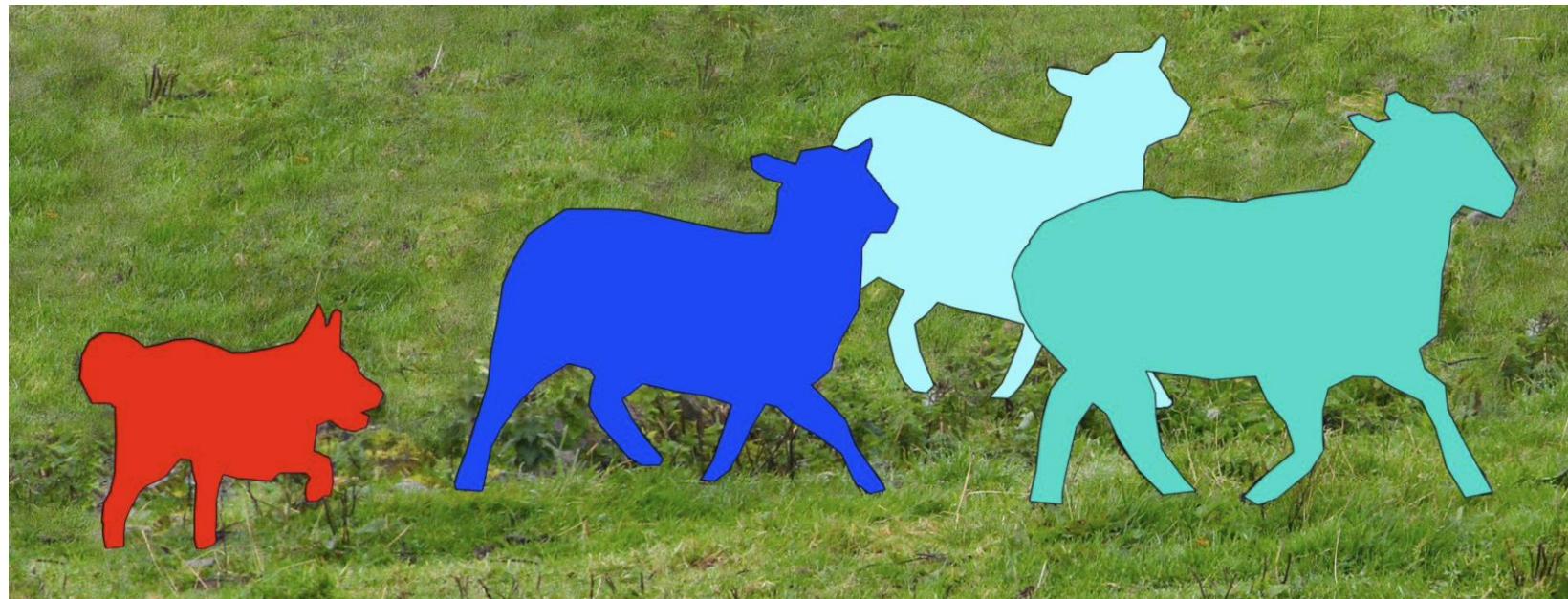
- **Segmentación de instancias:** igual, sólo que considera a cada objeto de la misma clase como un objeto individual



Fuente: Object Detection and Instance Segmentation: A detailed overview. Shaunk Halbe, 2020

- **Modelos de Visión para Segmentación:**

- **Segmentación de instancias:** igual, sólo que considera a cada objeto de la misma clase como un objeto individual



- **Modelos de Visión para Segmentación:**
 - **Segmentación de instancias:** Ejemplos:
 - Mask R-CNN, basada en Faster RCNN, con una capa de segmentación de instancias para generar una máscara binaria precisa cada cada objeto detectado
 - YOLACT, que en lugar de un post procesamiento tras la detección y segmentación utiliza un enfoque basado en máscaras de instancia a través de un conjunto de convoluciones
 - DetectoRS, que se funda en combinar métodos basados en bordes y regiones
 - SOLO, que consiste en una arquitectura de segmentación de objetos que usa una combinación de convoluciones y pooling para generar máscaras de instancia precisas
 - Etc.

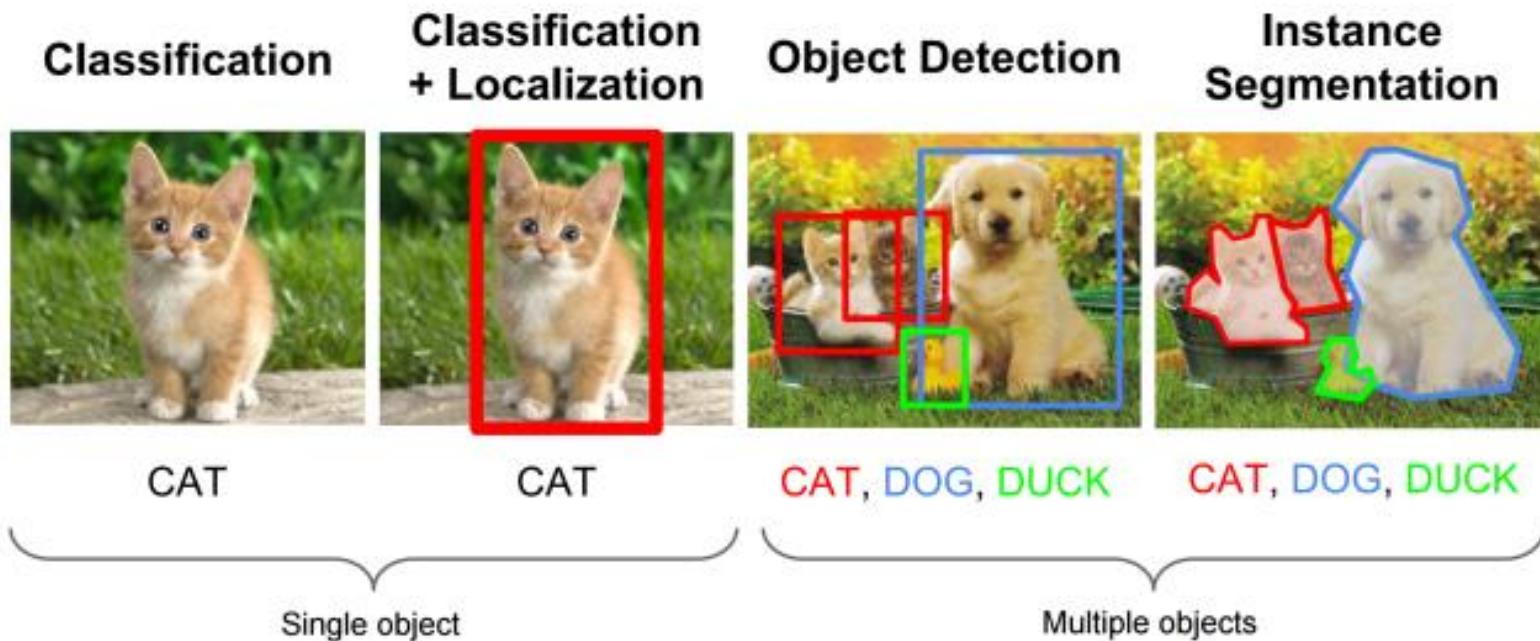
- **Modelos de Visión para Segmentación:**

- **Segmentación panóptica:** a cada pixel se la asocian dos valores: la etiqueta de su clase, y un número de instancia (combina los conceptos de segmentación de instancias y semántica)



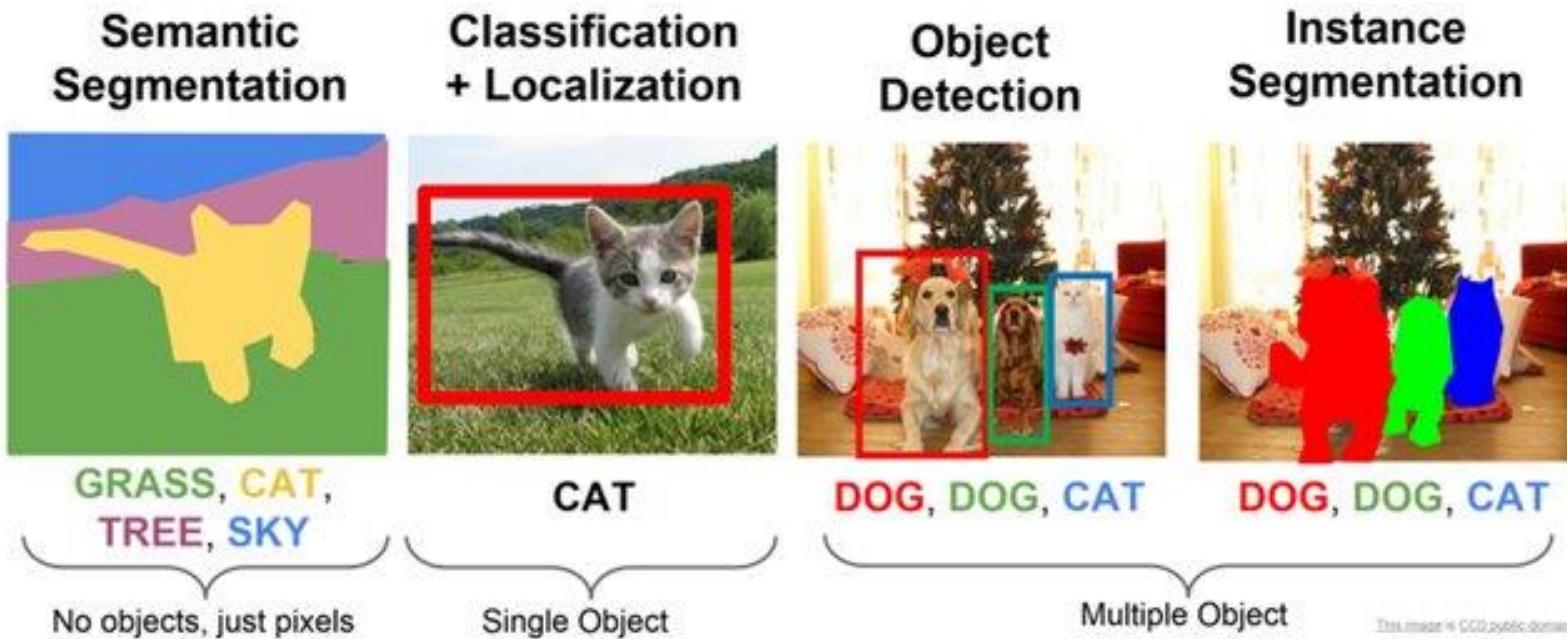
Fuente: Object Detection and Instance Segmentation: A detailed overview. Shaunak Halbe, 2020

- Comparación entre ellos:



Fuente: Object Detection and Instance Segmentation: A detailed overview. Shaunk Halbe, 2020

- Comparación entre ellos:

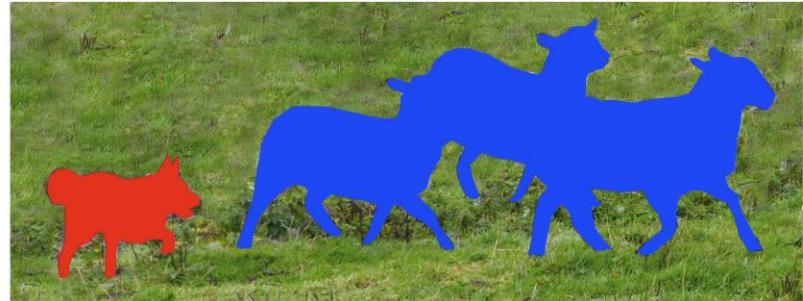


Fuente: Li, Johnson and Yeung, 2017

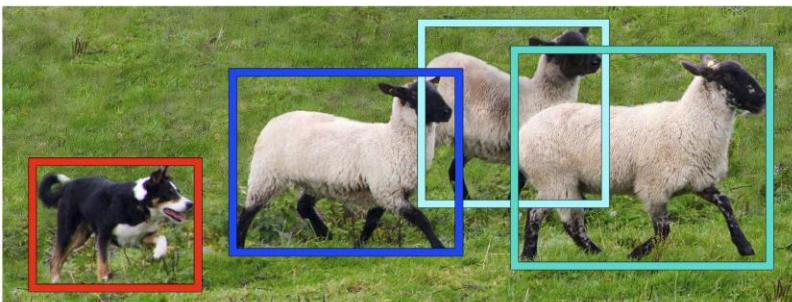
- Comparación entre ellos:



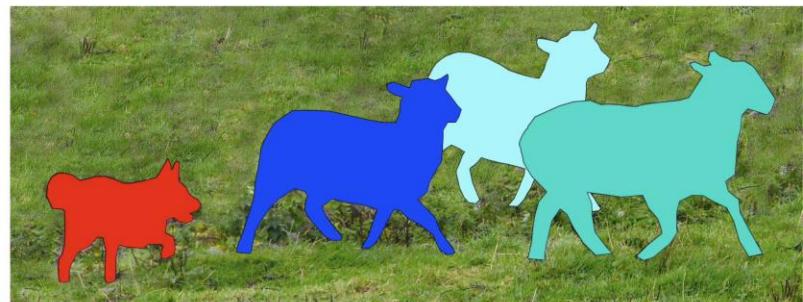
Image Recognition



Semantic Segmentation



Object Detection



Instance Segmentation

- **Imágenes estáticas vs. video:**

- Para procesar video, el modelo más elemental consiste en separar en fotogramas la secuencia de video, es decir, seleccionar una serie de fotogramas de periodicidad inferior al fps de dicho video, y procesarlos con el modelo de visión apropiado para la aplicación, dentro de los indicados en la pantalla anterior: clasificar, segmentar o detectar.
- Pero se han desarrollado modelos específicos para procesar video, como los siguientes:
- Convolutional LSTM, para predicción de seguimiento de objetos en movimiento, flujo óptico, etc
- TwoStreams Convolutional Networks, para la clasificación de acción en el video, a base de fusionar la salida de dos CNNs, una para procesar flujo óptico y la otra para fotogramas de video
- SlowFast Networks, I3D, RNNs...etc.

Tema 6: Deep Learning

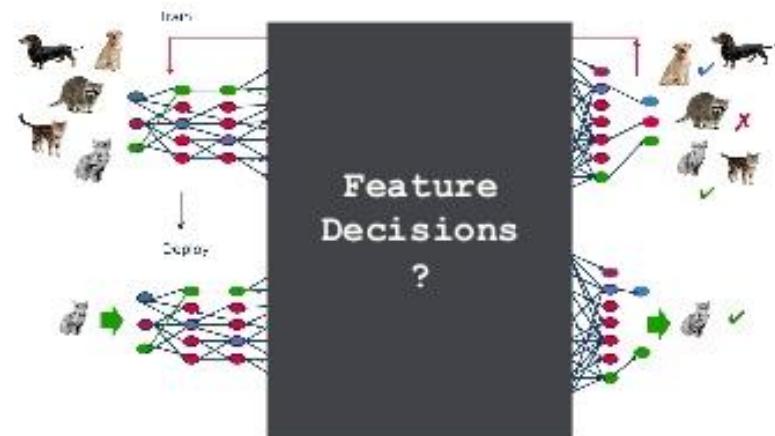
Índice:

1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
- 7. Las críticas al Deep Learning: Ética de la IA**
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

7. Las críticas al Deep Learning. Ética de la IA

Deep Learning Features

- Advantage:
 - Features do not have to be predetermined
- Disadvantage:
 - Decisions are a black box



Tema 6: Deep Learning

Índice:

1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
- 8. Herramientas Prácticas y Configuraciones**
9. Bibliografía

8. Herramientas Prácticas y Configuraciones

• TensorFlow

- Es una biblioteca de código abierto para desarrollar modelos de aprendizaje automático, de Google Brain, bien documentada y soportada
- Es una plataforma integral, válida tanto para principiantes como para expertos, pues ofrece múltiples capas de abstracción.
- Multiplataforma, pero internamente los cálculos los realiza mediante binarios muy eficientes implementados en C++ y CUDA
- Keras es una API o biblioteca de Python, de alto nivel, para implementar RNAs, creada por François Chollet (ingeniero de Google), código abierto bajo licencia permisiva MIT.
- Actualmente está incluido en TensorFlow.
- Curva de aprendizaje suave, muy difundida (la segunda tras TensorFlow).
- Se puede usar como plataforma de prototipado



Comparadas:

- TensorFlow es más flexible y potente que Keras, pero más compleja de aprender y manejar, incluso empleada en alto nivel.
- Se suele considerar a Keras como preferible para conjuntos menores, por su sencillez, expresividad y menor eficiencia.
- TensorFlow actualmente proporciona una API funcional de Keras, por tanto, se ejecuta sobre TensorFlow, CNTK y Theano como una envoltura de estos *backend*.
Microsoft Cognitive Toolkit
↑
- La comunidad de TensorFlow es mucho mayor que el de Keras.
- Keras:
 - Se ejecuta en CPU y GPU
 - Fácil de usar, minimizando lo que tiene que hacer el usuario, y con respuestas sobre errores claras
 - Modular: los modelos se construyen por bloques configurables
 - Fácil de ampliar: también permite crear nuevos bloques personalizados para investigación



Instalación de un entorno de trabajo con TensorFlow y Keras :

- Instalar Keras: visitar página web y seguir sus instrucciones (https://keras.io/getting_started/)
- Como se indica allí también, previamente se debe instalar TensorFlow (<https://www.tensorflow.org/install>)

- Los modelos sencillos se pueden probar sin tener GPU, pero a medida que se avance, estos pueden tardar mucho tiempo en entrenamiento
- Otra opción es utilizar el entorno Google Colab, donde se emplean GPUs y TPUs remotas: es un entorno en la nube de Google, y se usa desde un navegador web, con los *Jupyter notebook* en Python. Desde allí se puede usar Keras, TensorFlow y PyTorch.



- **AAD Entregable en Moodle:**
Comprendión práctica de cómo entrenar un modelo de DL para una tarea robótica sencilla

1. Problema a resolver: **Distinguir carretera.**

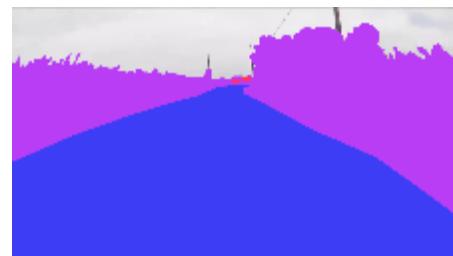


2. Materiales:

- Dataset: [Kaggle – Road segmentation dataset](#)
- Herramienta: Google colab + Tensorflow/Keras
- Modelo: Segmentación



3. Resultado Final: **Segmentar la imagen propuesta.**



Tema 6: Deep Learning

Índice:

1. Introducción
2. Definiciones y fundamentos
3. RNs Densamente conectadas
4. RNs Convolucionales
5. Falta de datos: Data Augmentation & Transfer Learning
6. Otros modelos de RNs Profundas
7. Las críticas al Deep Learning: Ética de la IA
8. Herramientas Prácticas y Configuraciones
9. Bibliografía

9. Bibliografía

- Deep Learning, Introducción práctica con Keras (Primera Parte), Jordi Torres, Colección Watch This Space, Lulu Press, Inc. 2018
- Deep Learning on Windows. Building Deep Leasrning Computer Vision Systems on Microsoft Windows. Thimira Amaratunga. Apress 2021
- Deep Learning, Introducción práctica con Keras (Segunda Parte), Jordi Torres, Colección Watch This Space, Lulu Press, Inc. 2019
- Deep Learning, Francisco Herrera, Máster en Ciencia de Datos e Ingeniería de Computadores, 2017
(<https://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/SIGE/Tema06-Deep-learning.pdf>)

- Conferencia Sergio Guadarrama: *Campus Experts Summit: TensorFlow, Machine Learning for Everyone with Sergio Guadarrama*, Google for Startups 2017
<https://www.youtube.com/watch?reload=9&v=dCvIHsnWJKg>
- Deep Learning, Ian Goodfellow and Yoshua Bengio and Aaron Courville, MIT Press Book <https://www.deeplearningbook.org/> (libre como HTML)
- Wikipedia...

END