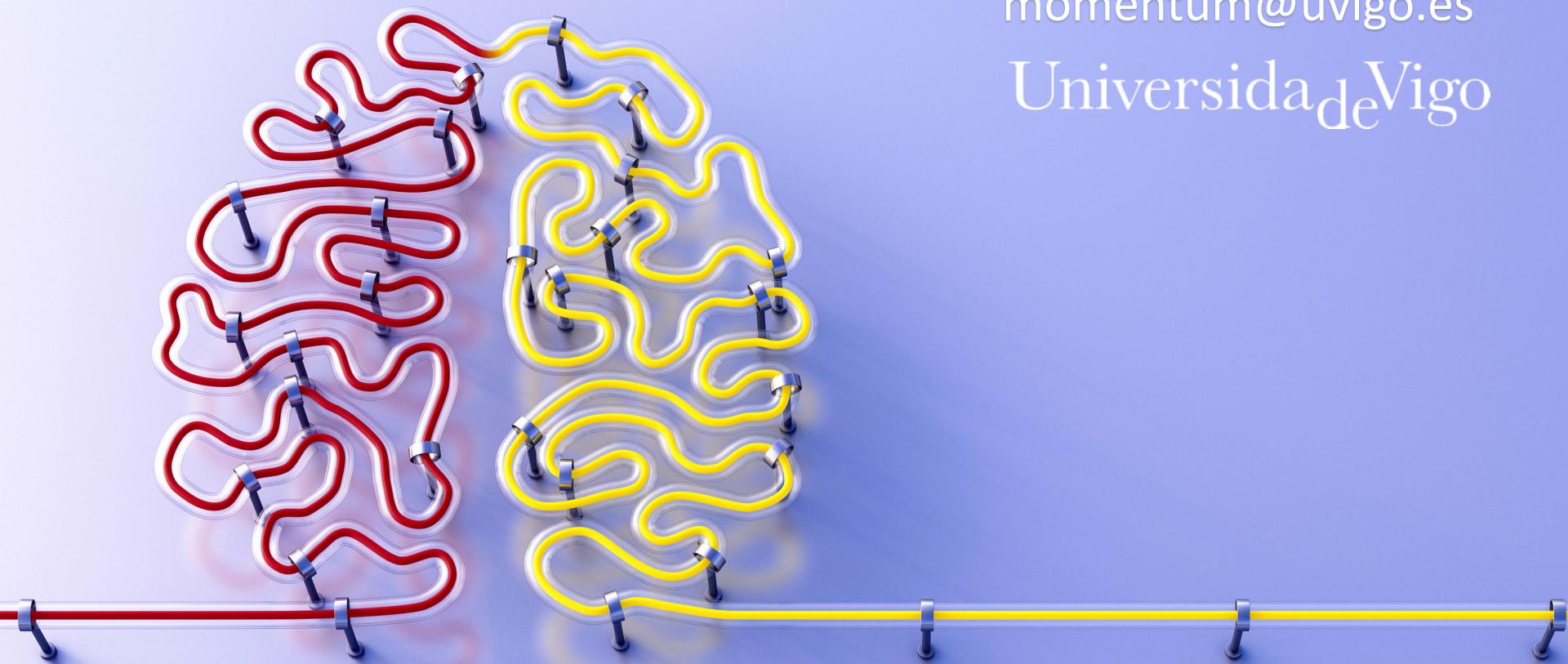


# Deep Learning

Amador Rodríguez Diéguez  
[momentum@uvigo.es](mailto:momentum@uvigo.es)

Universida<sub>d</sub>eVigo



# Índice del curso

## Secciones:

- Introducción y conceptos básicos .
- Preparación de datos
- Redes neuronales y componentes
- Entrenamiento
- Hiperparámetros
- Regularización
- Arquitecturas
  - ✓ CNN
  - ✓ RNN
  - ✓ ...

# ARTIFICIAL INTELLIGENCE

IS NOT NEW

## ARTIFICIAL INTELLIGENCE

Any technique which enables computers to mimic human behavior



1950's

1960's

1970's

1980's

## MACHINE LEARNING

AI techniques that give computers the ability to learn without being explicitly programmed to do so



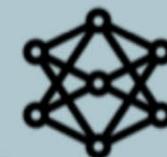
1990's

2000's

2010s

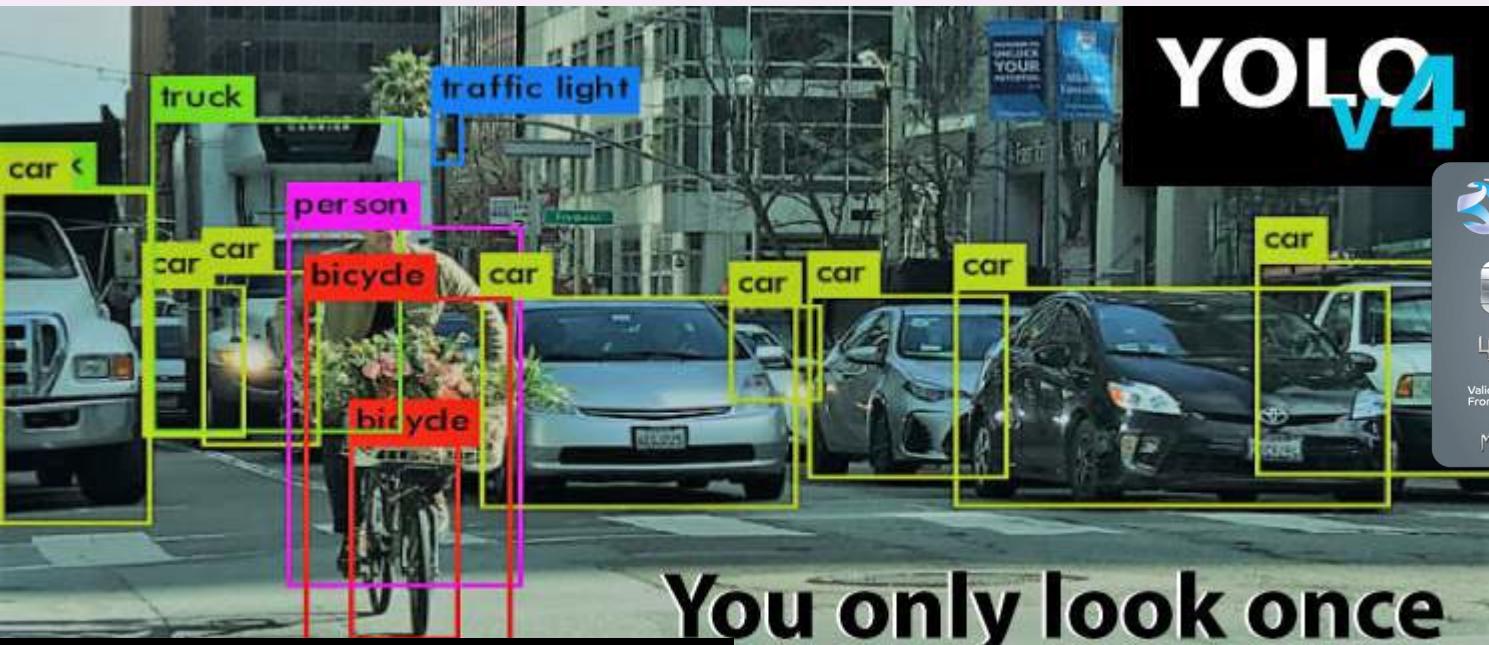
## DEEP LEARNING

A subset of ML which make the computation of multi-layer neural networks feasible



# 20 DEEP LEARNING Applications

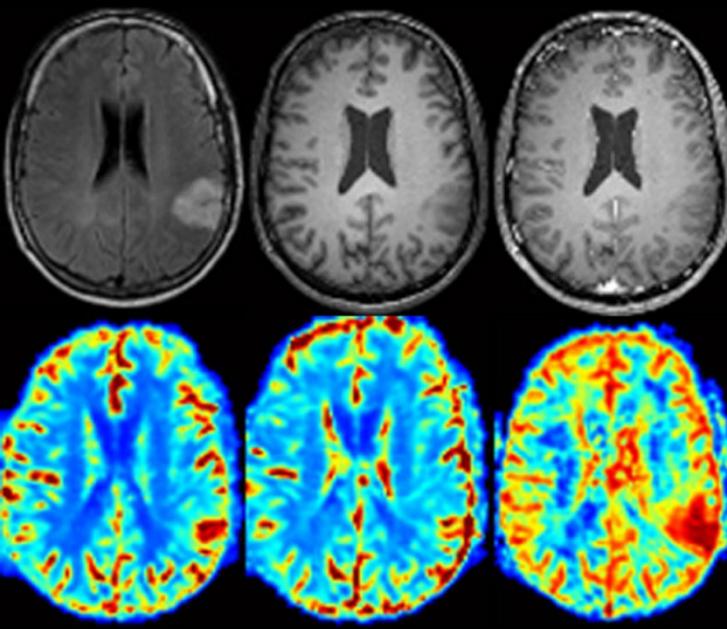




YOLO  
v4



You only look once



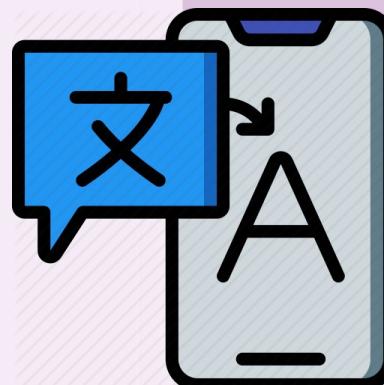
## COMPORTAMIENTO DOW JONES

\*Comportamiento en un mes

29.170,49  
07/02/2020

Cifras en puntos

23.835,96  
09/03/2020



## TYPES OF MACHINE LEARNING

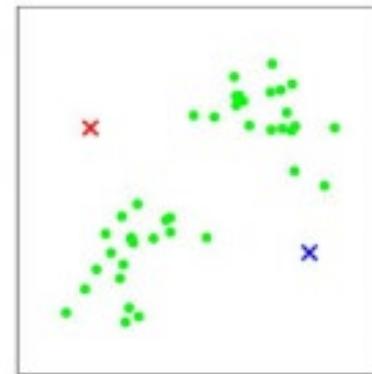


# Ejemplo

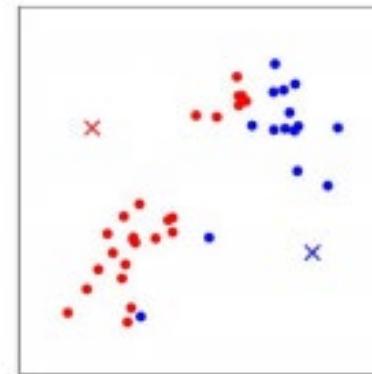
El método *K-means* es un ejemplo de aprendizaje automático (*machine learning*) que no está en la categoría de Deep Learning:



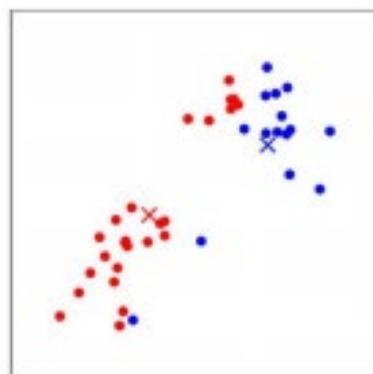
(a)



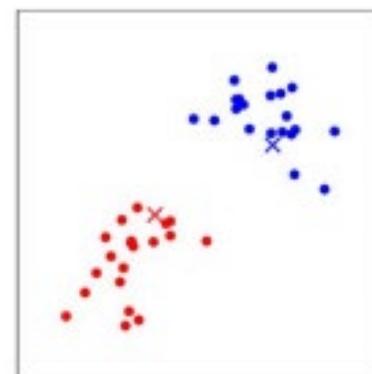
(b)



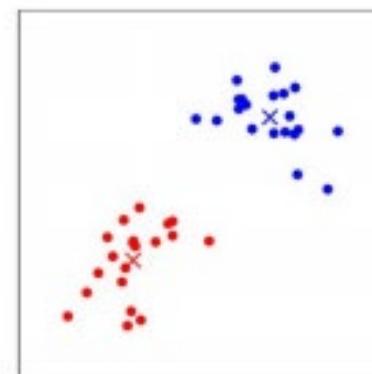
(c)



(d)

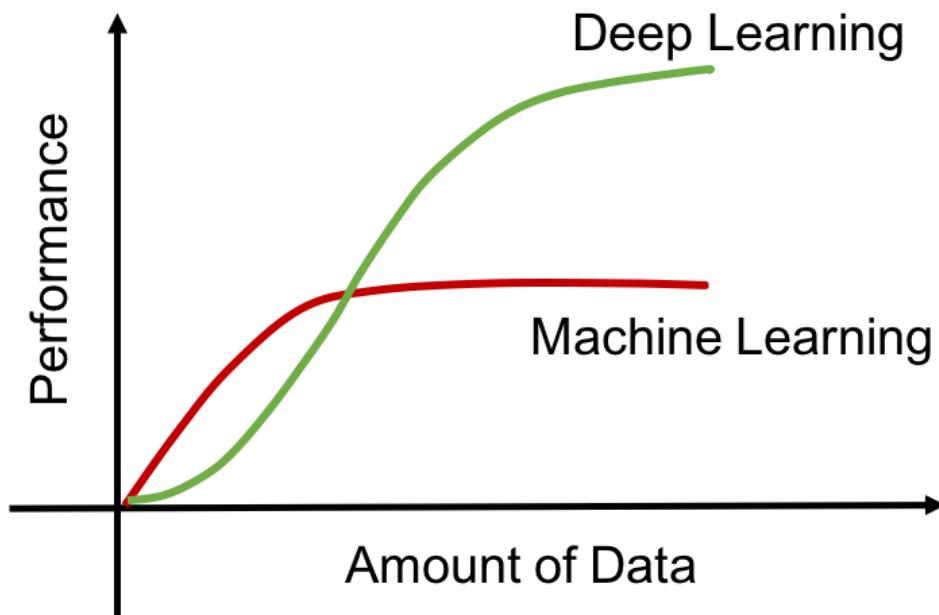


(e)



(f)

# Deep Learning



# ¿Por qué ahora?

## Big data

muchísima información  
disponible y gran capacidad  
de almacenamiento



Google Trends



IMAGENET

## Hardware

GPUs y TPUs  
Disponibles también  
en la nube.



GPU



TPU

## Software

Nuevos modelos y  
técnicas. Frameworks  
potentes y maduros.

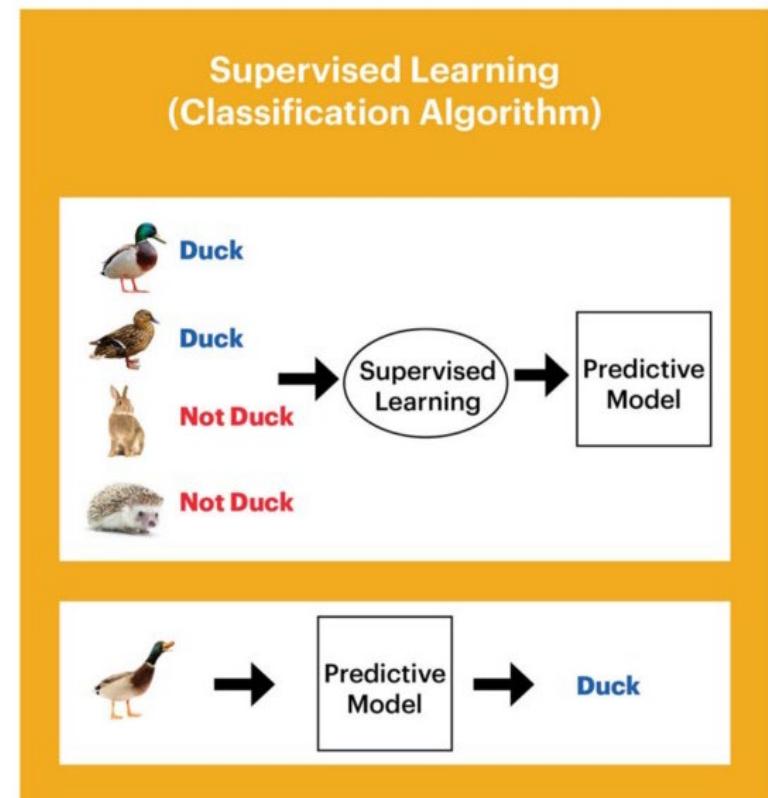


# Aprendizaje supervisado

En el **aprendizaje supervisado**, los algoritmos trabajan con datos “etiquetados”, intentando encontrar una función que, dadas las variables de entrada, les asigne la etiqueta de salida adecuada. El algoritmo se entrena con un histórico de datos (*dataset*) y así aprende a asignar la etiqueta de salida adecuada a un nuevo valor, es decir, predice el valor de salida.

Por ejemplo, un detector de spam, analiza el histórico de mensajes, viendo qué función puede representar, según los parámetros de entrada que se definan (el remitente, si el destinatario es individual o parte de una lista, si el asunto contiene determinados términos, etc), la asignación de la etiqueta “spam” o “no es spam”.

Una vez definida esta función, al introducir un nuevo mensaje no etiquetado, el algoritmo debería ser capaz de asignarle la etiqueta correcta.



# Aprendizaje supervisado: clasificación y regresión

El aprendizaje supervisado se suele usar en problemas de **clasificación** o de **regresión**.

El objetivo de la **clasificación** (también llamada *regresión logística*) es, asignar la instancia cuya información usamos como entrada a la red, una categoría de entre varias predefinidas. La salida del algoritmo suele ser un valor de probabilidad de que la entrada corresponda a cada una de las clases disponibles.

Un tipo de clasificación especialmente interesante es la **clasificación binaria**, donde sólo existen dos categorías.

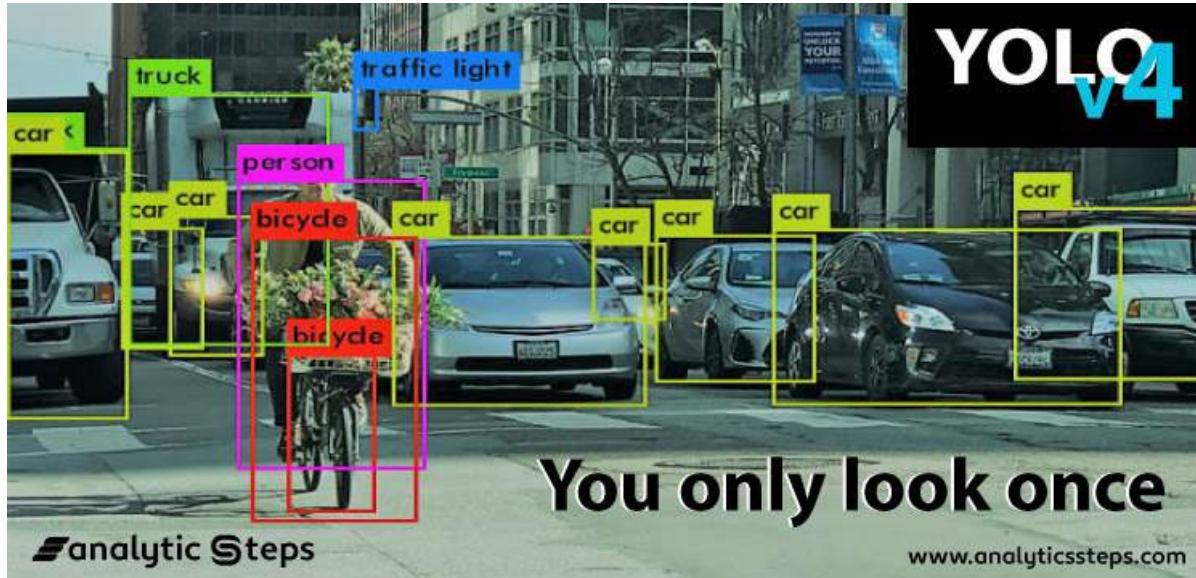
Ejemplos de clasificación:

- ¿Comprará el cliente este producto? [sí, no]
- ¿Subirá el IBEX mañana? [sí, no]
- Elemento detectado por una cámara: peatón, coche, señal de tráfico, semáforo...

Por otra parte, el objetivo de la **regresión** será un valor numérico. Por ejemplo:

- Predecir por cuánto se va a vender una propiedad inmobiliaria.
- Estimar cuánto tiempo va a tardar un vehículo en llegar a su destino
- Estimar cuántos productos se van a vender.

# Aprendizaje supervisado



Bicicleta:	0.04
Persona:	0.01
Coche:	0.83
Semáforo	0.02
Camión:	0.10

# Aprendizaje no supervisado

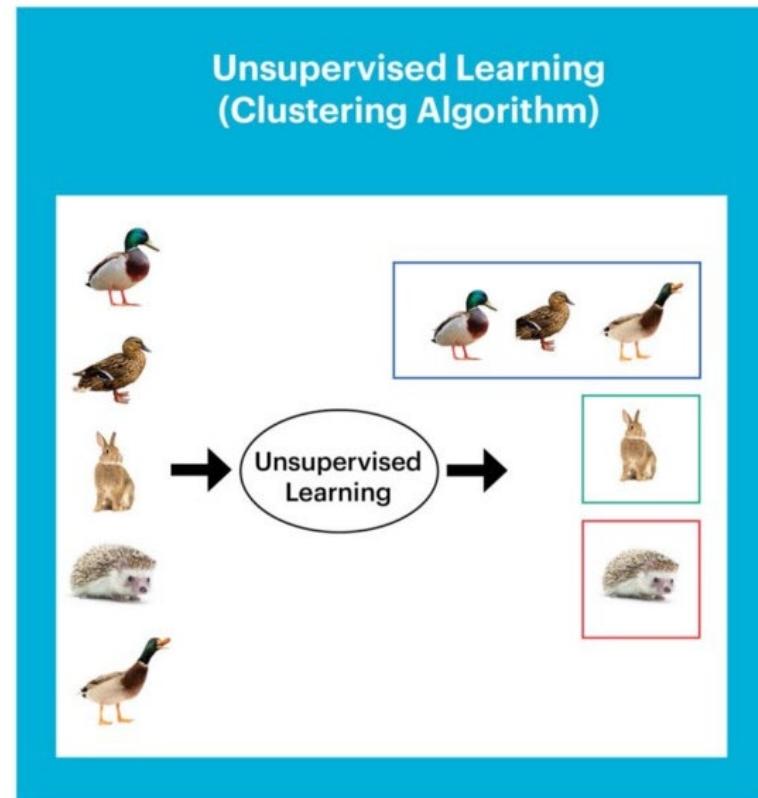
El **aprendizaje no supervisado** tiene lugar cuando no se dispone de datos etiquetados para el entrenamiento. Sólo conocemos los datos de entrada, pero no existen datos de salida que correspondan a una determinada entrada. Por tanto, sólo podemos describir la estructura de los datos, para intentar encontrar algún tipo de organización que simplifique el análisis. Por ello, tienen un carácter exploratorio.

Por ejemplo, las tareas de ***clustering***, buscan agrupamientos basados en similitudes, pero nada garantiza que éstas tengan algún significado o utilidad. En ocasiones, al explorar los datos sin un objetivo definido, se pueden encontrar correlaciones poco prácticas.

El aprendizaje no supervisado se suele usar en:

- Problemas de ***clustering***.
- Agrupamientos de co-ocurrencias.
- Perfilado (***profiling***).

Si embargo, los problemas que implican tareas de encontrar similitudes o reducción de datos, pueden ser supervisados o no.

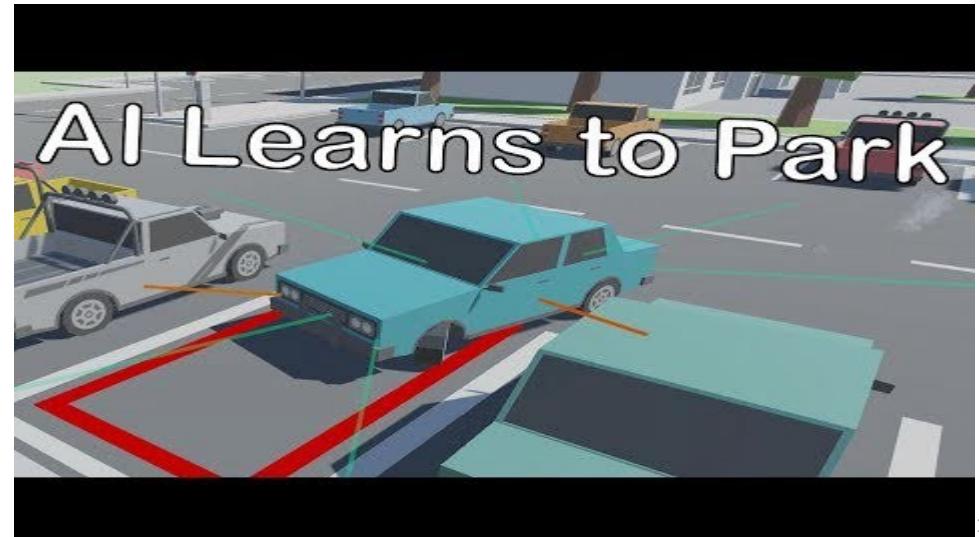
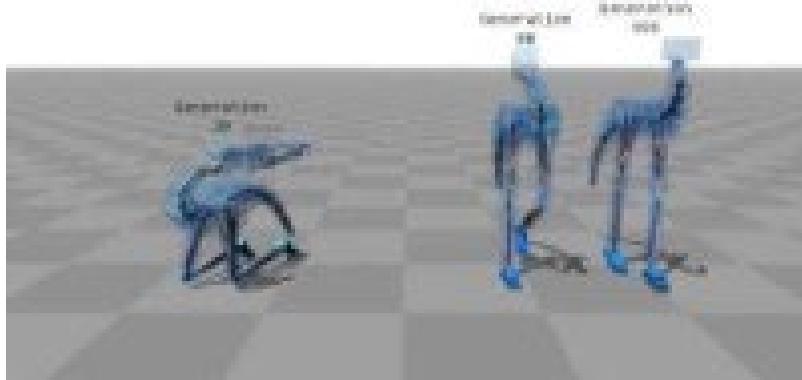


# Aprendizaje por refuerzo

Otro de los problemas habituales de este tipo son los que se resuelven mediante el **aprendizaje por refuerzo**.

En estos problemas no se conoce la solución y la forma de entrenar el modelo es mediante la introducción de recompensas **positivas o negativas** en función de los resultados.

La utilizad de este enfoque se encuentra en problemas para los que se puede asignar una recompensa o penalizaciones, pero no se conoce cómo llegar al resultado. El mejor proceso para llegar al resultado es lo que descubre el algoritmo, ya que conoce las recompensa o la penalización asociada a cada una de sus posibles acciones y resultados.

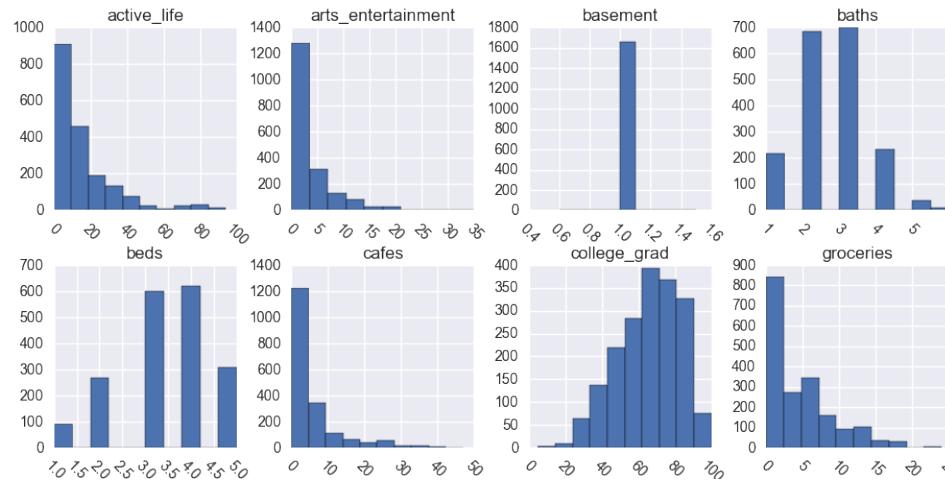


# Preparación de los datos

- Es imprescindible asegurarse que la información con la que entrenamos la red sea representativa de lo que se encontrará el sistema en producción.
- Principio básico: *Garbage in, garbage out.*
- Fases:
  - **Análisis exploratorio de los datos:**
    - Permite conocer la información disponible de cara a implementar las fases de limpieza e ingeniería de datos.
    - Es un proceso breve.
  - **Limpieza de la información (*data cleaning*):**
    - Es un proceso crítico y que consume gran parte de la duración del proyecto.
    - Incluye la selección de variables o características (*feature selection*).
  - **Ingeniería de datos (*feature engineering*)**
    - Consiste en crear nuevas variables de entrada a partir de las ya existentes.
    - Es la fase que más influye en el rendimiento del modelo por que permite que los algoritmos se centren en la información clave e introducir información experta en el tema (médica, financiera, etc.)

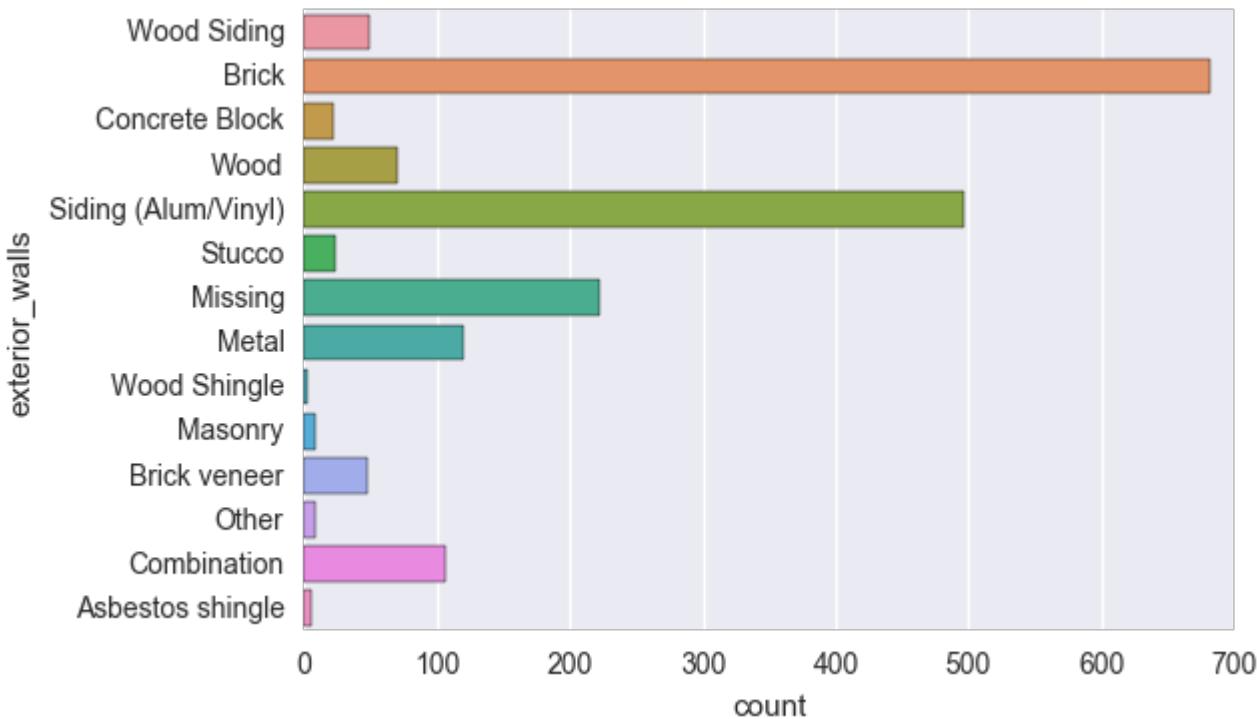
# Preparación de los datos: análisis exploratorio

- **Cuestiones básicas:** ¿de cuántas observaciones se dispone? ¿con cuántas variables (características) se trabajará? ¿de qué tipos de datos son las variables? ¿cuál es la variable objetivo? ¿tienen sentido los valores de las columnas (características)? ¿están esos valores en la escala adecuada? ¿es previsible que los valores ausentes sean un problema?
- **Visualizar gráficamente las distribuciones de las características numéricas** (histogramas). Esto permite detectar distribuciones inesperadas, valores atípicos (*outliers*) que no tienen sentido, características que deberían ser binarias, errores potenciales de medida, etc.



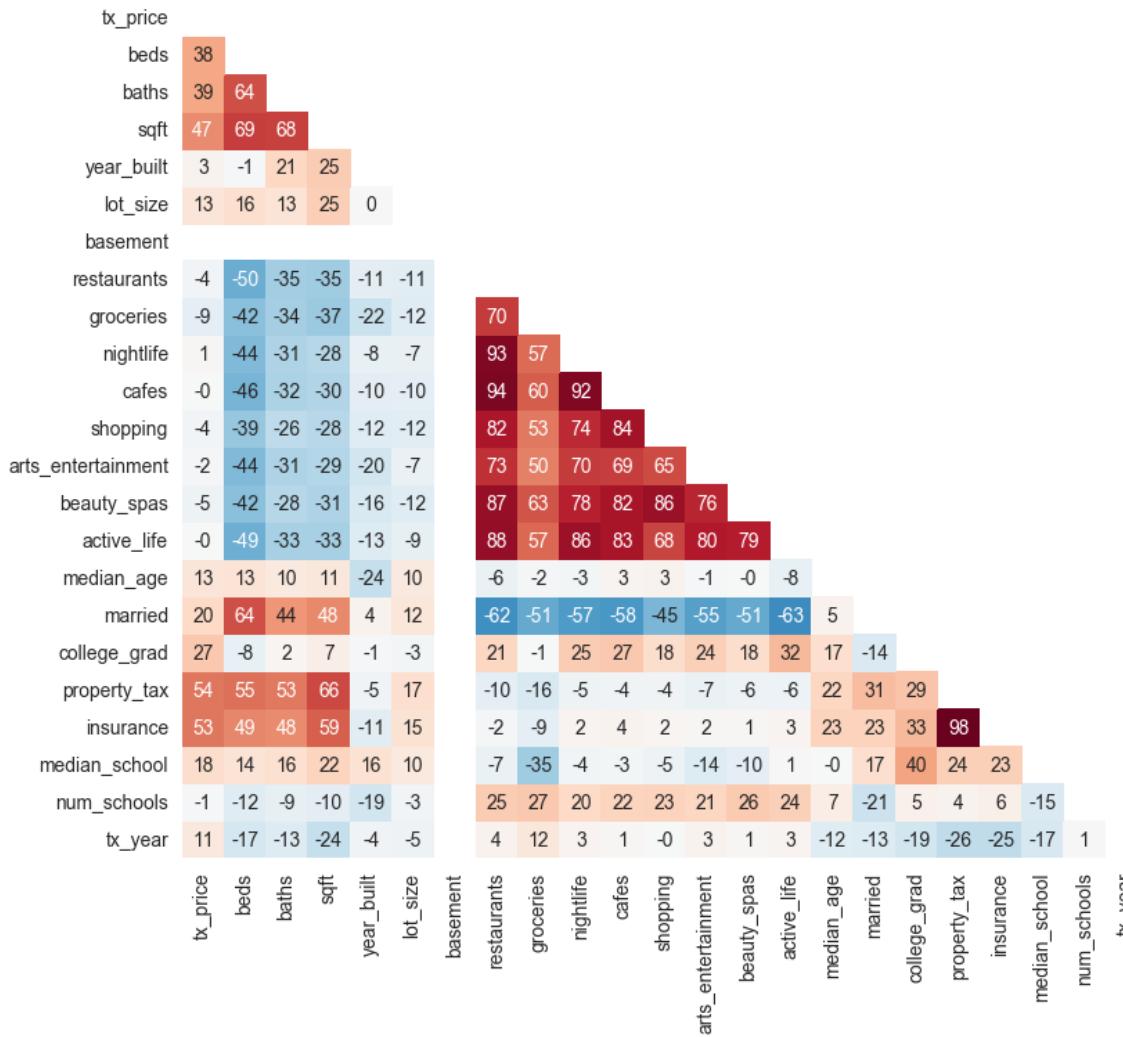
# Preparación de los datos: análisis exploratorio

- **Visualización gráfica de las distribuciones de las características categóricas** (histogramas). Permite detectar clases dispersas (con pocas observaciones), las cuales no suelen tener mucha influencia en el sistema, pero en algunos casos favorece el sobreajuste.



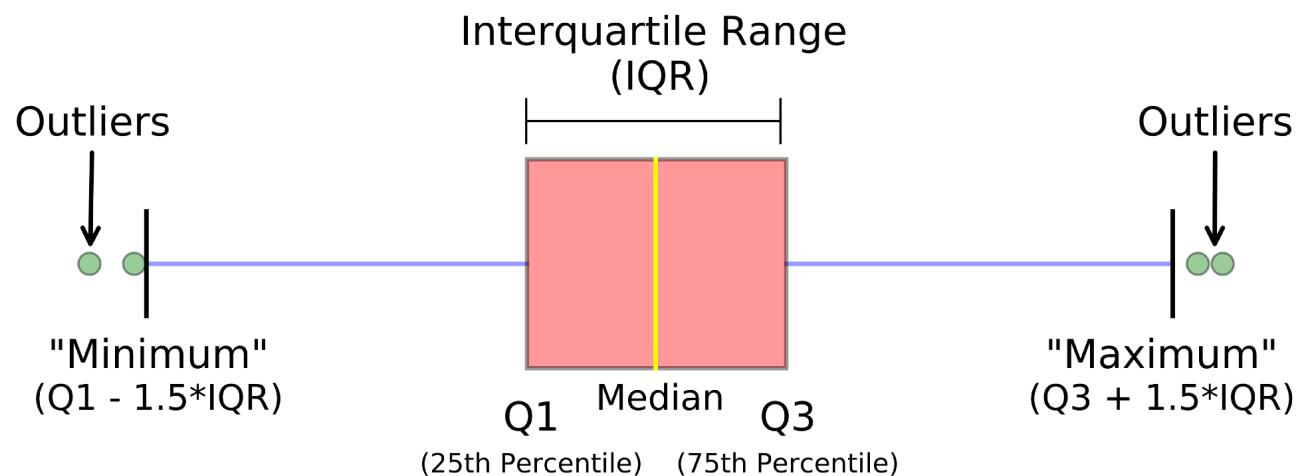
# Preparación de los datos: análisis exploratorio

- **Estudio de correlaciones**, por medio de mapas de calor (*heatmaps*). Se suele buscar las variables con mayor correlación entre ellas, así como las más correladas con el objetivo.



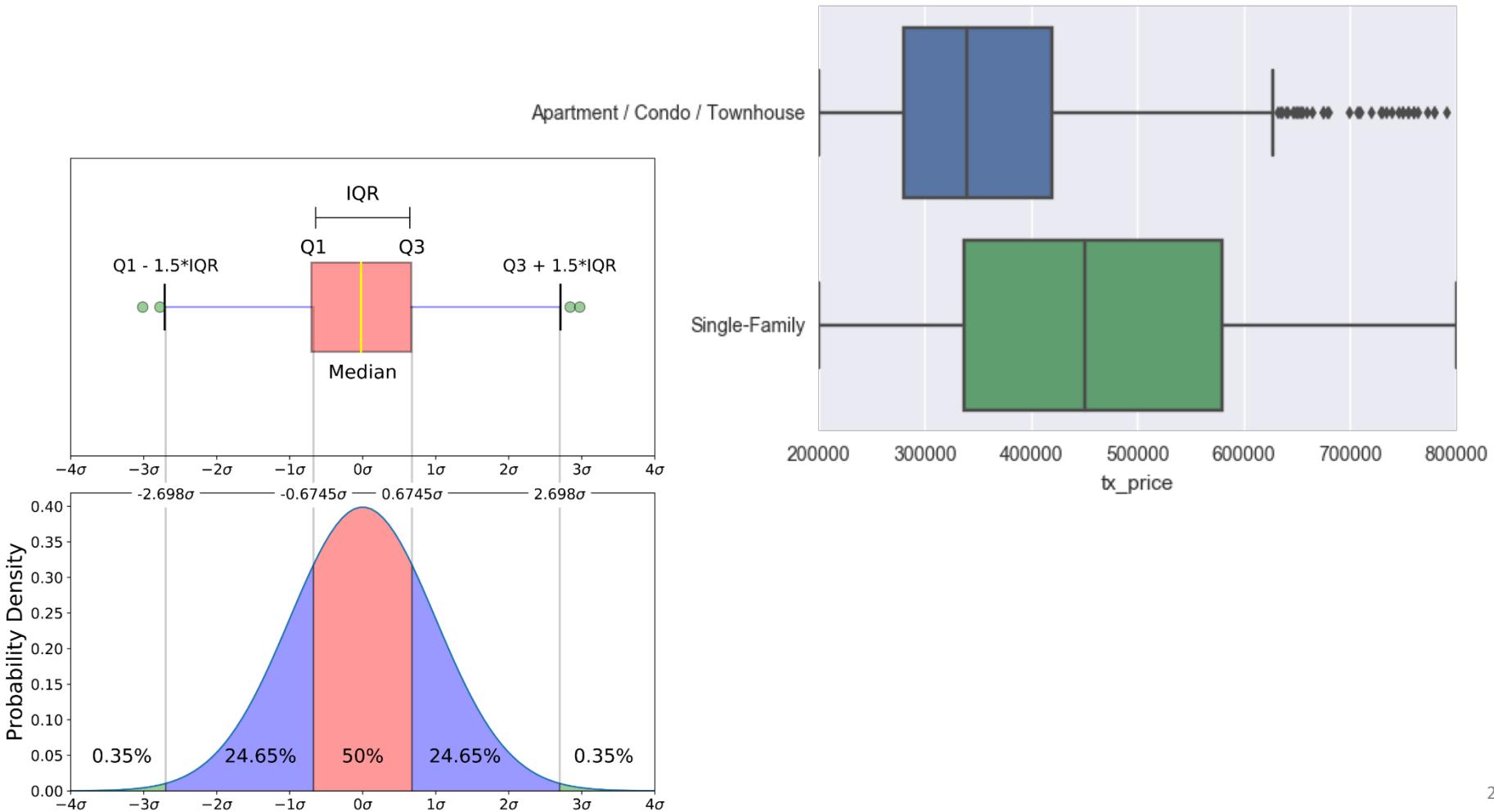
# Preparación de los datos: análisis exploratorio

- **Visualización de segmentaciones:** permite observar la relación entre variables categóricas y numéricas. Se usan los gráficos de cajas (*box plots*):
  - Mediana.
  - Los cuartiles (Q1 a Q4) definidos por los percentiles del 25%, 50%, 75% y 100%.
  - Rango intercuantílico (*interquartile range - IQR*): entre los percentiles 25% y 75%.
  - Máximos y mínimos calculados con respecto al IQR.



# Preparación de los datos: análisis exploratorio

En el ejemplo se pueden observar las medidas típicas: medianas, máximos, mínimos, etc. Los valores máximos y mínimos (200K y 800K) sugieren que los valores están truncados (o redondeados), lo que puede influir en la generalización del modelo.



# Preparación de los datos: limpieza (*data cleaning*)

## 1. Eliminación de observaciones:

- Observaciones duplicadas: típicamente proviene de la fusión de distintos *datasets* o *web scraping*.
- Información irrelevante: por ejemplo, observaciones de casas unifamiliares, cuando el problema intenta predecir el precio de viviendas en edificios.

## 2. Filtrado de valores atípicos (*outliers*):

- Se deben eliminar los que se sepa (o sea muy sospechoso) que es una lectura errónea o que no aporta información al sistema.

## 3. Arreglar errores estructurales:

- Si la información proviene de distintas fuentes, puede tener formatos distintos para los mismos atributos.
- Por ejemplo, “23.53 €” o “23.53 Eur”; o usar nombres de países en distinto formato, como por ejemplo “Estados Unidos” o “EEUU”; o tener una columna “cliente” con nombre y apellidos, y otros registros, nombre y apellidos en distintas columnas.
- Eliminar caracteres que no aportan información: cambios de línea, #, &, espacios en blanco al principio o al final, etc.

# Preparación de los datos: limpieza (*data cleaning*)

## 4. Reducir la información:

- Eliminar columnas (*atributo* o *característica*) que no aportan información (*attribute sampling*), que permite reducir complejidad. Por ejemplo, en algunos *datasets* aparecen columnas que sólo contienen un valor.
- Eliminar filas (*observaciones*) en la que falta información, o ésta es errónea o los datos son poco representativos para las predicciones. Se conoce como *record sampling*.
- Agregación: por ejemplo, puede ser interesante sustituir los datos de ventas diarias por ventas semanales. De este modo se reduce el número de filas.
- Eliminar demasiadas filas (por valores ausentes, por ejemplo) puede introducir sesgos o empeorar el *dataset*.

## 5. Gestión de datos ausentes:

- Eliminación.
- Asignación de valores numéricos (por ejemplo, 0) o por valores categóricos (por ejemplo, el texto “n/a” o “no disponible”)
- Si son columnas numéricas, por promedios, medianas, valores interpolados, etc.
- En columnas categóricas, por el valor más frecuente.
- Si se asigna un valor, se puede crear otra columna (variable) que indique esta situación.

# Preparación de los datos: limpieza (*data cleaning*)

## 6. Eliminación de características no utilizadas:

- Identificadores, variables que no van a estar disponibles en producción.

## 7. Selección de características (*feature selection*):

- Concepto clave con una gran influencia en el sistema.

- Reduce el sobreajuste.
- Mejora las predicciones.
- Reduce el tiempo de entrenamiento.

- Técnicas para evaluar la necesidad de una variable:

- **Univariate selection:**

- Tests estadísticos para seleccionar las características con mayor influencia en la variable de salida.
    - Clase *selectKBest* de *scikit-learn*.

- **Feature importance:** se aplican clasificaciones basadas en árboles.

- **Mapas de calor (heatmaps).**

# Preparación de los datos: ingeniería de datos (*data engineering*)

## 1. Cambio de formato:

- Cuando hay estructuras anidadas (por ejemplo, XML o JSON) convertirlas a estructuras tabulares.
- Cambiar el tipo de dato de las columnas cuando el *dataset* es muy grande: ahorra memoria.
- En algunos casos, los sistemas necesitan entradas numéricas en lugar de categóricas: “Sí” → 1, “No” → 0.

## 2. Descomposición de datos complejos en múltiples partes y generación de nuevos datos (*data enrichment*):

- Por ejemplo, usando ventas diarias en lugar de semanales.
- Convertir *timesteps* en formatos de fecha y hora.
- Generando el día de la semana a partir de la fecha, o la duración de un evento en a partir de fechas de inicio y fin.

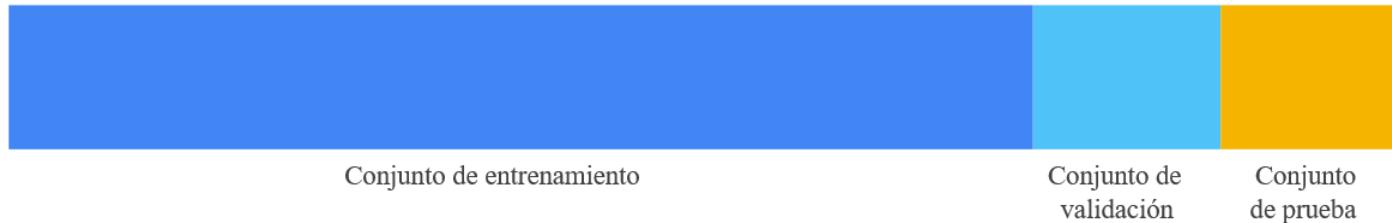
## 3. Reescalado de los datos: normalización (*ya visto*)

## 4. Discretización: agrupando rangos de valores en grupos.

- Por ejemplo, agrupando el nivel adquisitivo de los clientes o su edad en tramos.

# Datos de entrenamiento, validación y prueba

El paso final tras la preparación de los datos consiste en dividirlos en tres conjuntos: **entrenamiento (*training*)**, **validación (*validation*)** y **prueba (*test*)**.



- Conjunto de **entrenamiento**: Para entrenar pesos e hiperparámetros
- Conjunto de **validación**: Para comprobar con datos no conocidos. Permite detectar sobreajuste.
- Conjunto de **prueba**: Permite comparar modelos distintos, así como tener una medida con datos que no han sido usados previamente.

Dado que en Deep Learning se trabaja con *datasets* muy grandes, típicamente los conjuntos de prueba y validación estarán entre el 1% y el 5% del total de datos.

# Datos de entrenamiento, validación y prueba

Algunos *frameworks* tienen funciones que generan los tres conjuntos de datos de forma **aleatoria**, pero en muchos casos esto no cumple los requisitos necesarios:

- Por ejemplo cuando se trabaja con series temporales.
- También es necesario considerar si los datos que el modelo va a usar en **producción son cualitativamente distintos de los de entrenamiento**.

Por ejemplo, si entrenamos un sistema de reconocimiento de emociones a partir de una imagen del rostro. Si usamos un número reducido de personas para el entrenamiento, en muchos casos ocurrirá que en producción se encontrará con caras de personas desconocidas.

Por ello, se debe reservar para los conjuntos de validación y prueba imágenes de personas que no hayan participado en el entrenamiento.

En general, es importante que los tres conjuntos comparten las mismas propiedades estadísticas.

# Datos de entrenamiento, validación y prueba

**Error de sesgo (bias)**: error de predicción alto en los datos de entrenamiento.

**Error de varianza (variance)**: diferencia entre los errores en el conjunto de entrenamiento y en el de validación.

El entrenamiento es cíclico hasta que los dos errores anteriores son aceptables.

Ejemplo de situaciones :

- **High variance**. Error de entrenamiento bajo (p. ej. 1%) y error de validación alto (p. ej. 10%): situación de sobreajuste (*overfitting*) y se necesita regularizar.
- **High Bias**. Error de entrenamiento y de validación altos y parecidos (p. ej. 10% y 11%, resp.): situación de subajuste (*underfitting*) y se necesitan más datos, o una red mayor o una arquitectura distinta).

# Perceptrón: propagación hacia adelante (forward propagation)

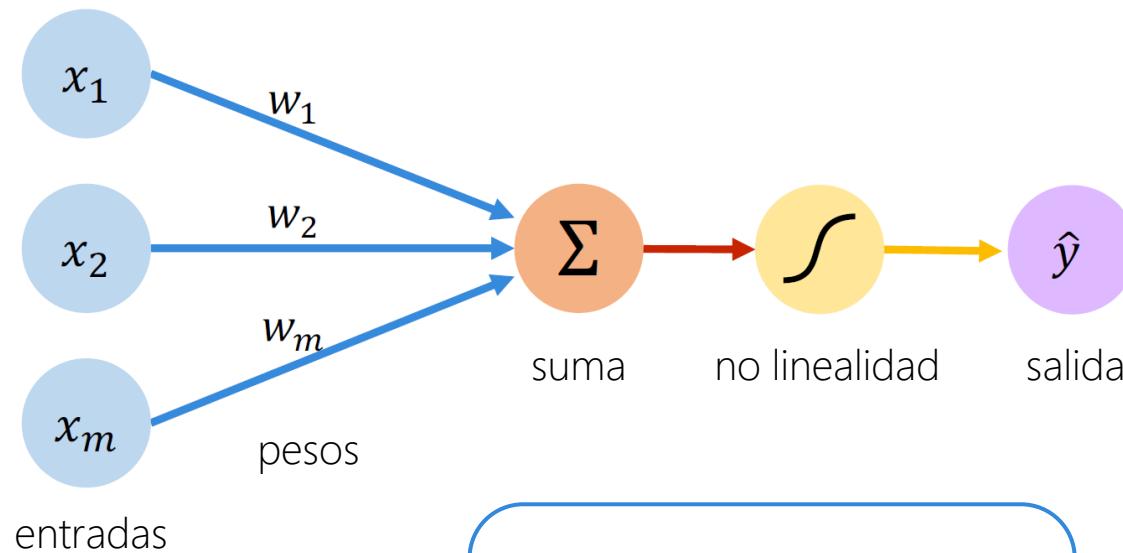


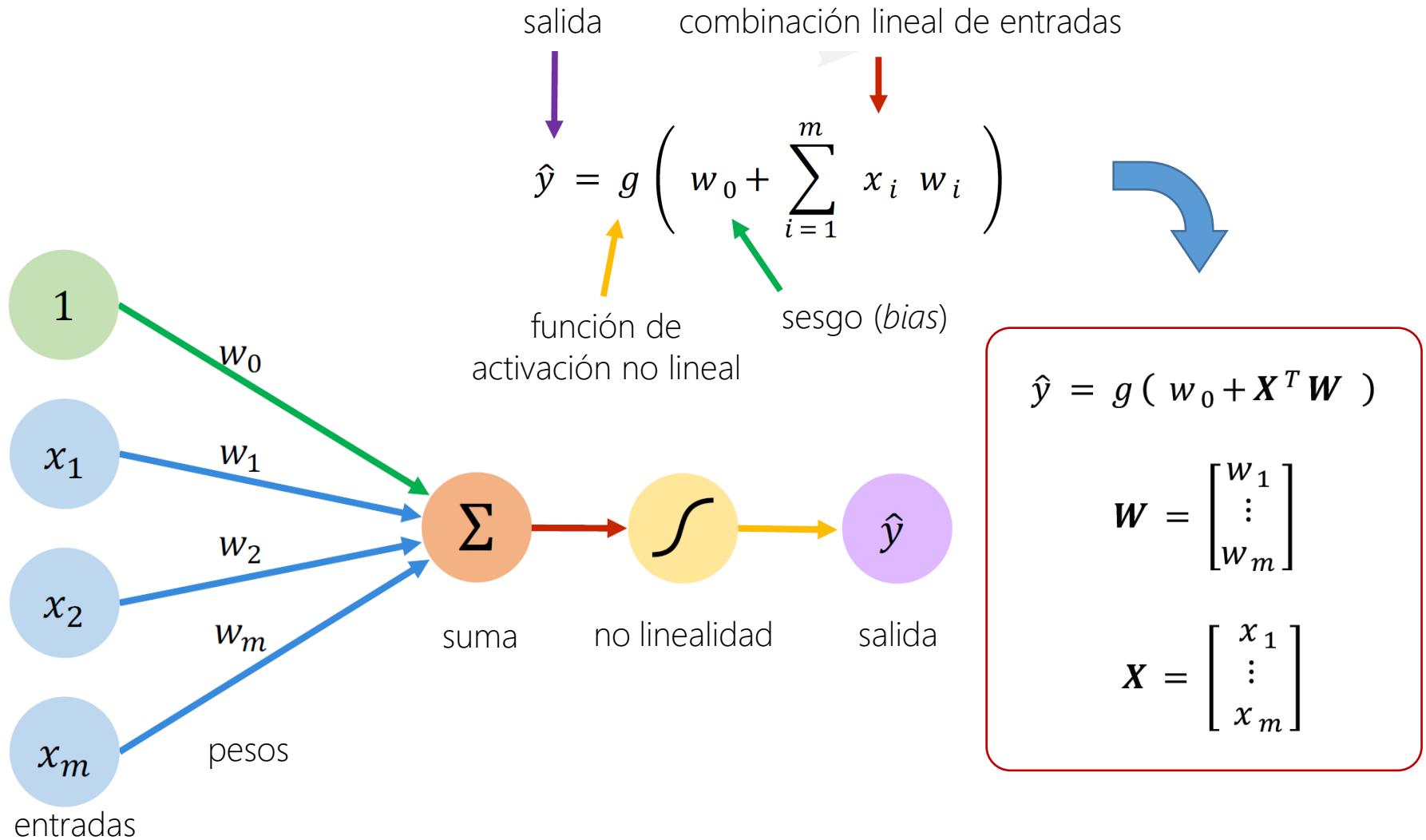
Diagram illustrating the forward propagation formula:

$$\hat{y} = g \left( \sum_{i=1}^m x_i w_i \right)$$

Legend:

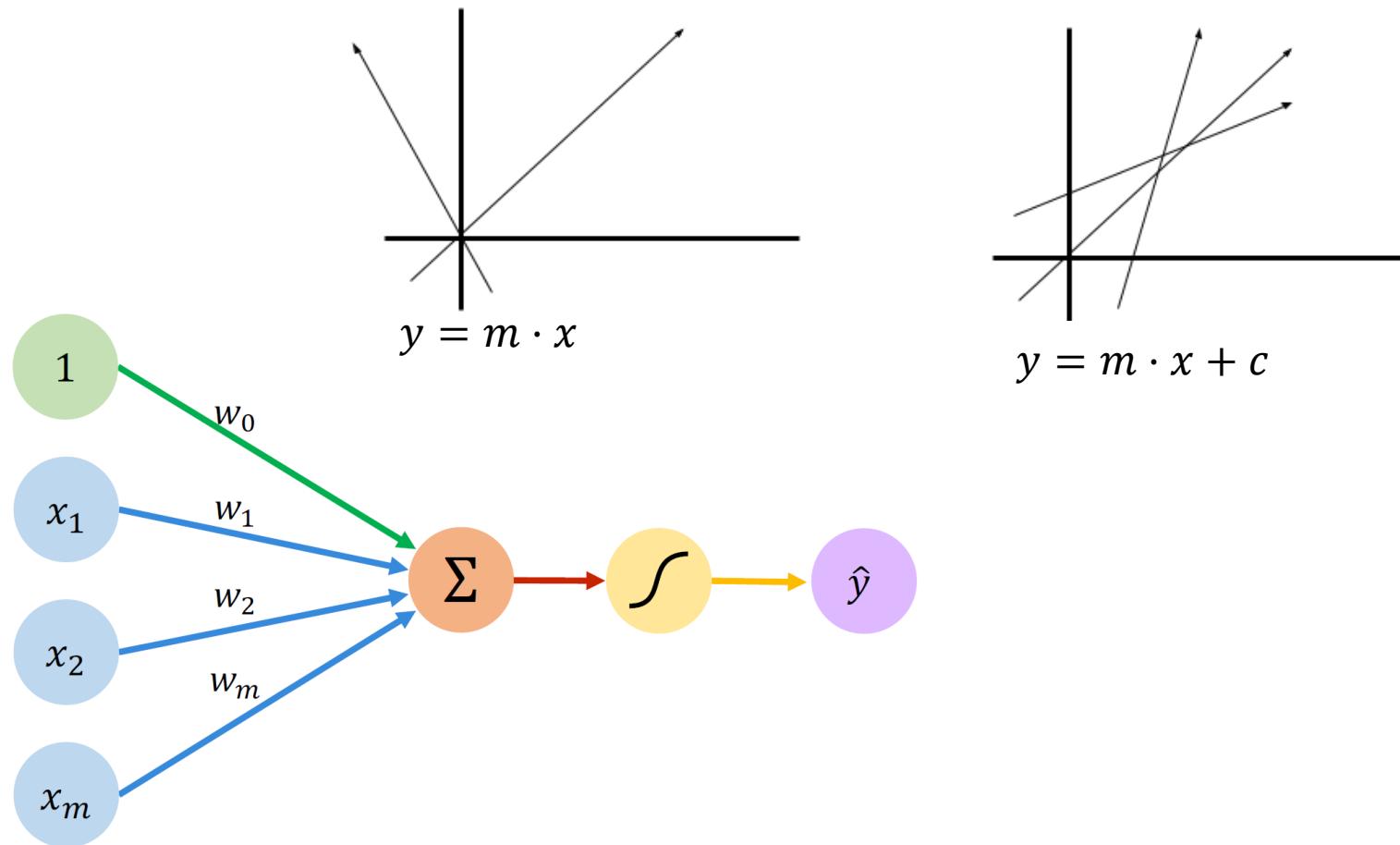
- salida (Output):  $\hat{y}$
- combinación lineal de entradas (Linear combination of inputs):  $\sum_{i=1}^m x_i w_i$
- función de activación no lineal (Non-linear activation function):  $g$

# Perceptrón: propagación hacia adelante con sesgo (*bias*)



# Sesgo (*bias*)

El sesgo (*bias*) hace que se puedan representar más valores, sin que el modelo pase forzosamente por el origen:

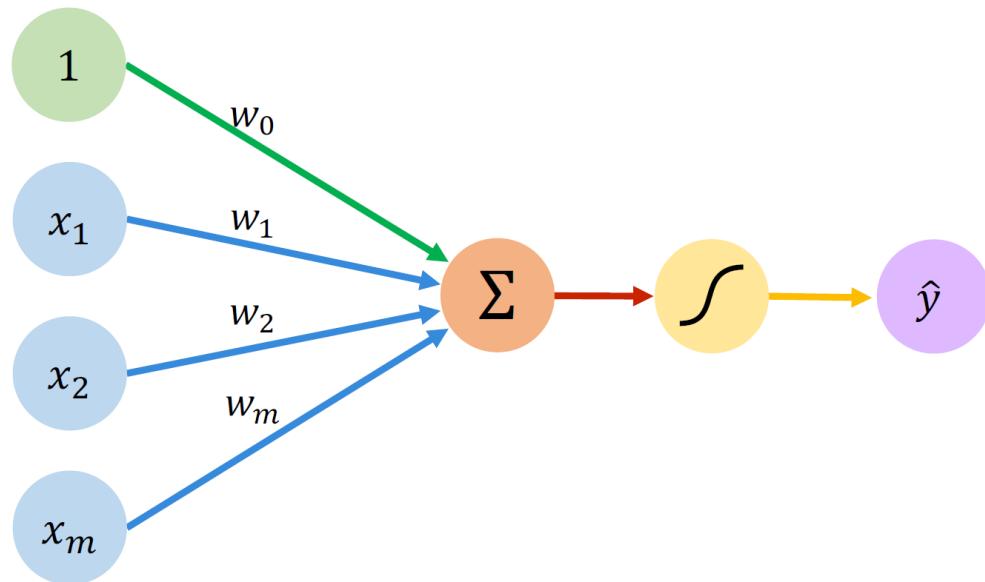


# Funciones de activación

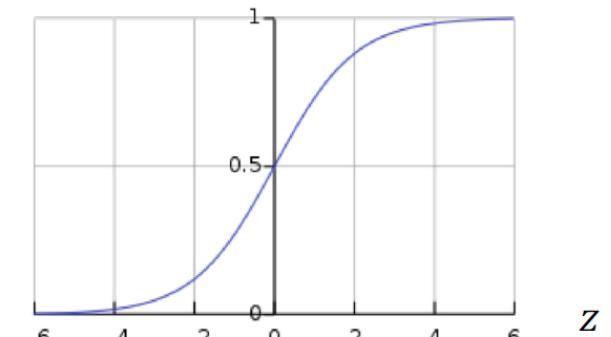
La función de activación decide si la neurona se activa (1) o no (0) en función de los valores de las entradas.

Las funciones de activación introducen alinealidad en la red y esto es lo que le permite a la red aprender y realizar tareas más complejas.

La red se comporta mejor si la función de activación cambia de 0 a 1 sin discontinuidades.



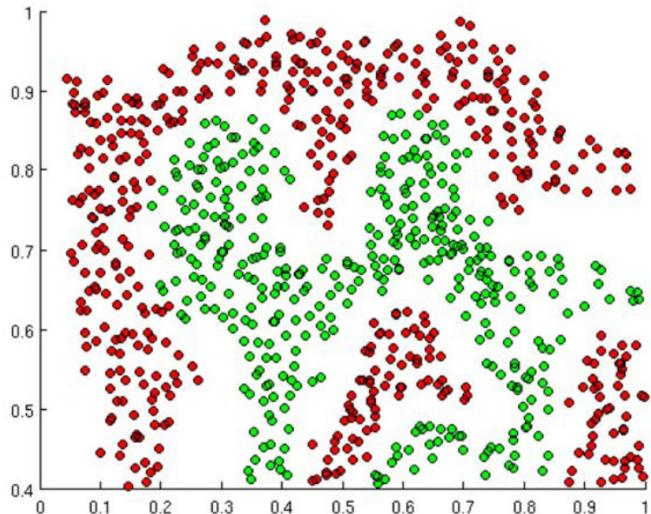
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



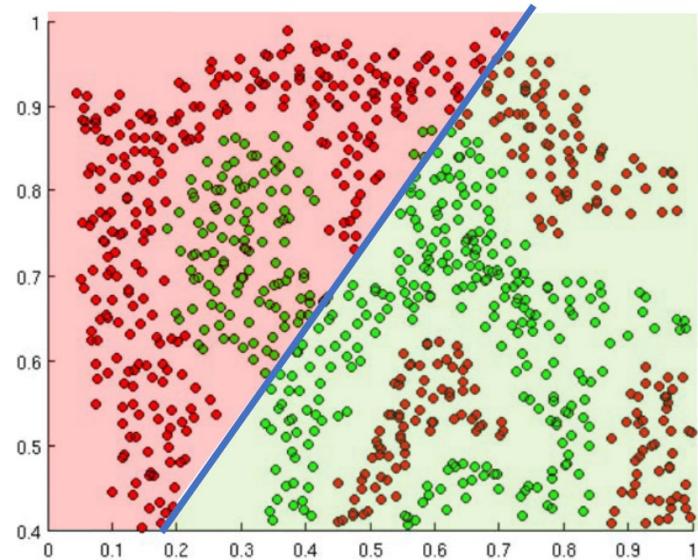
$$\hat{y} = g(w_0 + X^T W)$$

# Objetivo de las funciones de activación

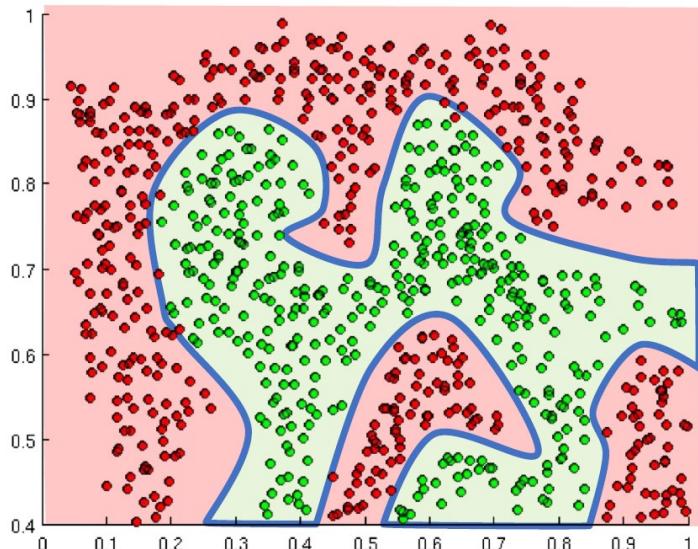
El objetivo de las funciones de activación es el de introducir no linealidades.



lineal



no lineal



*Ejemplo: cómo distinguir los puntos verdes de los rojos.*

# Funciones de activación

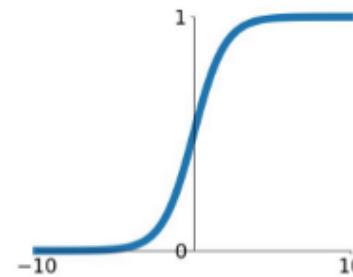
La función **sigmoide** presenta los siguientes problemas:

- El cálculo de exponentiales consume mucho tiempo.
- Se produce el problema de desvanecimiento del gradiente: hay momentos en los que el gradiente se hará tan pequeño haciendo que los pesos apenas cambien su valor.
- No es útil en problemas de regresión.

La función de **tangente hiperbólica (tanh)** también presenta el problema de desvanecimiento del gradiente, pero su comportamiento general es mejor que el de la función sigmoide.

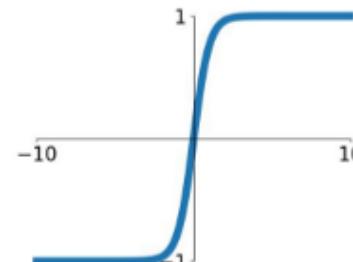
## Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



## tanh

$$\tanh(x)$$



# Funciones de activación

La función **ReLU (Rectified Linear Units)**, en comparación con las dos anteriores, tiene un coste computacional muy bajo y acelera la convergencia de los algoritmos de descenso de gradiente. Además, no presenta el problema de desvanecimiento de gradiente.

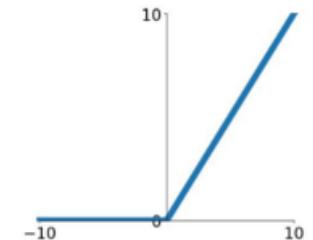
Sin embargo presenta el problema “ReLU inactiva”: al ser cero para todos los valores negativos de entrada, cuando la neurona entra en esa zona, es probable que no vuelva a salir de ese estado.

Esta función se suele usar solo en las capas ocultas y apenas nunca en redes recurrentes.

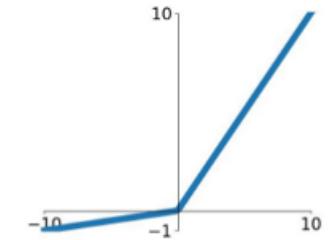
La función **Leaky ReLU** es una variante de la anterior y pretende solucionar el problema provocado por los valores negativos.

La solución consiste en presentar para los valores negativos presentan una recta con una pequeña pendiente negativa.

**ReLU**  
 $\max(0, x)$



**Leaky ReLU**  
 $\max(0.1x, x)$



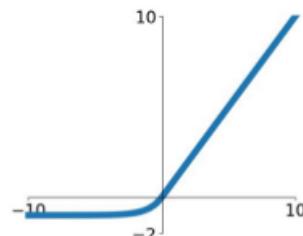
# Funciones de activación

**PReLU (Parametric ReLU)** da a las neuronas la capacidad de escoger qué pendientes mejor para la región negativa.

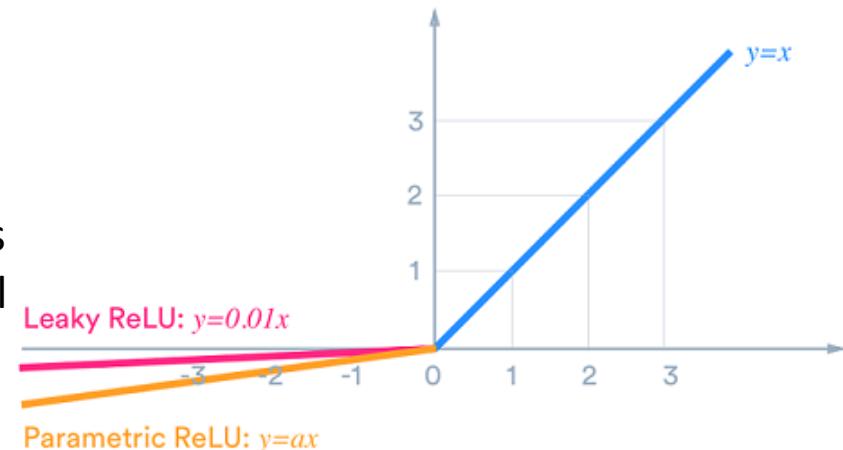
**ELU (Exponential Linear Unit)** da mejores resultados que ReLU. Para entradas positivas se comporta igual que ReLU, pero crece exponencialmente para valores negativos.

## ELU

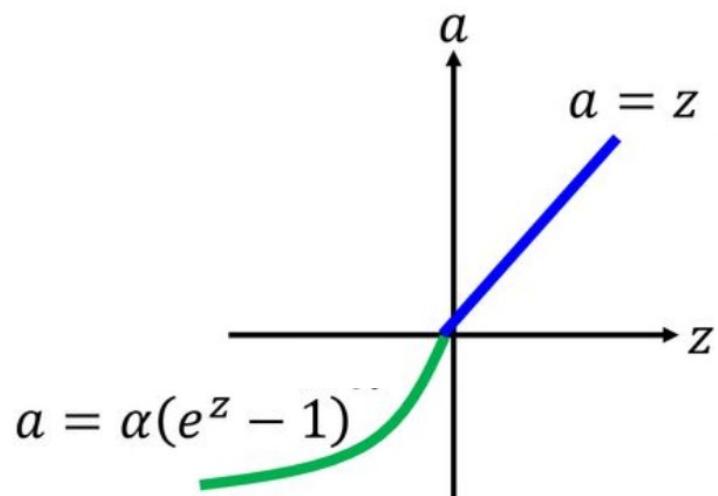
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Con la función de activación **SELU**, cada capa conserva la media y varianza de la capa anterior, por lo cual, se acelera el proceso de aprendizaje.



## Scaled ELU (SELU)



# Funciones de activación

La función de activación **Maxout** es una generalización de las funciones ReLU y ReLU con pérdidas. Es una función que devuelve el máximo de las entradas, diseñada para ser utilizada junto con la técnica de regularización de abandono (*dropout*).

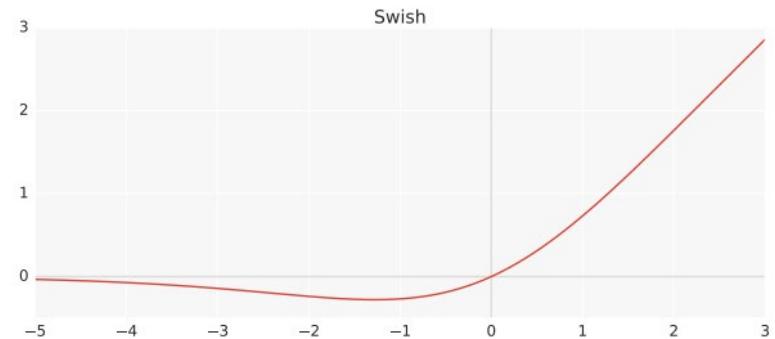
Esta función proporciona todos los beneficios de una unidad ReLU (régimen lineal de operación, sin saturación) y no tiene su inconveniente: ReLU inactiva. Los parámetros de esta función necesitan ser entrenados, por lo que se duplica el número de parámetros de cada neurona y, por lo tanto, se necesita entrenar un mayor número de parámetros.

## Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

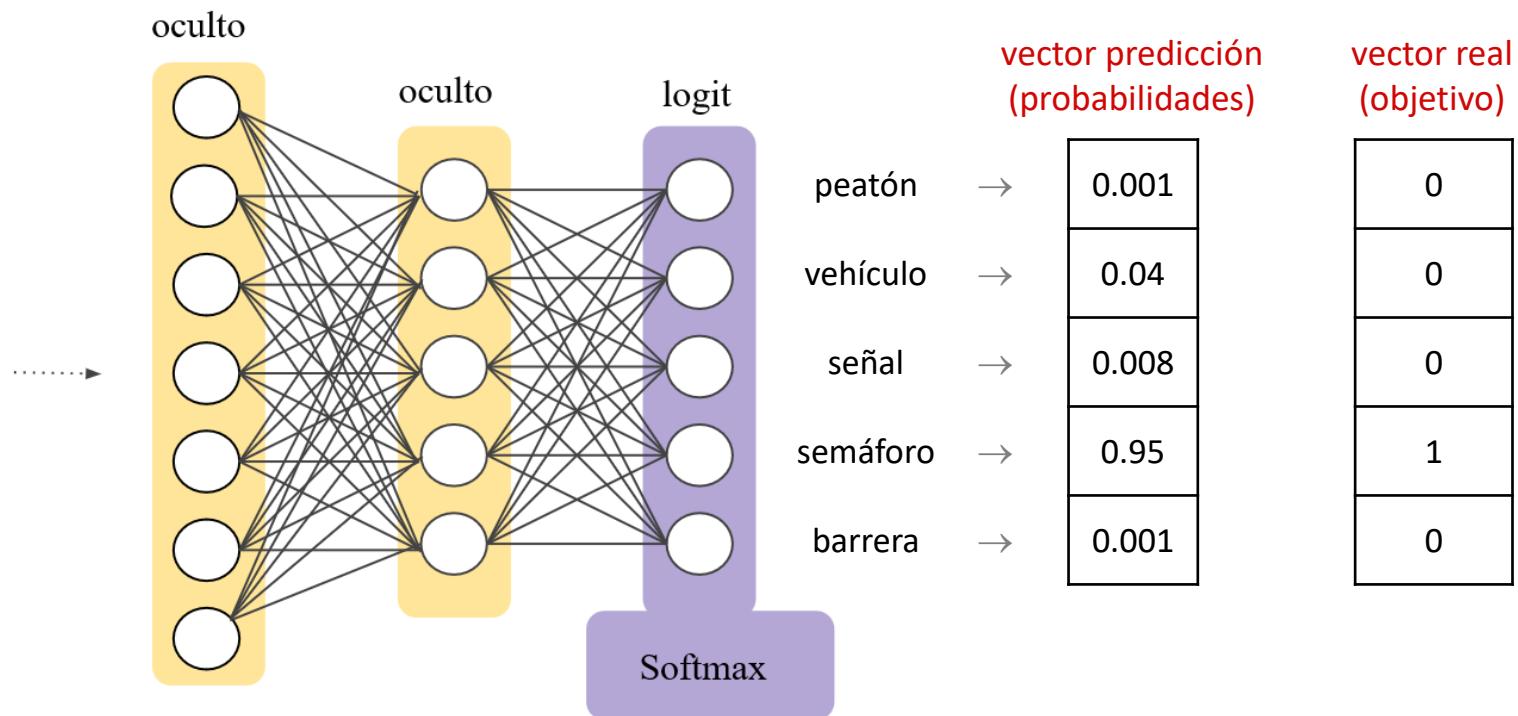
**Swish** es una función desarrollada por Google con un comportamiento mejor que ReLU y una carga computacional similar.

$$f(x) = x \cdot \text{sigmoid}(x)$$



# Funciones de activación

La función de activación **Softmax** asigna probabilidades decimales a cada clase cuando la decisión consiste en escoger una clase de entre  $n$  disponibles. El resultado es un vector de  $n$  valores, con las probabilidades de cada clase, cuyo total deben sumar 1.



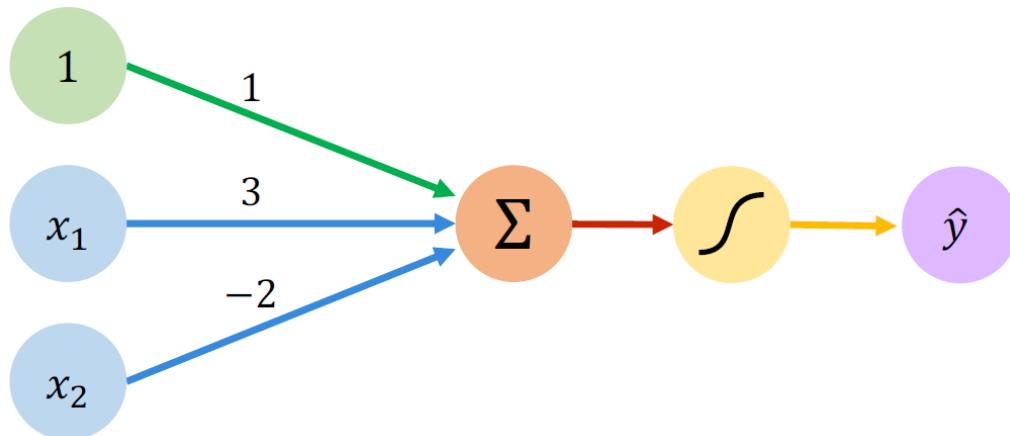
$$\text{softmax}(\mathbf{x}) = \frac{1}{\sum_{i=1}^n e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

# Funciones de activación

## Criterio de elección:

- Usar ReLU en capas ocultas, vigilando la tasa de aprendizaje (*learning rate*) y el porcentaje de neuronas inactivas.
- Si ReLU da problemas, probar Leaky ReLU, PReLU o Maxout.
- Las funciones sigmoide y tanh no se deben usar en redes con muchas capas debido al problema de desvanecimiento del gradiente.

# Ejemplo de perceptrón



$$w_0 = 1$$

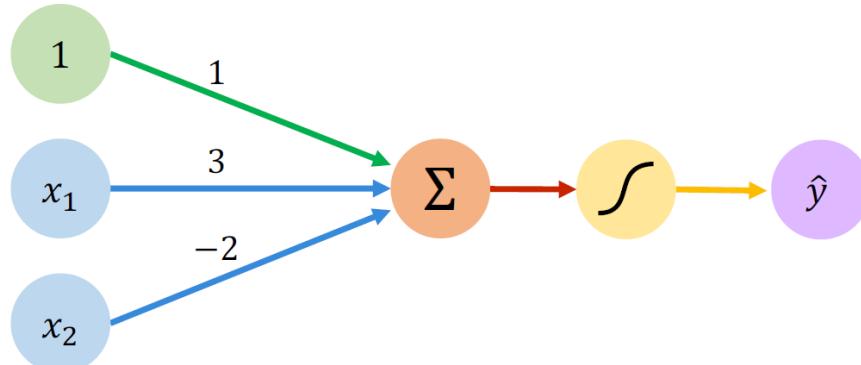
$$w = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$



$$\begin{aligned}\hat{y} &= g(w_0 + X^T w) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

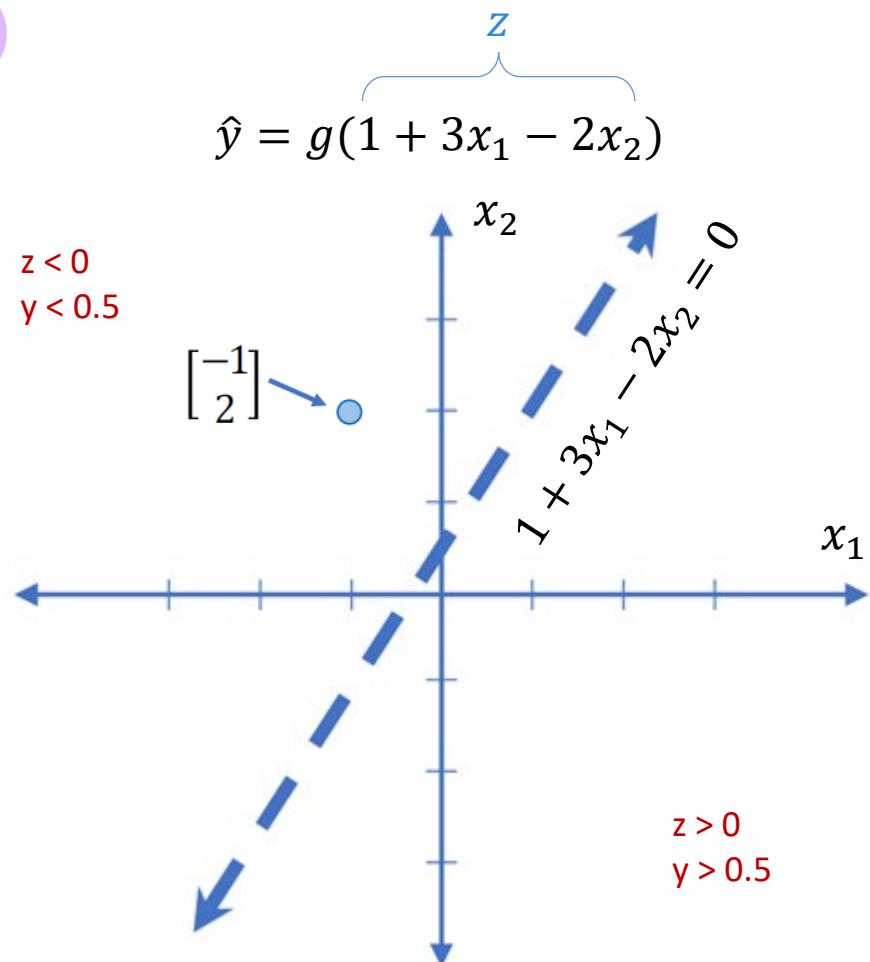
es una línea 2D

# Ejemplo de perceptrón

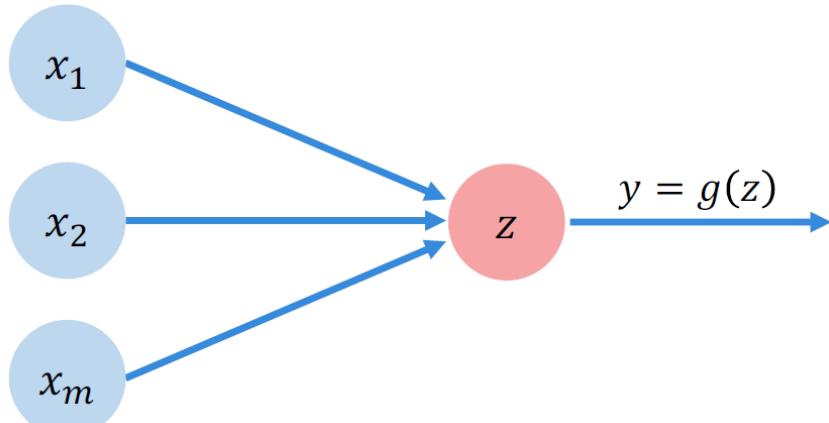


Si la entrada es:  $X = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

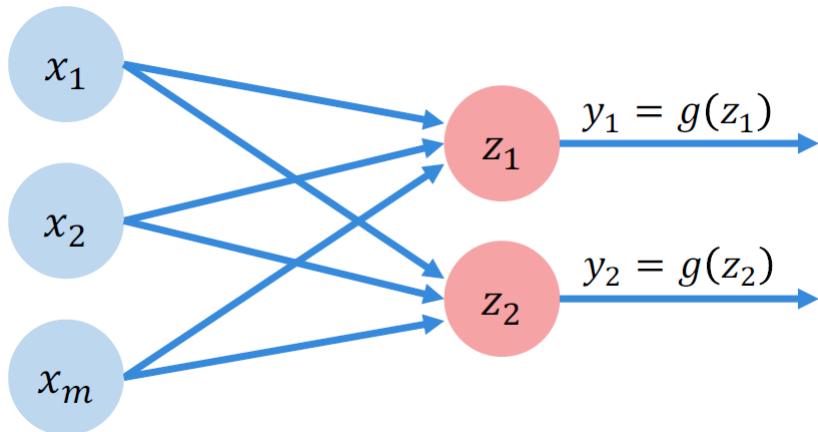
$$\begin{aligned}\hat{y} &= g(1 + (3 * -1) - (2 * 2)) \\ &= g(-6) \approx 0.002\end{aligned}$$



# Perceptrón simplificado y de múltiple salida



$$z = w_0 + \sum_{j=1}^m x_j w_j$$



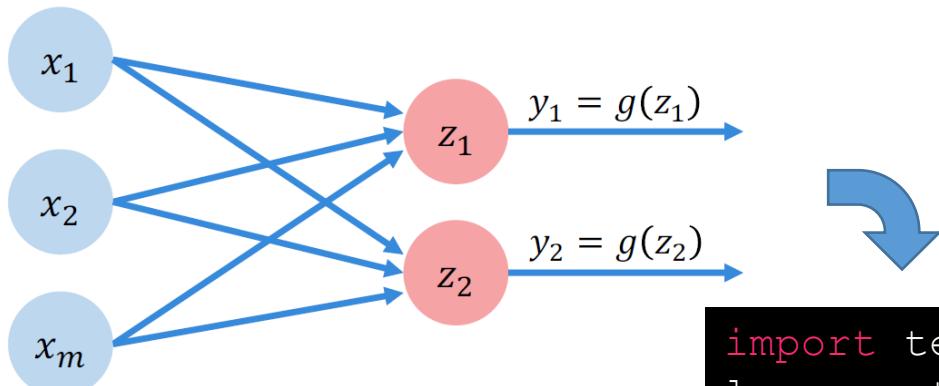
$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

**Capa densa:** todas las entradas están conectadas con todas las salidas.

# Programación de una capa densa:

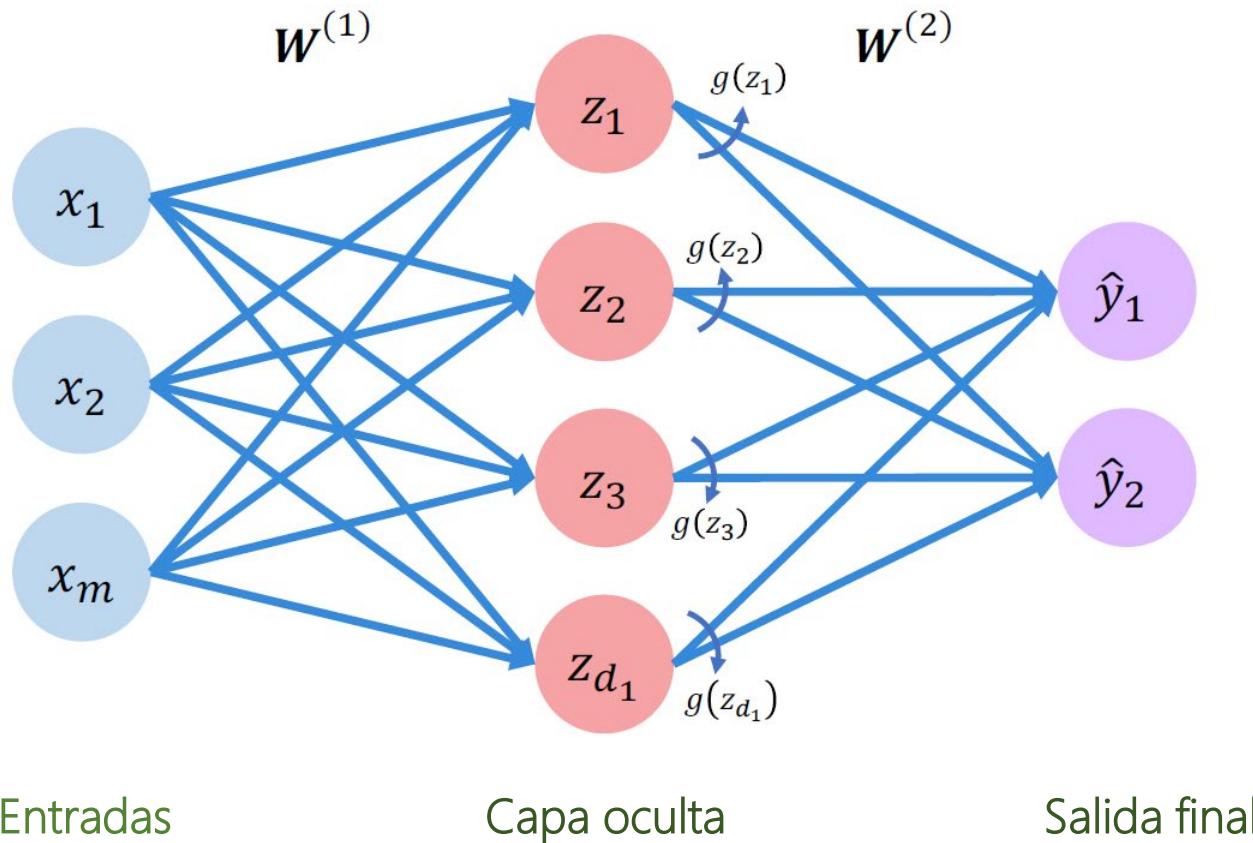
```
class CapaDensa(tf.keras.layers.Layer):
    def __init__(self, dim_ent, dim_sal):
        super(CapaDensa, self).__init__()
        # Inicializa pesos y sesgo (bias)
        self.W = self.añade_peso([dim_ent, dim_sal])
        self.b = self.añade_peso([1, dim_sal])

    def call(self, entradas):
        # propagación hacia delante de las entradas
        z = tf.matmul(entradas, self.W) + self.b
        # Activación no lineal
        salida = tf.math.sigmoid(z)
        return salida
```



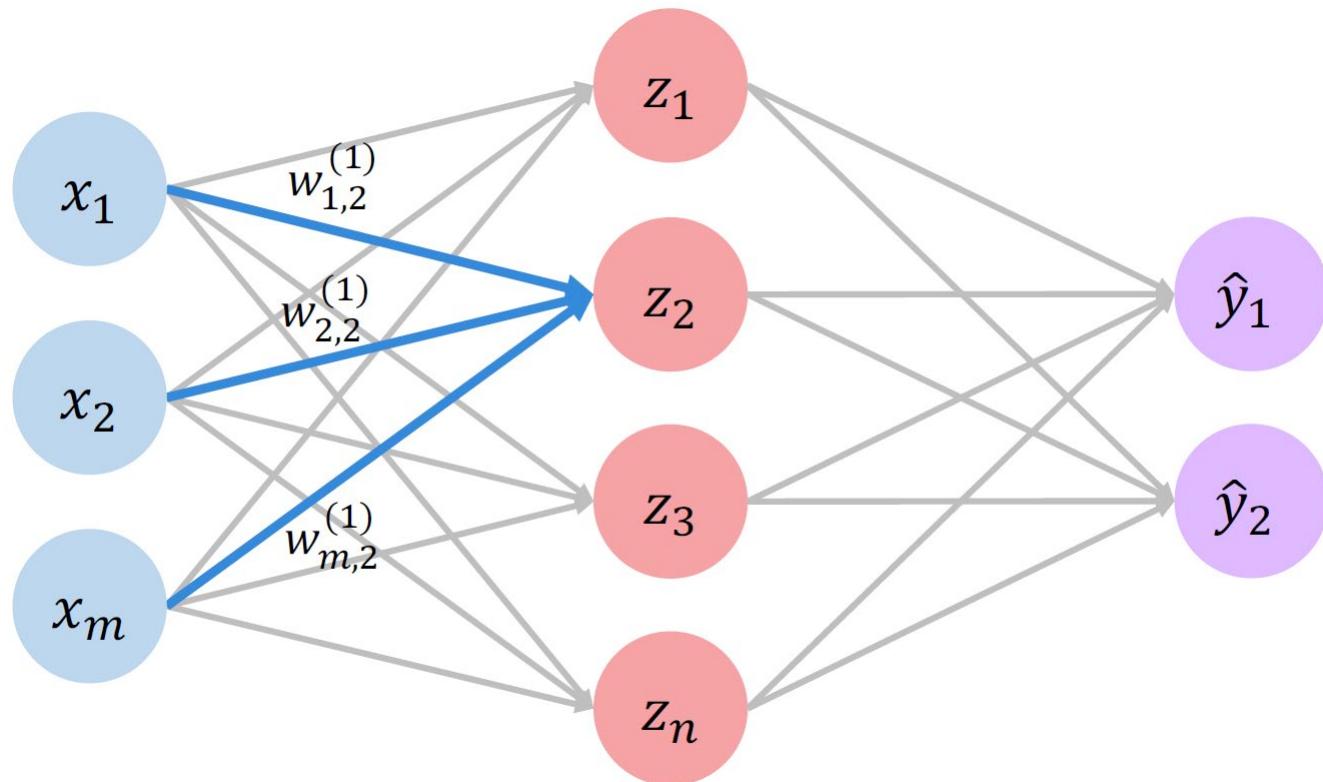
```
import tensorflow as tf
layer = tf.keras.layers.Dense(units=2)
```

# Red neuronal de capa única



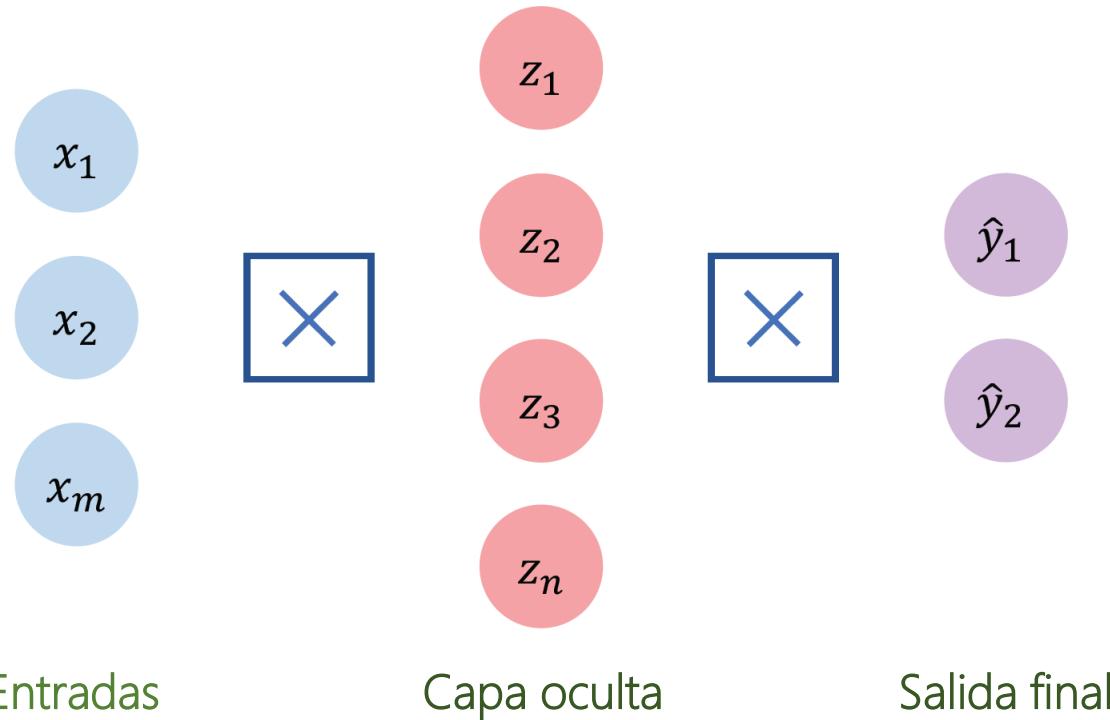
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)} \quad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)}\right)$$

# Red neuronal de capa única



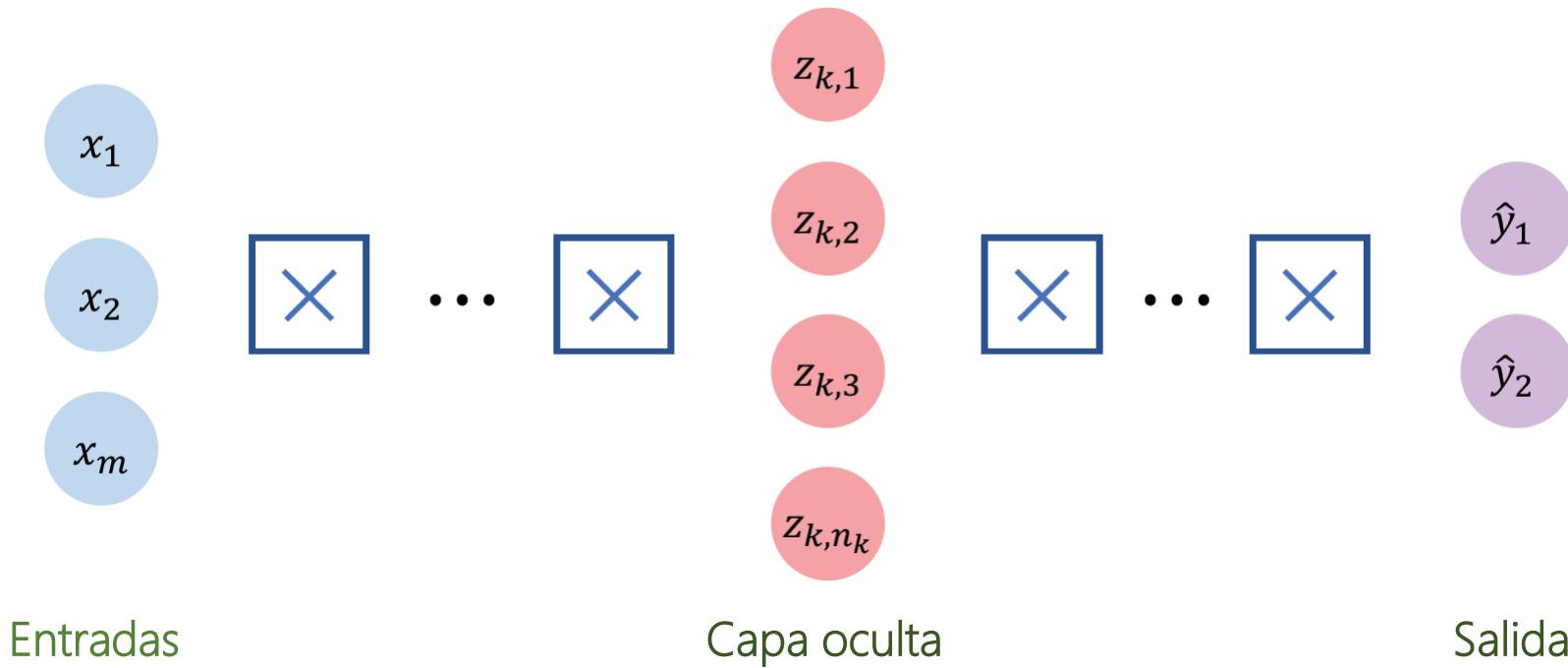
$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^m x_j w_{j,2}^{(1)} = w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)}$$

# Red neuronal de capa única



```
import tensorflow as tf  
  
model = tf.keras.Sequential([  
    tf.keras.layers.Dense(n),  
    tf.keras.layers.Dense(2)  
])
```

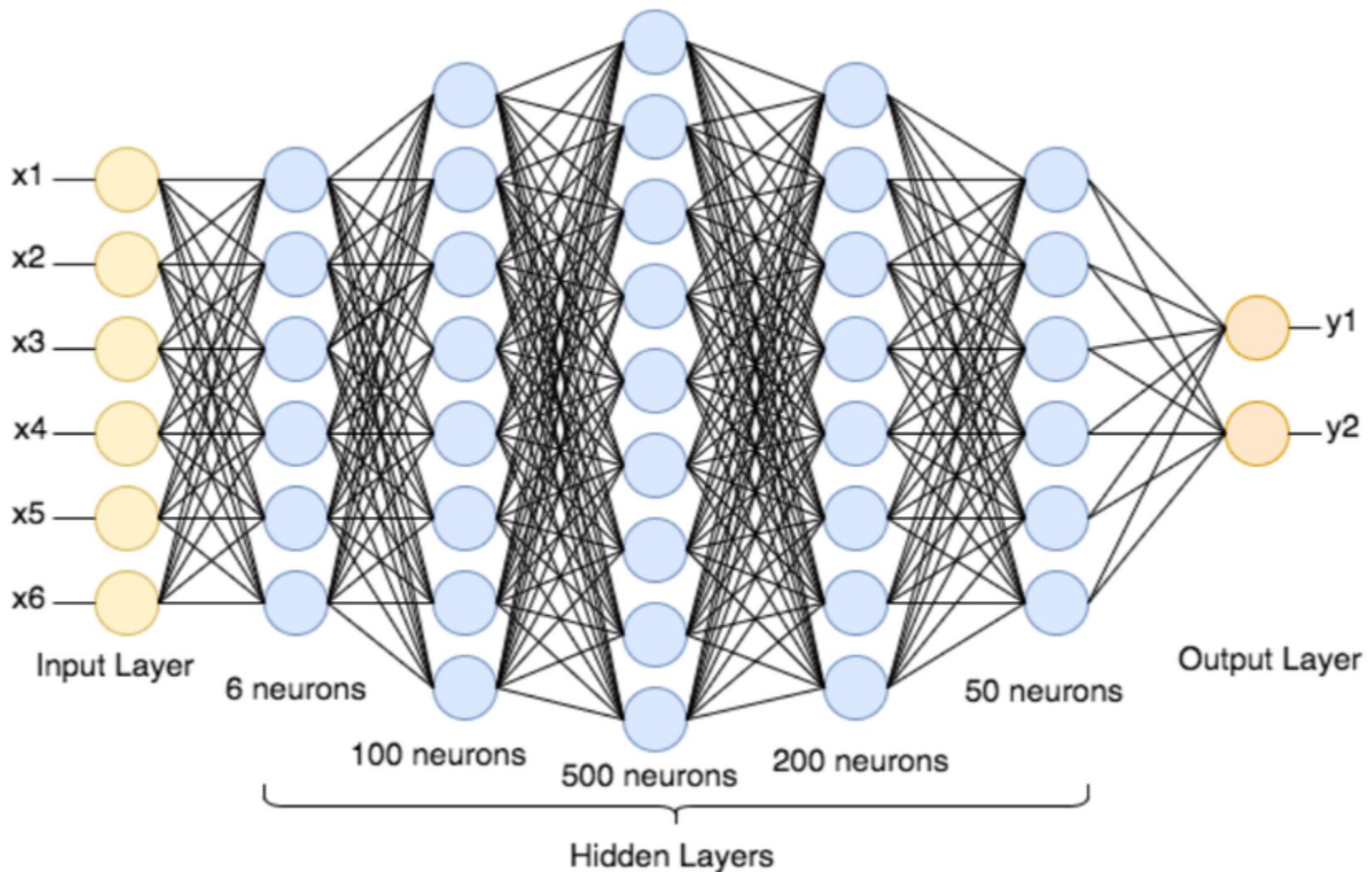
# Red neuronal profunda



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$

```
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Dense(n1),
    tf.keras.layers.Dense(n2),
    ...
    tf.keras.layers.Dense(2) ])
```

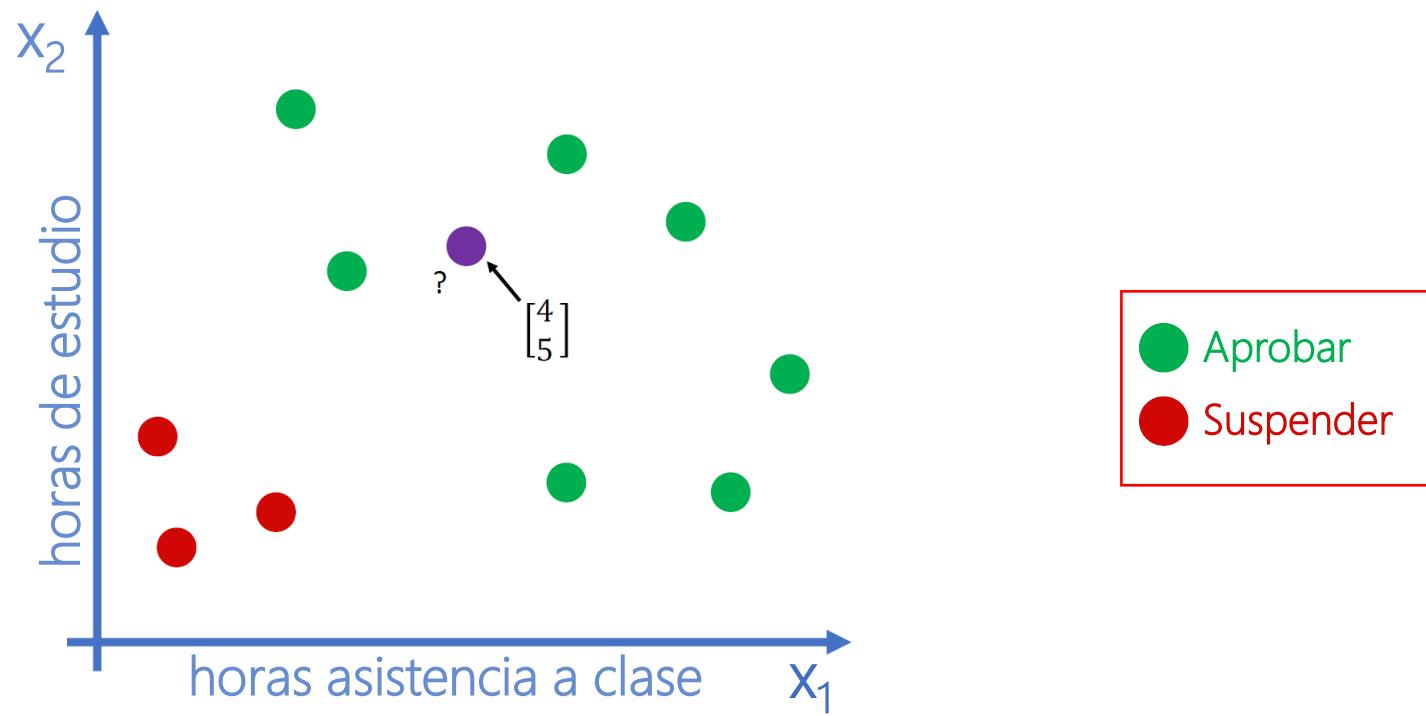
# Red neuronal profunda



# Ejemplo de uso

**Ejemplo de modelo sencillo: aprobar un examen.**

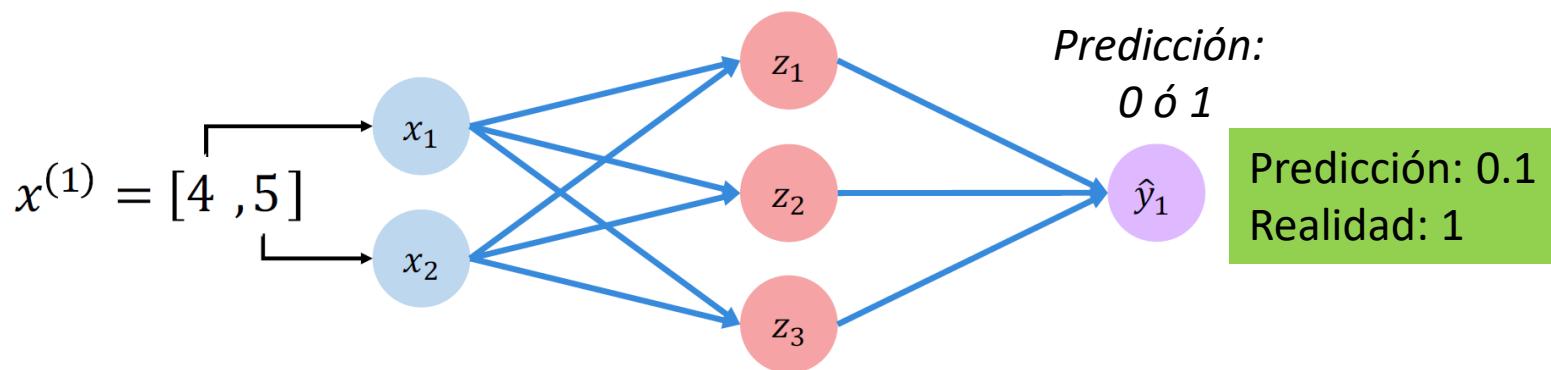
- $x_1$ : número de horas de asistencia a clase
- $x_2$ : horas de estudio



# Función de pérdida (*loss function*)

Ejemplo de modelo sencillo: aprobar un examen.

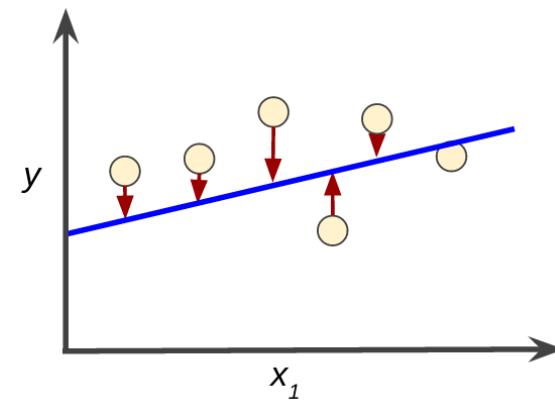
- $x_1$ : número de horas de asistencia a clase
- $x_2$ : horas de estudio



La **pérdida** (*loss*) es una medida del error entre la predicción y el valor real a predecir.

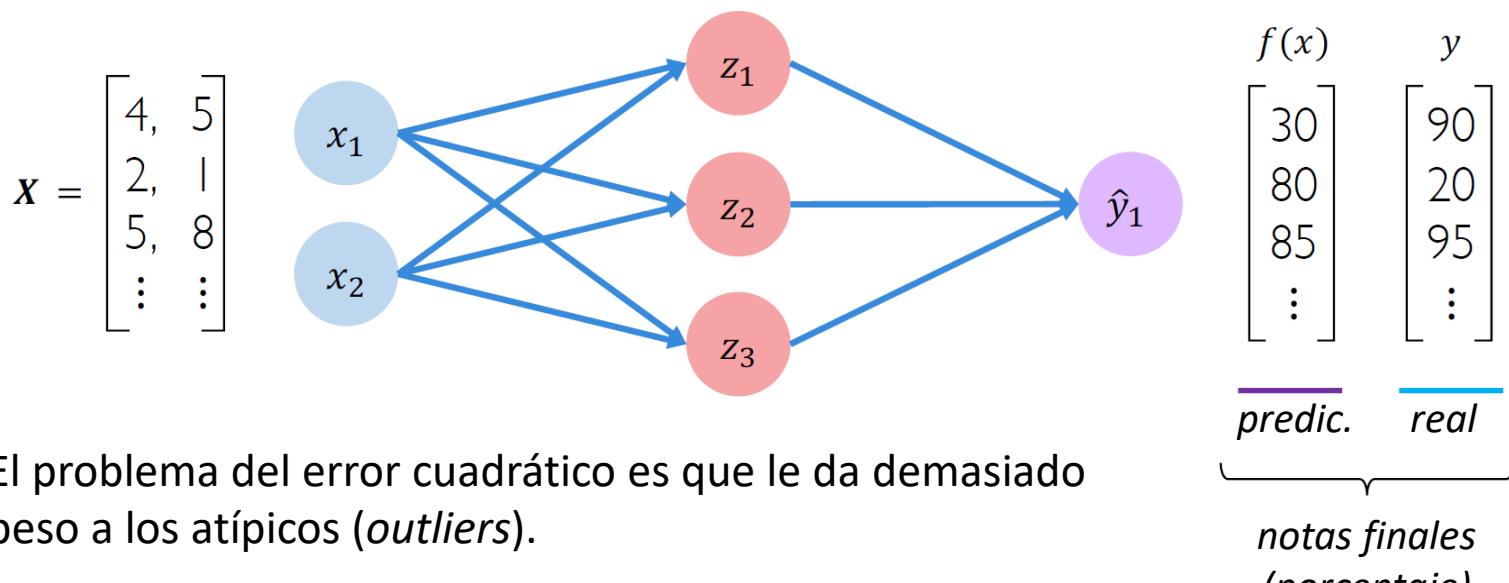
Se define para un único ejemplo, no sobre todo el conjunto de datos.

$$\mathcal{L} \left( \underbrace{f(x^{(i)}; \mathbf{W})}_{\text{predicción}}, \underbrace{y^{(i)}}_{\text{real}} \right)$$



# Pérdida de error cuadrático/absoluto

Se usan con modelos de regresión que generan números reales continuos, como por ejemplo estimar el precio de venta de un inmueble, o la nota de un examen:



El problema del error cuadrático es que le da demasiado peso a los atípicos (*outliers*).

El problema del error absoluto es que el gradiente es grande incluso si el error es pequeño.

error absoluto (L1)

$$MAE(r, p) = |r - p|$$

error cuadrático (L2)

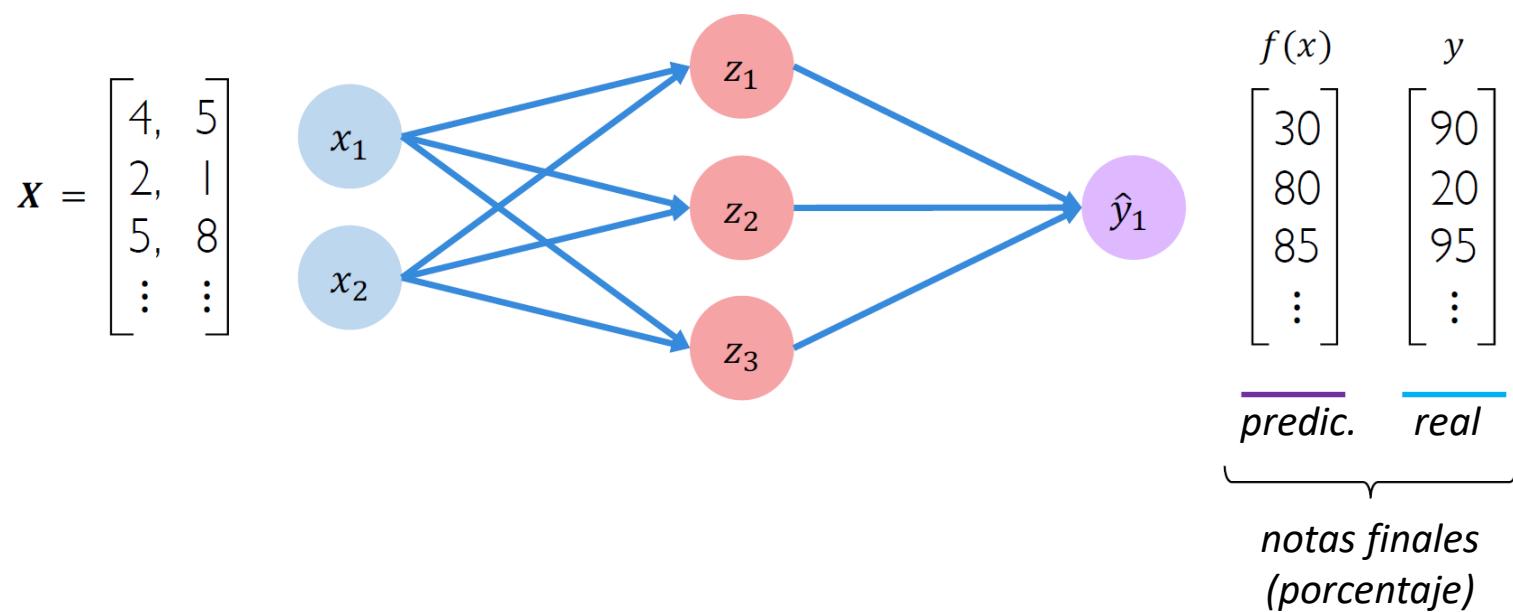
$$MSE(r, p) = (r - p)^2$$

donde:

$r$ : valor real,  
 $p$ : predicción

# Función de pérdida de Huber

La función de Huber es básicamente el error absoluto que se convierte en cuadrático cuando el error es pequeño. Es diferenciable en 0 y menos sensible a los casos atípicos (*outliers*).



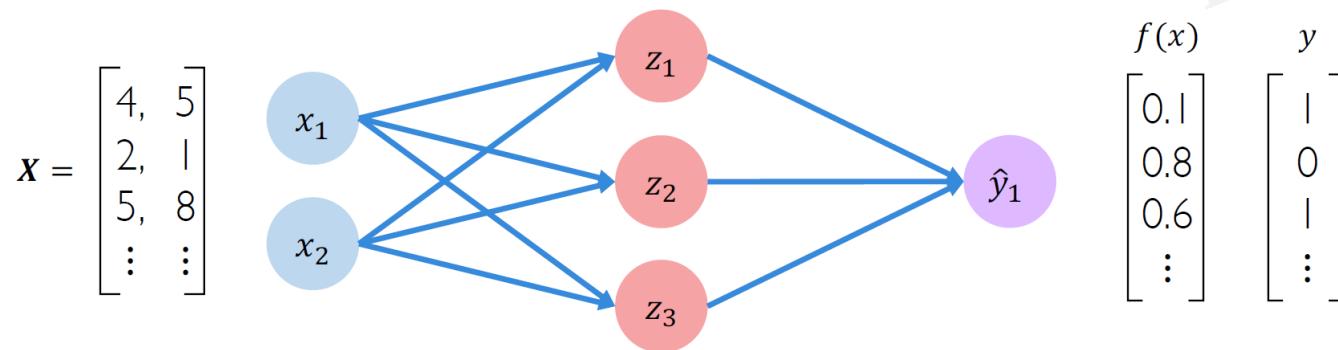
Huber

$$L_\delta(r, p) = \begin{cases} 0.5 \cdot (r - p)^2 & \text{para } |y - p| < \delta \\ \delta \cdot |y - p| - 0.5 \cdot \delta^2 & \text{resto} \end{cases}$$

donde:  
 $r$ : valor real,  
 $p$ : predicción  
 $\delta$ : parámetro Huber

# Pérdida de entropía cruzada binaria (*binary crossentropy loss*)

Se usa con modelos de clasificación binaria, los cuales generan una probabilidad entre 0 y 1. Por ejemplo, si lloverá mañana o si se superará un examen o no.



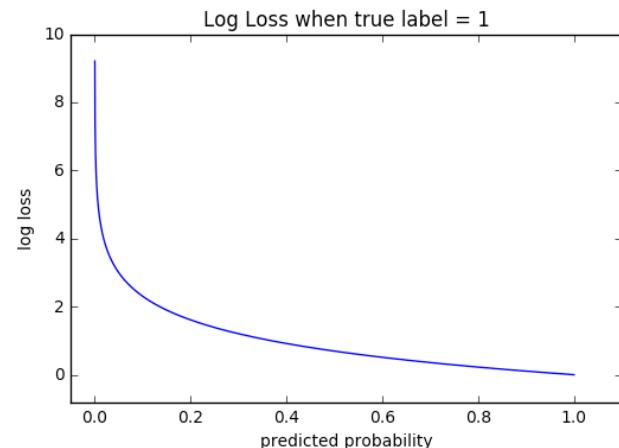
$$BCE(r, p) = -(r * \log(p) + (1 - r) * \log(1 - p))$$

donde:

$r$ : valor real,

$p$  : predicción

En todas las diapositivas se usa *log* para denotar el logaritmo natural

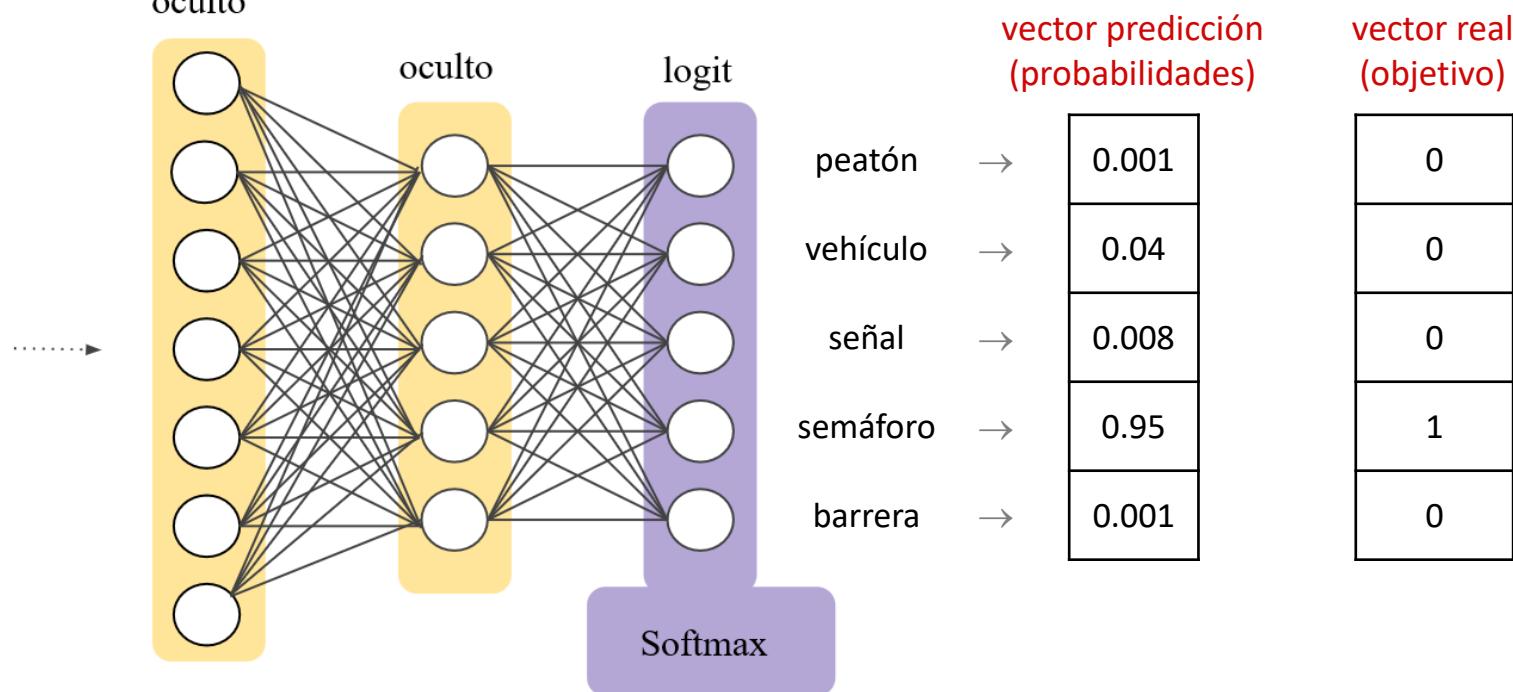


# Entropía cruzada categórica (categorical crossentropy loss)

El resultado (predicción) de la función *softmax* es un vector con las probabilidades de cada categoría (clase). El vector real (objetivo), será cero excepto en la posición correspondiente a la clase a la que corresponde el ejemplo (one-shot encoding).

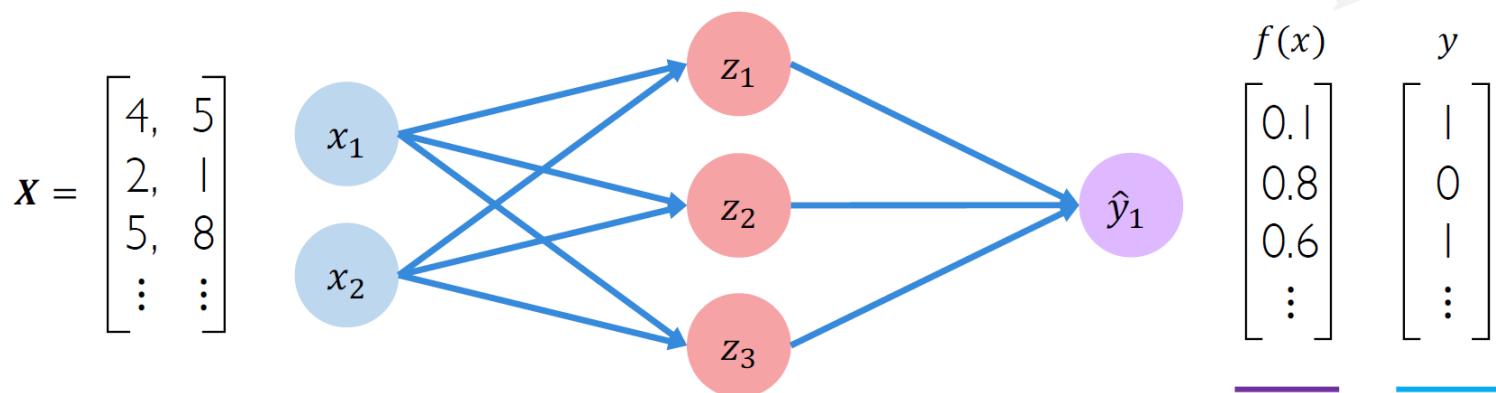
$$CCE(r, p) = \sum_i^n -r_i \cdot \log(p_i)$$

$r_i$ : componente  $i$  del vector real  
 $p_i$ : componente  $i$  del vector predicción



# Funciones de coste

Mide la pérdida total sobre todo el conjunto de datos (*dataset*) a partir de las funciones de pérdida. Por tanto existe una gran variedad de funciones de coste.



**Función de coste**  
**Función objetivo**  
**Riesgo empírico**

$$J(W) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

*predicción*      *real*

# Funciones de coste

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}\left(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{predicción}}, \underbrace{y^{(i)}}_{\text{real}}\right)$$

error cuadrático  
medio

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \left( \underbrace{y^{(i)}}_{\text{real}} - \underbrace{f(x^{(i)}; \mathbf{W})}_{\text{predicción}} \right)^2$$

entropía cruzada  
binaria

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{real}} \log\left(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{predicción}}\right) + (1 - \underbrace{y^{(i)}}_{\text{real}}) \log\left(1 - \underbrace{f(x^{(i)}; \mathbf{W})}_{\text{predicción}}\right)$$

entropía cruzada  
categórica

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{real}} \log\left(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{predicción}}\right) + (1 - \underbrace{y^{(i)}}_{\text{real}}) \log\left(1 - \underbrace{f(x^{(i)}; \mathbf{W})}_{\text{predicción}}\right)$$

# Entrenamiento de redes neuronales

# Optimización de pérdidas (*loss optimization*)

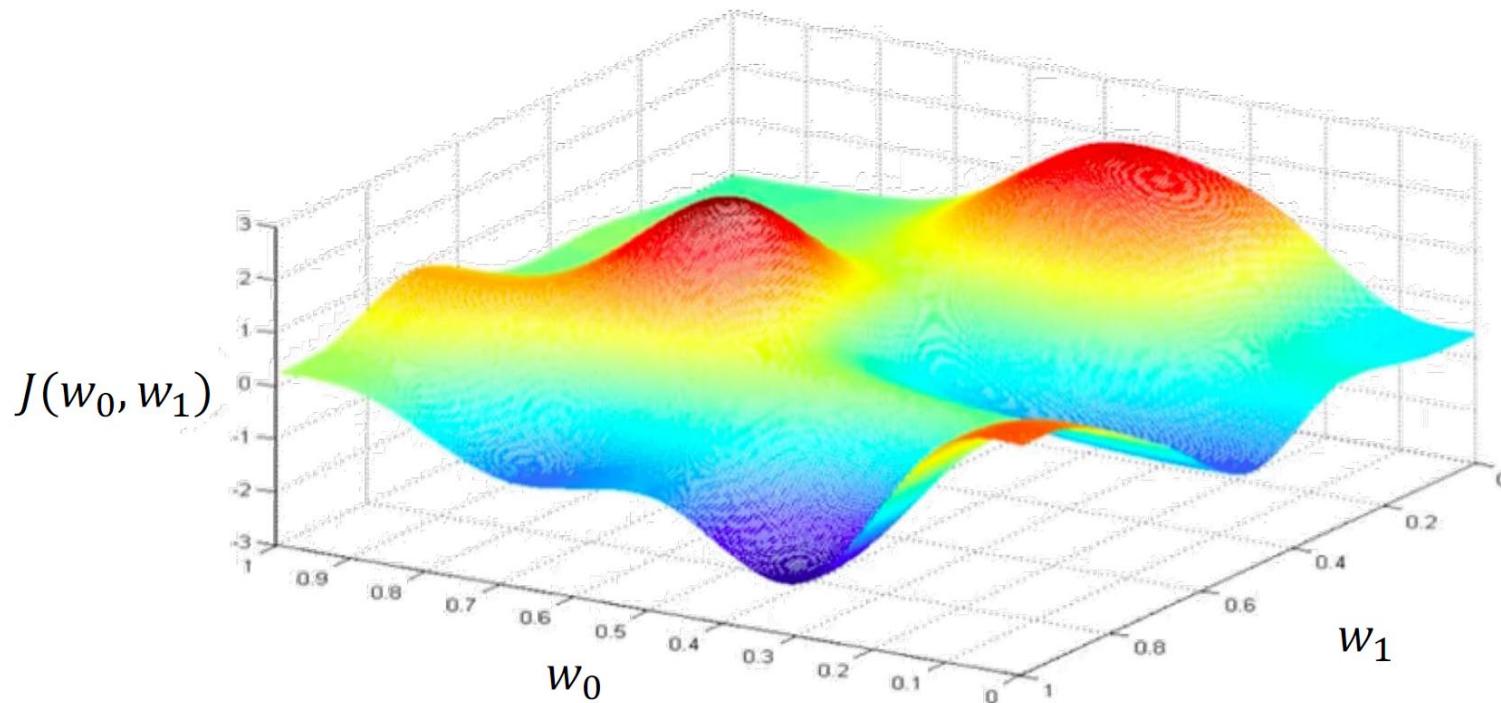
El objetivo es conseguir los pesos de la red que minimicen el coste.

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} J(\mathbf{W})$$

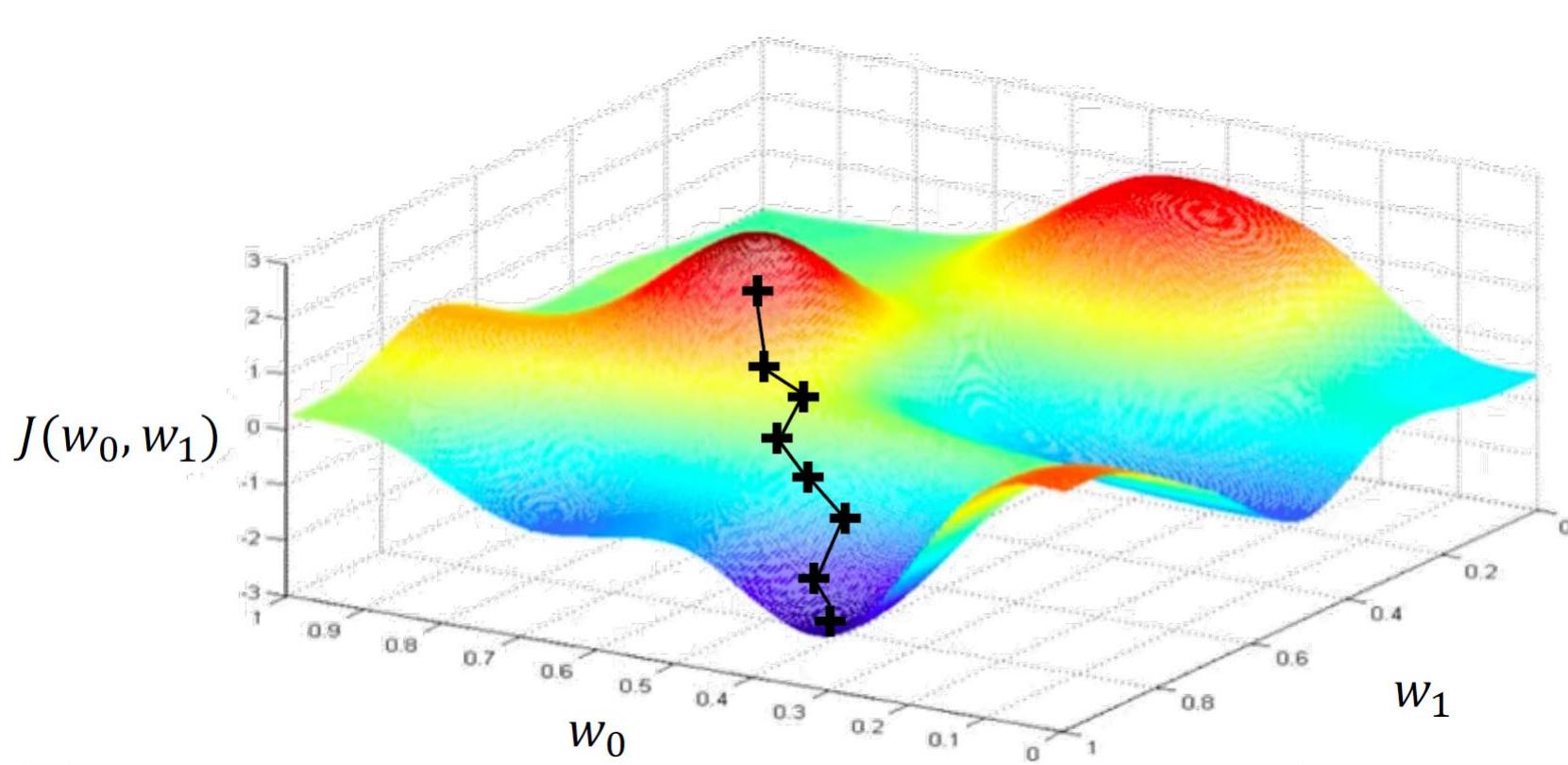


$$\mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \dots\}$$



# Descenso de gradiente (*gradient descent*)

El algoritmo de minimización consiste en escoger unos pesos aleatorios, calcular el gradiente y desplazarse en la dirección opuesta a dicho gradiente. El proceso se repite hasta que converja.



# Descenso de gradiente (*gradient descent*)

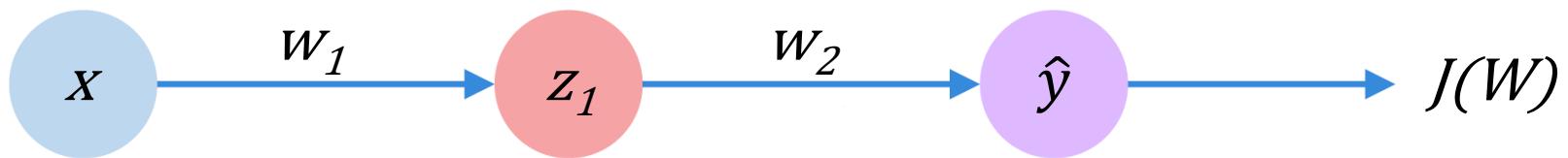
El algoritmo anterior se denomina de descenso de gradiente:

1. Inicializar los peso aleatoriamente  $\sim N(0, \sigma^2)$
2. Repetir hasta la convergencia:
  - i. Calcular el gradiente:  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
  - ii. Actualizar los pesos:  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
3. Devolver los pesos

```
import tensorflow as tf
pesos = tf.Variable([tf.random.normal()])
while True: # bucle infinito
    with tf.GradientTape() as g:
        loss = compute_loss(pesos)
        gradiente = g.gradient(loss, pesos)
    pesos = pesos - lr * gradiente
```

# Cálculo del gradiente: retropropagación (backpropagation)

Backpropagation es un método de cálculo de gradientes que parte de la siguiente pregunta: ¿cuánto afecta un pequeño cambio en un peso ( $w_2$ , por ejemplo) en la pérdida final  $J(W)$ ?



Se aplica la regla de la cadena:

$$\frac{\partial J(W)}{\partial w_2} = \underline{\frac{\partial J(W)}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial w_2}}$$

Se repite esto para cada peso de la red usando los gradientes de capas posteriores.

$$\frac{\partial J(W)}{\partial w_1} = \underline{\frac{\partial J(W)}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial w_1}}$$

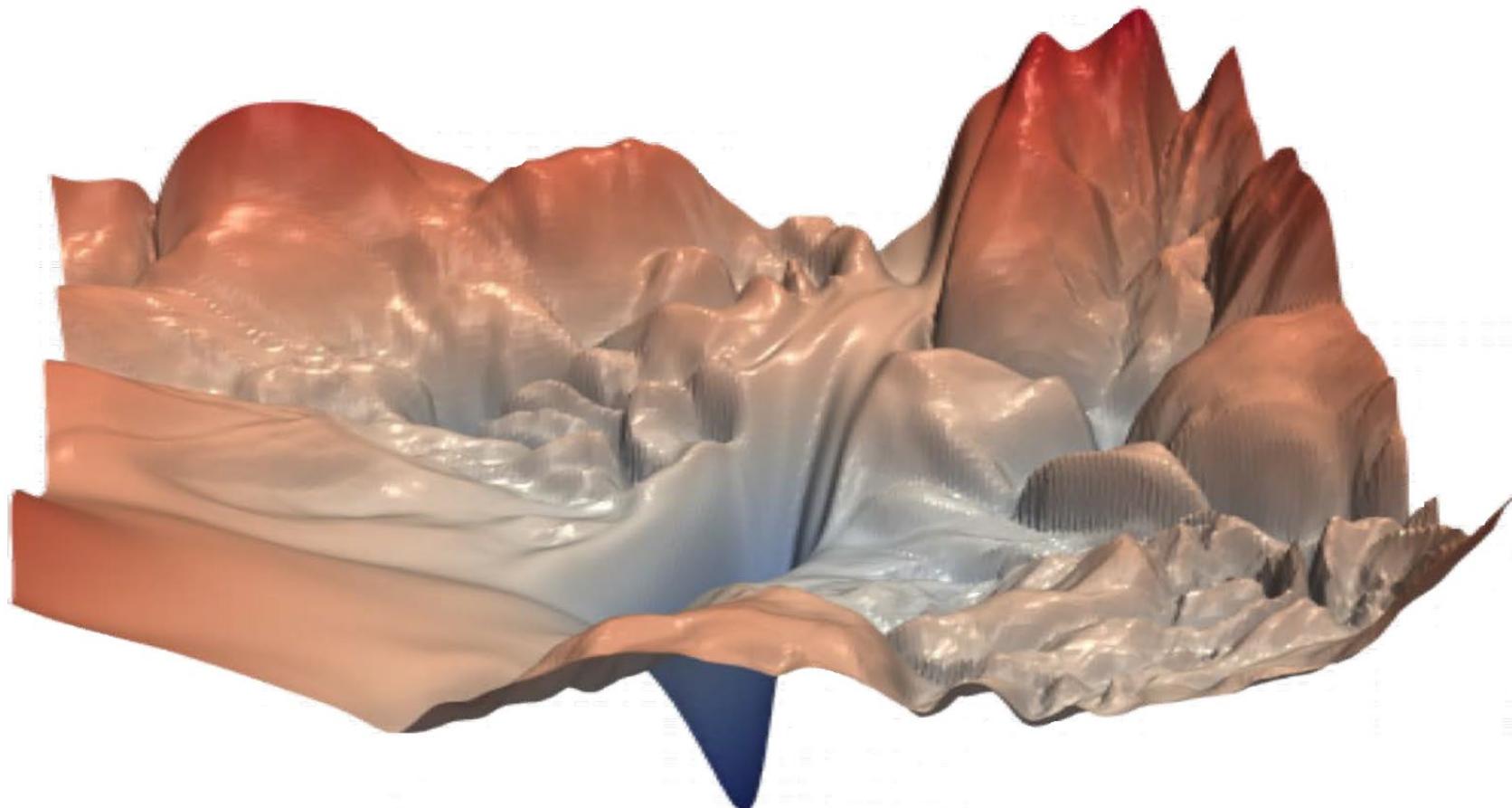


$$\frac{\partial J(W)}{\partial w_1} = \underline{\frac{\partial J(W)}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial z_1}} * \underline{\frac{\partial z_1}{\partial w_1}}$$

# Optimizadores

# Dificultad en el entrenamiento de redes neuronales

Las funciones de pérdidas son difíciles de optimizar.

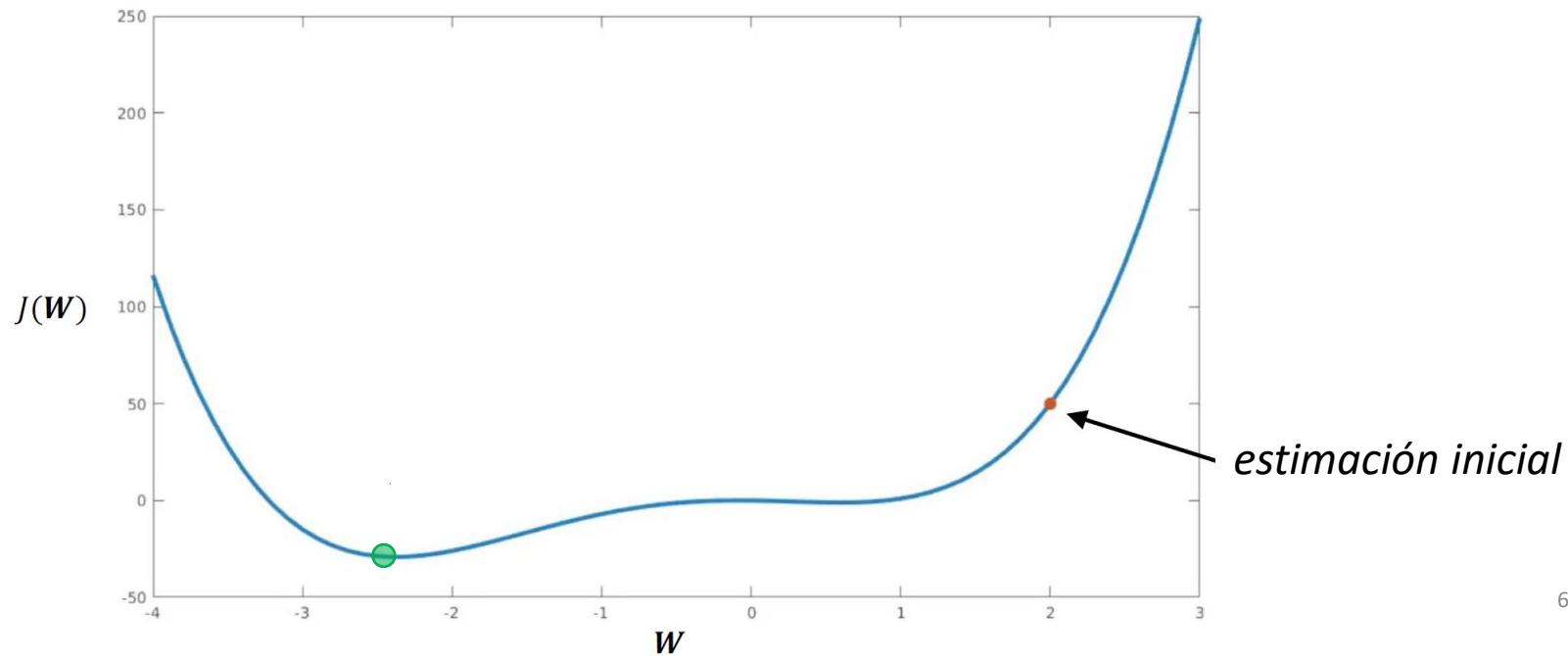


# Tasa de aprendizaje (*learning rate*)

Hay que determinar un valor para la tasa de aprendizaje ( $\eta$ ) del método de optimización por descenso de gradiente:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

- Como se deduce de la fórmula, la tasa de aprendizaje determina cuánto cambian los pesos en dirección opuesta al gradiente.
- Si es pequeña, la convergencia es lenta y puede quedar atrapada en un mínimo local.
- Si es grande, puede sobreoscilar, ser inestable o divergir. La convergencia es más rápida.



# Cálculo de la tasa de aprendizaje constante

La opción por defecto en la mayoría de las implementaciones de SGD es la de usar una tasa de aprendizaje constante.

El método ***learning rate range*** permite establecer un valor adecuado para cada problema:

- Durante una repetición (*epoch*) se comienza a aplicar el SGD con una tasa de aprendizaje muy pequeña ( $10^{-8}$ , por ejemplo)
- Para cada mini-lote se multiplicará por un factor hasta que alcance un valor muy alto (1 ó 10, por ejemplo). El factor suele ser  $(\text{valor\_final}/\text{valor\_inicial})^{1/(\text{num\_ejemplos}-1)}$
- En cada iteración se registra la pérdida que se dibujará frente a la tasa de aprendizaje.

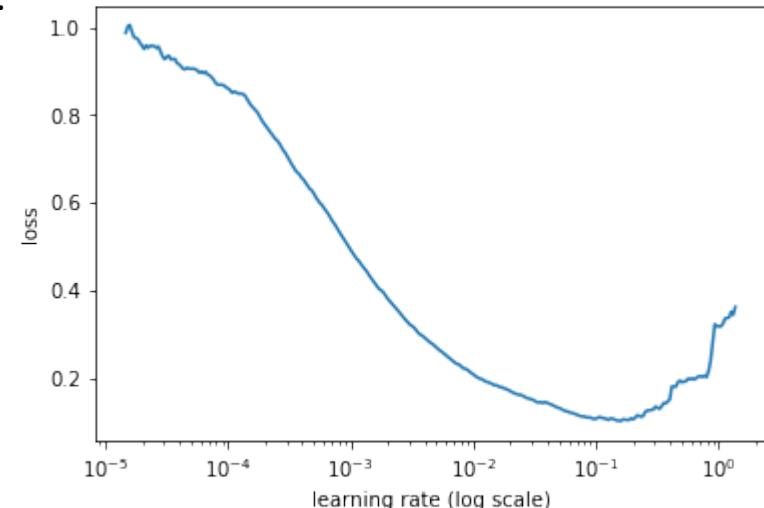
La tasa de aprendizaje correspondiente al mínimo ya es demasiado alta, dado que estamos en la frontera entre la mejora y el empeoramiento rápido.

La tasa a escoger será un orden de magnitud anterior.

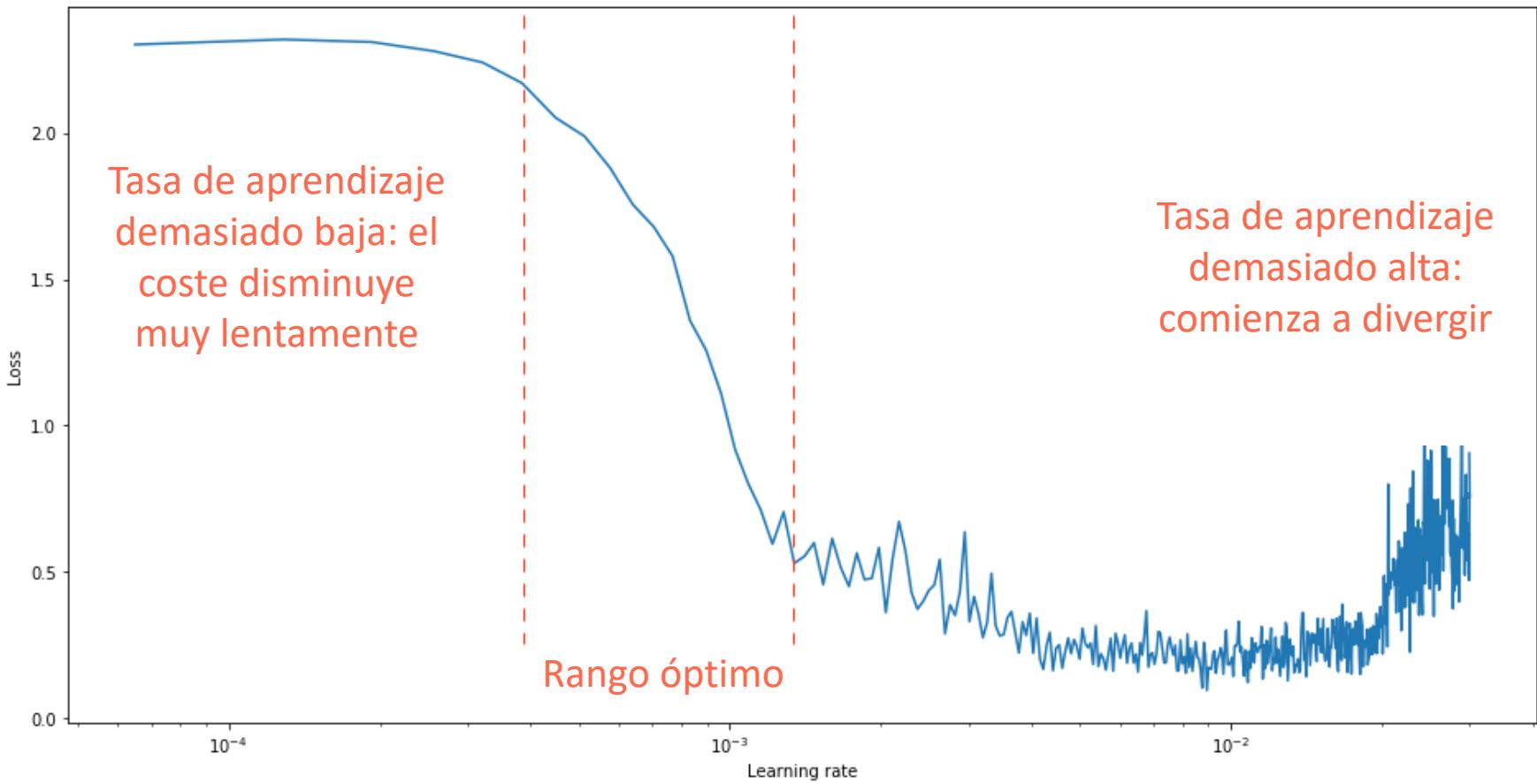
En el ejemplo de la gráfica, el mínimo se alcanza aproximadamente para una tasa de  $10^{-1}$  por lo que se escoge el valor  $10^{-2}$ .

Este método se puede aplicar con cualquier variante del SGD y cualquier tipo de red.

Simplemente se necesita una repetición o menos para ajustar el valor.



# Cálculo de la tasa de aprendizaje constante



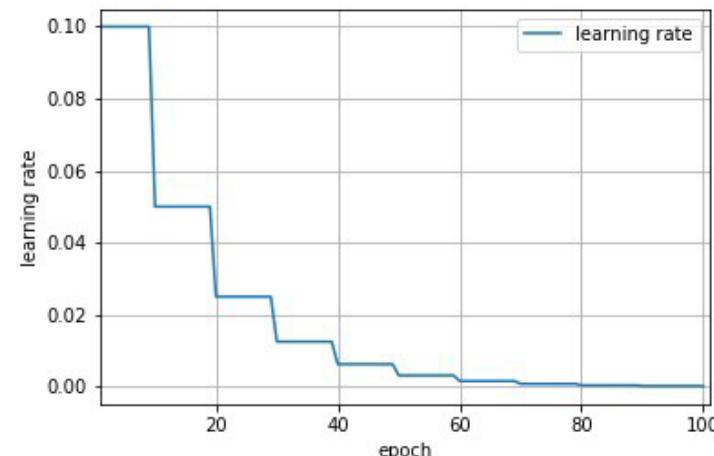
# Tasa de aprendizaje decreciente

## Decrecimiento temporal (*time-based decay*):

- La tasa de aprendizaje se reduce en cada repetición (*epoch*) según la siguiente fórmula:  
$$\text{learn\_rate} = \text{learn\_rate\_inicial} / (1 + k \cdot \text{epoch})$$
 donde **k** es el factor de decrecimiento y **epoch** el número de repetición.
- Una práctica común es buscar el valor ideal para una tasa constante (diapositiva anterior) y tomar ese valor como inicial en el proceso.

## Decrecimiento por pasos (*step decay*):

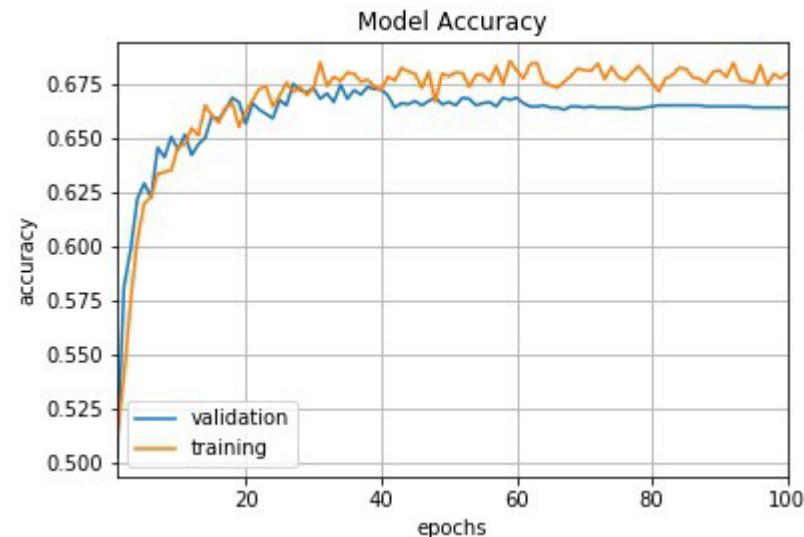
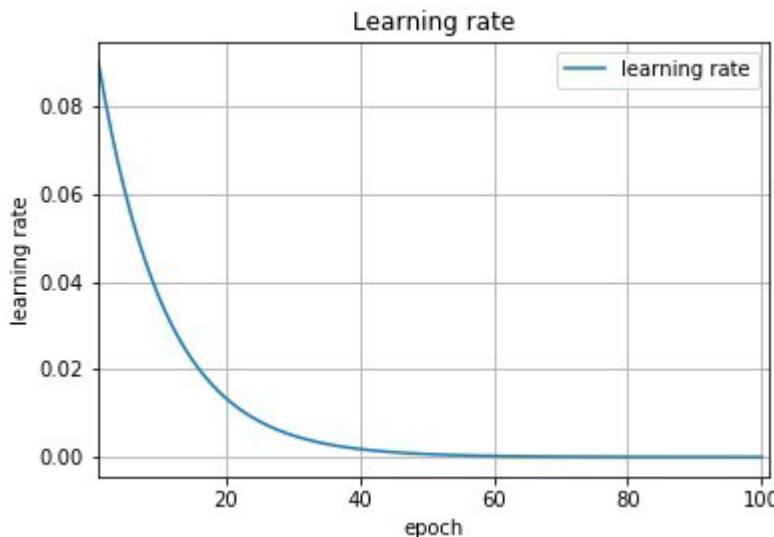
- La tasa de aprendizaje se reduce en un factor cada n repeticiones (*epochs*):  
$$\text{learn\_rate} = \text{learn\_rate\_inicial} * k^{(\text{epoch}/n)}$$
 donde **k** es el factor de decrecimiento, **epoch** el número de repetición.
- Típicamente se reduce a la mitad cada 10 repeticiones ( $k=0.5$  y  $n=10$ ):  
$$\text{learn\_rate} = \text{learn\_rate\_inicial} * 0.5^{(\text{epoch}/n)}$$
- Este método es mucho más utilizado que el anterior.



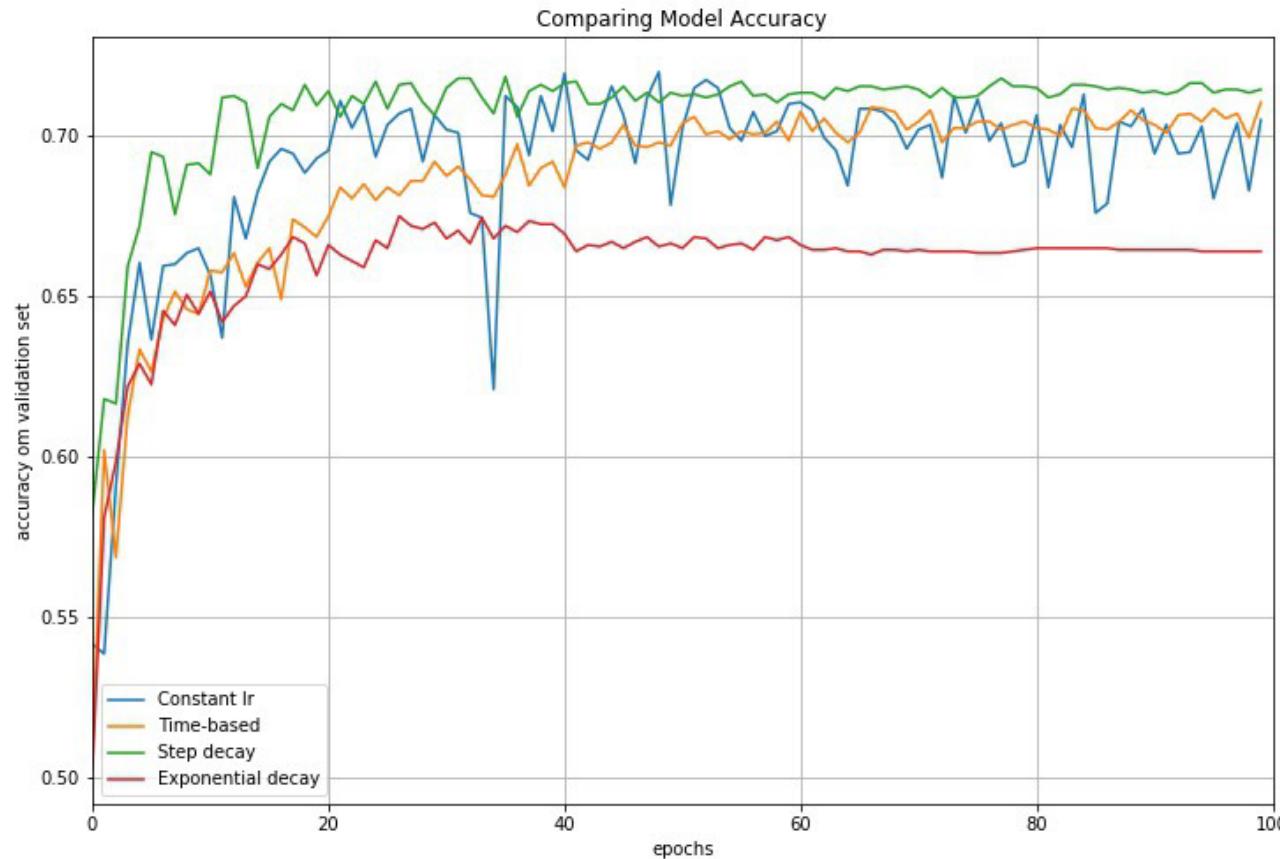
# Tasa de aprendizaje decreciente

**Decrecimiento exponencial (exponential decay):**

- La tasa de aprendizaje se reduce en cada repetición (*epoch*) según la siguiente fórmula:  
$$\text{learn\_rate} = \text{learn\_rate\_inicial} \cdot e^{(-k \cdot \text{epoch})}$$
 donde  $k$  es el factor de decrecimiento y  $\text{epoch}$  el número de repetición.
- Nuevamente, es típico buscar el valor ideal para una tasa y tomar ese valor como inicial en el proceso. Un valor típico de  $k$  es 0.1.



# Tasas contantes y decrecientes



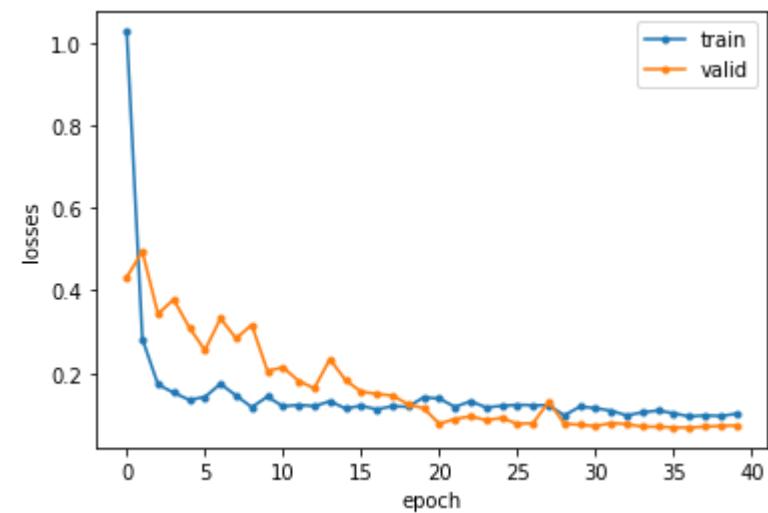
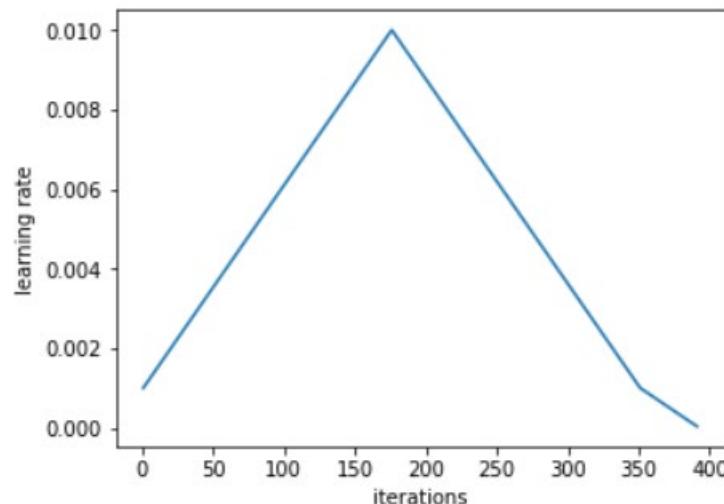
Inconveniente de las tasas de aprendizaje monótonamente decrecientes:

- El modelo es sensible al valor inicial de la tasa.
- No hay garantías de que el modelo esté en áreas de pérdidas bajas cuando la tasa sea baja.

Solución: tasas de aprendizaje cíclicas.

# Tasas de aprendizaje cíclicas: Política de un ciclo

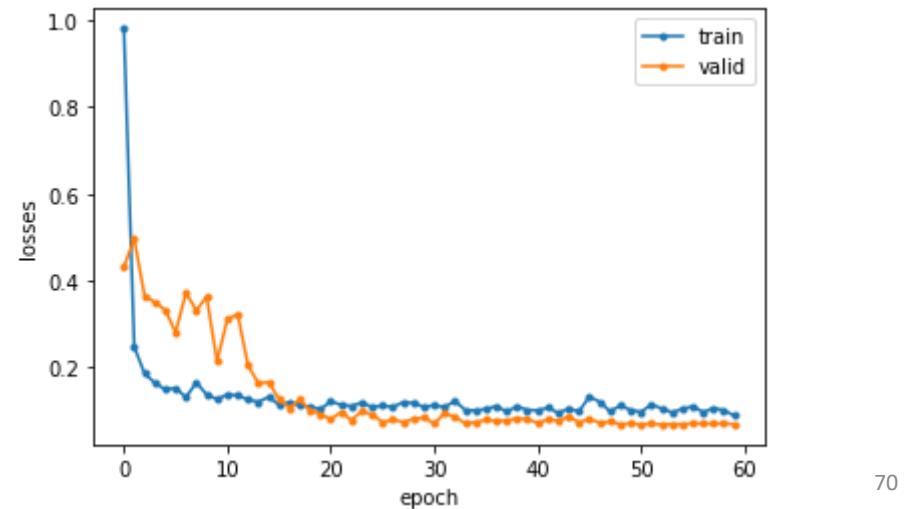
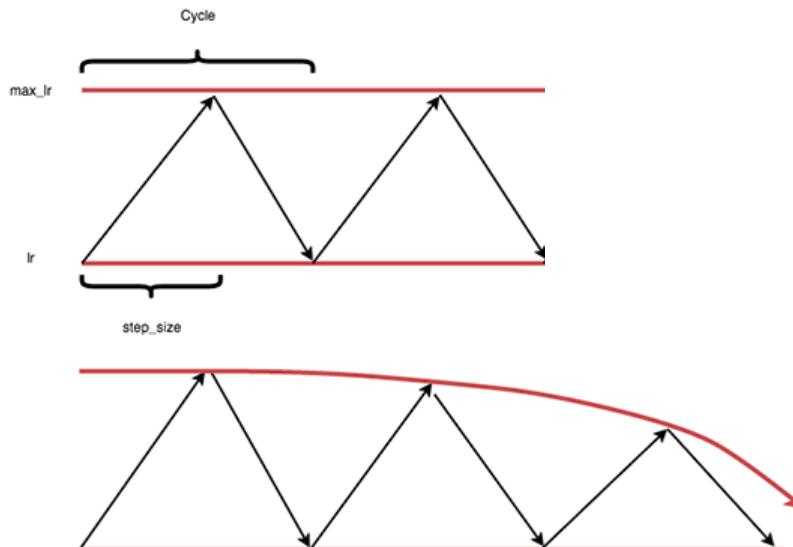
- *One cycle policy*: Consiste en hacer un ciclo con dos pasos de igual tamaño: uno desde una tasa de aprendizaje pequeña hasta una grande, y el segundo, desde la grande volver a la pequeña. Ambos pasos son lineales. La máxima es la calculada con el método de *learning range* visto anteriormente.
- Como mínima suele tomar un valor 10 veces menor que la máxima.
- La longitud del ciclo debería ser algo menor que el número total de repeticiones (*epochs*)
- En la última parte del entrenamiento, se debería permitir que la tasa de aprendizaje disminuyera por debajo del mínimo en varios órdenes de magnitud.



- Esta estrategia también es útil con el momento: primero decrece linealmente y finalmente se le permite volver a crecer (forma de V).

# Tasa de aprendizaje cíclicas (*cyclical learning rates*)

- Se define los límites inferior y superior de la tasa y se le hace oscilar entre ambas, incrementándola y decrementándola lentamente tras cada actualización de lote.
- Esto disminuye la probabilidad de que el sistema se atasque en un mínimo local o un punto de silla y lo hace menos sensible al valor inicial de la tasa de aprendizaje.
- Parámetros a utilizar:
  - Tamaño del lote: número de ejemplos por actualización de los pesos
  - Iteración: número de lotes en los que se divide el *dataset* de entrenamiento.
  - Ciclo: número de iteraciones que tarda en tomar dos veces seguidas el valor mínimo .
  - Tamaño de paso: número de iteraciones en medio ciclo. Se recomienda entre dos y ocho veces el número de iteraciones por *epoch* durante el entrenamiento.
  - Existen variantes que reducen monótonamente la amplitud de los ciclos.



# Optimizadores

Se han desarrollado una gran cantidad de algoritmos basados en el de descenso de gradiente para mejorar su rendimiento. La mayoría están ya implementados en diversas bibliotecas.

- SGD → tf.keras.optimizers.SGD
- Adam → tf.keras.optimizers.Adam
- Adadelta → tf.keras.optimizers.Adadelta
- Adagrad → tf.keras.optimizers.Adagrad
- RMSProp → tf.keras.optimizers.RMSProp
- ...

```
import tensorflow as tf
modelo = tf.keras.Sequential([...])
optimizador = tf.keras.optimizer.SGD() # escogemos un optimizador
while True: # bucle infinito

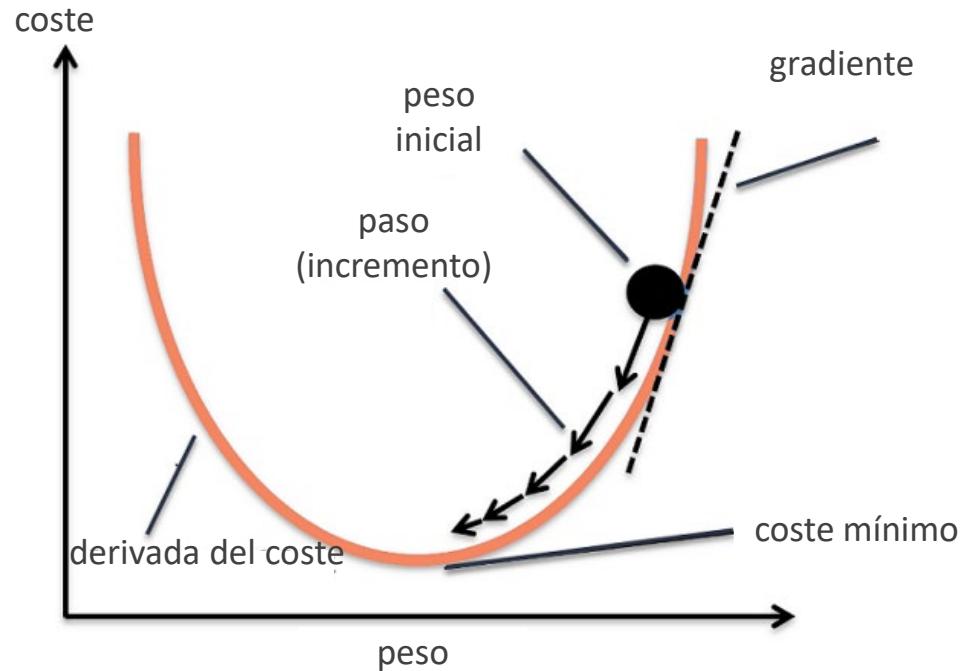
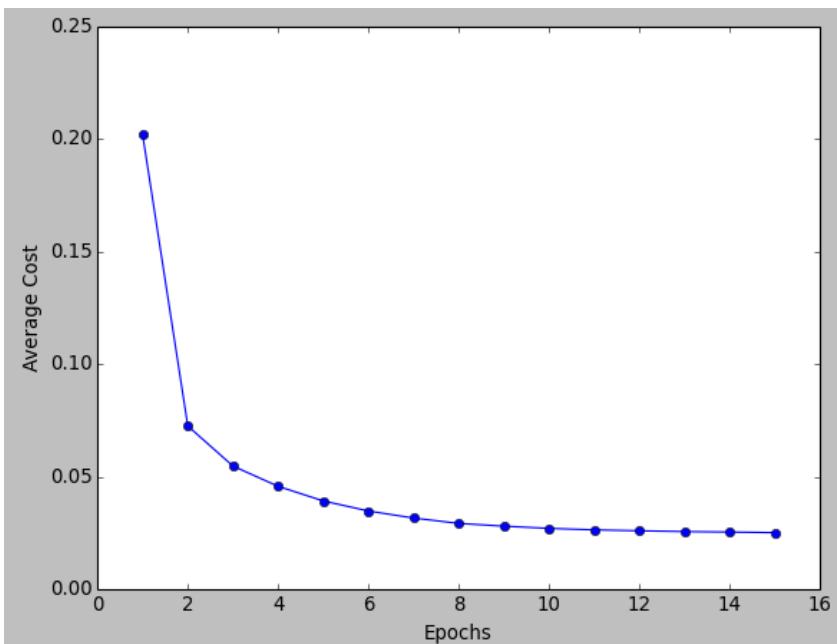
    prediccion = modelo(x) # paso hacia adelante por la red
    with tf.GradientTape() as tape:
        perdida = compute_loss(y, prediccion) # calcula la pérdida

    # actualiza los pesos usando el gradiente:
    grads = tape.gradient(perdida, modelo.trainable_variables)
    optimizador.apply_gradients(zip(grads, modelo.trainable_variables)))
```

# Batch Gradient Descent

*Batch Gradient Descent*: para calcular los incrementos en los pesos de cada paso en el algoritmo de *Descenso de Gradiente* habría que hacer pasar todos los datos por el algoritmo, calcular la función de error y las derivadas parciales. Y esto tendría que repetirse, hasta alcanzar un mínimo de la función.

Esto supone una elevada carga computacional, ya que en la mayoría de los casos el conjunto de datos es muy grande.

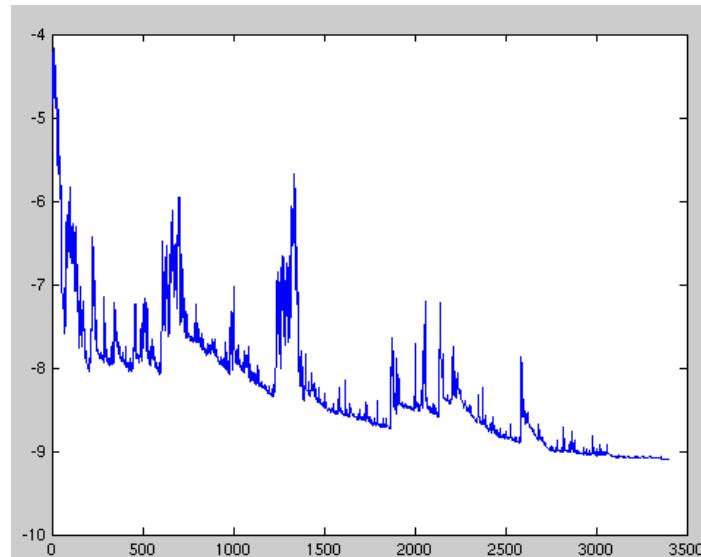


# Descenso de Gradiente Estocástico (SGD)

El optimizador **Descenso de Gradiente Estocástico** (*Stochastic Gradient Descent*) simplifica el cálculo usando solo una muestra aleatoria del *dataset* (con uno o más ejemplos) cada vez que realiza el cálculo del gradiente.

Esto supone que el tiempo necesario para el cálculo es muchísimo menor, aunque el algoritmo irá moviéndose hacia el mínimo de forma menos coherente en cada iteración.

Una ventaja de esta "incoherencia" es que puede resultarle más sencillo escapar de un mínimo local.



# Descenso de Gradiente en Mini-Lotes (MB-GD)

En lugar de alimentar la red con una única muestra, se introducen N muestras en cada iteración.

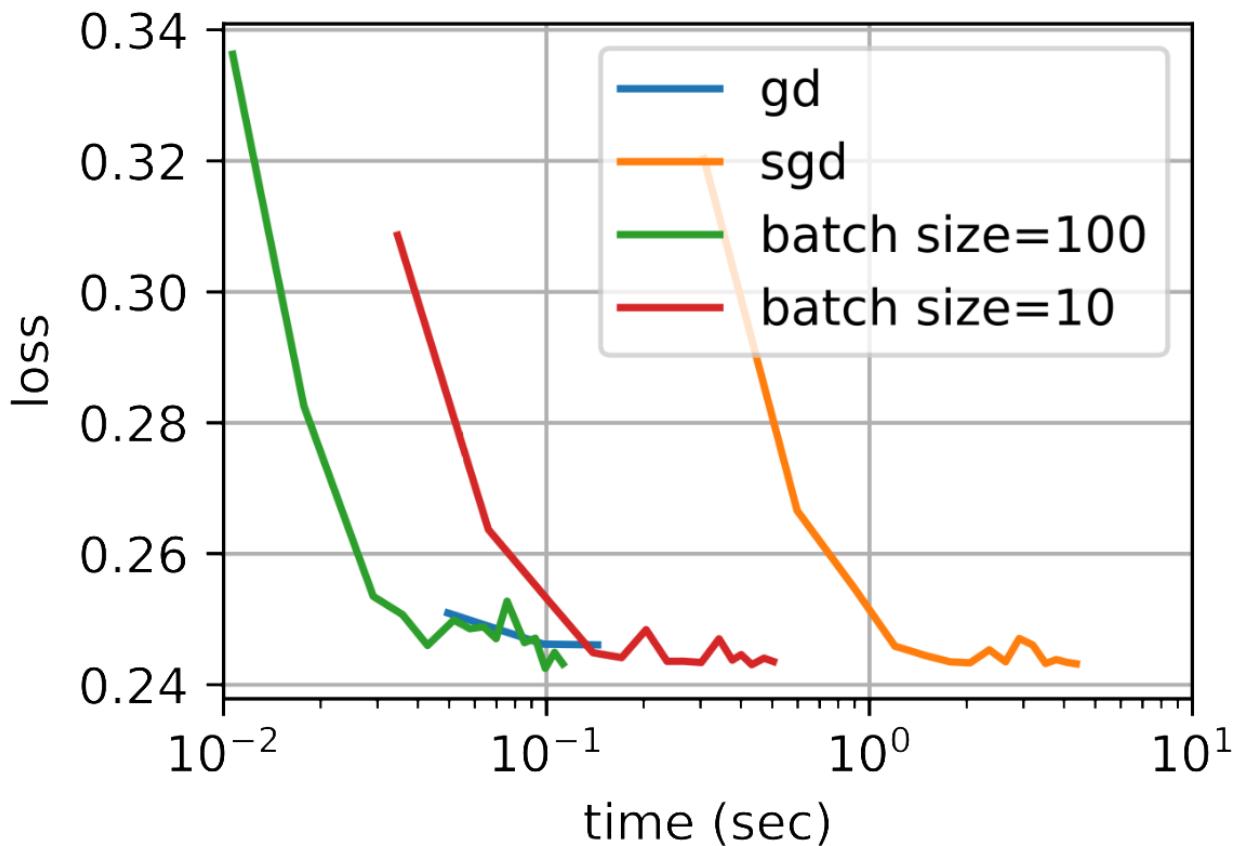
Conserva las ventajas del gradiente estocástico y consigue, además, que el entrenamiento sea más rápido debido a la paralelización de las operaciones.

Este es el enfoque más utilizado ya que proporciona un buen balance entre aleatoriedad y tiempo de entrenamiento.

Este algoritmo converge más rápido que el Descenso de Gradiente Estocástico, pero tiene mayor probabilidad de caer en un mínimo local.

Los tamaños más típicos de los mini-lotes son potencia de dos entre 32 y 1024. El tamaño dependerá de la potencia de cálculo. También es hacer el tamaño proporcional al número de clases en el conjunto de datos.

# Descenso de Gradiente



# Optimizadores: Métodos del Momento

Al *Descenso de Gradiente Estocástico* le cuesta moverse en zonas donde las pendientes de las distintas dimensiones son muy diferentes. El método del **momento** (*momentum*) le permite navegar en las direcciones relevantes y eliminar las oscilaciones que producen las irrelevantes.

De forma intuitiva, el *momentum* acelera el descenso en direcciones similares a las anteriores. Para ello, guarda un vector que representa la media en ventana de los anteriores vectores de descenso, y si el nuevo vector es similar al vector de *momentum* aceleramos su descenso

Para ello, cuando calcula una nueva dirección, se le suma la dirección anterior escalada con un factor que estará entre 0 y 1 (el valor de 0.9 es bastante típico):

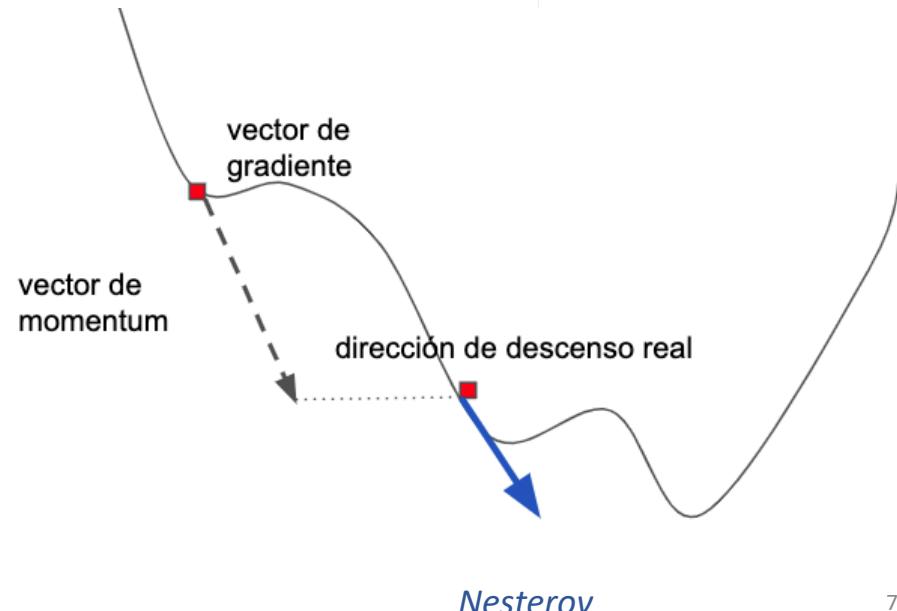
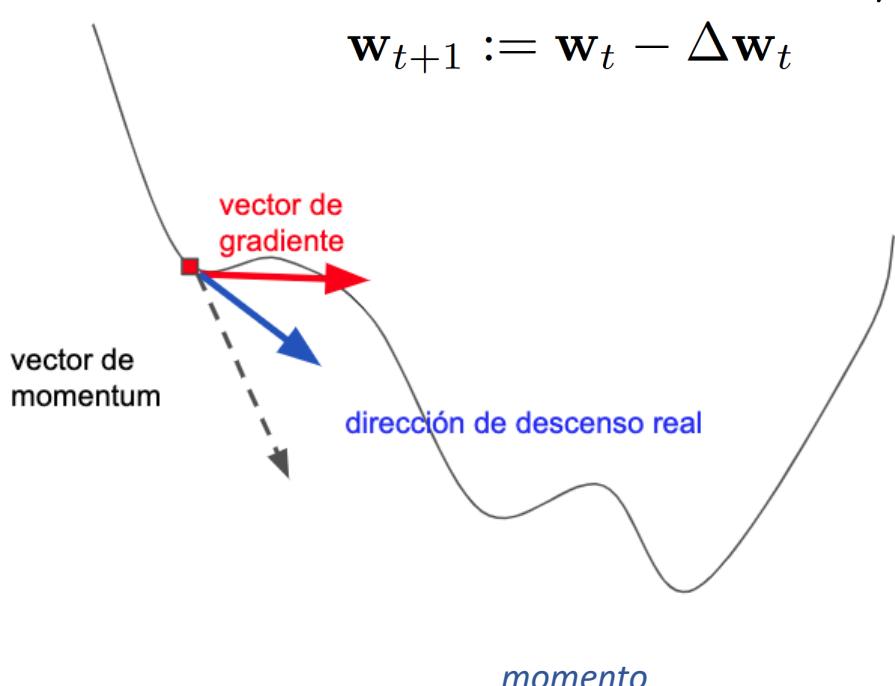
$$\begin{aligned}\Delta \mathbf{w}_t &:= \alpha \cdot \Delta \mathbf{w}_{t-1} + \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_t) \\ \mathbf{w}_{t+1} &:= \mathbf{w}_t - \Delta \mathbf{w}_t\end{aligned}$$

# Optimizadores: Métodos del Momento y Nesterov

El problema es que cuando se está cerca del objetivo, el momento suele ser grande y puede hacer que pierda el mínimo o se oscile alrededor.

Una variante destacable de este método es el algoritmo **Gradiente Acelerado de Nesterov** (*Nesterov Accelerated Gradient*): para el cálculo del descenso, en un primer momento confía en el vector de momentum y una vez descendido en su dirección se calcula el nuevo gradiente desde ese punto.

$$\begin{aligned}\Delta \mathbf{w}_t &:= \alpha \cdot \Delta \mathbf{w}_{t-1} + \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}_t - \alpha \cdot \Delta \mathbf{w}_{t-1}) \\ \mathbf{w}_{t+1} &:= \mathbf{w}_t - \Delta \mathbf{w}_t\end{aligned}$$



# Optimizadores: Algoritmo de Gradiente Adaptativo

**AdaGrad** (*Adaptive Gradient Algorithm*) es una modificación del método de *Descenso de Gradiente Estocástico* que escala de forma adaptable el parámetro de tasa de aprendizaje para garantizar que el proceso de entrenamiento no sea ni demasiado lento ni demasiado volátil e impreciso. Es especialmente útil para redes de dimensión muy grande.

Debido a esto, es **adecuado para datos dispersos** (procesado de lenguaje natural o reconocimiento de imágenes). Otra ventaja es que básicamente **elimina la necesidad de ajustar la tasa de aprendizaje**.

**Cada parámetro tiene su propia tasa de aprendizaje** y debido a las peculiaridades del algoritmo la tasa de aprendizaje decrece de forma monótona. Esto causa el mayor problema: en algún momento la tasa de aprendizaje es tan pequeña que el sistema deja de aprender.

**AdaDelta** es una variación de AdaGrad en la que en vez de calcular el escalado del factor de entrenamiento de cada dimensión teniendo en cuenta el gradiente acumulado desde el principio de la ejecución, se restringe a una ventana de tamaño fijo de los últimos n gradientes.

# Optimizadores: RMSProp

**RMSProp** (*Root Mean Square Propagation*) es método en el que, al igual que AdaGrad, cada peso dispone de su propio valor de tasa de aprendizaje, produciéndose una menor oscilación en el camino hacia el mínimo.

La mayor diferencia con AdaGrad es la forma de calcular la tasa de aprendizaje. Para cada parámetro, se divide la tasa de aprendizaje por la media móvil de las magnitudes de los gradientes recientes de ese parámetro.

La siguiente fórmula calcula un factor de atenuación ( $v_t$ ) de la tasa de aprendizaje del peso en cuestión, como la suma ponderada ( $\rho$ ) del factor de atenuación anterior ( $v_{t-1}$ ) y del cuadrado de la componente del gradiente de la función de coste en la dirección del peso ( $g_t$ ).

$$v_t = \rho \cdot v_{t-1} + (1 - \rho) \cdot g_t^2$$

La segunda fórmula calcula el nuevo peso ( $w_{t+1}$ ) como la suma del anterior ( $w_t$ ) menos el gradiente de la función de coste ( $\nabla w_t$ ), aplicándole una tasa de aprendizaje ( $\eta$ ) atenuada por la raíz cuadrada del factor calculado en la fórmula anterior ( $v_t$ ):

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \epsilon} \cdot g_t$$

# Optimizadores: Adam

**Adam** (*Adaptive momentum estimation*), además de usar tasas de aprendizaje para cada parámetro, también usa separadamente cambios en el momento de cada uno.

Es, por tanto, una combinación de RMSProp y del método del momento. Para cada peso:

$$\begin{aligned}\nu_t &= \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t \\ s_t &= \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2 \\ \omega_{t+1} &= \omega_t - \eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t\end{aligned}$$

$\eta$ : tasa de aprendizaje

$g_t$ : gradiente en dirección del peso

$\nu_t$ : promedio exponencial de los gradientes en la dirección del peso.

$s_t$ : promedio exponencial de los cuadrados de los gradientes en la dirección del peso.

$\beta_1, \beta_2$ : hiperparámetros .

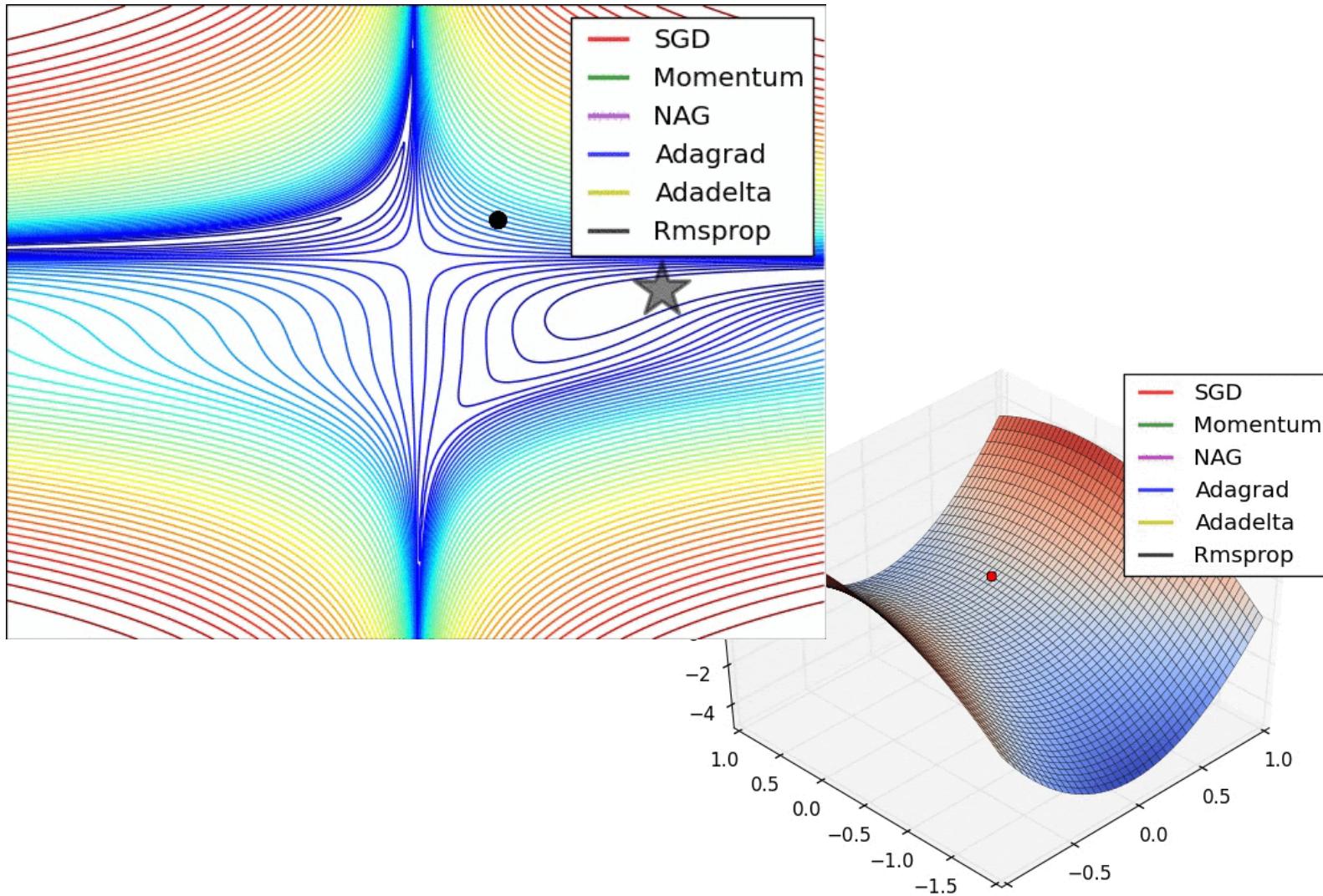
Las dos primeras ecuaciones calculan la media exponencial del gradiente, así como los cuadrados del gradiente para cada parámetro, respectivamente.

La tasa de aprendizaje se escala por el promedio del gradiente y se divide por la raíz del promedio cuadrático del promedio exponencial del cuadrado de los gradientes.

El hiperparámetro  $\beta_1$  se suele establecer cerca de 0.9 y  $\beta_2$  cerca de 0.99.  $\epsilon$  vale típicamente  $10^{-10}$ .

# Optimizadores

Además de los presentados, existen otros métodos, que como hemos visto suelen ser variantes de los métodos básicos basados en el momento y la tasa de aprendizaje.



# Convergencia

Una situación que se puede dar es que el algoritmo circule alrededor de la solución óptima (es decir, el conjunto de pesos con menor coste) sin nunca converger.

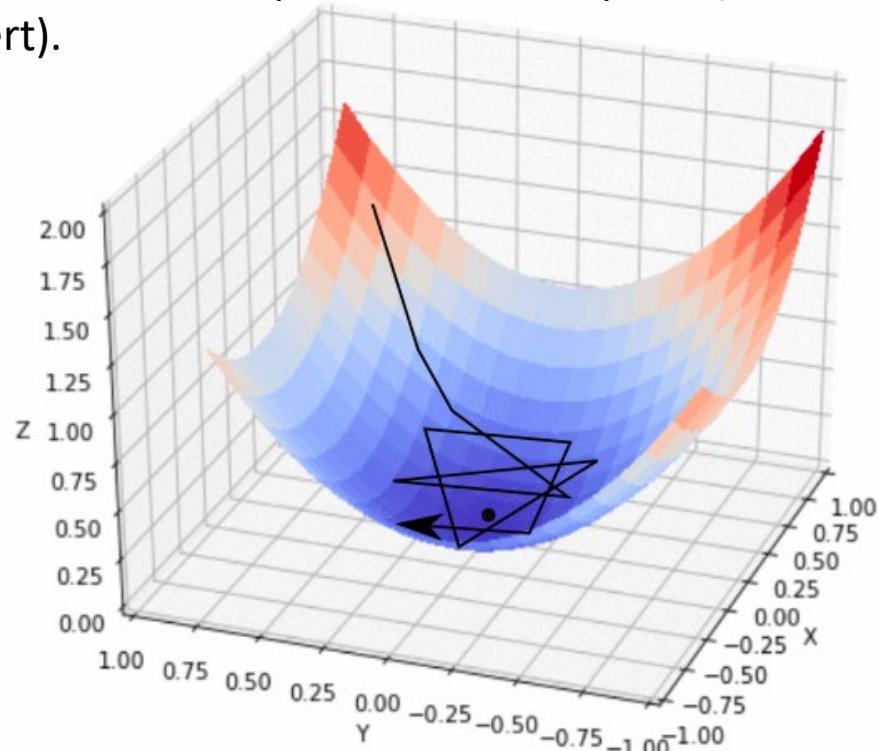
Un síntoma de esta situación es la subida y bajada de la función de coste a lo largo de las repeticiones.

Un método de combatir este problema consiste en combinar los pesos conseguidos hacia el final del proceso de entrenamiento, por medio de un promediado temporal (conocido como promedia Polyak o de Polyak-Ruppert).

La variante más utilizada es la de hacer el promedio aplicando una ponderación que reduce exponencialmente el peso de los pesos más antiguos:

$$v_t = \beta \cdot v_{t-1} + (1 - \beta) \cdot w_t$$

Este método está incluido en buena parte de los optimizadores.



# Guardar y cargar modelos con Keras

## Guardar y cargar el modelo completo

- El método `model.save(ruta)` guarda en un único fichero HDF5 lo siguiente:
  - ✓ Arquitectura.
  - ✓ Pesos.
  - ✓ Configuración del entrenamiento (pérdidas y optimizador).
  - ✓ Estado del entrenamiento, que permite continuarlo desde el mismo punto.
- El método `model.load_model(ruta)` carga el modelo completo.

También compila el modelo usando la configuración de entrenamiento guardada en el fichero (sólo si el modelo estaba compilado cuando se guardó).

```
from keras.models import load_model  
...  
# una vez creado el modelo y, opcionalmente, compilado y entrenado:  
modelo.save('modeloABC.h5') # crea el fichero HDF5  
del modelo # elimina el modelo  
...  
modelo = load_model('modeloABC.h5') # recupera el modelo
```

# Guardar y cargar modelos con Keras

## Guardar y cargar solo la arquitectura de un modelo

Los métodos `model.to_json()` y `model.to_yaml()` representan el modelo como una cadena de texto en formato json o yaml. Son legibles y editables. Sólo es necesario usar uno de ellos.

```
json_string = modelo.to_json() # convertir modelo a texto en formato JSON  
yaml_string = model.to_yaml() # formato YAML (se recomienda JSON)  
with open("modeloEjemplo.json", "w") as json_file: # guardar string en fichero en Python  
    json_file.write(modelo_json)
```

Los métodos `model_from_json(json_string)` y `model_from_yaml(yaml_string)` generan un nuevo modelo a partir de las strings `json` o `yaml` que lo representa.

```
modelo = model_from_json(json_string)  
modelo = model_from_yaml(yaml_string)
```

## Guardar y cargar sólo los pesos del modelo

El método `model.save_weights('ficheroPesos.h5')` guarda los pesos en un fichero con formato HDF5. Para cargarlos, se usa el método `model.load_weights('ficheroPesos.h5')`

```
modelo.save_weights('my_model_weights.h5')  
...  
modelo.load_weights('ficheroPesos.h5')
```

# Regularización

# Datos de entrenamiento y prueba

Un modelo de aprendizaje automático tiene como objetivo realizar buenas predicciones sobre datos nuevos nunca antes vistos. Para ello se divide el conjunto de datos en tres subconjuntos:

- Conjunto de entrenamiento: para entrenar un modelo (~96%).
- Conjunto de validación: para probar el modelo (~2%).
- Conjunto de prueba: para comparar modelos (~2%).

Un buen rendimiento en el conjunto de desarrollo y de prueba es un indicador útil de buen rendimiento en los datos nuevos en general, suponiendo lo siguiente:

- Que sea lo suficientemente grande como para generar resultados significativos desde el punto de vista estadístico.
- Que sea representativo de todo el conjunto de datos. En otras palabras, no se debe elegir un conjunto de prueba con características diferentes al del conjunto de entrenamiento.
- Nunca se deben usar los datos de prueba ni de validación para el entrenamiento

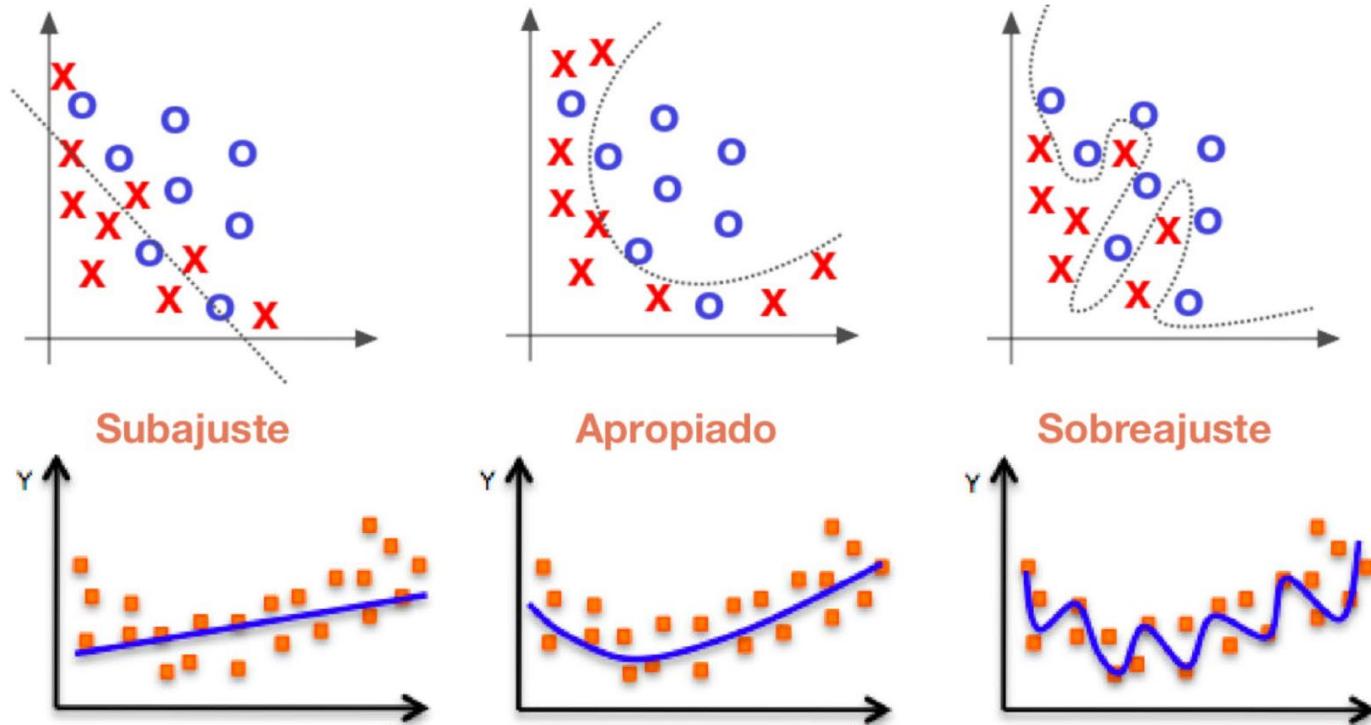
■ entrenamiento ■ validación ■ pruebas (test)



# Sobreajuste y subajuste

Se define **subajuste** (*underfitting*) de un modelo como la incapacidad para capturar la tendencia subyacente de los datos. Típicamente se debe a haber entrenado el sistema con pocos datos o a haber hecho una mala selección de las características a utilizar. También se denominan sistemas con **sesgo alto** (*high bias*).

Se define **sobreajuste** (*overfitting*) como el ajuste excesivo de un modelo a los datos de entrenamiento, dificultando la generalización y, por tanto, la predicción ante datos nuevos. También se denominan sistemas con **varianza alta** (*high variance*).



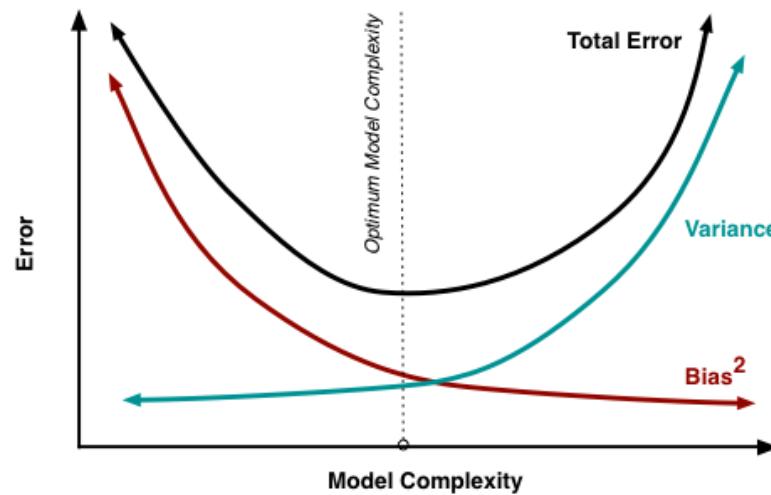
# Regularización

El aumento de las características (*features*) y del número de capas, aumenta el riesgo de sobreajuste, sobre todo cuando no se dispone de una gran cantidad de datos.

Por ello, la mejor forma de reducir el sobreajuste (*overfitting*) es **conseguir más datos** para el entrenamiento. Una técnica bastante utilizada es la del aumento de datos (*data augmentation*).

Cuando esto no es posible, se puede probar a aplicar **técnicas de regularización o a reducir la capacidad de la red**:

- Arquitecturas más reducidas: menos capas, menos unidades por capas, etc.
- Reducción de los pesos: técnicas que reducen los pesos o penalizan la complejidad de la red (L1 y L2), etc.
- Añadir ruido: dropout.



# Regularización: Aumento de los datos *(data augmentation)*

La idea es aplicar diversas transformaciones sobre las entradas originales, obteniendo muestras ligeramente diferentes pero iguales en esencia, lo que permite a la red desenvolverse mejor en la fase de inferencia.

Esta técnica se utiliza mucho en el campo de la visión artificial porque funciona extraordinariamente bien.

Dentro de dicho contexto, una misma imagen de entrada será procesada por la red neuronal tantas veces como repeticiones (*epochs*) ejecutemos en entrenamiento, provocando que la red acabe memorizando la imagen si entrenamos demasiado.

Lo que haremos es aplicar transformaciones de forma aleatoria cada vez que volvamos a introducir la imagen a la red:

- Imagen especular (horizontal o vertical)
- Rotar la imagen X grados.
- Recortar, añadir relleno, redimensionar, ...
- Aplicar deformaciones de perspectiva
- Ajustar brillo, contraste, saturación, ...
- Introducir ruido, defectos, ...
- Combinaciones de las anteriores

# Regularización: Aumento de los datos *(data augmentation)*

De esta forma se dispondrá de más información para entrenamiento sin necesidad de obtener muestras adicionales, y también sin alargar los tiempos.

Lo mejor es que si por ejemplo nuestra red se dedica a clasificar imágenes o detectar objetos, esta técnica conseguirá que el modelo sea capaz de obtener buenos resultados para imágenes tomadas desde distintos ángulos o bajo distintas condiciones de luz.

Por tanto conseguiremos que la red no sobreajuste y que generalice mejor.



# Regularizaciones L1 y L2

La idea básica detrás de estos tipos de regularización consiste en minimizar la pérdida junto con **la complejidad del modelo**:

$$Coste = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + f(w)$$

Si se penaliza la complejidad del modelo, se hace más difícil que el modelo se adapte a cada uno de los ejemplos que le suministramos, por lo que le resulta más complicado memorizarlos.

Las funciones más habituales con las que se mide la complejidad del sistema ( $f$  en la fórmula anterior) son las normas L1 y L2 (norma euclídea):

$$\|\mathbf{w}\|_1 = |w_1| + |w_2| + \dots + |w_N|$$

$$\|\mathbf{w}\|_2 = \left( |w_1|^2 + |w_2|^2 + \dots + |w_N|^2 \right)^{\frac{1}{2}}$$

## Regularización: L2

La idea detrás de este tipo de regularización es reducir el valor de los parámetros para que sean pequeños.

Utiliza el cuadrado de la norma euclídea para medir la complejidad del sistema:

$$\text{Coste L2} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n} \sum_j \omega_j^2 = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n} \sum_{l=1}^L \sum_i \sum_j \left( w_{i,j}^{(l)} \right)^2$$

*suma sobre todas las capas*

Si el nuevo término es demasiado grande, el modelo será demasiado simple y se podría producir un problema de subajuste, ya que la función de pérdidas apenas influiría.

Si, por el contrario, es demasiado pequeño, no se habrá conseguido el objetivo de reducir el sobreajuste.

Por este motivo, se multiplica ese sumando por una constante pequeña, el hiperparámetro  $\lambda$ : 0.1, 0.01, ...

L2 es útil cuando hay características codependientes (por ejemplo, embarazo y género), ya que la codependencia tiende a aumentar la varianza de los coeficientes

$$\omega_{i,j} := \omega_{i,j} - \eta \cdot \left( \frac{\partial L}{\partial \omega_{i,j}} + \frac{2\lambda}{n} \omega_{i,j} \right)$$

# Regularización: L1

Al igual que la regularización L2, busca penalizar la complejidad. En el caso de la regularización L1, se fomenta la dispersión (*sparsity*): que, ante una entrada, se active el menor número posible de neuronas. Dicho de otra forma, que **los pesos de las características que tienen poca información se hagan cero**.

Esto presenta numerosas ventajas en la práctica.

La implementación es similar a la regularización L2, pero se usa la norma 1 en lugar de la norma euclídea:

$$\text{Coste L1} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n} \sum_j |\omega_j|$$

Como conclusión, L1 es útil para selección de características, eliminando las que no tienen influencia en el resultado.

# Regularización: Decaimiento de pesos (weight decay)

La técnica de **decaimiento de pesos** podríamos decir que es similar a la regularización L2, pero, en lugar de introducir la penalización como un sumando en la función de coste, la añadimos como un término extra en la fórmula de actualización de los pesos:

$$\omega_{i,j} := \omega_{i,j} - \eta \cdot \left( \frac{\partial L}{\partial \omega_{i,j}} \right) + \lambda \omega_{i,j}$$

A pesar de que ambas técnicas son muy similares, el decaimiento de pesos obtiene mejores resultados que L2 en optimizadores con gradientes adaptativos (como, por ejemplo, Adam).

A diferencia de L1 y L2, Keras no implementa la regularización por decaimiento de pesos, aunque es posible programarla como extensión de la clase Regularizer.

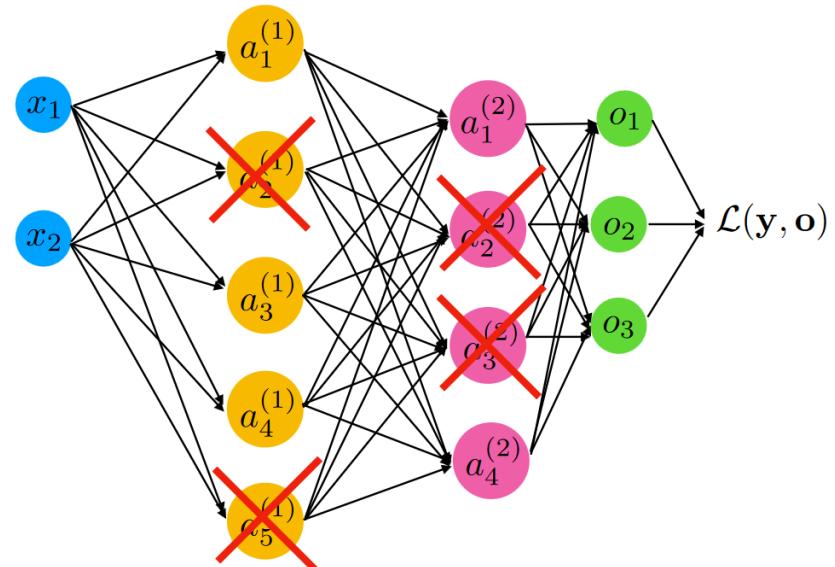
# Regularización: *Dropout*

Por cada nueva entrada a la red en fase de entrenamiento, se desactivará aleatoriamente un porcentaje de las neuronas en cada capa oculta (probabilidad de descarte previamente definida). Dicha probabilidad puede ser igual para toda la red, o distinta en cada capa. Los valores más utilizados para la probabilidad de *dropout* están entre 0.2 y 0.5.

Dado que en fase de inferencia se usan todas las neuronas, se hace necesario compensar esa diferencia, la opción más eficiente es multiplicar los valores de activación en la fase de entrenamiento por un factor de  $1/(1-p)$ , donde  $p$  es la probabilidad de *dropout*. Este es el enfoque llamado **dropout invertido** es el utilizado por la mayoría de los *frameworks* de deep learning.

El principal motivo por el que este método funciona tan bien es porque la red aprende a no depender demasiado de conexiones particulares y de esa forma tendrá en cuenta más conexiones.

Es importante apuntar que, aunque no se tiene sentido aplicarlo si el sistema no sobreajusta, no es infrecuente ver cómo en estos casos se aumenta la capacidad de la red hasta que sobreajusta y luego se aplica *dropout*.



# Normalización y estandarización de los datos

Supongamos un escenario en el cual se dispone de dos entradas a una RN, en la que los valores de la primera están entre 0 y 1 y los de la segunda varían entre 0 y 0.01. Dado que la tarea de la red es aprender cómo combinar ambas entradas por medio de una serie de combinaciones lineales y activaciones no lineales, los parámetros asociados a cada entrada también tendrán escalas distintas. En la práctica esto puede producir que la función de coste dé más importancia a los gradientes de unos parámetros que a los de otros.

Una buena política es tratar de mantener las entradas con valores pequeños, típicamente en el rango de 0 a 1 (**normalización**) o con media cero y desviación estándar 1 (**estandarización**).

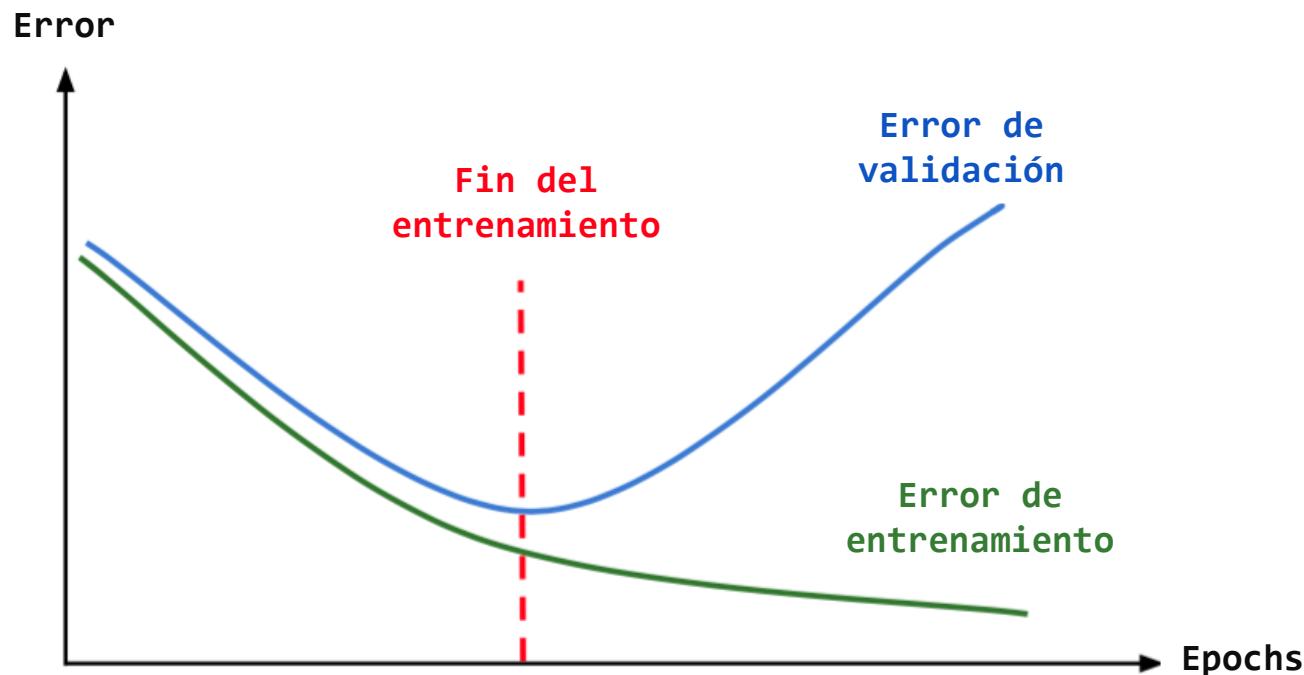
Normalizando o estandarizando las entradas, aceleramos el proceso de aprendizaje.

Además, es práctico que las entradas estén, aproximadamente, en un rango de -1 a 1 para eliminar problemas de precisión de los números en punto flotante, dado que los microprocesadores pierden precisión cuando los operandos son muy grandes o muy pequeños.

Por otra parte, si las entradas y las salidas objetivo están en una escala muy distinta, los parámetros por defecto de la red (como por ejemplo, la tasa de aprendizaje) serán más difíciles de calcular.

# Regularización: Parada temprana (*early stopping*)

Esta técnica monitoriza el error de validación y detiene el entrenamiento cuando detecta que deja de disminuir. Dado que el sobreajuste comienza cuando el error de validación aumenta, esta estrategia se considera un método de regularización.



# Regularización: Parada temprana (*early stopping*)

También se usa esta técnica para no tener que estimar cuántas repeticiones (*epochs*) de entrenamiento se deben aplicar: dejamos de entrenar la red cuando detectemos que el error de validación aumenta de forma sostenida.

```
Train on 84 samples, validate on 28 samples
Epoch 1/200
84/84 - 0s - loss: 0.0181 - accuracy: 1.0000 - val_loss: 0.2464 - val_accuracy: 0.9286
Epoch 2/200
84/84 - 0s - loss: 0.0253 - accuracy: 0.9881 - val_loss: 0.2413 - val_accuracy: 0.9286
Epoch 3/200
84/84 - 0s - loss: 0.0184 - accuracy: 1.0000 - val_loss: 0.2083 - val_accuracy: 0.9286
Epoch 4/200
84/84 - 0s - loss: 0.0146 - accuracy: 1.0000 - val_loss: 0.2026 - val_accuracy: 0.9286
Epoch 5/200
84/84 - 0s - loss: 0.0166 - accuracy: 1.0000 - val_loss: 0.2169 - val_accuracy: 0.9286
Epoch 6/200
84/84 - 0s - loss: 0.0124 - accuracy: 1.0000 - val_loss: 0.2900 - val_accuracy: 0.9286
Epoch 7/200
84/84 - 0s - loss: 0.0337 - accuracy: 0.9881 - val_loss: 0.2882 - val_accuracy: 0.9286
Epoch 8/200
84/84 - 0s - loss: 0.0376 - accuracy: 0.9762 - val_loss: 0.4433 - val_accuracy: 0.8571
Epoch 9/200
84/84 - 0s - loss: 0.1473 - accuracy: 0.9524 - val_loss: 0.3199 - val_accuracy: 0.8929
```

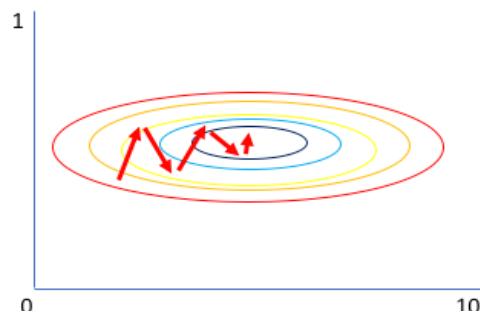
# Escalado y normalización

Si la distribución de los valores de una entrada es normal (gaussiana), debería estandarizarse. En otro caso, debería normalizarse. Si el rango de valores ya es similar a una distribución  $N(0,1)$ , no sería necesario modificarlo.

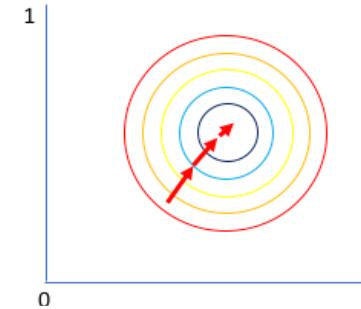
Por ejemplo, cuando la información de entrada son imágenes, se suele escalar el valor de cada canal RGB de los píxeles (típicamente, un entero entre 0 y 255) por un factor 1/255 de forma que la intensidad esté entre 0 y 1.

En los problemas de regresión, reducir la escala de la variable objetivo puede reducir el tamaño de los gradientes usados para actualizar los pesos y, resultar en un proceso de entrenamiento más estable.

La normalización de las entradas solo afecta a la primera capa oculta, para el resto se utiliza otra técnica: **normalización de lotes** (*batch normalization*).



*El gradiente del mayor parámetro domina la actualización*



*Ambos parámetros se actualizan en la misma proporción*

# Normalización de lotes (*batch normalization*)

Dado que normalizar las entradas de la red mejora el entrenamiento de la red, la idea es aplicar la misma estrategia el resto de las capas, y no solo a la capa de entrada.

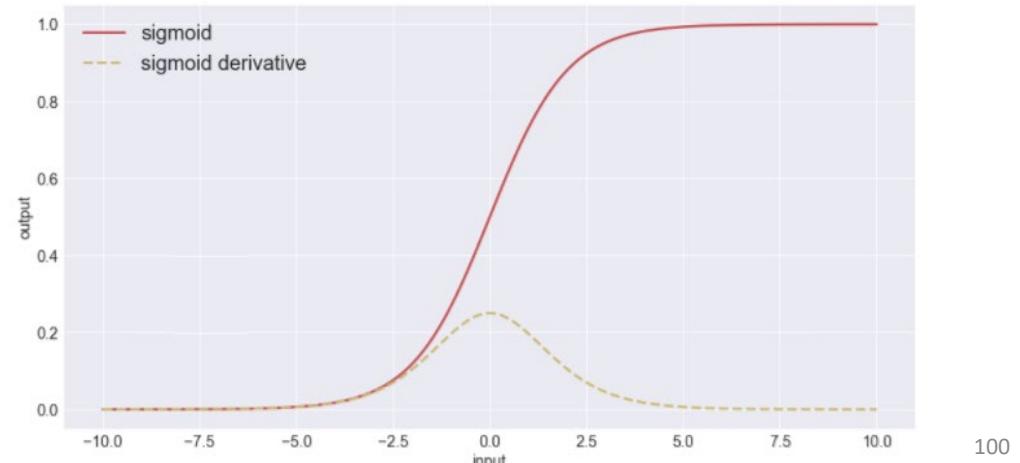
La normalización de lotes consiste básicamente en añadir un paso extra, **justo antes de la función de activación o inmediatamente tras ella**, normalizando las entradas de las capas ocultas.

Lo ideal es que la normalización se hiciera usando la media y la varianza de todo el conjunto de entrenamiento, pero si estamos aplicando el descenso del gradiente por minilotes para entrenar la red, se usará la media y la varianza de cada minilote de entrada.

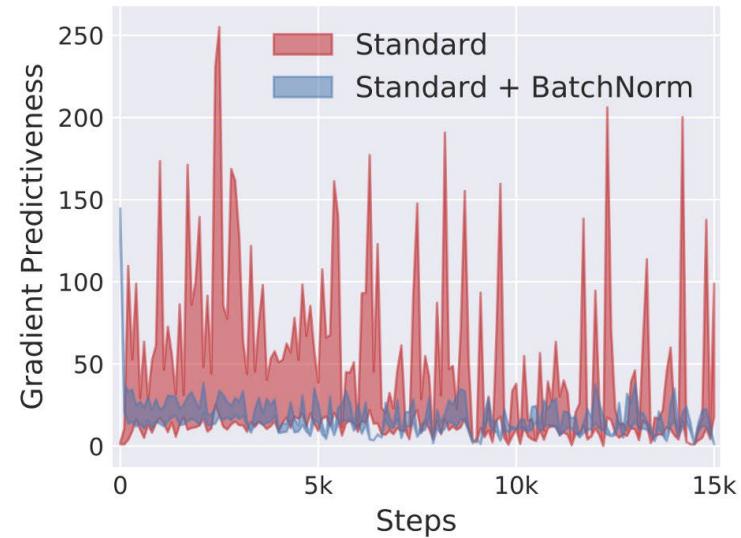
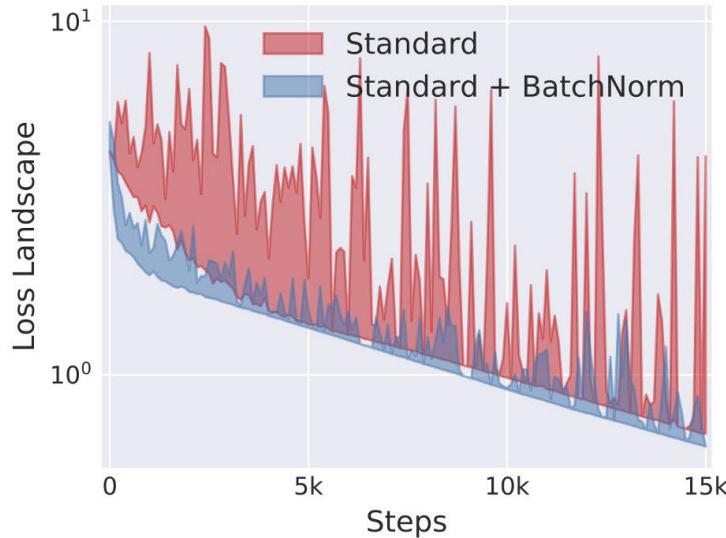
Dado que, durante el entrenamiento los pesos se van a modificar en proporción a la derivada de la función de activación, vemos como en la sigmoide, las zonas de los extremos tienen una derivada casi nula, por lo que los pesos apenas cambiarán. Por ello, esta técnica ayuda a reducir los problemas de desvanecimiento o explosión del gradiente, a la vez que mejora la estabilidad del entrenamiento y a la tasa de convergencia.

El método se puede interpretar y aplicar como capas extra que realizan la normalización.

**La normalización de lotes acelera el aprendizaje y hace el sistema menos sensible a los hiperparámetros.**



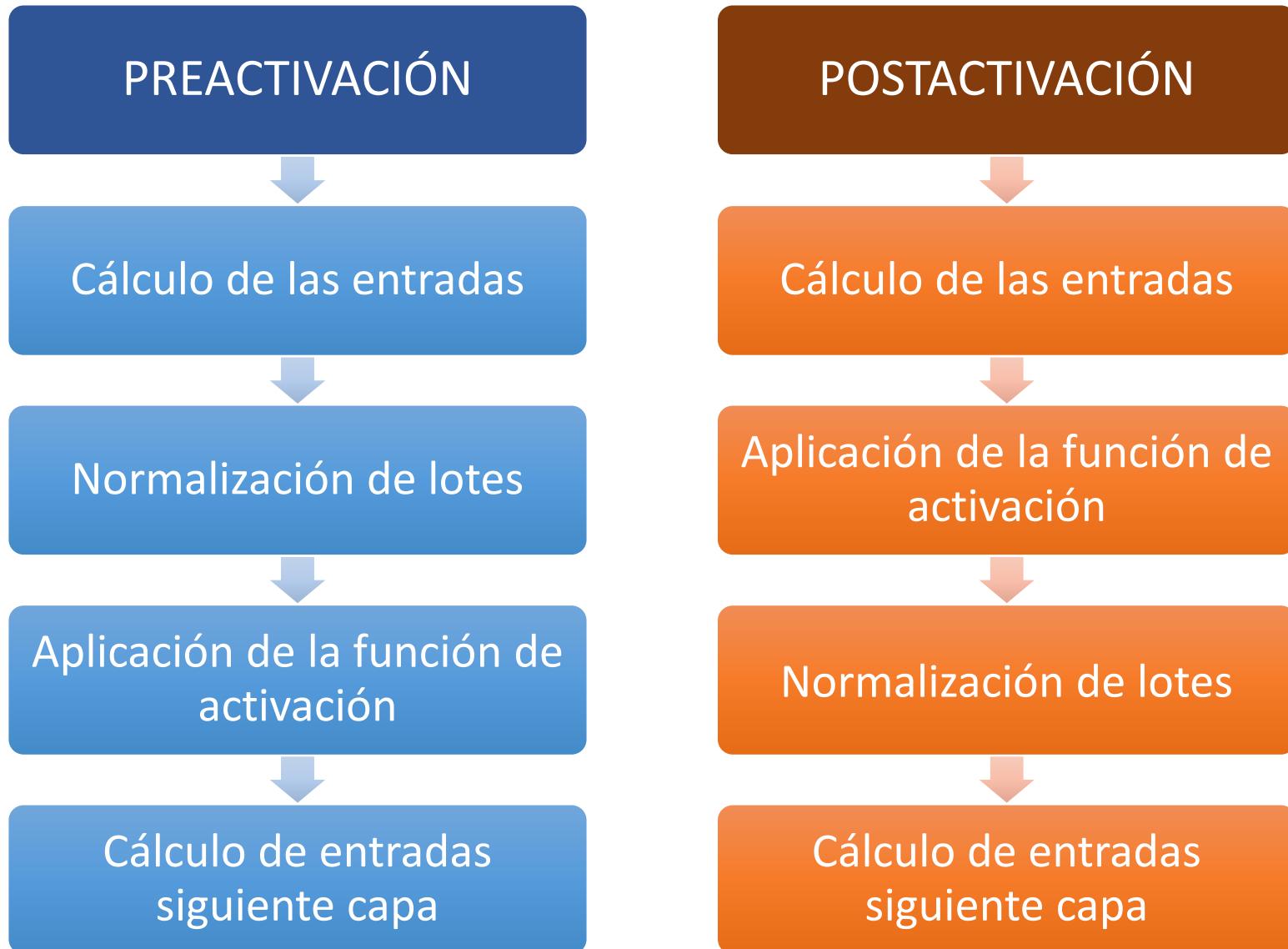
# Regularización: Normalización de lotes (batch normalization)



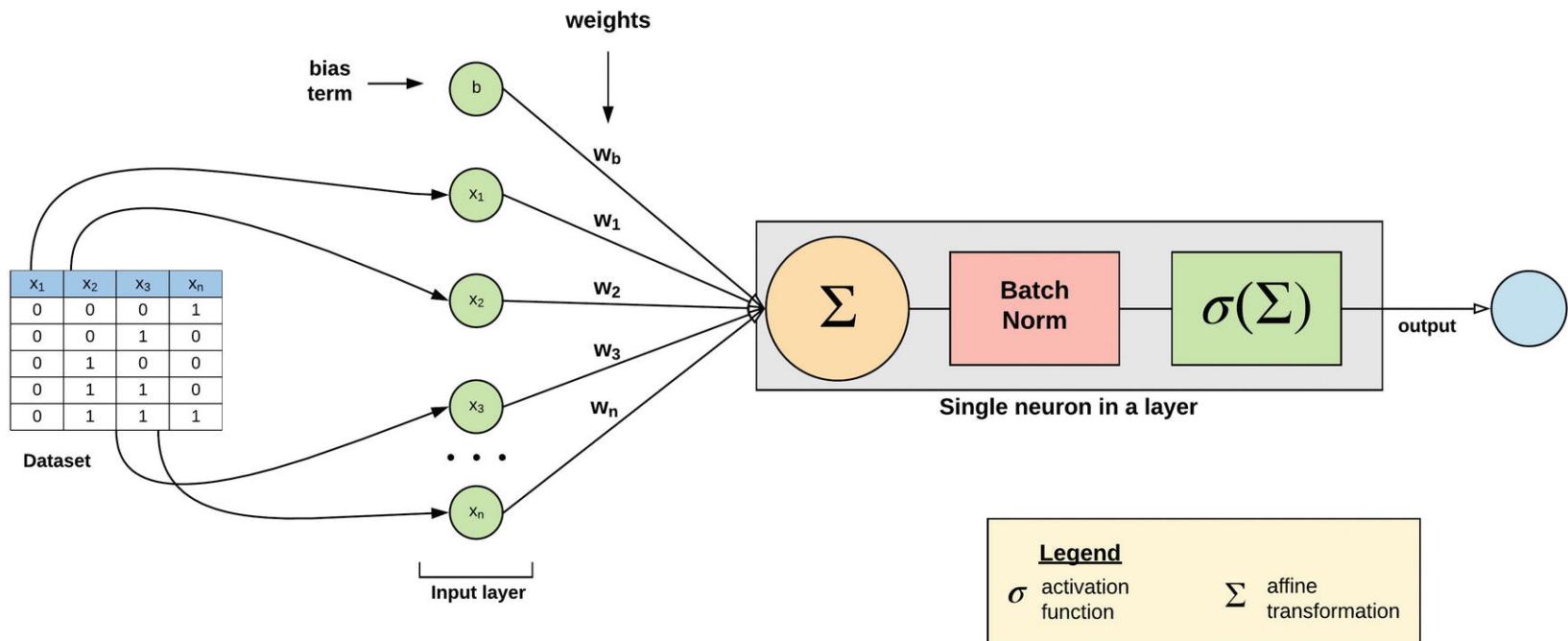
La normalización de lotes incrementa la estabilidad de una red neuronal normalizando la salida de cada capa: le resta la media del lote y la divide por la desviación estándar.

Esta técnica que se verá más adelante, además acelerar el aprendizaje, tiene un efecto regularizador.

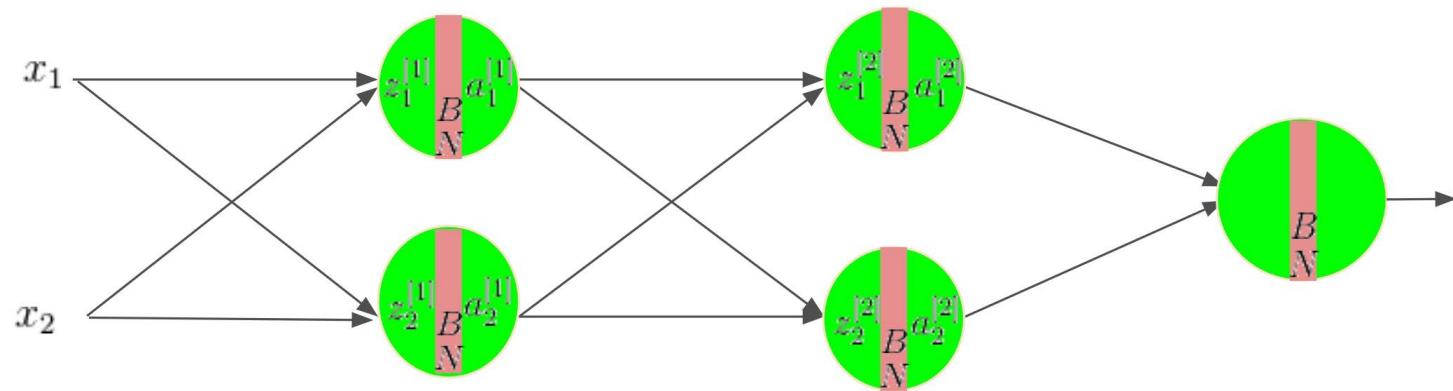
# Normalización de lotes: variantes



# Normalización de lotes antes de la función de activación



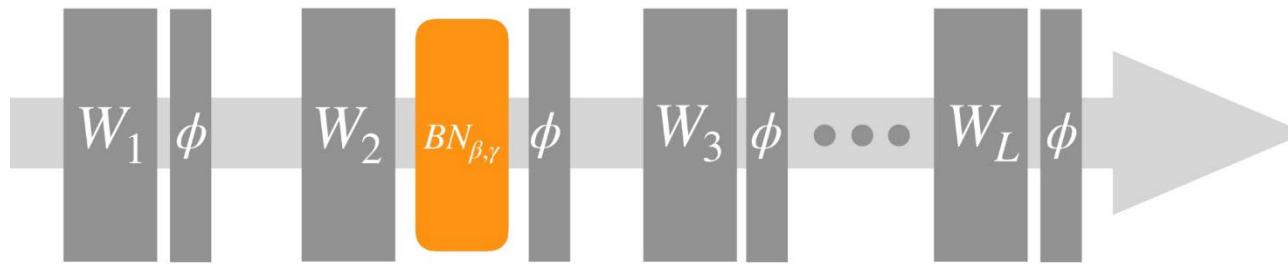
# Normalización de lotes antes de la función de activación



$$\begin{aligned} z^{[l]} = W^{[l]} a^{[l-1]} \longrightarrow & \mu^{[l]} = \frac{1}{m} \sum_i z^{[l](i)} \\ & \sigma^{[l]2} = \frac{1}{m} \sum_i (z^{[l](i)} - \mu^{[l]})^2 \longrightarrow a^{[l]} = g^{[l]}(\tilde{z}^{[l]}) \\ & z_{norm}^{[l](i)} = \frac{z^{[l](i)} - \mu^{[l]}}{\sqrt{\sigma^{[l]2} + \epsilon}} \\ & \tilde{z}^{[l](i)} = \gamma^{[l]} z_{norm}^{[l](i)} + \beta^{[l]} \end{aligned}$$

# Normalización de lotes de la salida de una capa

La salida de cada neurona se normalizará de forma independiente, lo que quiere decir que en cada iteración se calculará la media y la varianza de cada salida para el mini-lote en curso.

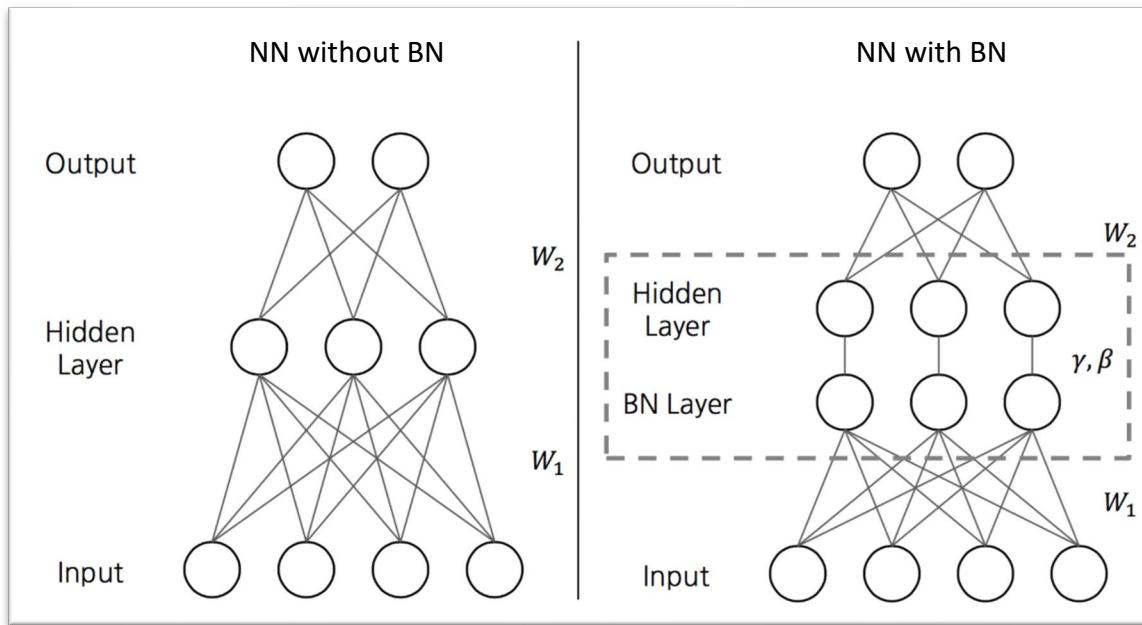


$$BN(y_j)^{(b)} = \gamma \cdot \left( \frac{y_j^{(b)} - \mu(y_j)}{\sigma(y_j)} \right) + \beta$$

A continuación de la normalización se añaden 2 parámetros: un sesgo (*bias*) como sumando, y otra constante similar a un sesgo pero que aparece multiplicando cada activación. Esto se hace para que el rango de la entrada escale fácilmente hasta el rango de salida, lo que ayudará mucho a nuestra red a la hora de ajustar a los datos de entrada, y reducirá las oscilaciones de la función de coste. Como consecuencia de esto podremos aumentar la tasa de aprendizaje (no hay tanto riesgo de acabar en un mínimo local) y la convergencia hacia el mínimo global se producirá más rápidamente.

El método funciona mejor cuanto mayor sea el mini-lote. En la práctica se buscan mini-lotes con un tamaño potencia de 2.

# Normalización de lotes de la salida de una capa



# Normalización de lotes antes y después de la función de activación en Keras

Antes de la función de activación:

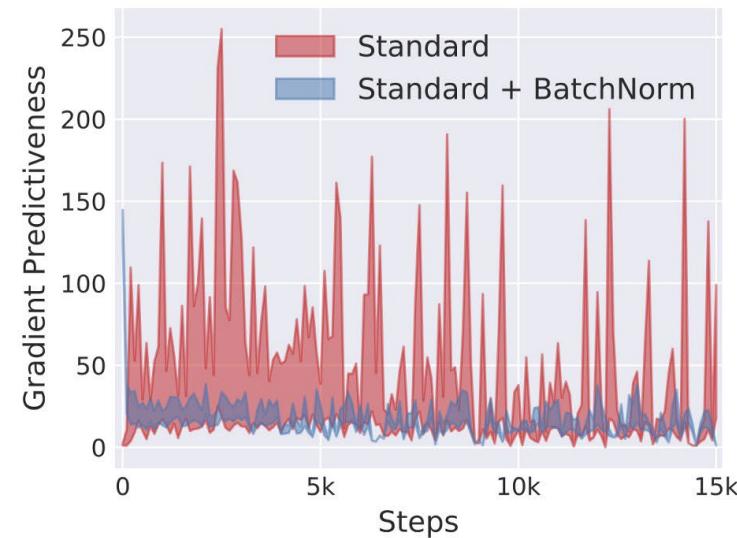
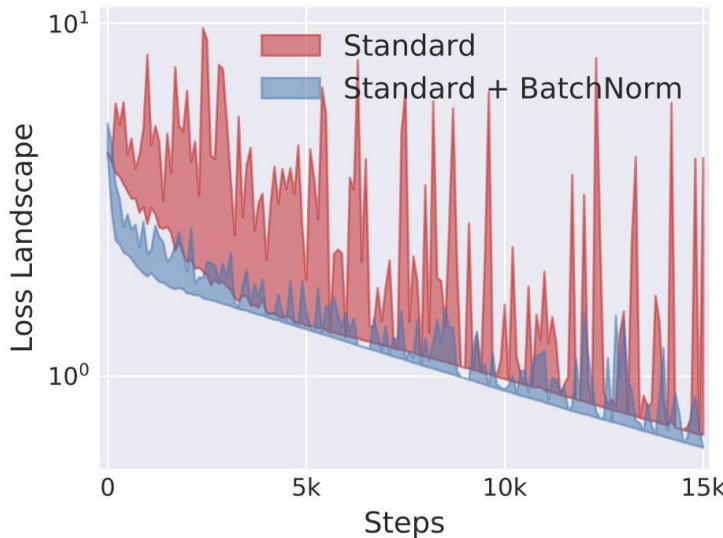
```
modelo = Sequential()  
modelo.add(Dense(32))  
modelo.add(BatchNormalization())  
modelo.add(Activation('relu'))
```

Tras la función de activación:

```
modelo = Sequential  
modelo.add(Dense(32, activation='relu'))  
modelo.add(BatchNormalization())
```

# Normalización de lotes: pautas

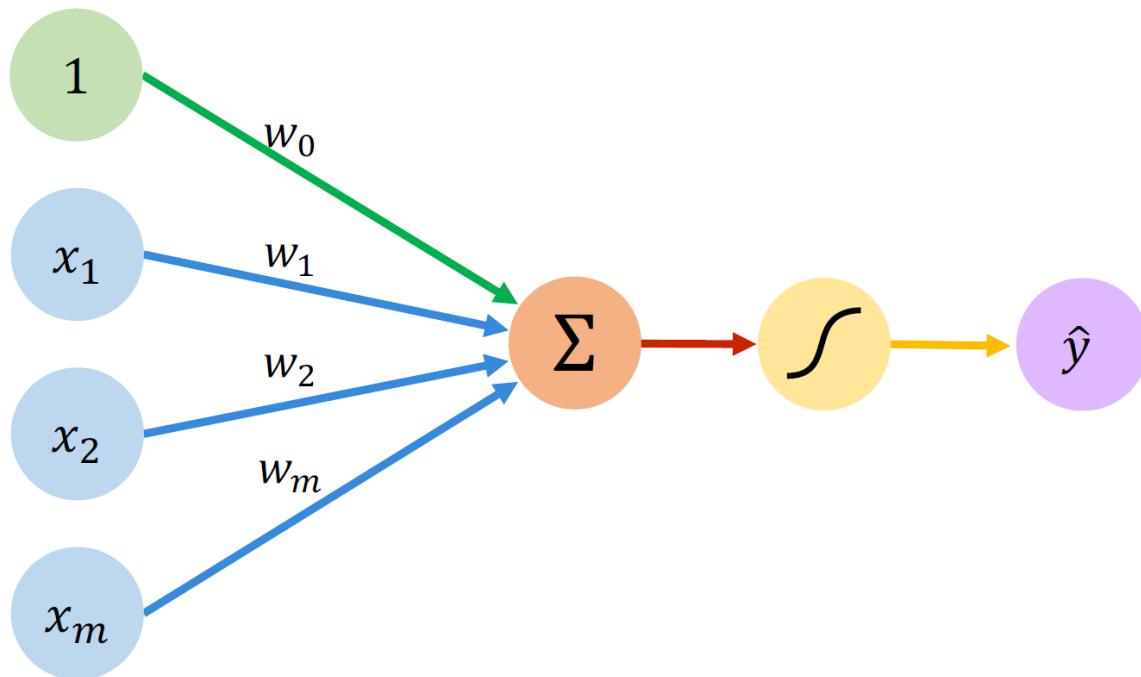
- Se puede usar en casi todo tipo de redes: MLP, CNN, RNN...
- En la práctica suele funcionar mejor la preactivación.
- Se pueden usar tasas de aprendizaje mayores con tasas de decaimiento también mayores.
- El aprendizaje dependerá menos de la técnica de inicialización de pesos.
- No se debe combinar con *Dropout*, dado que la desactivación aleatoria de unidades introduce ruido en los estadísticos usados para la normalización.



# Hiperparámetros y diseño de la red

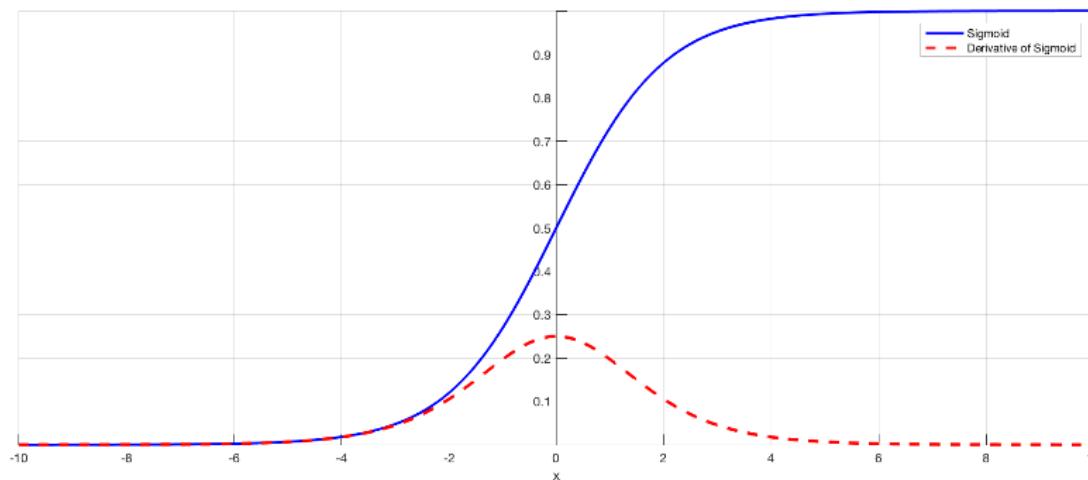
# Inicialización de los pesos

- En la práctica se suele inicializar el peso del sesgo (*bias*) de cada neurona a 0.
- Sin embargo, si los pesos se inicializan a 0, todas las neuronas tendrían el mismo comportamiento.



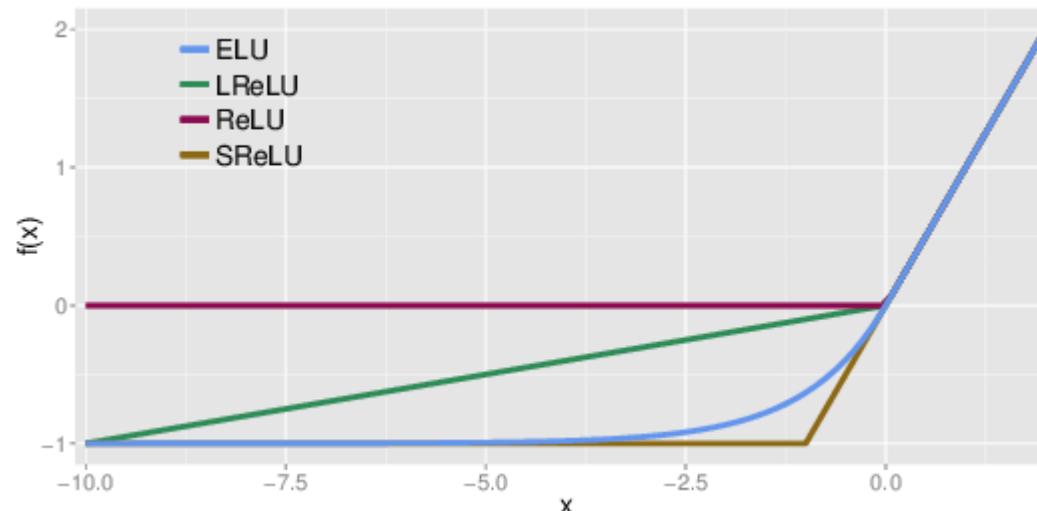
# Inicialización de los pesos

- Si se inicializan con valores muy altos o muy bajos, dependiendo de las funciones de activación que se usen, podría provocar que el aprendizaje fuese lento.
- Por ejemplo, en el caso de la función *sigmoide*, los valores extremos de la entrada corresponden a salidas en las zonas planas, por lo que la derivada será muy baja y, por tanto, los cambios lentos (desvanecimiento del gradiente).



# Inicialización de los pesos (versiones de Keras)

- La técnica de inicialización **He (o Kaiming)** se aplica cuando se usa ReLU o Leaky ReLU. Los pesos son valores aleatorios con distribución gaussiana de media 0 y varianza  $2/n$ , donde  $n$  es el número de entradas de la capa (el peso del sesgo se inicializa a 0).
- La técnica **Xavier** (también llamada **Glorot**) se aplica con activación  $tanh$ . La versión normal usa una distribución gaussiana de media 0 y varianza  $2(n_{ent} + n_{sal})^{-1}$  donde  $n_{ent}$  es el número de entradas de la capa y  $n_{sal}$  el de salidas. Existe otra versión (**Glorot uniforme**) donde la distribución es uniforme en el intervalo  $\pm\sqrt{6/(n_{ent} + n_{sal})}$
- Cuando la función de activación es SELU o ELU, se suele utilizar la inicialización **LeCun**, donde la media es 0 y la varianza es  $1/n_{ent}$ , donde  $n_{ent}$  es el número de entradas a la capa.



## Diseño de la red:

# Selección del optimizador y la función de pérdidas

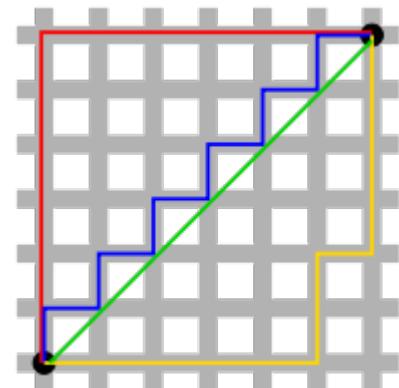
El objetivo de la optimización es calcular eficientemente los pesos que minimicen la función de pérdidas.

En muchos casos, la función de pérdidas mide la distancia. Se puede definir la distancia entre dos puntos de datos de múltiples formas. Esta definición dependerá del tipo de datos y del problema a resolver. Por ejemplo, cuando se procesa lenguaje natural (NPL), la distancia más usada es la de *Hamming* (mide cuántos componentes son distintos entre dos datos dados).

Otras medidas típicas son la distancia euclídea y la distancia Manhattan, que se define como la suma de las distancias en cada dimensión:

### Funciones de pérdida

- MSE: típica en regresión.
- Entropía cruzada categórica: para clasificación multiclase.
- Entropía cruzada binaria: para clasificación binaria.



# Recorte y escalado de gradiente (gradient clipping and scaling)

- Explosión del gradiente: cuando los valores de actualización de los pesos son excesivamente grandes, existe el peligro de que estos excedan el rango (*overflow*) o la precisión (*underflow*) del tipo de dato usado.
- Este problema se puede reducir con una tasa de aprendizaje pequeña, variables objetivo escaladas y una función de pérdidas estándar, pero no es totalmente evitable.
- La técnica de recorte del gradiente limita los valores del gradiente antes de aplicarlos. El recorte de gradiente ayuda a garantizar la estabilidad numérica y previene el crecimiento excesivo de gradientes.
- Este método se usa muy habitualmente con redes RNN.
- Una técnica similar, aunque menos utilizada, es la de escalado del gradiente, la cual normaliza el vector de gradiente de modo que su magnitud (norma) sea igual a un valor determinado, como por ejemplo 1.0.

# Ajuste de la tasa de aprendizaje

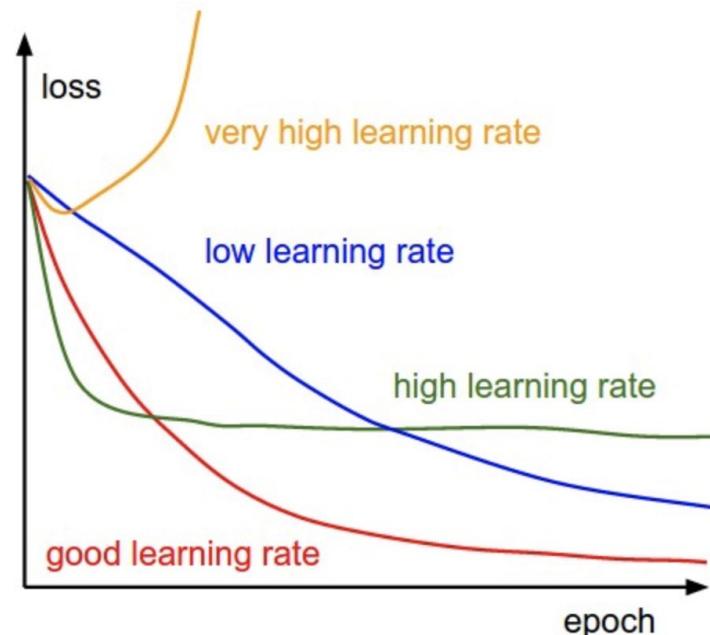
Algunos de los hiperparámetros más importantes que se pueden optimizar en SGD son la tasa de aprendizaje, el momento, el decaimiento de pesos regularización y el parámetro de Nesterov.

La tasa de aprendizaje controla el peso al final de cada lote, y el momento controla cuánto dejar que la actualización anterior influya en la actualización de peso actual.

El decaimiento de pesos indica la disminución de la tasa de aprendizaje sobre cada actualización, y el parámetro de Nesterov toma el valor "True" o "False" dependiendo de si queremos aplicar el *momento de Nesterov*.

Los valores típicos de esos hiperparámetros so

- Tasa de aprendizaje: 0.01
- Decaimiento de pesos:  $-10^{-6}$
- Momento: 0.9
- Nesterov: True.



# Diseño de la red: Inicialización de hiperparámetros

Simplemente como orientación, se muestran a continuación recomendaciones y valores típicos de los hiperparámetros de las redes:

## Hiperparámetros de aprendizaje:

- Tasa de aprendizaje inicial: 0.01.
- Tamaño de los mini-lotes: 32 muestras (depende del nº de procesadores).
- Número de iteraciones de entrenamiento: fijarlo muy alto y usar *early stopping*.
- Momento: 0.9.
- Hiperparámetros distintos para cada capa: posible pero muy poco utilizado.

## Hiperparámetros del modelo:

- Número de nodos: Usar modelos grandes con regularización.
- Regularización de pesos: L2 para penalizar pesos grandes.
- Regularización de actividad: penalizar activaciones de muchas neuronas, es decir, buscar dispersión (*sparsity*) por medio de la regularización L1.
- Función de activación: ReLU o variantes.
- Inicialización de pesos: multiplicar el valor aleatorio obtenido de una distribución normal estándar por  $(2/n)^{0.5}$  (n es el tamaño de la capa anterior).
- *Dropout*: 0.1 para la capa de entrada y entre 0.5 y 0.8 para las internas.

# Diseño de la red: Ajuste de hiperparámetros

Además de los valores típicos visto en la transparencia anterior, para la tasa de aprendizaje, se recomienda tener en cuenta lo siguiente:

- Si el mejor valor se encuentra en el límite del intervalo de búsqueda, es recomendable expandir la búsqueda en esa dirección.
- Debería considerarse buscar en una escala logarítmica, por lo menos inicialmente (por ejemplo, 0.1, 0.01, 0.001, etc.).

## Estrategias de búsqueda de hiperparámetros:

- Tratar de ajustar cada hiperparámetro secuencial e independientemente del resto.
- Búsqueda multi-resolución: aumentar la precisión de forma iterativa.
- Definir una rejilla con todos los hiperparámetros y testear por coordenadas.

Estas tres estrategias se pueden usar de modo independiente o en conjunto.

El método de búsqueda en rejilla (*grid search*), aunque es lento, es el más utilizado, dado que es fácilmente paralelizable. Además, se suele aplicar combinado con la búsqueda iterativa multi-resolución.

Una variante interesante consiste en definir los intervalos del *grid* de forma aleatoria en lugar de uniforme. La distribución de estos valores aleatorios podría estar sesgada en favor de los valores más típicos o esperables.

# Optimización en grid de hiperparámetros

- **Scikit-learn** es una biblioteca para aprendizaje automático de software libre para Python. Incluye múltiples algoritmos, entre ellos, clasificación y regresión.
- Los modelos creados con Keras pueden usarse con scikit-learn usando las clases **KerasClassifier** y **KerasRegressor**.
- Para ello es necesario crear el modelo, devolverlo dentro de una función y pasar esa función a las clases **KerasClassifier** y **KerasRegressor**.
- Los argumentos de la función que se quieran usar (como por ejemplo el número de repeticiones o *epochs*) se deben declarar pasar también a la clase en cuestión:

```
def crea_modelo(dropout=0.0):  
    ...  
    return model
```

```
model = KerasClassifier(build_fn=crea_modelo, dropout=0.2)
```

# Optimización en *grid* de hiperparámetros

- La clase **GridSearchCV** de **scikit-learn** implementa la optimización de hiperparámetros en *grid*.
- Al construir esta clase debe proporcionar en el argumento *param\_grid* un diccionario (o lista de diccionarios) con los hiperparámetros a evaluar: cada clave es el nombre de un hiperparámetro a optimizar y el valor asociado a esa clave es una matriz de valores a probar.
- Por defecto se optimiza la exactitud (*accuracy*), pero se pueden especificar otras en el argumento *score* del constructor de la clase *GridSearchCV*.
- Por defecto, la búsqueda de cuadrícula solo usará un hilo. Al establecer el argumento *n\_jobs* a -1 en el constructor de *GridSearchCV*, el proceso usará todos los núcleos en el equipo.
- A continuación, el proceso *GridSearchCV* construirá y evaluará un modelo para cada combinación de parámetros.

# Optimización en grid de hiperparámetros

```
grid_params = dict(epoches=[10,20,30], batch_size = [16, 32])
grid = GridSearchCV(estimator=modelo, param_grid=grid_params,
                     n_jobs=-1)

grid.fit(X, Y)
print("Mejor combinación de parámetros:", grid.best_params_)
```

- El método *fit* realiza la búsqueda y devuelve el resultado en un objeto.
- El argumento *n\_jobs* indica el número de trabajos en paralelo a utilizar. El valor -1 solicita el uso de todos los procesadores.
- El atributo **best\_score\_** da la mejor puntuación observada: **grid.best\_score\_**
- El atributo **best\_params\_** da la combinación de parámetros que dio la mejor puntuación: **grid.best\_params\_**

# Funciones de respuesta en Keras (*callbacks*)

- Las *callbacks* son clases preprogramadas que se aplicarán en determinadas etapas del entrenamiento (*fit*), la evaluación (*evaluate*) o la predicción (*predict*).
- Se puede utilizar *callbacks*, por ejemplo, para obtener una vista de los estados internos y las estadísticas del modelo durante el entrenamiento.
- Las *callbacks* de Keras ayudan a localizar errores de programación más rápidamente, a construir mejores modelos, etc.
- Permiten visualizar como se va desenvolviendo el entrenamiento y ayudar a prevenir el sobreajuste, implementando la finalización anticipada (*early stopping*) o personalizando la tasa de aprendizaje en cada iteración.
- Se puede usar más de una *callback* simultáneamente.
- Se pueden escribir *callbacks* a medida.
- Todas las *callbacks* tienen acceso al valor de las pérdidas y las métricas utilizadas al final del *batch* o *epoch* (diccionario *logs*).
- Además, tienen acceso a la red por medio del atributo `self.model`: Este atributo permite, por ejemplo, interrumpir el entrenamiento (`self.model.stop_training=True`), cambiar hiperparámetros (por ejemplo, `self.model.optimizer.learning_rate`), etc.

# Funciones de respuesta en Keras (*callbacks*)

- Las siguientes son las *callbacks* predefinidas más importantes:
  - **CSVLogger**: escribe un fichero de log en formato CSV con la información de las repeticiones, exactitud y pérdida, para poder analizarlo posteriormente.
  - **ModelCheckpoint**: guarda el modelo en formato *hdf5* en el disco tras cada *epoch*. Se puede especificar un fichero distinto para cada *epoch*. El fichero puede contener las pérdidas o la exactitud (*accuracy*) como parte del nombre del fichero. Esto es especialmente útil cuando las repeticiones (*epochs*) tardan mucho en completarse.
  - **EarlyStopping**: hace que el entrenamiento termine cuando se detecte sobreajuste. Se puede escoger distintas métricas para decidir el final del entrenamiento.
  - **ReduceLROnPlateau** : permite reducir la tasa de aprendizaje cuando una métrica ha dejado de mejorar.
  - **LearningRateScheduler**: permite especificar la tasa de aprendizaje a utilizar en función de la iteración (*epoch*).
  - **TensorBoard**: escribe ficheros de log que se podrán visualizar posteriormente con la herramienta de visualización *TensorBoard*.
  - **LambdaCallback**: permite crear funciones de *callback* sencillas (si no, se pueden crear funciones más complejas heredando de la clase `keras.callbacks.Callback`).

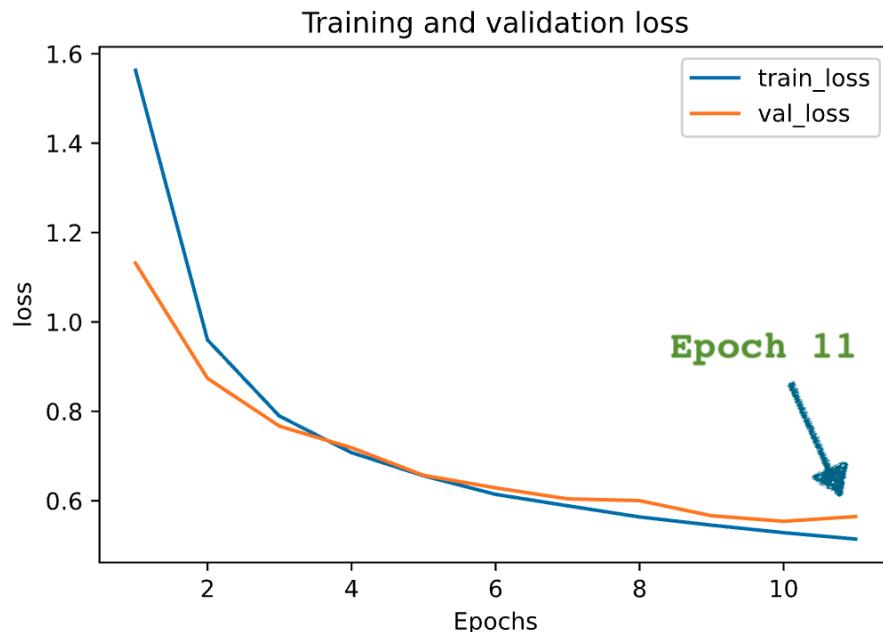
# Callbacks: EarlyStopping

Finaliza el entrenamiento cuando se detecte sobreajuste. Se puede escoger distintas métricas para decidir el final del entrenamiento.

```
from tensorflow.keras.callbacks import EarlyStopping  
early_stopping = EarlyStopping()  
...  
historial = modelo.fit(X_entr, y_entr, epochs=50, validation_split=0.20,  
                       batch_size=64, verbose=2, callbacks=[early_stopping])
```

# Callbacks: EarlyStopping

```
Epoch 1/50
8000/8000 - 6s - loss: 1.5632 - accuracy: 0.5504 - val_loss: 1.1315 - val_accuracy: 0.6605
.....
.....
Epoch 10/50
8000/8000 - 2s - loss: 0.5283 - accuracy: 0.8213 - val_loss: 0.5539 - val_accuracy: 0.8170
Epoch 11/50
8000/8000 - 2s - loss: 0.5141 - accuracy: 0.8281 - val_loss: 0.5644 - val_accuracy: 0.7990
```



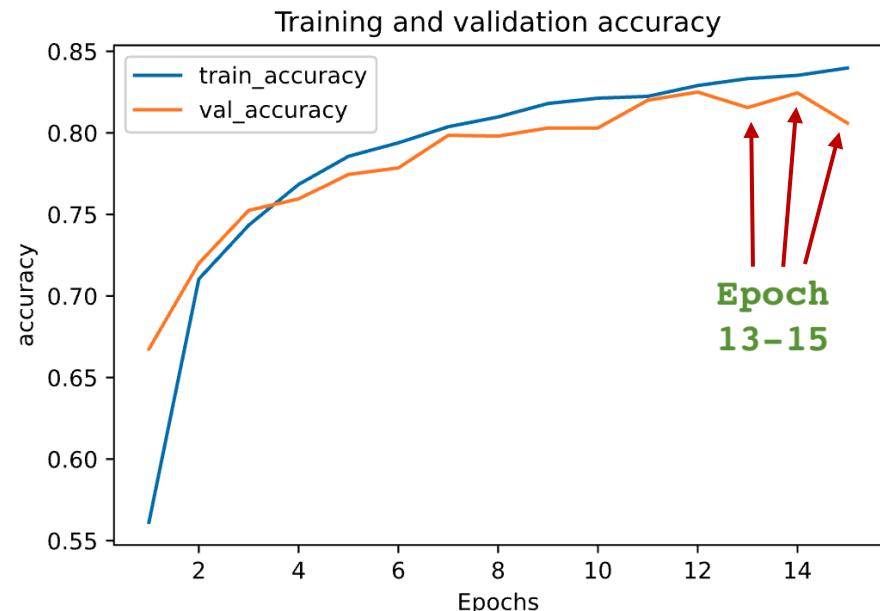
# Callbacks: EarlyStopping

## ARGUMENTOS:

- **monitor**: Métrica a monitorizar. Valor por defecto: 'val\_loss'.
- **patience**: Número de *epochs* sin mejora antes de terminar. Valor por defecto: 0.
- **min\_delta**: Cambio mínimo (valor absoluto) de la métrica que se considera mejora.
- **mode**: Valores posibles:
  - 'min': para cuando la métrica ha dejado de decrecer.
  - 'max': para cuando la métrica ha dejado de crecer.
  - 'auto': la dirección la deduce de la métrica utilizada

## Ejemplo:

```
es = EarlyStopping(  
    monitor='val_accuracy',  
    patience=3,  
    min_delta=0.001,  
    mode='max')
```



# Callbacks: CSVLogger

Escribe un fichero de log en formato CSV con la información de las repeticiones, exactitud y pérdida, para poder analizarlo posteriormente.

```
from tensorflow.keras.callbacks import CSVLogger
csv_log = CSVLogger("resultados.csv")
historial = modelo.fit(X_entr, y_entr, epochs=10, validation_split=0.20,
                       batch_size=64, verbose=2, callbacks=[csv_log])
...
// tras el entrenamiento se puede leer el fichero, con pandas p. ej.
pd.read_csv("resultados.csv", index_col='epoch')
```

# Callbacks: CSVLogger

epoch	accuracy	loss	val_accuracy	val_loss
0	0.540375	1.632193	0.6285	1.190760
1	0.688250	0.999455	0.6965	0.903381
2	0.728125	0.818999	0.7280	0.791272
3	0.755125	0.733289	0.7595	0.725264
4	0.772500	0.678768	0.7810	0.681306
5	0.789125	0.638452	0.7860	0.648267
6	0.796750	0.606171	0.7900	0.620067
7	0.806375	0.579147	0.7780	0.625771
8	0.810000	0.559040	0.7945	0.593174
9	0.816375	0.541813	0.8065	0.569976

Fichero  
generado por  
el ejemplo  
anterior:

Además del nombre del fichero, se le puede indicar, opcionalmente, el separador y el modo 'append' (añade al fichero en lugar de sobreescribir):

```
csv_log = CSVLogger(filename, separator=',', append=False)
```

# Callbacks: ReduceLROnPlateau

Permite reducir la tasa de aprendizaje cuando una métrica ha dejado de mejorar.

El argumento *patience* se utiliza para indicar el número máximo de *epochs* sin mejora antes multiplicar la tasa de aprendizaje por el factor indicado por el argumento *factor*:

$$\text{tasa\_nueva} = \text{tasa\_anterior} * \text{factor}$$

```
from tensorflow.keras.callbacks import ReduceLROnPlateau
ReducLROP = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                               patience=2, min_lr=0.001, verbose=2)
...
historial = modelo.fit(X_entr, y_entr, epochs=50, validation_split=0.2,
                       batch_size=64, verbose=2, callbacks=[reduce_lr])
```

- El argumento *min\_lr* establece el límite inferior de la tasa de aprendizaje

# Callbacks: ReduceLROnPlateau

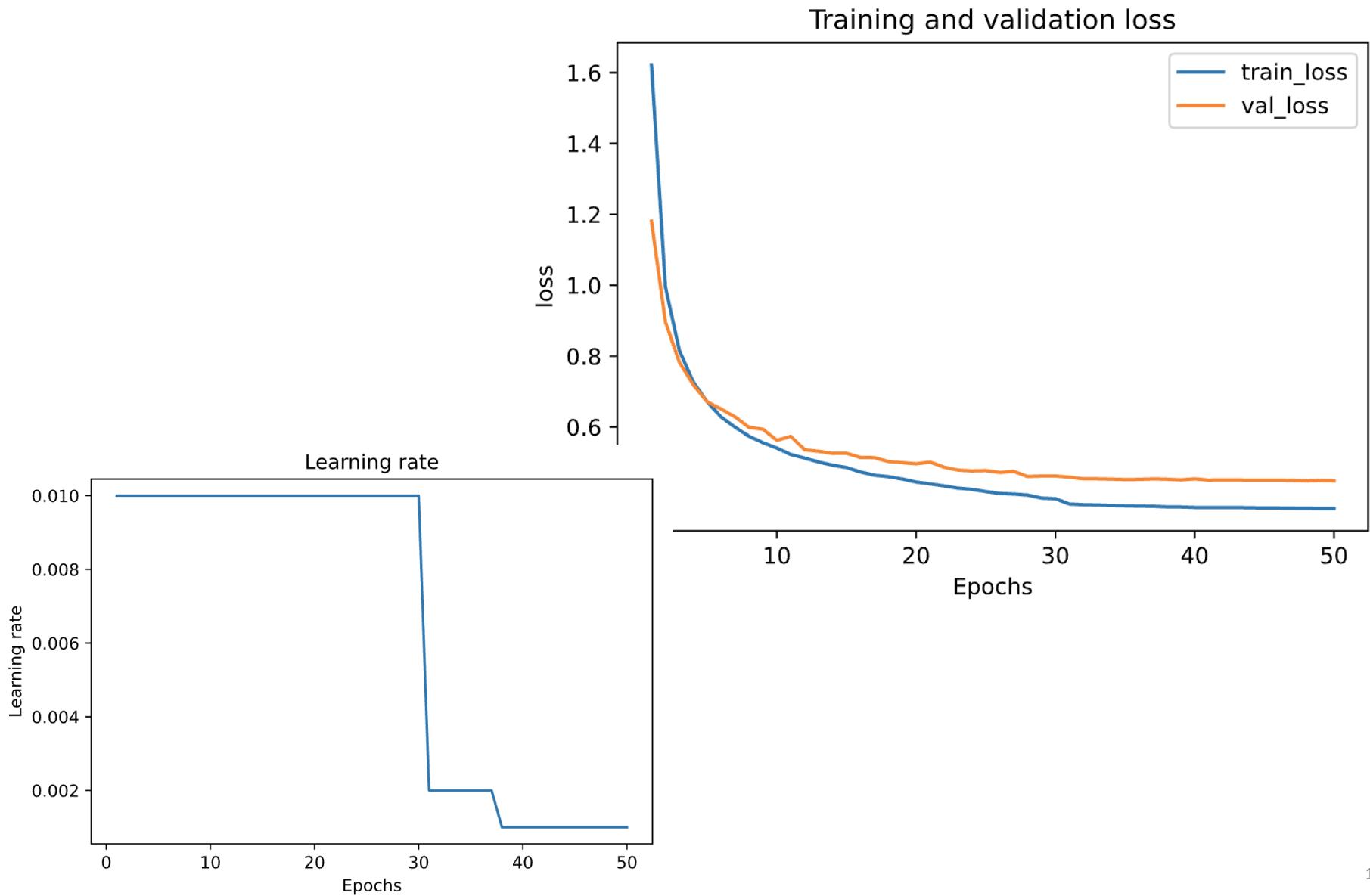
ReduceLROnPlateau callback actúa en los epochs 30 y 37:

```
Epoch 27/50  
8000/8000 - 1s - loss: 0.4108 - accuracy: 0.8624 - val_loss: 0.4747 - val_accuracy: 0.8325  
Epoch 28/50  
8000/8000 - 1s - loss: 0.4079 - accuracy: 0.8618 - val_loss: 0.4603 - val_accuracy: 0.8430  
Epoch 29/50  
8000/8000 - 1s - loss: 0.3994 - accuracy: 0.8656 - val_loss: 0.4617 - val_accuracy: 0.8360  
Epoch 30/50
```

```
Epoch 00030: ReduceLROnPlateau reducing learning rate to 0.001999999552965165. ←  
8000/8000 - 1s - loss: 0.3975 - accuracy: 0.8641 - val_loss: 0.4618 - val_accuracy: 0.8405  
Epoch 31/50  
8000/8000 - 1s - loss: 0.3826 - accuracy: 0.8712 - val_loss: 0.4584 - val_accuracy: 0.8400  
Epoch 32/50  
8000/8000 - 1s - loss: 0.3807 - accuracy: 0.8721 - val_loss: 0.4539 - val_accuracy: 0.8420  
Epoch 33/50  
8000/8000 - 1s - loss: 0.3799 - accuracy: 0.8714 - val_loss: 0.4538 - val_accuracy: 0.8455  
Epoch 34/50  
8000/8000 - 1s - loss: 0.3786 - accuracy: 0.8714 - val_loss: 0.4529 - val_accuracy: 0.8435  
Epoch 35/50  
8000/8000 - 1s - loss: 0.3778 - accuracy: 0.8726 - val_loss: 0.4516 - val_accuracy: 0.8440  
Epoch 36/50  
8000/8000 - 1s - loss: 0.3769 - accuracy: 0.8725 - val_loss: 0.4521 - val_accuracy: 0.8430  
Epoch 37/50
```

```
Epoch 00037: ReduceLROnPlateau reducing learning rate to 0.001. ←  
8000/8000 - 1s - loss: 0.3763 - accuracy: 0.8730 - val_loss: 0.4536 - val_accuracy: 0.8430  
Epoch 38/50  
8000/8000 - 1s - loss: 0.3746 - accuracy: 0.8733 - val_loss: 0.4526 - val_accuracy: 0.8445  
Epoch 39/50
```

# Callbacks: ReduceLROnPlateau



# Callbacks: LearningRateScheduler

Permite pasarle una función que devuelva la tasa de aprendizaje a utilizar en función de la iteración (epoch).

```
from tensorflow.keras.callbacks import LearningRateScheduler

def lr_reduc(epoch, lr):
    if epoch != 0 and epoch % 5 == 0:
        return lr * 0.2
    return lr

...
historial = modelo.fit(X_entr, y_entr, epochs=20, validation_split=0.2,
                       batch_size=64, verbose=2,
                       callbacks=[LearningRateScheduler(lr_reduc, verbose=1)])
```

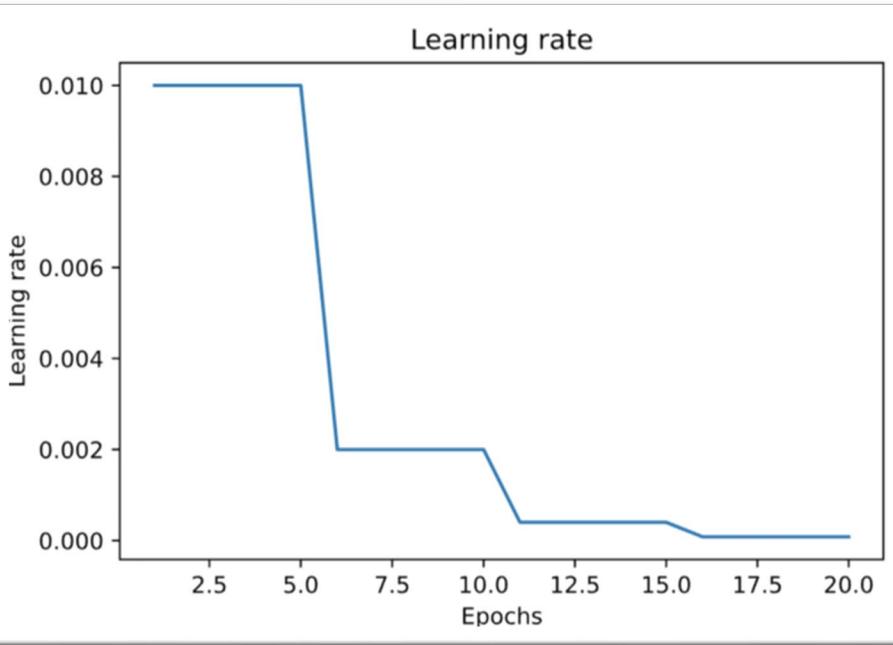
# Callbacks: LearningRateScheduler

*LearningRateScheduler* cambia la tasa de aprendizaje:

Train on 8000 samples, validate on 2000 samples

```
Epoch 00001: LearningRateScheduler reducing learning rate to 0.009999999776482582.  
Epoch 1/20  
8000/8000 - 3s - loss: 1.5567 - accuracy: 0.5614 - val_loss: 1.1245 - val_accuracy: 0.6720  
  
Epoch 00002: LearningRateScheduler reducing learning rate to 0.009999999776482582.  
Epoch 2/20  
8000/8000 - 1s - loss: 0.9460 - accuracy: 0.7106 - val_loss: 0.8574 - val_accuracy: 0.7265  
  
Epoch 00003: LearningRateScheduler reducing learning rate to 0.009999999776482582.  
Epoch 3/20  
8000/8000 - 1s - loss: 0.7757 - accuracy: 0.7487 - val_loss: 0.7673 - val_accuracy: 0.7380  
  
Epoch 00004: LearningRateScheduler reducing learning rate to 0.009999999776482582.  
Epoch 4/20  
8000/8000 - 1s - loss: 0.6925 - accuracy: 0.7740 - val_loss: 0.7008 - val_accuracy: 0.7545  
  
Epoch 00005: LearningRateScheduler reducing learning rate to 0.009999999776482582.  
Epoch 5/20  
8000/8000 - 1s - loss: 0.6433 - accuracy: 0.7851 - val_loss: 0.6497 - val_accuracy: 0.7720  
  
Epoch 00006: LearningRateScheduler reducing learning rate to 0.0019999999552965165.  
Epoch 6/20  
8000/8000 - 1s - loss: 0.6098 - accuracy: 0.7996 - val_loss: 0.6400 - val_accuracy: 0.7825  
  
Epoch 00007: LearningRateScheduler reducing learning rate to 0.001999999862164259.
```

# Callbacks: LearningRateScheduler



on 2000 samples

Epoch 00004: LearningRateScheduler reducing learning rate to 0.009999999776482582.

accuracy: 0.5614 - val\_loss: 1.1245 - val\_accuracy: 0.6720

Epoch 00005: LearningRateScheduler reducing learning rate to 0.009999999776482582.

accuracy: 0.7106 - val\_loss: 0.8574 - val\_accuracy: 0.7265

Epoch 00006: LearningRateScheduler reducing learning rate to 0.009999999776482582.

Epoch 3/20

8000/8000 - 1s - loss: 0.7757 - accuracy: 0.7487 - val\_loss: 0.7673 - val\_accuracy: 0.7380

Epoch 00004: LearningRateScheduler reducing learning rate to 0.009999999776482582.

Epoch 4/20

8000/8000 - 1s - loss: 0.6925 - accuracy: 0.7740 - val\_loss: 0.7008 - val\_accuracy: 0.7545

Epoch 00005: LearningRateScheduler reducing learning rate to 0.009999999776482582.

Epoch 5/20

8000/8000 - 1s - loss: 0.6433 - accuracy: 0.7851 - val\_loss: 0.6497 - val\_accuracy: 0.7720

Epoch 00006: LearningRateScheduler reducing learning rate to 0.001999999552965165.

Epoch 6/20

8000/8000 - 1s - loss: 0.6098 - accuracy: 0.7996 - val\_loss: 0.6400 - val\_accuracy: 0.7825

Epoch 00007: LearningRateScheduler reducing learning rate to 0.001999999862164259.

# Callbacks: ModelCheckpoint

- Los *checkpoints* permiten retomar un entrenamiento en el punto en el que se interrumpió.
- Permiten guardar el estado del modelo y su entrenamiento (al igual que *modelo.save*) o sólo los pesos (como *modelo.save\_weights*).
- Para ello se utiliza la *callback ModelCheckpoint*.
- Para continuar el entrenamiento de un modelo salvado por medio de un *checkpoint*, simplemente hay que cargar los datos del fichero guardado por el *checkpoint* y volver a llamar a *fit*. En este caso no se debe compilar el modelo cargado, ya entonces, el entrenamiento comenzaría desde el principio.
- Para la carga, se usan los métodos ya vistos, *modelo.load\_weights* o *load\_model*.
- **Algunos argumentos:**
  - **filepath**: Ruta en la que se guardará el fichero. Se puede indicar formato, como, por ejemplo, `weights.{epoch:02d}-{val_loss:.2f}`, que hará que el nombre contenga el número de epoch y las pérdidas en evaluación.
  - **save\_freq**: tomará el valor 'epoch' o un entero. En el primer caso, se guarda el modelo tras cada epoch. En el segundo, al final del número de batches indicados por el entero.
  - **save\_weights\_only**: *True*, para guardar solo los pesos (*modelo.save\_weights(ruta)*) y *False* para guardar todo (*modelo.save(ruta)*).
  - **save\_best\_only**: si vale *True*, solo guarda el mejor modelo.

# Callbacks: ModelCheckpoint

```
from tensorflow.keras.callbacks import ModelCheckpoint

ruta="pesos-{epoch:02d}-{val_accuracy:.2f}.hdf5"
checkpoint = ModelCheckpoint(ruta, monitor='val_accuracy', verbose=1,
                             save_best_only=True, mode='max')
lista_cbs = [checkpoint]
modelo.fit(X, Y, validation_Split = 0.33, epochs = 150, batch_size = 10,
            callbacks = lista_cbs, verbose = 0)
```

Para cargar el modelo y seguir entrenándolo en cualquier momento posterior:

```
modelo_nuevo = crear_modelo() # volvemos a crear un modelo como el guardado
modelo_nuevo.load_weights(...)
```

*Formato HDF5: Hierarchical Data Format, version 5.*

# Callbacks: LambdaCallback

Permite crear *callbacks* sencillas.

Se construye a partir de una serie de funciones anónimas que se llamarán en instantes de tiempo concretos:

- **on\_epoch\_begin / on\_epoch\_end**: Llamada al inicio/final de cada *epoch*.
  - Argumentos posicionales: *epoch* (número de epoch) y *logs*, que contiene los valores de pérdida (*loss*), exactitud, etc.
- **on\_batch\_begin / on\_batch\_end**: Llamada al inicio/final de cada lote (*batch*).
  - Argumentos posicionales: *batch* (número de lote) y *logs*, que contiene los valores de pérdida (*loss*), exactitud, etc.
- **on\_train\_begin / on\_train\_end**: Llamada al inicio/final del entrenamiento.
  - Argumento posicional: *logs*, que contiene los valores de pérdida (*loss*), exactitud, etc.

# Callbacks: LambdaCallback

```
from tensorflow.keras.callbacks import LambdaCallback

epoch_callback = LambdaCallback(
    on_epoch_begin=lambda ep,logs: print('Inicio epoch {}'.format(ep+1)))

batch_perd_callback = LambdaCallback(
    on_batch_end=lambda batch,logs: print('\nTras el batch nº {}, pérdida: {:.7.2f}'.format(batch, logs['loss'])))

fin_entr_callback = LambdaCallback(
    on_train_end = lambda logs: print('Entrenamiento finalizado'))

...

historial = modelo.fit(
    X_entr, y_entr, validation_split=0.20,
    verbose=False, epochs=2, batch_size=2000,
    callbacks=[epoch_callback, batch_perd_callback, fin_entr_callback]))
```

# Callbacks: LambdaCallback

## **Inicio epoch 1**

Tras el batch nº 0, pérdida: 0.41

Tras el batch nº 1, pérdida: 0.41

Tras el batch nº 2, pérdida: 0.42

Tras el batch nº 3, pérdida: 0.39

## **Inicio epoch 2**

Tras el batch nº 0, pérdida: 0.40

Tras el batch nº 1, pérdida: 0.42

Tras el batch nº 2, pérdida: 0.41

Tras el batch nº 3, pérdida: 0.41

**Entrenamiento finalizado**

# Matriz de confusión

La matriz de confusión, también llamada tabla de contingencia, es una herramienta que nos muestra el desempeño de un algoritmo de clasificación, describiendo cómo se distribuyen los valores reales y nuestras predicciones:

Suponiendo que la predicción de la red es positivo (cierto) o negativo (falso):

- **Verdaderos positivos:** Número de predicciones positivas que realmente lo eran.
- **Verdaderos negativos:** Número de predicciones negativas que realmente lo eran.
- **Falsos positivos:** Número de predicciones positivas cuando realmente el valor es tendría que ser negativo. Se les denomina errores de tipo I.
- **Falsos negativos:** Número de predicciones negativas cuando realmente el valor es positivo. A estos casos también se les denomina errores de tipo II.

		Predicción	
		Negativo (0)	Positivo (1)
Valor real	Negativo (0)	TN	FP
	Positivo (1)	FN	TP

# Matriz de confusión

Basándonos en esas cuatro categorías, se pueden calcular las siguientes métricas:

- **Exactitud (accuracy)**: Porcentaje total de los aciertos de nuestro modelo.
- **Sensibilidad (recall )**: Probabilidad de clasificar correctamente los casos positivos.
- **Precisión (valor de predicción positiva)**: Probabilidad de que una predicción positiva sea correcta
- **Especificidad (specificity)**: Probabilidad de clasificar correctamente los casos negativos.
- **Valor de predicción negativa**: Probabilidad de que una predicción negativa sea correcta.
- **Tasa de falso positivo (False positive rate)**: probabilidad de clasificar incorrectamente un negativo.
- **Error de clasificación**: Porcentaje de errores del modelo.

## CURVAS

- Las **curvas ROC** muestran la relación entre la tasa de verdaderos positivos y la de falsos positivos usando distintos umbrales de probabilidad.
- Las **curvas de Precisión-Sensibilidad** muestran la relación entre ambos parámetros para distintos umbrales de probabilidad.
- Las curvas ROC son útiles cuando las observaciones entre ambas clases están equilibradas, mientras que las curvas de Precisión-Sensibilidad son más útiles en conjuntos de datos desequilibrados.

# Medidas a partir de la matriz de confusión

$$\text{Sensibilidad (Recall, Sensitivity, True positive rate)} = \frac{VP}{\text{Total positivos}} = \frac{VP}{VP + FN}$$

$$\text{Precisión (Precision)} = \frac{VP}{\text{Total clasificados positivos}} = \frac{VP}{VP + FP}$$

$$\text{Exactitud (Accuracy)} = \frac{VP + VN}{\text{Total}} = \frac{VP + VN}{FP + FN + VP + VN}$$

$$\text{Tasa de error (Error rate)} = \frac{FP + FN}{\text{Total}} = \frac{FP + FN}{FP + FN + VP + VN}$$

$$\text{Especificidad (Specificity)} = \frac{VN}{\text{Total negativos}} = \frac{VN}{VN + FP}$$

$$\text{Tasa de predicción negativa} = \frac{VN}{\text{Total clasificados negativos}} = \frac{VN}{VN + FN}$$

$$\text{Tasa de falso positivo} = \frac{FP}{\text{Total de negativos}} = \frac{FP}{VN + FP}$$

$$\text{Coef. de correlación de Matthews (MCC)} = \frac{VP \cdot VN - FP \cdot FN}{\sqrt{(VP + FP)(VP + FN)(VN + FP)(VN + FN)}}$$

$$F-score = F_{\beta} = \frac{(1 + \beta^2)(\text{Precisión} \cdot \text{Sensibilidad})}{\beta^2 \cdot \text{Precisión} + \text{Sensibilidad}} \quad \text{con } \beta = 0.5, 1 \text{ ó } 2$$

# Medidas a partir de la matriz de confusión

## F-1 Score

Combina precisión y sensibilidad en una sola métrica por ello es de gran utilidad **cuando la distribución de las clases es desigual**. Es especialmente interesante cuando se busca equilibrio entre precisión y sensibilidad.

$$F1 - score = \frac{2 \cdot \text{Precisión} \cdot \text{Sensibilidad}}{\text{Precisión} + \text{Sensibilidad}}$$

Conforme a estas métricas tenemos cuatro casos posibles para cada clase:

- Alta precisión y alta sensibilidad: el modelo maneja perfectamente esa clase
- Alta precisión y baja sensibilidad: el modelo no detecta la clase muy bien, pero cuando lo hace es altamente fiable.
- Baja precisión y alta sensibilidad: La clase detecta bien la clase, pero también incluye muestras de otras clases.
- Baja precisión y baja sensibilidad : El modelo no logra clasificar la clase correctamente.

Cuando tenemos un *dataset* con desequilibrio, suele ocurrir que obtenemos un **alto valor de precisión en la clase mayoritaria y una baja sensibilidad en la clase minoritaria**.

# Medidas a partir de la matriz de confusión

Matriz de confusión

Verdaderos negativos 8	Falsos positivos 2
Falsos negativos 4	Verdaderos positivos 6

Tasa de error:  $6 / 20 = 0.3$

Exactitud (% de aciertos):  $14 / 20 = 0.7$

Sensibilidad:  $6 / 10 = 0.6$

Especificidad  $8 / 10 = 0.8$

Precisión:  $6 / 8 = 0.75$

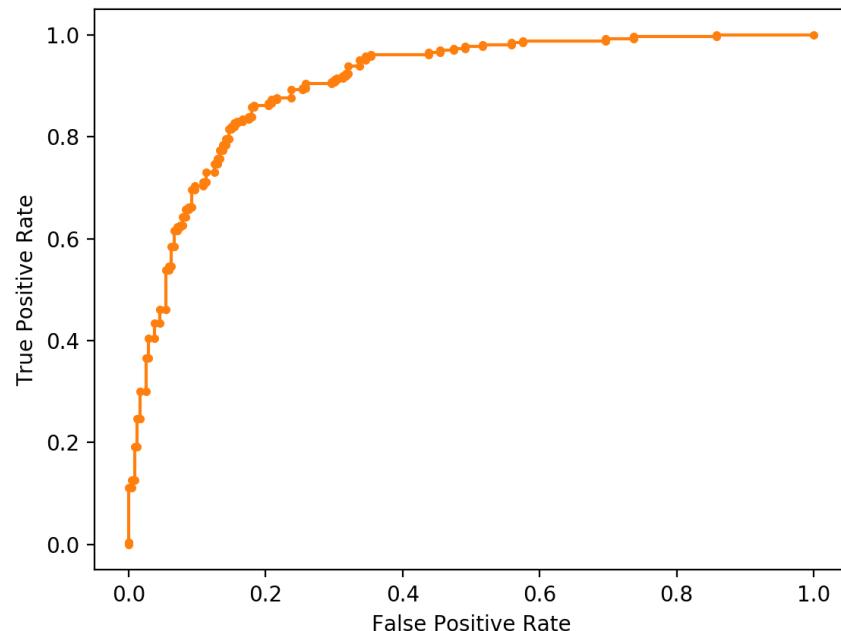
Tasa de falsos positivos:  $2 / 10 = 0.2$

# Curvas ROC

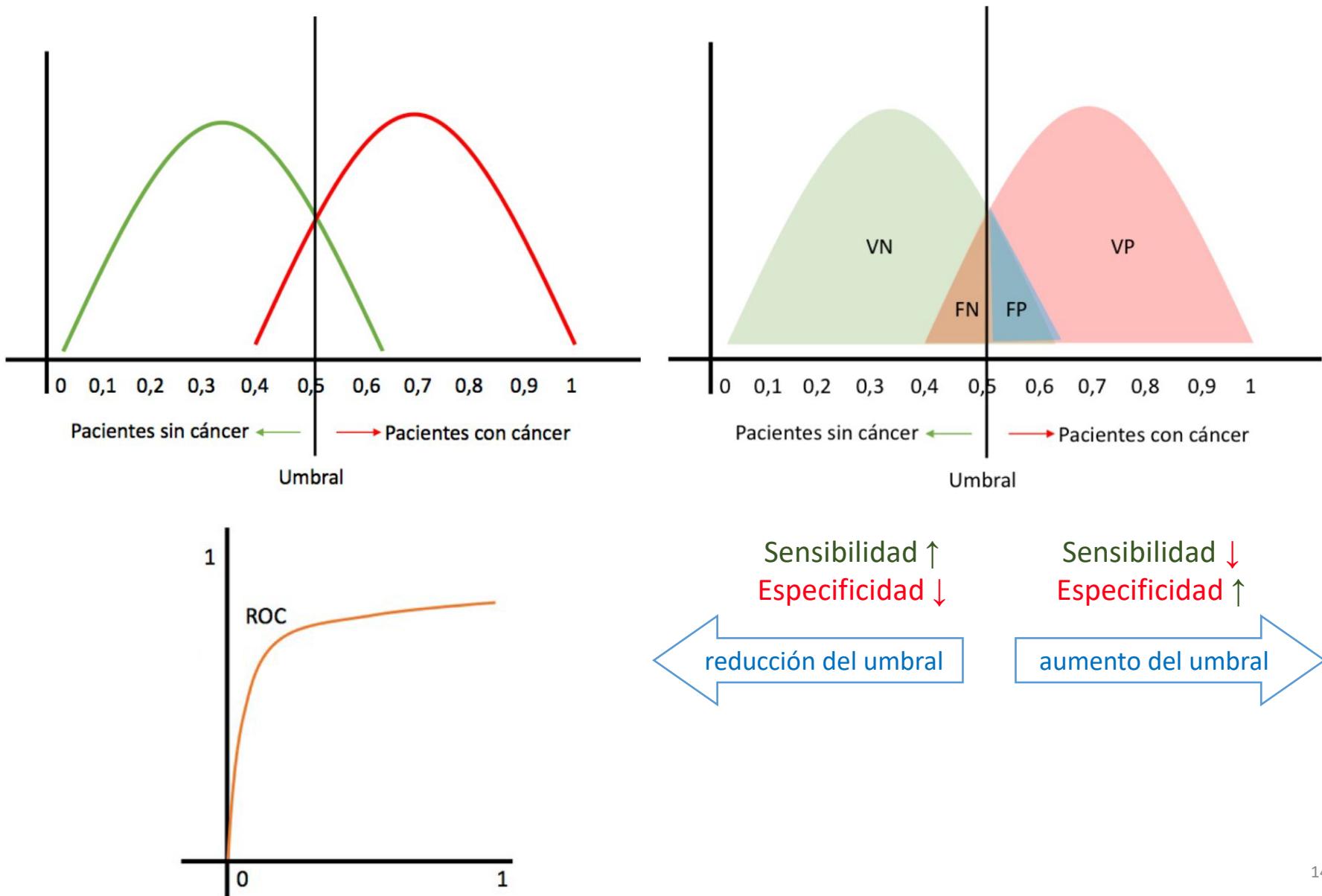
La **curva ROC** (*Receiver Operating Characteristic* o *Característica Operativa del Receptor*) es una representación gráfica de la relación entre la tasa de falsos positivos (eje x) y los verdaderos positivos o sensibilidad (*recall*) para distintos valores del umbral (entre 0 y 1).

La tasa de falsos positivos es la proporción entre los falsos positivos y el total de negativos reales, es decir, el porcentaje de negativos no detectados). Se denomina también como tasa de falsa alarma y como especificidad invertida, ya que se calcula como: *tasa de falsos positivos = 1 – especificidad*.

De modo intuitivo, representa la falsa alarma frente a los aciertos.



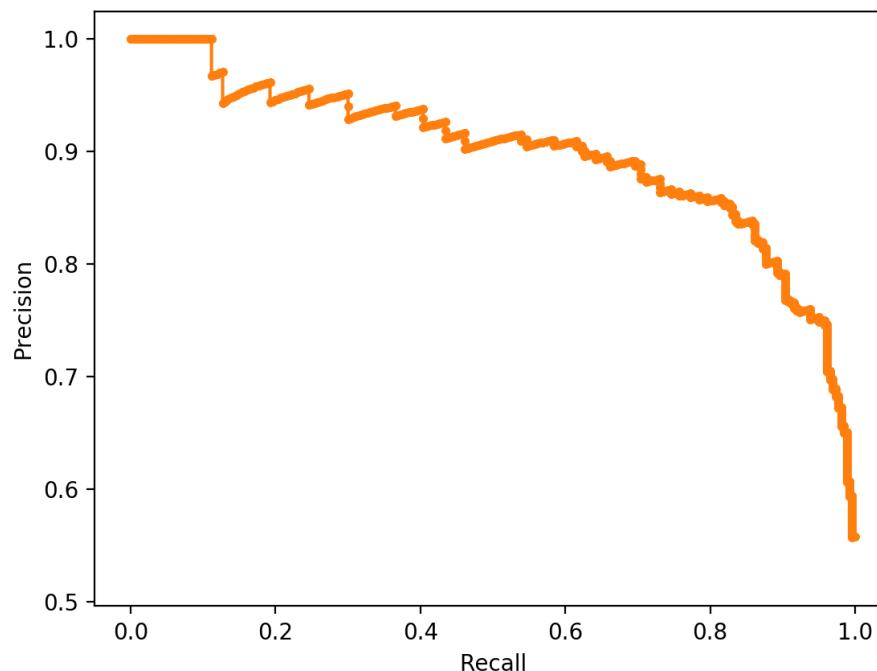
# Matriz de confusión



# Curvas Precisión-Sensibilidad

La **curva Precisión-Sensibilidad** representa la relación entre ambos valores para distintos valores del umbral (entre 0 y 1). Es más útil que la curva ROC cuando hay desequilibrio en la distribución de las dos clases.

Conviene destacar que no se tienen en cuenta los verdaderos negativos, porque el objetivo es detectar la clase minoritaria (verdaderos).



# Análisis de error

Supongamos el caso de una aplicación donde los datos de entrenamiento y los de validación tienen distinta distribución: supongamos que disponemos de 10K imágenes reales obtenidas de una app móvil y completamos con 200K imágenes obtenidas de Internet. Estas últimas probablemente tendrán mucha mayor calidad que las reales.

Si los tres *datasets* (entrenamiento, validación y pruebas o test) contienen ambos tipos de imagen en la misma proporción, a pesar de que los tres tendrán la misma distribución, la validación del sistema se hará con una mayoría de imágenes que no serán las que se encontrará el sistema en producción.



*Imagen de la web*



*Imagen real (app)*

# Análisis de error

Un enfoque mejor podría construir los *datasets* de validación y pruebas con la mitad de las imágenes reales (del app), 2.5K cada uno. El resto de las imágenes reales (5K) se usarían para el entrenamiento. Ahora, la validación del sistema se haría en condiciones similares a las reales, pero los datos no provendrían de la misma distribución, por lo que, para detectar sub o sobreajuste, ya no se pueden comparar las pérdidas en los conjuntos de entrenamiento y validación.

Para resolver el problema, se extraería del *dataset* de entrenamiento parte de las imágenes reales para formar un nuevo conjunto llamado puente (*bridge* o *train-dev*) con el mismo tamaño que los de validación y pruebas, y la misma distribución.

entrenamiento: 205K

puente: 2.5K

validación: 2.5K

pruebas: 2.5K

El análisis del sub o sobreajuste se replantearía así:

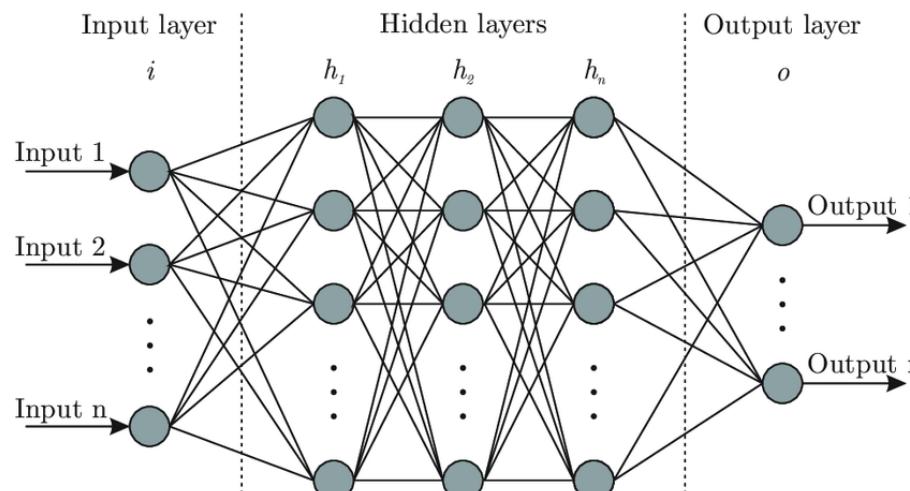
- Error de entrenamiento (bajo) y de puente y validación altos (p. ej. 10% y 11% resp.): Error de sobreajuste.
- Los tres errores altos y parecidos (10%, 11% y 13%, por ejemplo): subajuste.
- Errores aceptables en entrenamiento y puente (2% y 5%, p. ej.) y de validación alto (10%, p. ej.): se conoce como error de disparidad (*mismatch error*) y se debe a la diferencia de distribución de los *datasets*. Se deben analizar los datos y ver cuáles son las diferencias. En el caso de las imágenes anteriores podría probarse a hacer más borrosas las imágenes de internet para que fueran más parecidas a las reales.

# Proceso completo de diseño de una red neuronal

**NEURONAS DE ENTRADA:** una por cada característica de los datos. En el caso de imágenes, correspondería con el número de píxeles si la imagen está en una escala de gris, o el triple si es una imagen en color (RGB).

**NEURONAS DE SALIDA:** se corresponde con el número de predicciones.

- **Clasificación binaria:** Una neurona por cada clase positiva, que genera la probabilidad de esa clase positiva.
- **Clasificación multiclase:** Una neurona por clase usando la función de activación *softmax* sobre la capa de salida para asegurar que la probabilidad total suma uno.
- **Regresión:** Una neurona por cada valor de salida. Puede ser un único valor de salida (predicción del precio de una vivienda, por ejemplo) o multivalorado (cuatro píxeles donde se encuadra una cara en una imagen, por ejemplo).



# Proceso completo de diseño de una red neuronal

## CAPAS OCULTAS Y SU NÚMERO DE UNIDADES:

- Depende mucho del problema y la arquitectura de la red. En **general**, *entre una y cinco capas* suele ser suficiente para la mayoría de los problemas.
- Por eso puede ser interesante empezar con entre una y cinco capas y hasta 100 neuronas por capa. A continuación se incrementan ambos hasta que la red sobreajuste.
- Se puede hacer un seguimiento de la pérdida y la exactitud para ver qué capas o neuronas ocultas llevan a la menor pérdida.
- Con frecuencia se consigue mejor rendimiento aumentando el número de capas que el número de neuronas de cada capas.
- Para trabajar con **imágenes o reconocimiento de habla**, es frecuente trabajar con modelos pre-entrenados (YOLO, ResNet, ...), de modo que solo unas pocas capas necesitan entrenamiento.
- En general, se puede trabajar con el mismo número de neuronas en todas las capas ocultas. Con algunos *datasets* se usará una primera capa grande, seguida de capas más pequeñas.
- Si la red es demasiado pequeña, no será capaz de aprender los patrones subyacentes. Un enfoque para evitarlo es empezar con una red grande número de capas y de neuronas y después usar *dropout* y parar temprana (*early stopping*). Es recomendable probar distintas configuraciones.
- Otra estrategia es conseguir una red que sobreajuste y luego regularizar.

# Proceso completo de diseño de una red neuronal

## FUNCIONES DE PÉRDIDA (*loss funtions*)

- Regresión: Error cuadrático medio, excepto si hay muchos valores atípicos (*outliers*), en cuyo caso se prefiere el error medio absoluto o la función de Huber.
- Clasificación: la entropía cruzada es válida en la mayoría de los casos.

## TAMAÑO DEL MINILOTE

- Salvo casos muy excepcionales cuando se dispone de mucha potencia de cálculo, convienen lotes grandes (decenas de miles de ejemplos para imágenes) o incluso millones (para aprendizaje por refuerzo), aunque presenta problemas con la tasa de aprendizaje y, por tanto, con la convergencia.
- La consecuencia es que mini-lotes con un tamaño de 32 suelen ser prácticos.
- Excepto si se dispone de una capacidad de cálculo enorme, la recomendación podría ser empezar con lotes pequeños e ir aumentándolos poco a poco, monitorizando el rendimiento.

## NÚMERO DE REPETICIONES (*epochs*)

- Es recomendable empezar con un número de repeticiones muy grande y usar parada temprana (*early stopping*) cuando el comportamiento empieza a empeorar.

# Proceso completo de diseño de una red neuronal

## ESCALADO DE CARACTERÍSTICAS (FEATURES) DE LOS DATOS DE ENTRADA

- Todas las características (*features*) deberían tener una escala similar antes de introducirlas a la red. Esto acelera la convergencia.

## TASA DE APRENDIZAJE

- Es un parámetro crítico para la red.
- Se debe empezar por un valor pequeño ( $10^{-6}$ ) e ir multiplicándolo lentamente por una constante hasta que alcanza un valor muy alto (10, por ejemplo). Medir el rendimiento del sistema frente a la tasa de aprendizaje, (su logaritmo, típicamente) para escoger el mejor valor.
- Es de destacar el método de tasas de aprendizaje cíclicas de Leslie Smith: incrementarla hasta que se disparan las pérdidas. La tasa elegida será un orden inferior a la que minimiza las pérdidas.

## MOMENTO

- En general, conviene un valor de momento cercano a uno. Un buen valor inicial para *datasets* pequeños puede ser 0.9, aumentando progresivamente para tamaños mayores.
- Es útil activar el parámetro de Nesterov, ya que el momento se calculará teniendo en cuenta el gradiente de la función de coste.

# Proceso completo de diseño de una red neuronal

## DESVANECIMIENTO Y EXPLOSIÓN DE GRADIENTES

- Funciones de activación

- Capas ocultas:

- En general, el rendimiento mejora en este orden (de peor a mejor): logistic → tanh → ReLU → Leaky ReLU → ELU → SELU.
    - ReLU es la más popular y por tanto la mejor opción inicial a pesar de que está perdiendo peso progresivamente en favor de ELU y SELU.
    - Es interesante recordar:
      - Para combatir el sobre-ajuste: ReLU
      - Para reducir la latencia en tiempo de ejecución: Leaky ReLU
      - Para *datasets* de entrenamiento muy grandes: PReLU (parametric ReLU)
      - Para tiempos de inferencia rápidos: Leaky ReLU
      - Si la red no autonormaliza: ELU
      - Si queremos una función de activación general robusta: SELU
    - Es interesante probar con diferentes funciones de activación.

# Proceso completo de diseño de una red neuronal

- Activación en la capa de salida:

- **Regresión:** En este tipo de problemas no se necesita ninguna función de activación en la capa de salida (también llamada activación lineal). En caso de necesitar que la salida esté entre -1 y 1 se puede usar *tanh*, o la función logística ( $f(x)=1/(1+\exp(-x))$ ) para rangos entre 0 y 1 (que es una generalización de la sigmoide). En los casos en los que se necesite sólo salidas positivas, se puede utilizar la activación *softplus* ( $f(x)=\ln(1+\exp x)$ ), que resulta ser la integral de la función logística.
- **Clasificación:** se usaría la función sigmoide para clasificación binaria (asegura salida entre 0 y 1) y *softmax* para clasificación multiclas (asegura que las probabilidades de salida suman 1).

- Inicialización de los pesos

- Una inicialización adecuada de los pesos acelera considerablemente la velocidad de convergencia. El tipo de inicialización depende de la función de activación.  
Sugerencias:
  - ReLU o Leaky ReLU: inicialización *He*.
  - SELU o ELU: inicialización *LeCun*.
  - Softmax, función logística (sigmoide), o tanh: inicialización *Xavier (Glorot)*.
- La mayoría de los métodos de inicialización usan distribuciones normales o uniformes.

# Proceso completo de diseño de una red neuronal

- **Normalización de lotes (*BatchNorm*)**

- Usar *BatchNorm* nos permite utilizar tasas de aprendizaje mayores, lo cual acelera la convergencia y reduce los problemas de desvanecimiento del gradiente.
- El inconveniente es que incrementa ligeramente los tiempos de entrenamiento debido a que aumenta la carga computacional necesaria en cada capa.

- **Recorte de gradiente**

- Se recomienda probar diferentes valores del umbral.

- **Parada temprana (*Early Stopping*)**

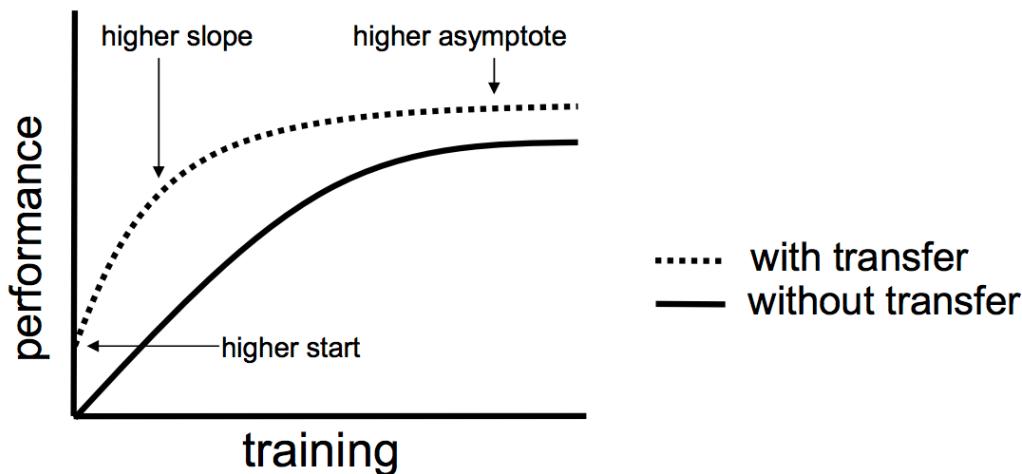
## DROPOUT

- Hace la red más robusta, distribuyendo el conocimiento por toda la red, de modo que no dependa de neuronas concretas.
- La proporción de *dropout* suele estar alrededor de 0.5, siendo típico 0.3 para RNNs y 0.5 para CNNs.
- Cuanto mayor es una capa, mayor ratio de *dropout* se usa.
- Hay que evitar aumentar tanto la ratio que se entre en subajuste (*underfitting*).
- Se debe probar con distintos valores en las primeras capas.
- No se usa en las capas de salida.

# Aprendizaje por transferencia (*transfer learning*)

Con el aprendizaje por transferencia, en lugar de comenzar el proceso de aprendizaje desde cero, se comienza haciendo uso de modelos pre-entrenados: modelos que fueron entrenados con un ingente conjunto de datos de referencia para resolver un problema similar al que queremos abordar.

Debido al coste computacional del entrenamiento de tales modelos, así como en la complejidad a la hora de elegir la arquitectura óptima, el *Transfer Learning* de modelos bien conocidos y precisos se ha convertido es una práctica común, sobre todo cuando se trabaja en análisis de imágenes.



# Arquitecturas

En función del tipo de datos que manejen y del tipo de resultado que se busque de una red neuronal, existen múltiples topologías de redes neuronales.

Estas arquitecturas se pueden organizar en cuatro grandes grupos:

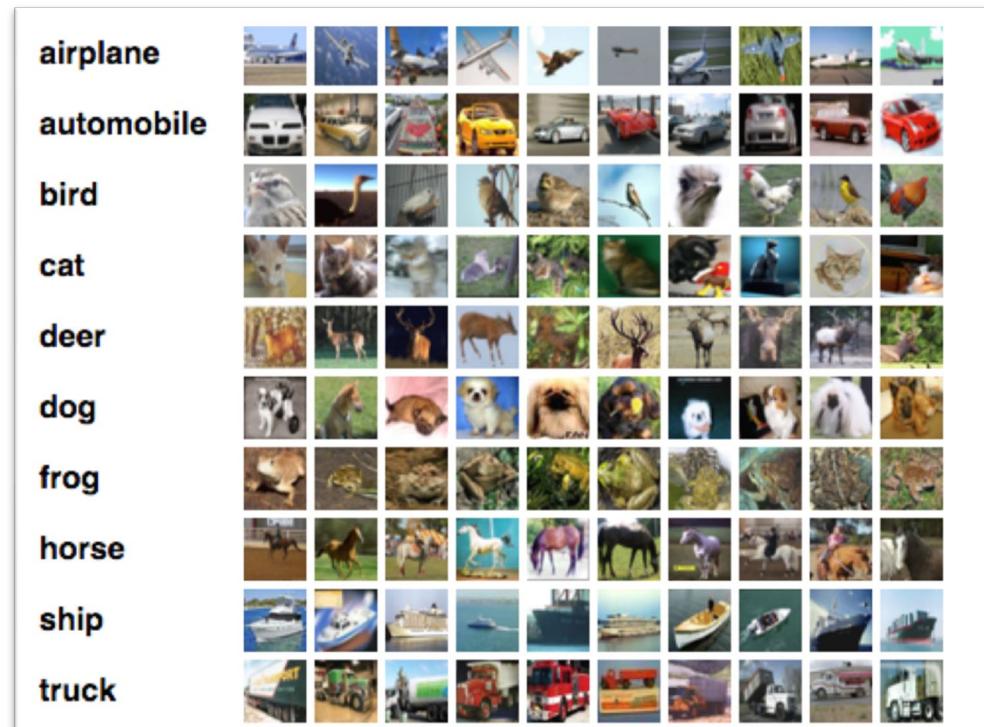
- Redes pre-entrenadas sin supervisión (*Unsupervised Pretrained Networks – UPN* ). En esta categoría destacan tres tipos de redes:
  - Autoencoders
  - Redes de creencia profunda (*Deep Belief Networks – DBN* )
  - Redes generativas antagónica (*Generative Adversarial Networks – GAN* )
- Redes neuronales convolucionales (*Convolutional Neural Networks – CNN* )
- Redes neuronales recurrentes (*Recurrent Neural Networks*)

# Redes neuronales convolucionales (CNN)

- Objetivo: detectar características de alto nivel en los datos por medio de convoluciones.
- Son la mejor opción para el reconocimiento de objetos en imágenes.
- También es una arquitectura eficaz para el análisis de sonido y de texto, cuando se consideran las palabras como unidades de texto discretas.
- Tienden a ser más útiles cuando los datos de entrada tienen algún tipo de estructura. Ejemplos de esto son las imágenes y el audio, donde aparecen patrones de forma repetida y, valores de entrada vecinos están relacionados.
- Aplicaciones:
  - Procesado de imagen, reconocimiento y clasificación
  - Reconocimiento de vídeo
  - Procesado de lenguaje natural
  - Reconocimiento de patrones
  - Motores de recomendación

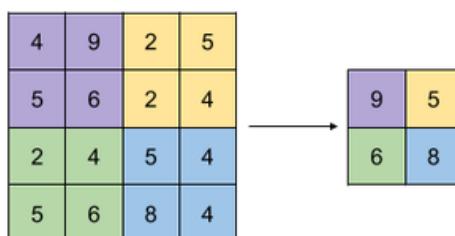
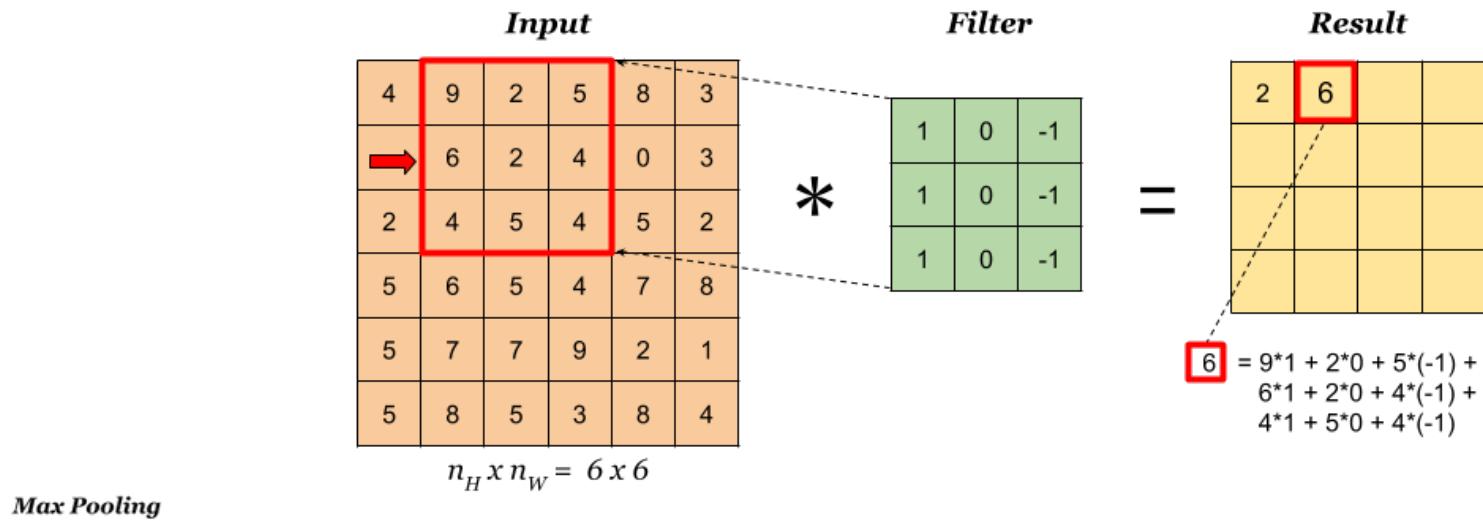
## CIFAR-10 Dataset:

Conjunto de 60.000 imágenes en color de 32x32 píxeles divididas en 10 clases (6.000 imágenes por clase)



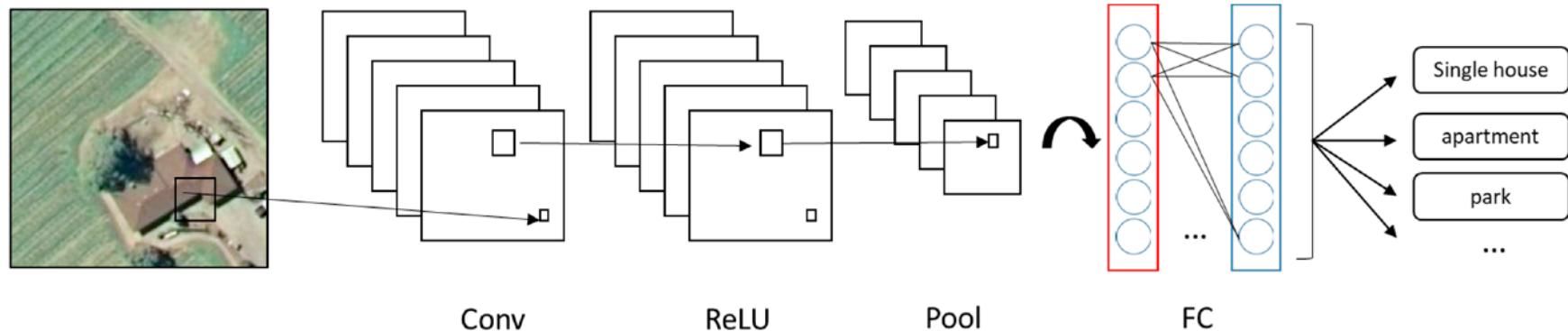
# Redes neuronales convolucionales (CNN)

- Convolución: es una operación donde se multiplica, posición por posición, una región de la imagen por una matriz de valores llamado núcleo (*kernel* en inglés).  
La convolución permite detectar bordes y esquinas en las imágenes.
- Pooling: Operación que reduce una imagen dividiéndola en áreas y sustituyendo cada una de ellas por uno de los valores (máximo o promedio, típicamente).

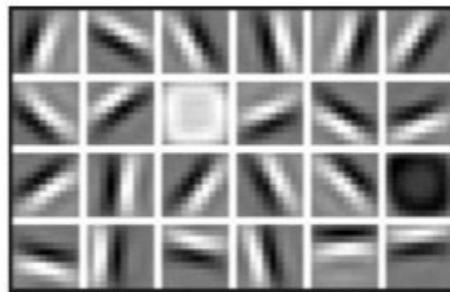


# Redes neuronales convolucionales (CNN)

- La arquitectura de una CNN es una secuencia de operaciones de convolución y pooling, que termina en una o varias capas densamente conectadas y finalmente una activación softmax si la red se usa para clasificación multiclase (p. ej. detectar si en la escena hay peatones, semáforos, vehículos, etc.) o una activación binaria si es un problema de clasificación binaria (p. ej. ¿hay algún peatón en la escena?)
- La combinación de los distintos componentes de la red genera distintas arquitecturas: VGG, ResNet, Inception, etc.



# Redes neuronales convolucionales (CNN)



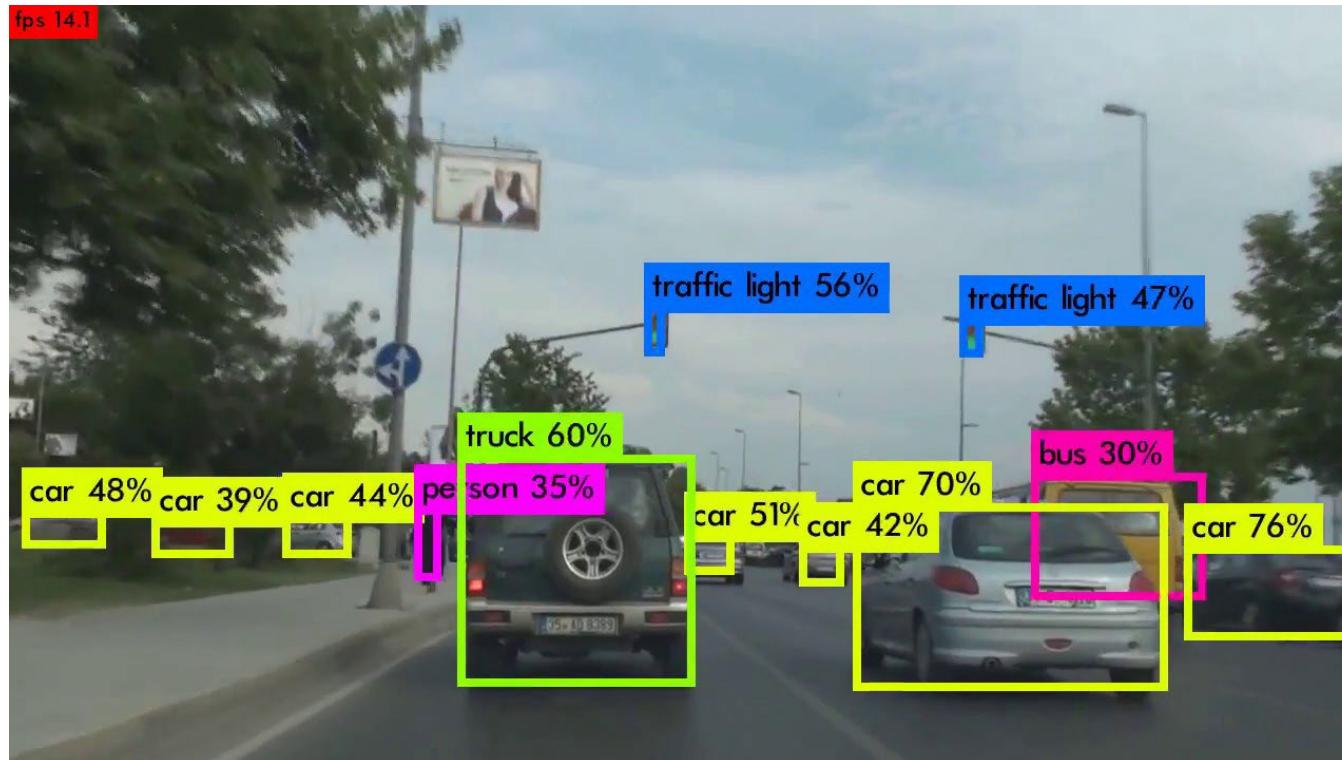
First Layer Representation



Second Layer Representation

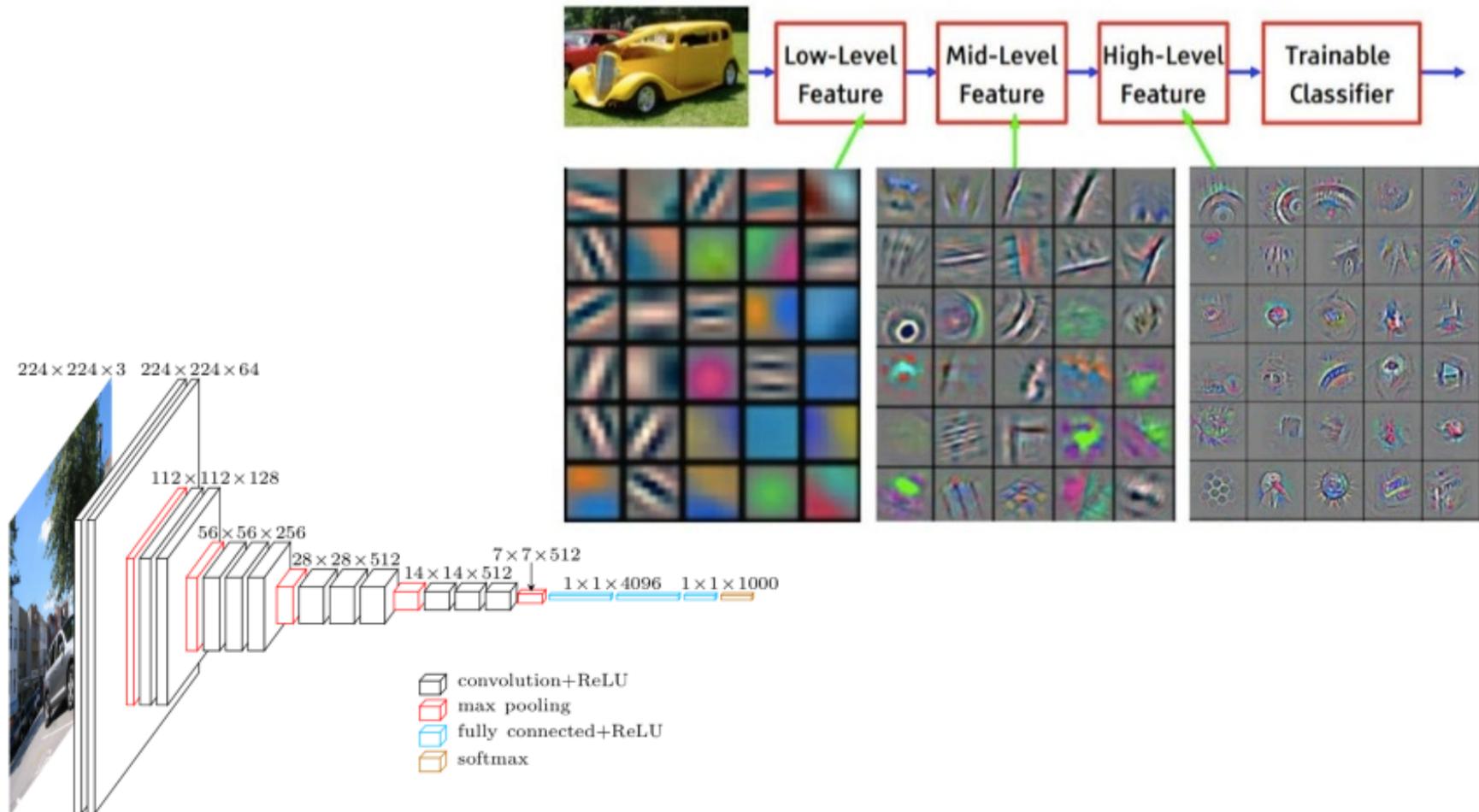


Third Layer Representation



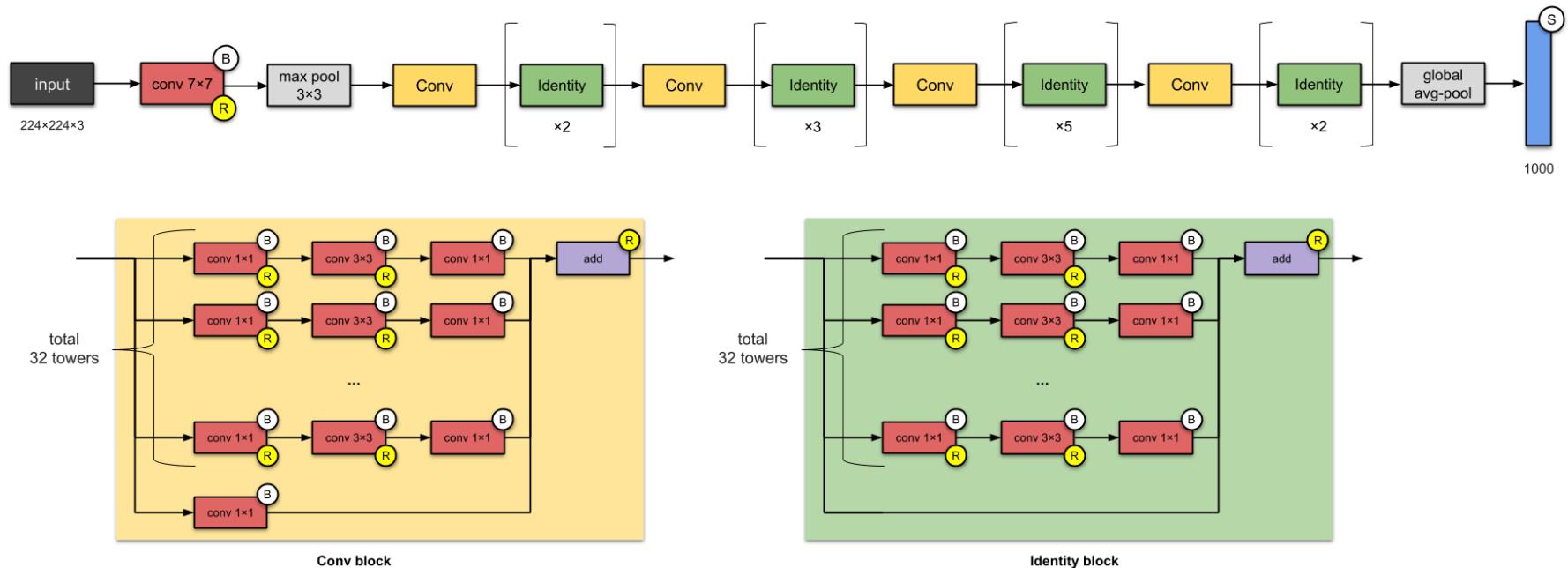
# Aprendizaje por transferencia en CNNs

Dado que las primeras capas convolucionales de las CNN extraen características básicas de la imagen (rectas, círculos, ...) es típico "congelar" esas primeras capas, y re-entrenar las últimas. Cuanto más se parezcan los datos del preentrenamiento a los nuevos datos, más capas se podrán congelar, y por tanto, menor tiempo de entrenamiento.



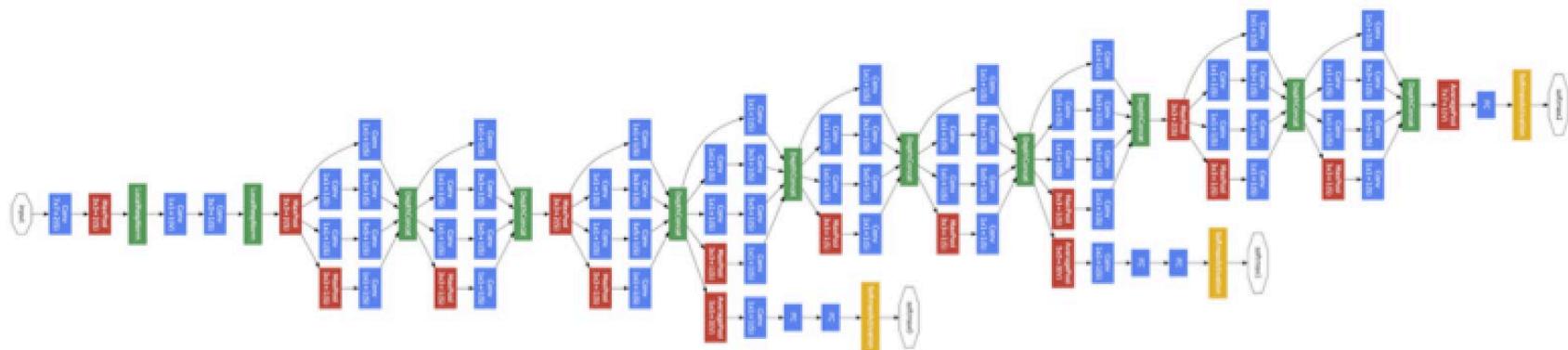
# Redes neuronales convolucionales (CNN)

ResNeXt-50



# Redes neuronales convolucionales (CNN)

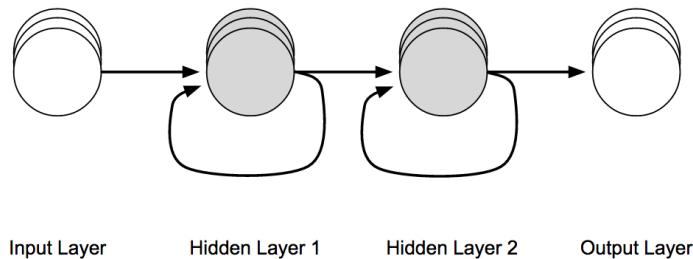
GoogleNet



Convolution  
Pooling  
Softmax  
Other

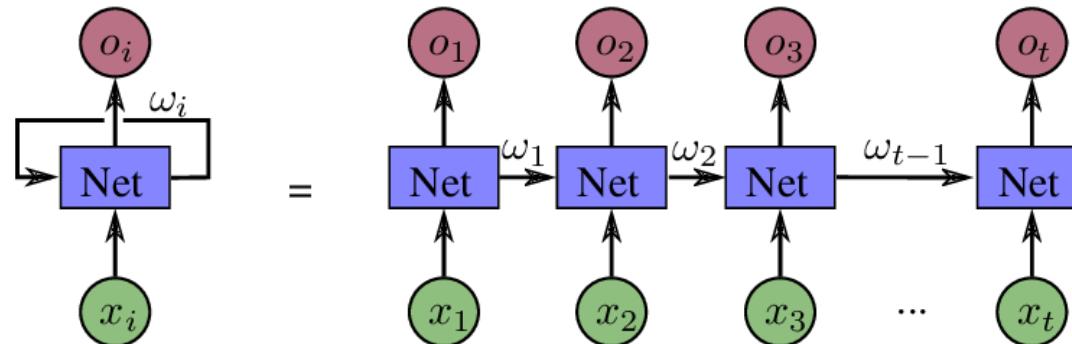
# Redes neuronales recurrentes (RNN)

- Las redes neuronales recurrentes, o *Recurrent Neural Networks* en inglés, tienen como objetivo el análisis de series temporales de datos, permitiendo tratar la dimensión tiempo. Su objetivo es reconocer patrones secuenciales y predecir el escenario más probable.
- Se usan para el procesado de sonido, lenguaje natural escrito, series de datos temporales.
- También se entran con los métodos de descenso de gradiente y retropropagación
- A diferencia de las redes tradicionales, en éstas, cada capa oculta conserva información secuencial de los pasos previos, de forma que la salida de pasos anteriores se introduce como entrada del paso actual, usando los mismos pesos y sesgos (*bias*) de modo repetido.

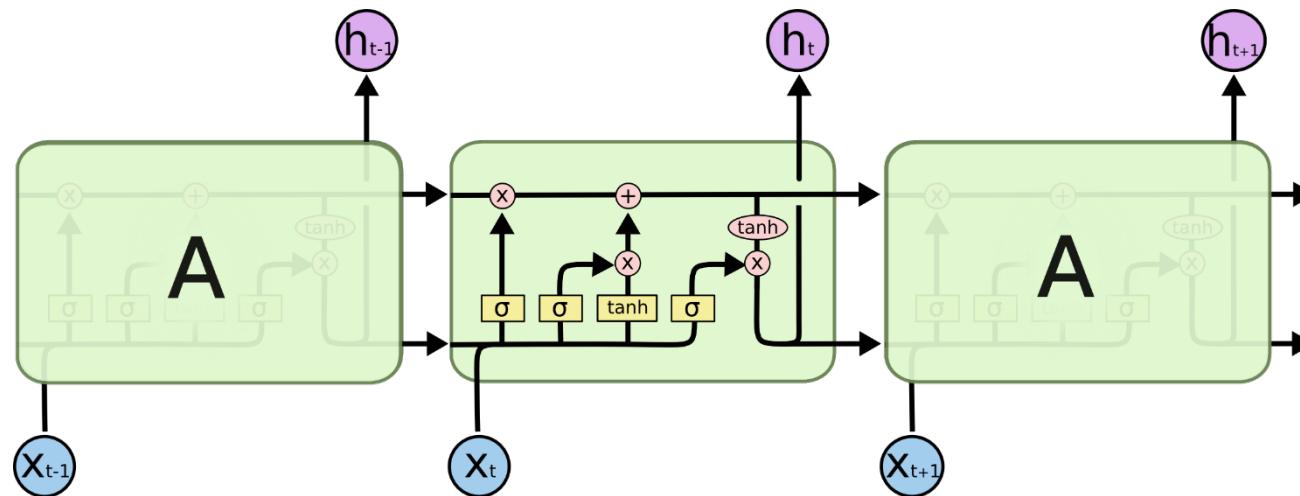


- Las redes de propagación hacia adelante (feed-forward networks) generan una salida para cada entrada (relación 1 a 1). En el caso de las RNNs, esta restricción no se cumple.
- Las RNNs se usan para análisis de sentimientos (*sentiment analysis*), subtulado de imágenes, reconocimiento del habla, procesado de lenguaje natural, traducción automática, predicción de búsquedas, clasificación de video, etc.

# Redes neuronales recurrentes (RNN)

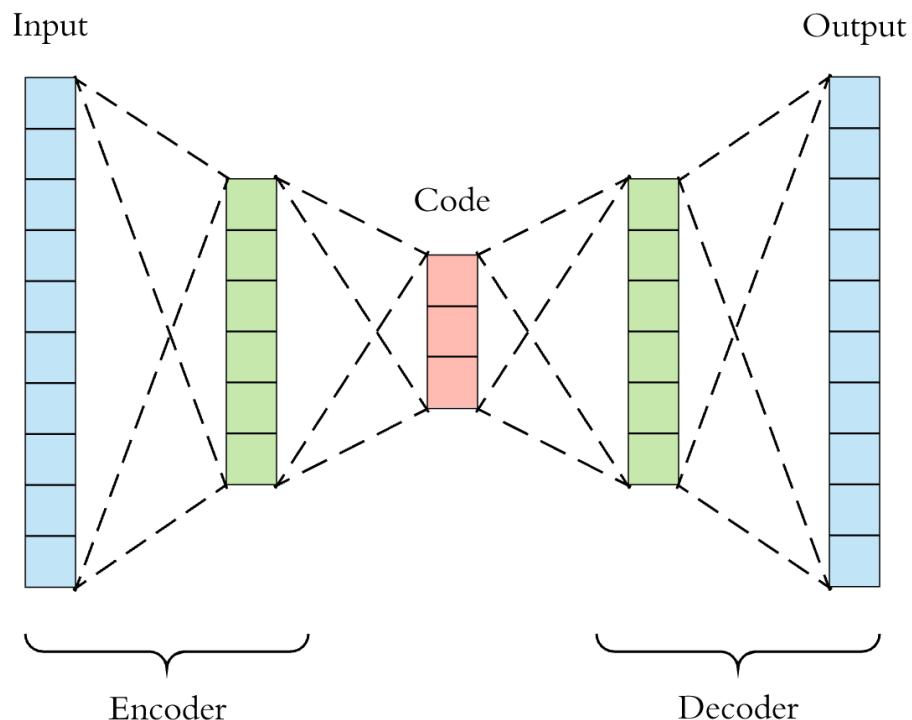


- Las redes LSTM (Long-short Term Memory) representan una de las configuraciones más utilizadas de redes RNN.



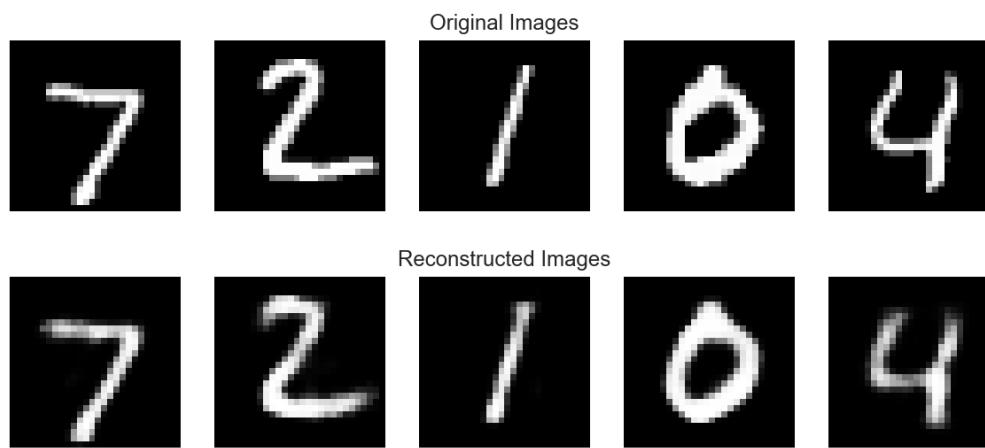
# Autoencoders

- Son un tipo concreto de red neuronal prealimentada (feedforward neural network) en la cual la salida (objetivo) es igual a la entrada: comprime la entrada en un código de menor dimensión, de modo que se pueda reconstruir la entrada a partir de dicho código.
- De este modo, el código se convierte en una compresión (o resumen) de la entrada.
- Esta codificación se conoce también como representación de espacio latente (*latent-space representation*).
- Un autoencoder está compuesto por tres elementos:
  - **Codificador:** comprime la entrada y genera el código.
  - **Código:** representación comprimida de la entrada
  - **Descodificador:** reconstruye la entrada a partir únicamente del código.



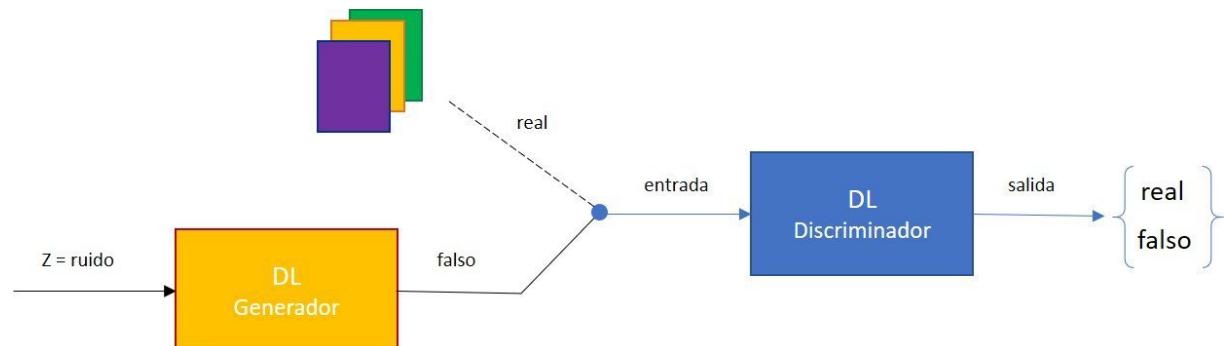
# Autoencoders

- Hay cuatro hiperparámetros que hay que establecer antes de entrenar un autoencoder:
  - **Tamaño del código:** número de nodos de la capa intermedia. Cuanto menor sea el tamaño, mayor la compresión.
  - **Número de capas:** puede tener cualquier profundidad.
  - **Número de nodos por capa:** habitualmente usan capas de tamaño decreciente hasta una capa y a partir de ahí vuelve a aumentar su tamaño.
  - **Función de pérdidas:** típicamente se usa *error medio cuadrático* o *entropía binaria cruzada*.
- Sus principales usos son para eliminación de ruido y *outliers*, y reducción de dimensiones.



# Redes generativa antagónicas (GAN)

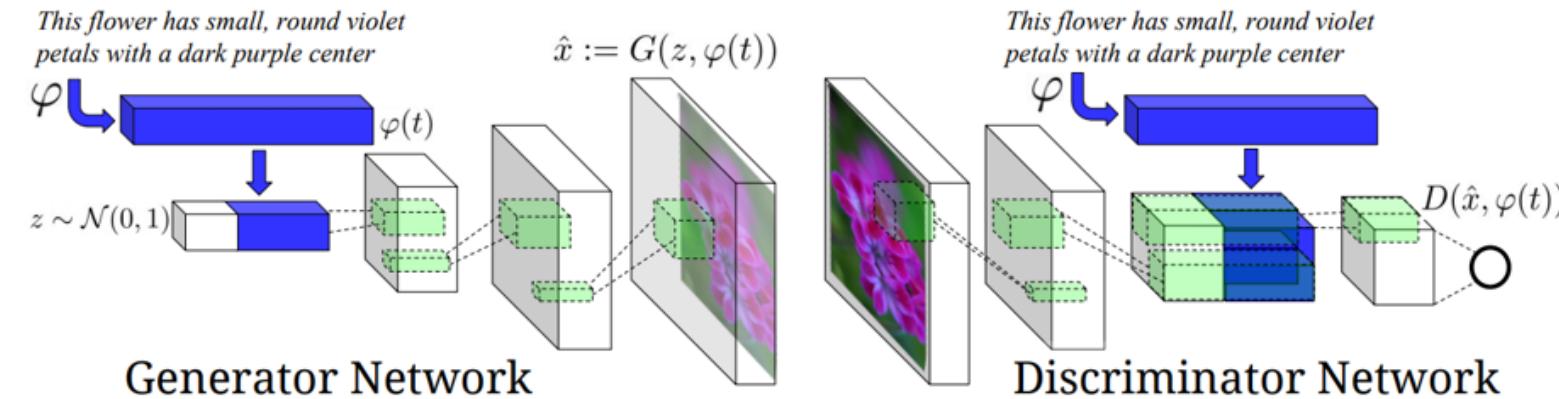
- El objetivo principal de las GAN es generar datos desde cero: dado un conjunto de datos de entrenamiento, la red descubre y aprende patrones en ellos y aprende así a generar nuevos datos.
- Para ello las GAN emplean dos redes neuronales y las enfrentan. La primera red es el "generador" y la segunda es el "discriminador".
- Ambas redes fueron entrenadas con un mismo conjunto de datos, pero la primera (**red generadora**) debe intentar crear variaciones de los datos que ya ha visto. Por ejemplo, en el caso de los rostros de personas que no existen, debe crear variaciones de los rostros que ya ha visto.
- La **red discriminadora** debe identificar si ese rostro que está viendo forma parte del entrenamiento original o si es un rostro falso que creó la red generativa. Cuanto más actúan, la red generativa se hace mejor creando y a la red discriminadora se le hace más difícil detectar si el rostro es falso.
- Hasta ahora las GAN se han usado principalmente en imágenes, aunque también con música.



# Redes generativa antagónicas (GAN)



Rostros generados mediante una GAN.

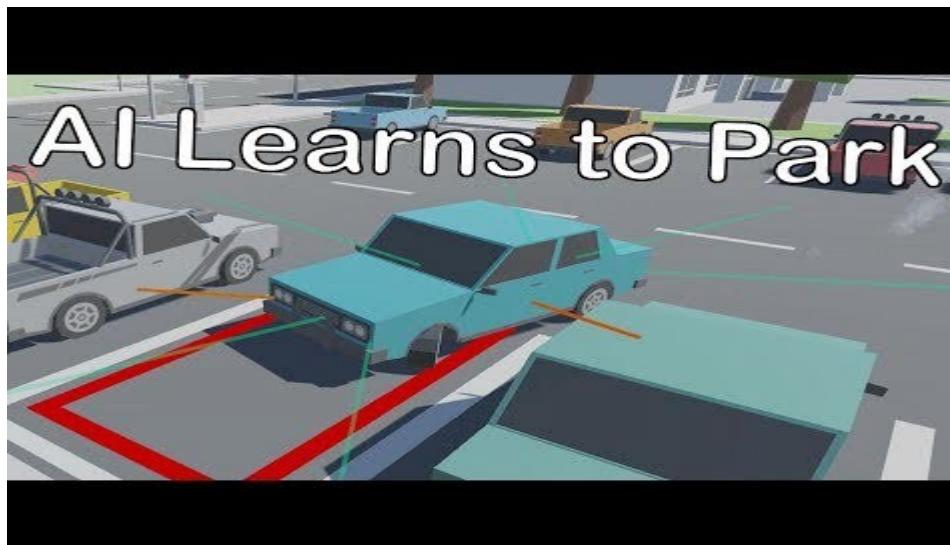


Generación de imágenes a partir de descripciones de texto.

# Aprendizaje por refuerzo

En aprendizaje por refuerzo (*Reinforcement Learning*) no tenemos una “etiqueta de salida”, por lo que no es de tipo supervisado y si bien estos algoritmos aprenden por sí mismos, tampoco son de tipo no supervisado, en donde se intenta clasificar grupos teniendo en cuenta alguna distancia entre muestras.

El *Reinforcement Learning* intentará hacer aprender a la máquina basándose en un esquema de “**premios y castigos**” en un entorno en donde hay que tomar acciones y que está afectado por múltiples variables que cambian con el tiempo.



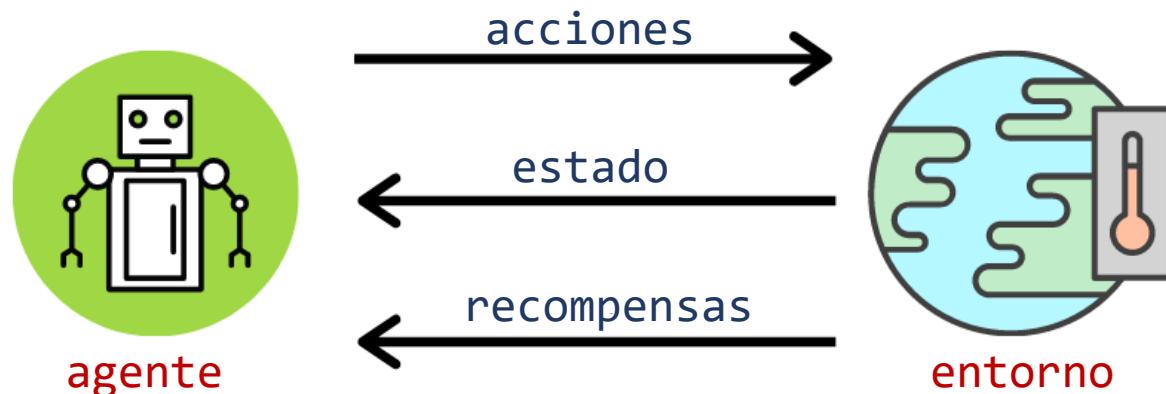
# Aprendizaje por refuerzo: Componentes

El *Reinforcement Learning* define los elementos “agente” y “entorno”:

- **Agente**: será el modelo que se quiere entrenar y que aprenda a tomar decisiones.
- **Entorno**: representa el ambiente donde interactúa y contiene las limitaciones y reglas aplicables en cada momento.

Ambos interactúan por medio de los siguientes elementos:

- **Acciones**: las posibles operaciones que puede realizar en un momento determinado el *Agente*.
- **Estado**: variables que representan la situación del *Entorno*.
- **Recompensas**: consecuencia de las acciones efectuadas por el *Agente*, que pueden ser premios o penalizaciones.



# Aprendizaje por refuerzo: Exploración

## Dilema de exploración/explotación

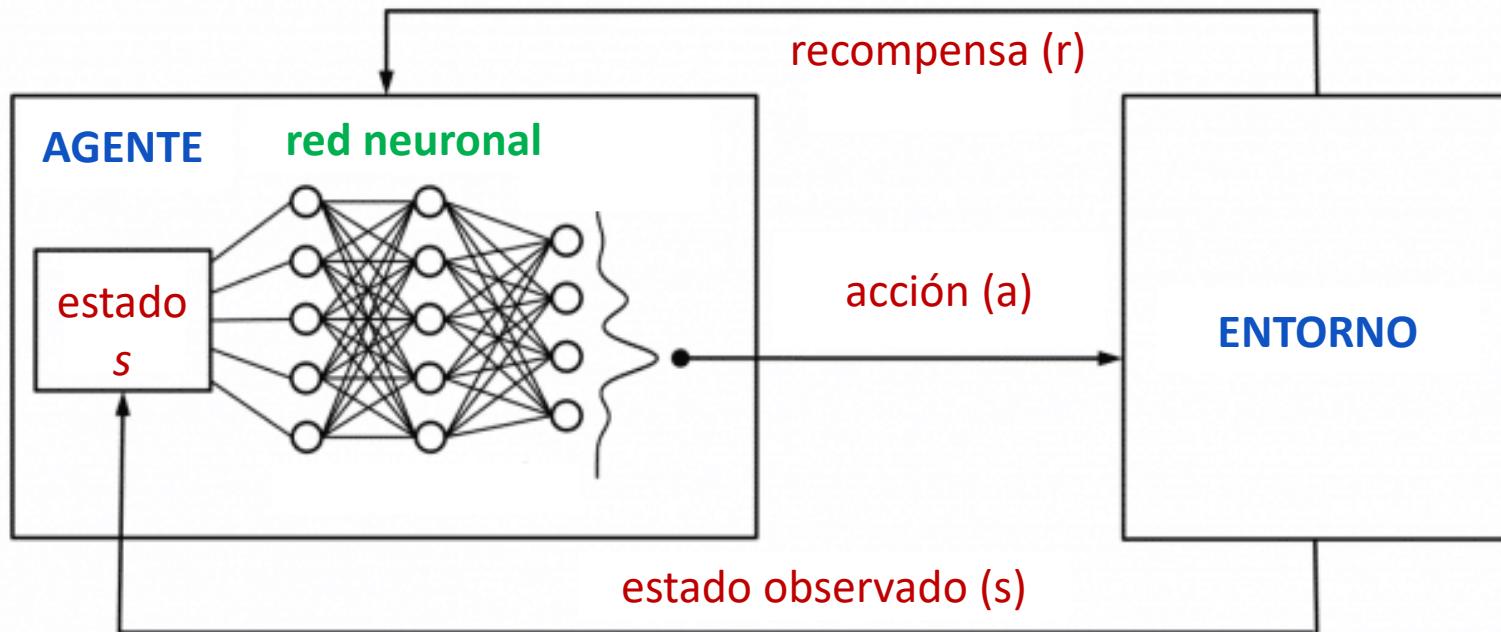
Una vez que el agente descubre que hay acciones que dan recompensa, podría renunciar a explorar acciones nuevas, y recurrir a las que sabe que son recompensadas.

Es necesario lograr un equilibrio entre “explorar lo desconocido y explotar los recursos”.

El agente explorará el ambiente e irá aprendiendo “cómo moverse” y cómo ganar recompensas (y evitar las penalizaciones). Al final almacenará el conocimiento en unas normas también llamadas “políticas”.



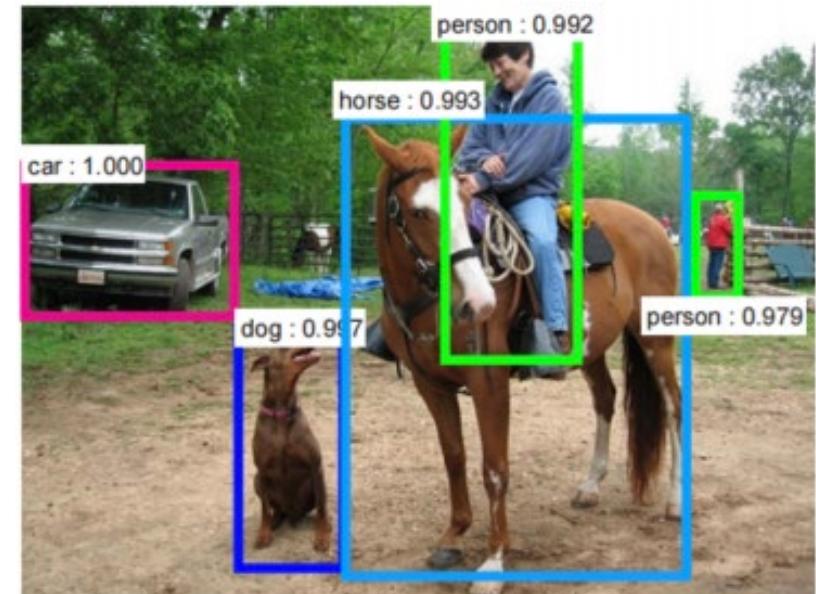
# Deep Reinforcement Learning



# Redes Neuronales Convolucionales CNN

# Redes Neuronales Convolucionales (CNN)

- Objetivo: detectar características de alto nivel en los datos por medio de convoluciones.
- Son la mejor opción para el reconocimiento de objetos en imágenes.
- También es una arquitectura eficaz para el análisis de sonido y de texto, cuando se consideran las palabras como unidades de texto discretas.
- Tienden a ser más útiles cuando los datos de entrada tienen algún tipo de estructura. Ejemplos de esto son las imágenes y el audio, donde aparecen patrones de forma repetida y, valores de entrada vecinos están relacionados.
- **Aplicaciones:**
  - Procesado de imagen, reconocimiento y clasificación
  - Reconocimiento de vídeo
  - Procesado de lenguaje natural
  - Reconocimiento de patrones
  - Motores de recomendación



# Objetivos de las CNN

## Clasificación de una imagen:

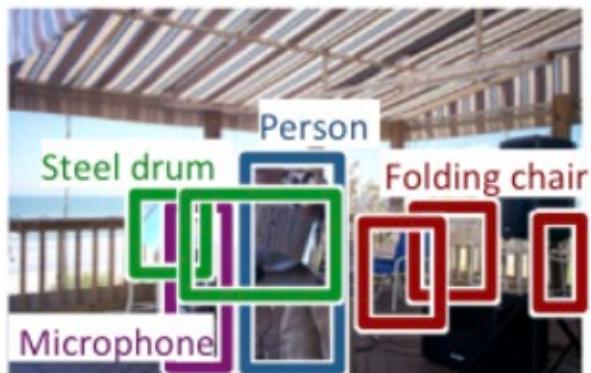
- Predecir el tipo o clase de una imagen. Por ejemplo, reconocer un dígito.
- Entrada: imagen con un único objeto.
- Salida: una etiqueta de clase.

## Localización de objetos:

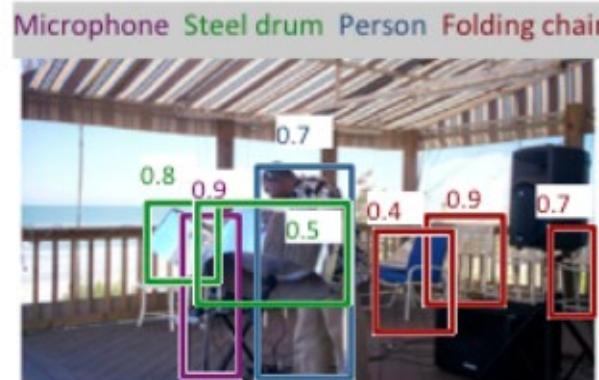
- Localizar la presencia de objetos en la imagen e indicar su ubicación
- Entrada: imagen con uno o más objetos.
- Salida: uno o más cuadros delimitadores (*bounding boxes*) indicando la ubicación de los objetos detectados.

## Detección de objetos:

- Igual que el anterior, pero indicando la clase de cada objeto detectado.
- La salida será un conjunto de cuadros delimitadores con una etiqueta cada uno.



Ground truth



AP: 1.0 1.0 1.0 1.0

# Representación de las imágenes

Los ordenadores perciben las imágenes como matrices bidimensionales de valores numéricos.

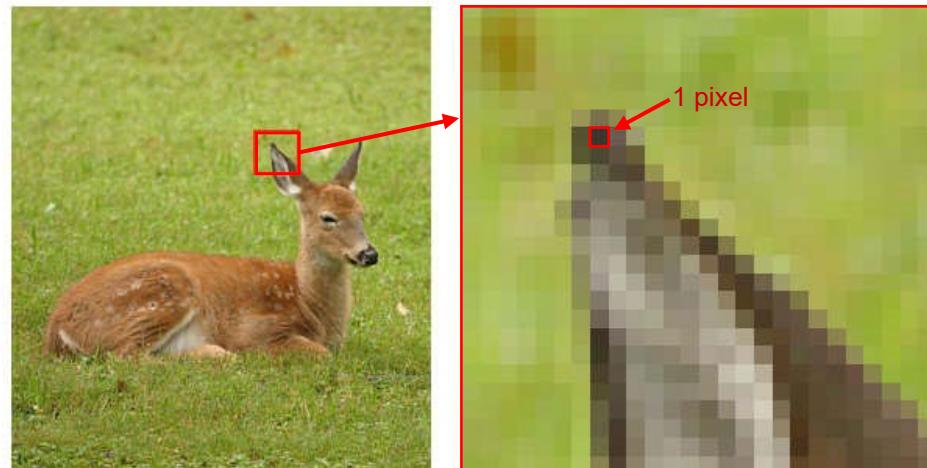
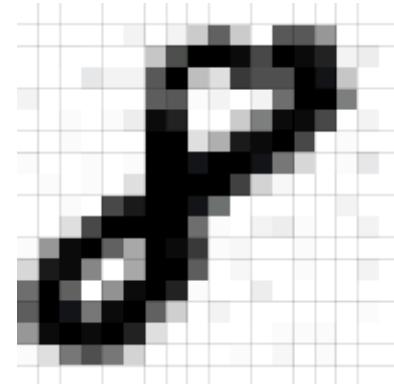
- En el caso de imágenes en escala de gris, cada imagen sólo necesita una matriz bidimensional con los valores de intensidad de cada pixel.



What We See

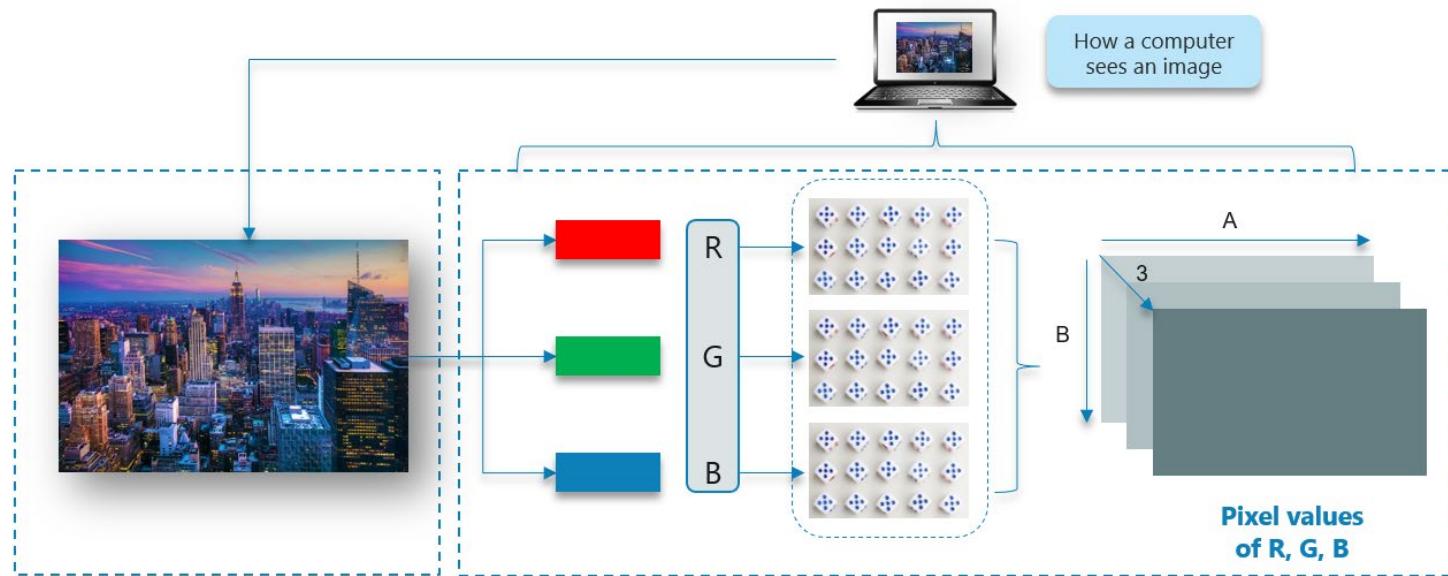
```
08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08  
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00  
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65  
52 70 95 25 04 60 11 42 69 24 68 56 01 52 56 71 37 02 36 91  
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80  
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50  
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70  
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21  
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72  
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95  
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92  
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57  
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58  
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40  
04 52 08 83 97 35 99 16 07 97 57 32 16 26 79 33 27 98 66  
88 36 65 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69  
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36  
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16  
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54  
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
```

What Computers See



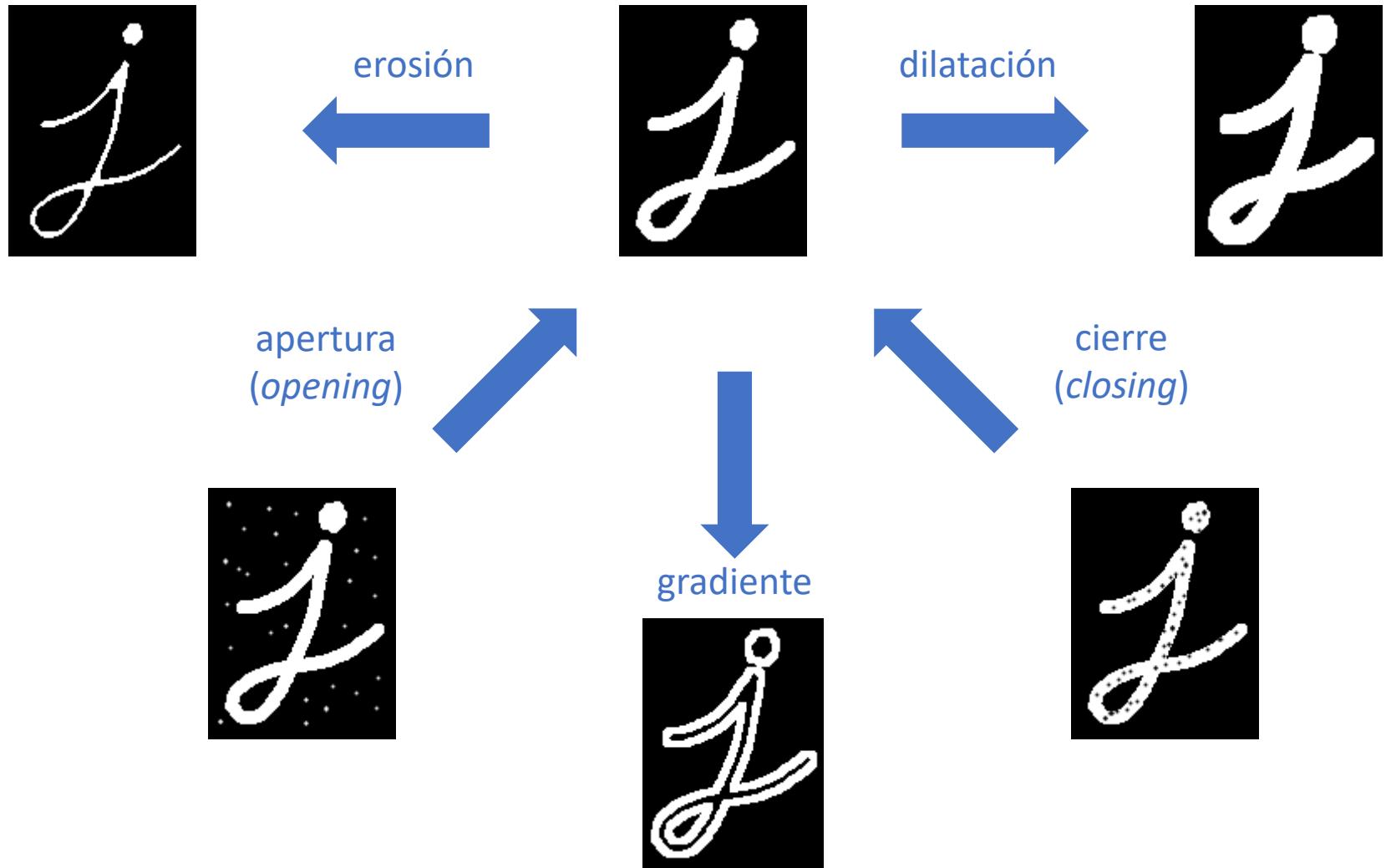
# Representación de las imágenes

- Las imágenes en color se suelen representar con tres matrices, donde cada una de ellas representa la intensidad de las componentes roja, verde y azul de cada pixel.

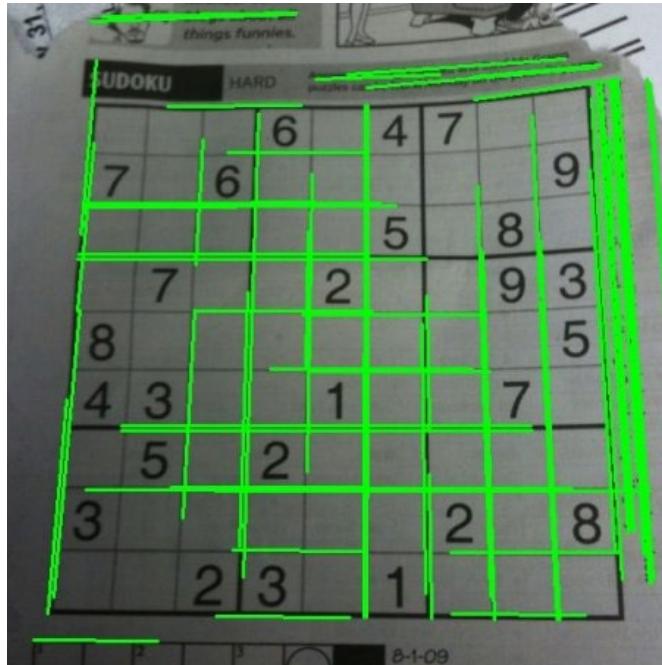


Debe tenerse en cuenta que una imagen es un **aproximación** de la escena real.

# Transformaciones morfológicas



# Transformada de Hough



Detección de rectas



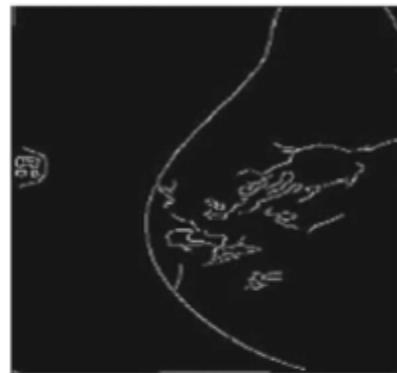
Detección de círculos

# Detección de bordes (edge detection)

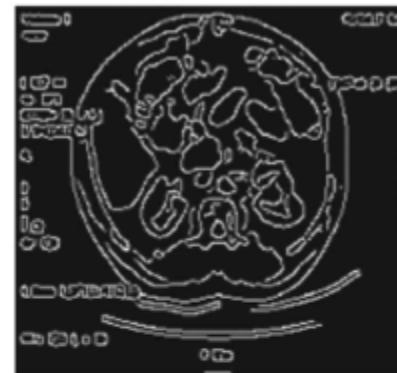
Identificación de los píxeles en los cuales hay un cambio brusco (discontinuidad) en el brillo de la imagen.



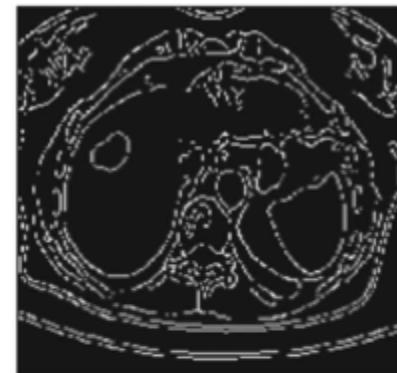
A



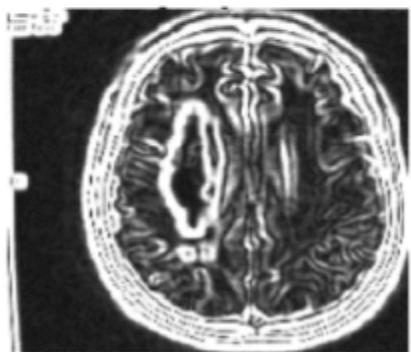
B



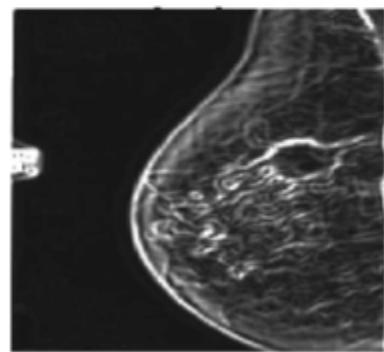
C



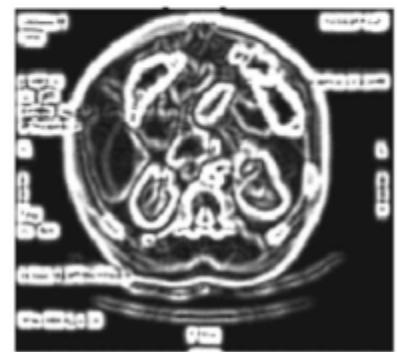
D



E



F



G



H

# Aumento del contraste (*sharpening*)

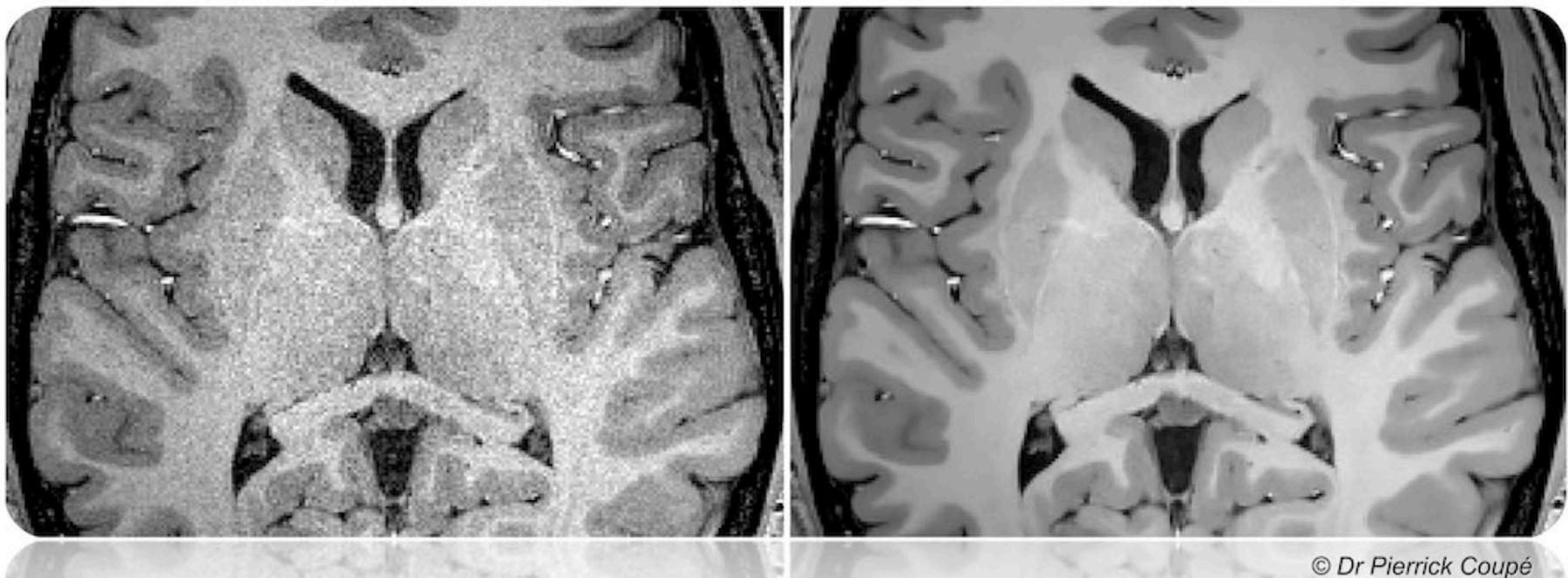


Original

Sharpened

# Eliminación de ruido

El ruido digital es la variación aleatoria (que no se corresponde con la realidad) del brillo o el color en imágenes digitales debido al dispositivo de entrada.



© Dr Pierrick Coupé

# Enfoque

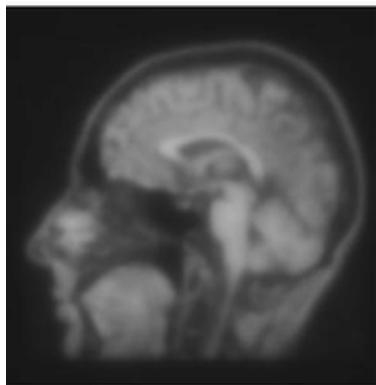


Corrupted license plate

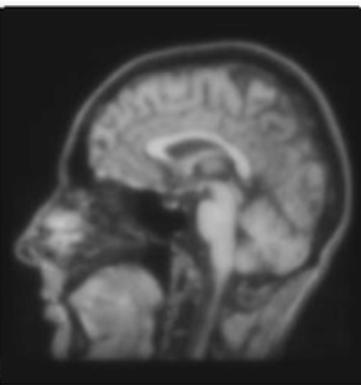


Restored license plate

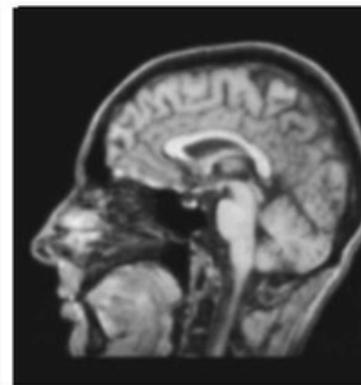
$t=1.0$



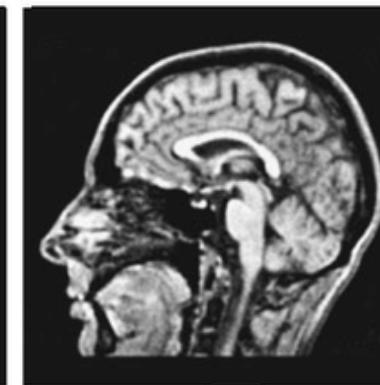
$t=0.8$



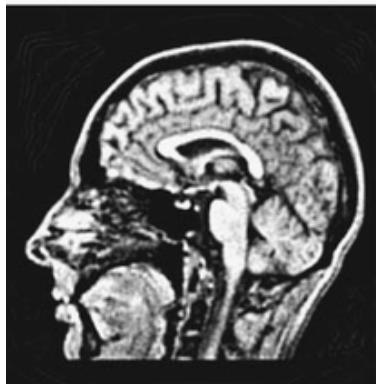
$t=0.6$



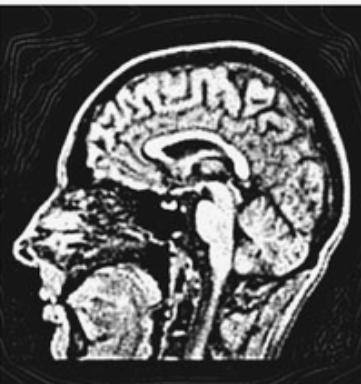
$t=0.4$



$t=0.3$



$t=0.2$



$t=0.1$



$t=0.0$

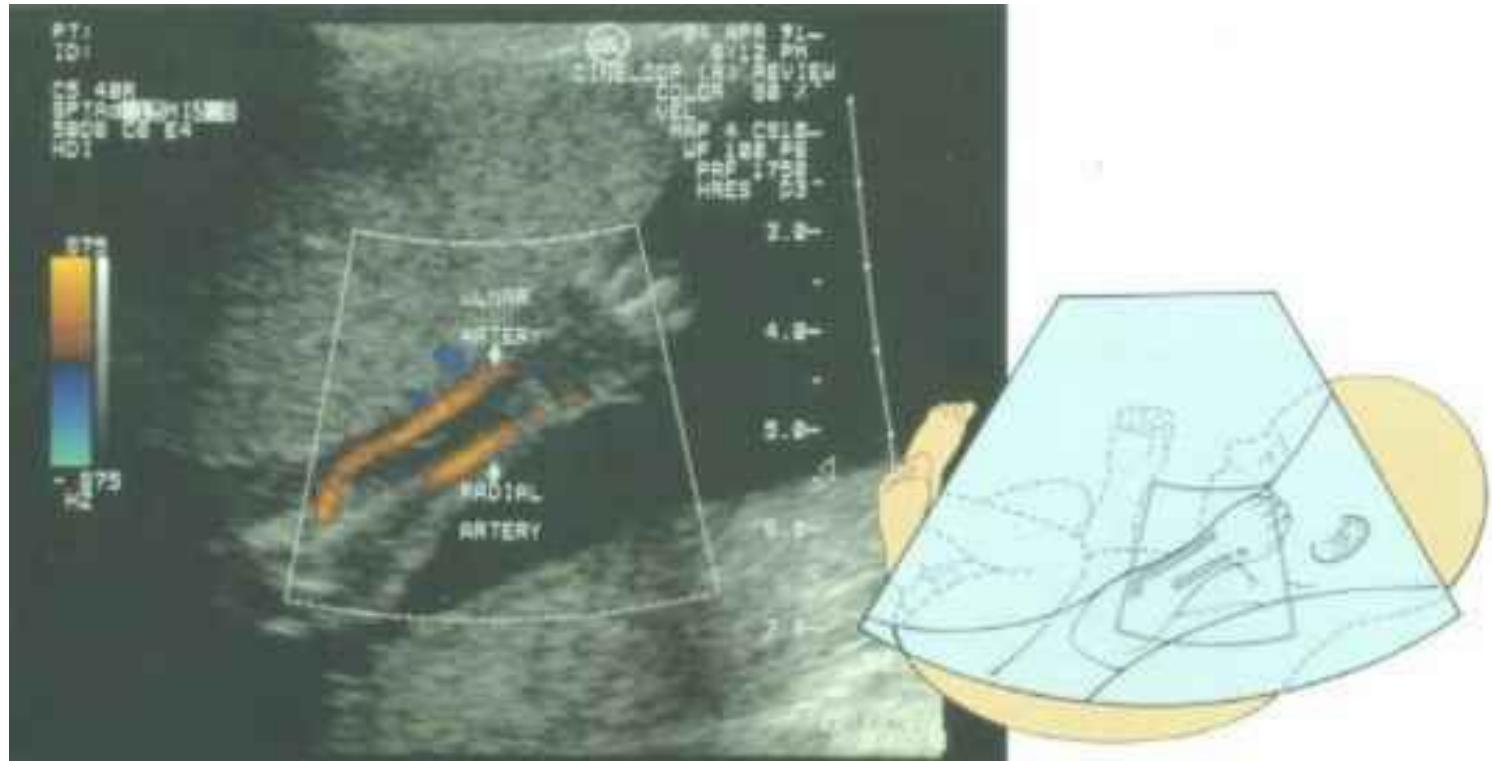


# Segmentación

La segmentación de imágenes es el proceso de particionar una imagen en partes o regiones.

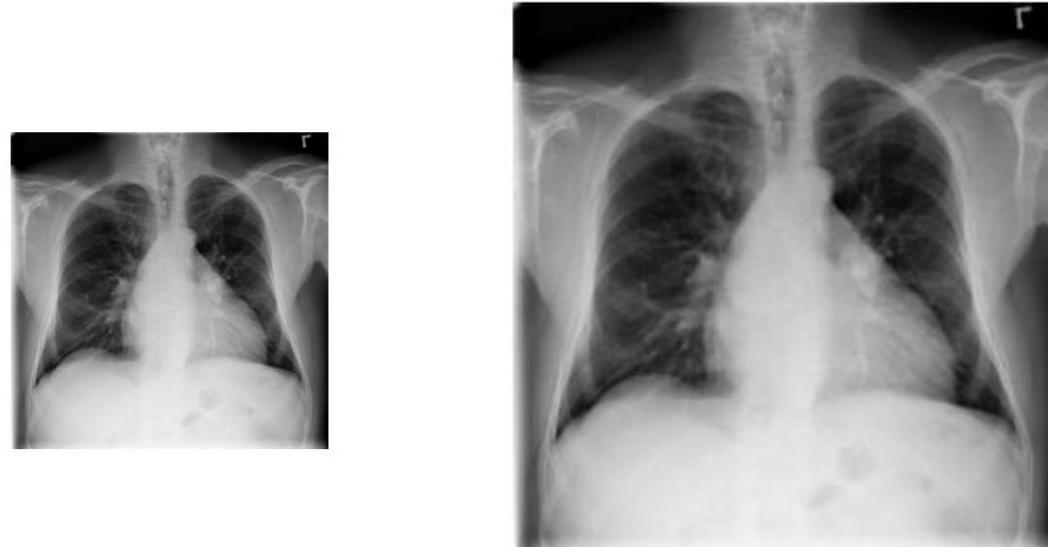
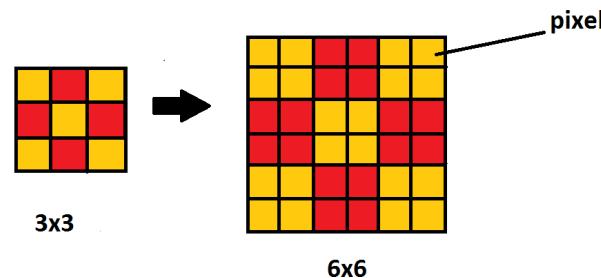


# Combinación de técnicas



# Transformaciones geométricas: escalado (zoom)

Si el factor de escalado es mayor que 1, los píxeles nuevos se calculan replicando o interpolando los existentes



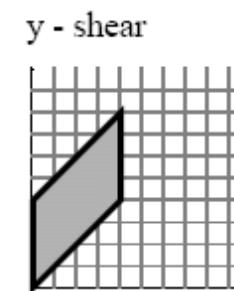
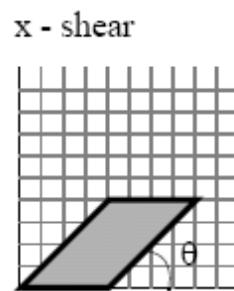
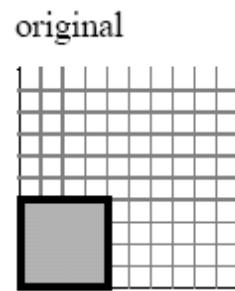
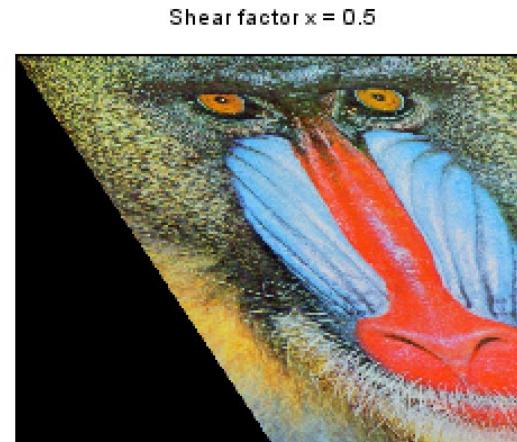
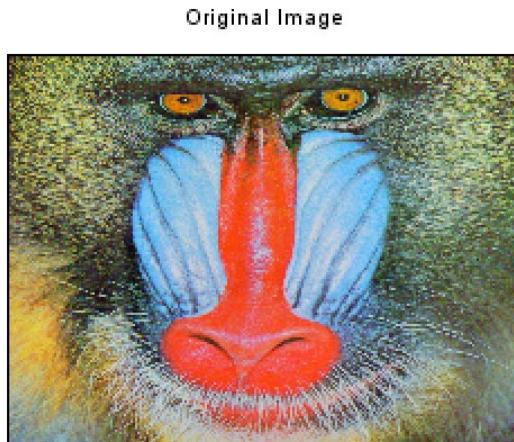
# Transformaciones geométricas: rotación



$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

# Transformaciones geométricas: *shearing*

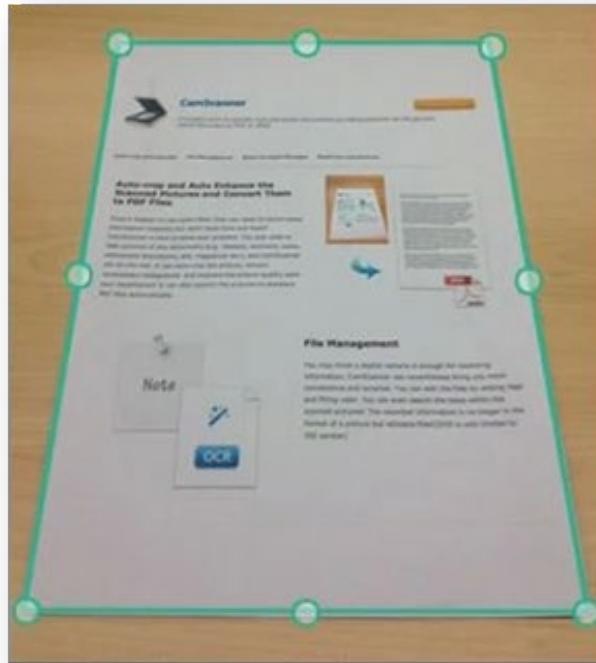


$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

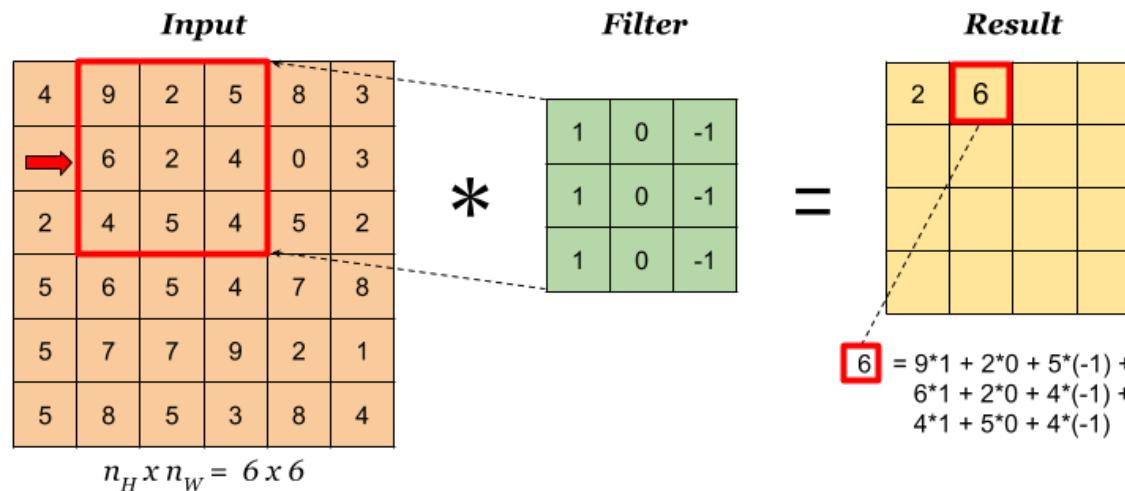
$$a = \cotg (\theta)$$

# Transformaciones geométricas: perspectiva



# Convolución

- La convolución es una operación donde se multiplica, posición por posición, una región de la imagen por una matriz de valores llamado filtro, núcleo o *kernel*.
- La convolución permite extraer características de las imágenes, como por ejemplo bordes y esquinas. El resultado se conoce como mapa de características (*feature map*).



Visualization of the receptive field

0	0	0	0	0	0	0	30
0	0	0	0	50	50	50	
0	0	0	20	50	0	0	
0	0	0	50	50	0	0	
0	0	0	50	50	0	0	
0	0	0	50	50	0	0	
0	0	0	50	50	0	0	

Pixel representation of the receptive field

\*

0	0	0	0	0	0	30	0
0	0	0	0	0	30	0	0
0	0	0	30	0	0	0	0
0	0	0	30	0	0	0	0
0	0	0	30	0	0	0	0
0	0	0	30	0	0	0	0
0	0	0	0	0	0	0	0

Pixel representation of filter

# Convolución y detección de bordes



Vertical edges



Horizontal edges

10	10	10	10	0	0	0	0	0
10	10	10	10	0	0	0	0	0
10	10	10	10	0	0	0	0	0
10	10	10	10	0	0	0	0	0
10	10	10	10	0	0	0	0	0
10	10	10	10	0	0	0	0	0
10	10	10	10	0	0	0	0	0
10	10	10	10	0	0	0	0	0
10	10	10	10	0	0	0	0	0

\*

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

Vertical

0	0	30	30	0	0
0	0	30	30	0	0
0	0	30	30	0	0
0	0	30	30	0	0
0	0	30	30	0	0

# Relleno de la convolución

La imagen resultante de la convolución tendrá dimensiones menores que la imagen original.

- Si, por ejemplo, la imagen original tiene dimensiones  $n \times n$  y el filtro  $f \times f$ , la imagen resultante será  $(n-f-1) \times (n-f-1)$ .
- En algunos casos se aplica un relleno con ceros alrededor de la imagen original. Si no hay relleno, se denomina “convolución válida”
- Si el relleno consigue que la imagen resultante del filtrado tenga las mismas dimensiones que el original la convolución se denomina “same convolution”.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 10 & 10 & 10 & 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} * \begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix} = \begin{matrix} -20 & 0 & 0 & 20 & 20 & 0 & 0 & 0 \\ -30 & 0 & 0 & 30 & 30 & 0 & 0 & 0 \\ -30 & 0 & 0 & 30 & 30 & 0 & 0 & 0 \\ -30 & 0 & 0 & 30 & 30 & 0 & 0 & 0 \\ -30 & 0 & 0 & 30 & 30 & 0 & 0 & 0 \\ -30 & 0 & 0 & 30 & 30 & 0 & 0 & 0 \\ -30 & 0 & 0 & 30 & 30 & 0 & 0 & 0 \\ -20 & 0 & 0 & 20 & 20 & 0 & 0 & 0 \end{matrix}$$

Vertical

# Convolución de imágenes RGB

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

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

Kernel Channel #1

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

Kernel Channel #2

0	1	1
0	1	0
1	-1	1

Kernel Channel #3

308

+

-498

+

164

+ 1 = -25

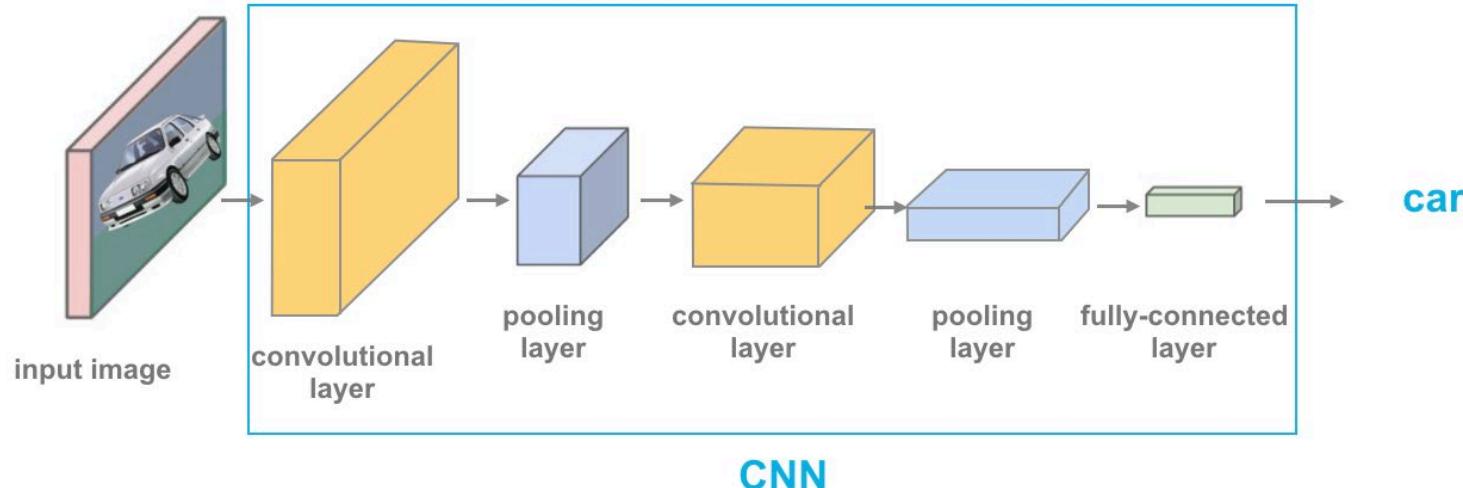
Bias = 1

-25				...
				...
				...
				...
...	...	...	...	...

- El número de canales del filtro debe ser el mismo que los de la imagen original. Si trabajamos con imágenes en color tendremos tres canales: RGB.
- Los valores resultantes de aplicar un filtro a su canal se suman con los resultantes de los otros dos filtros. De ese modo, la imagen resultante sólo tiene un canal.

# Redes neuronales convolucionales (CNN)

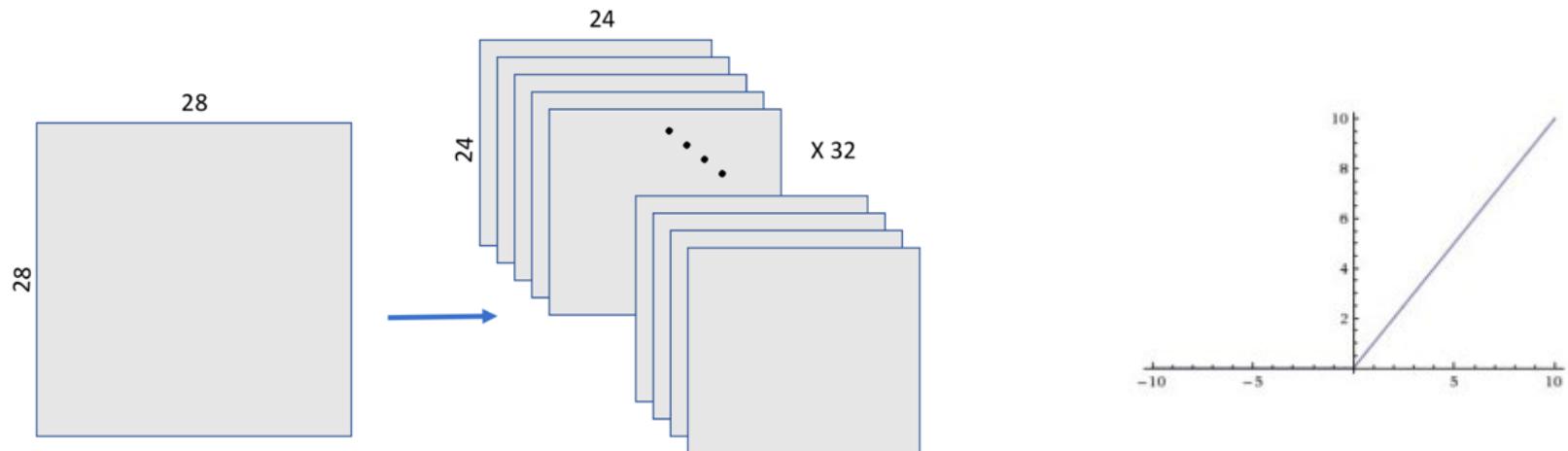
- La arquitectura de una CNN es una secuencia de operaciones de convolución y *pooling*, que termina en una o varias capas densamente conectadas y finalmente una activación *softmax* si la red se usa para clasificación multiclase (p. ej. detectar si en la escena hay peatones, semáforos, vehículos, etc.) o una activación binaria si es un problema de clasificación binaria (p. ej. ¿hay algún peatón en la escena?)
- La combinación de los distintos componentes de la red genera distintas arquitecturas: VGG, ResNet, Inception, etc.



# Capas convolucionales

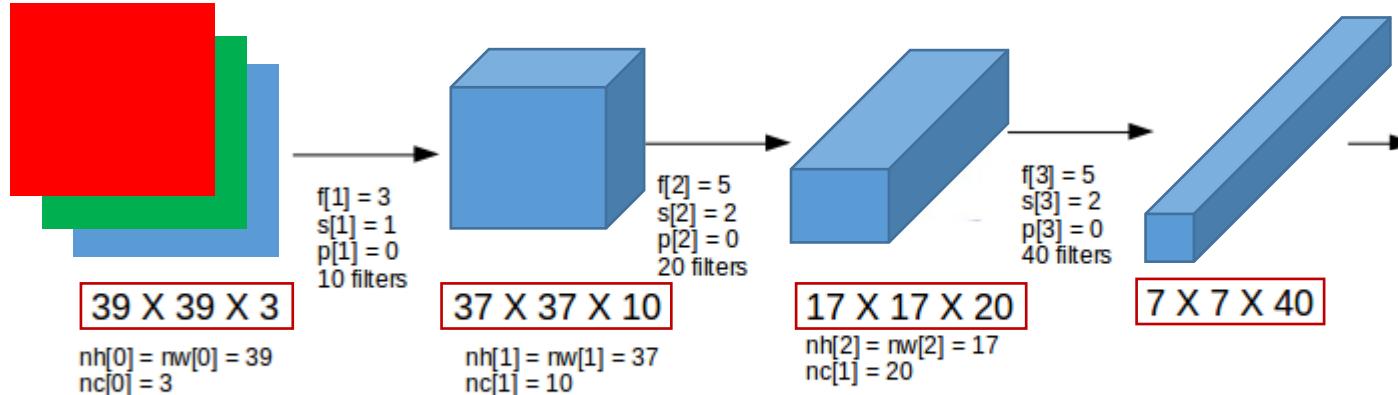
Las primeras capas de una CNN se usan para aplicar convoluciones:

- El paso (*stride*) indica cada cuántos píxeles se aplica el filtro. Si es mayor que uno, la imagen de salida será de un tamaño menor a la original.
- Una primera capa de neuronas ocultas conectadas a las neuronas de la capa de entrada realizará las operaciones convolucionales. No todas las neuronas de entrada están conectadas con todas las neuronas de este primer nivel de neuronas ocultas.
- En general, sobre la imagen original se aplican más de un filtro, obteniendo una nueva imagen con una profundidad (*depth*) igual al número de filtros aplicados.



- A cada pixel resultante se le aplica una función no lineal, que típicamente será la función de activación ReLU: convierte en cero los valores negativos.

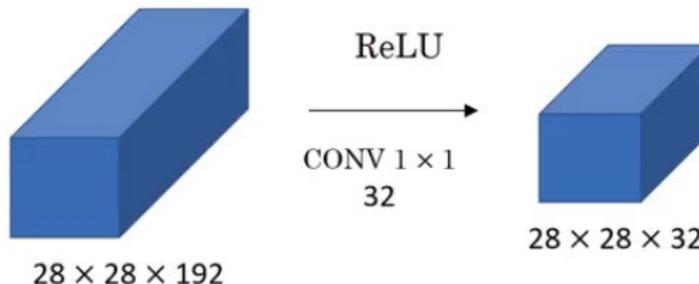
# Capas convolucionales: dimensiones



$$\begin{aligned} nh[1] &= nw[1] = (n + 2p - f) / s + 1 \\ nh[1] &= nw[1] = (39 + 0 - 3) / 1 + 1 \\ nh[1] &= nw[1] = 37 \end{aligned}$$

- $nh[c]$ ,  $nw[c]$ ,  $nc[c]$ : alto, ancho y profundidad (nº de canales) en la capa 'c' (0 para la imagen de entrada)
- $f[c]$ : dimensiones del filtro de la capa 'c'
- $s[c]$ : paso (*stride*) de aplicación del filtro en la capa 'c'
- $p[c]$ : relleno (*padding*) en la capa 'c'

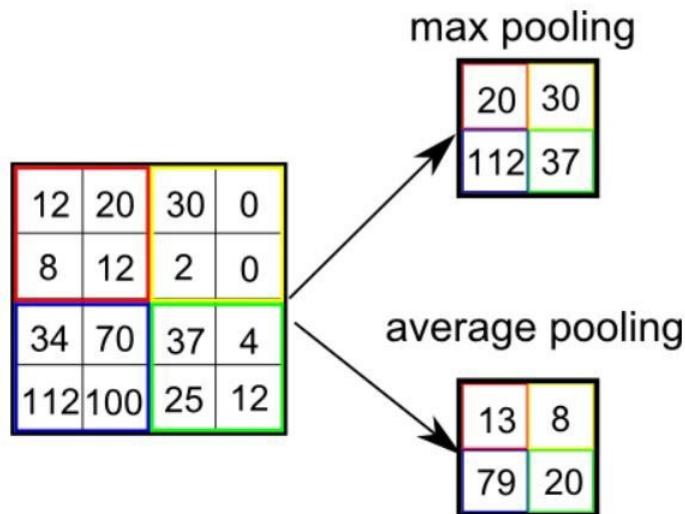
Las **convoluciones 1 X 1** se utilizan para reducir el número de canales.



# Pooling

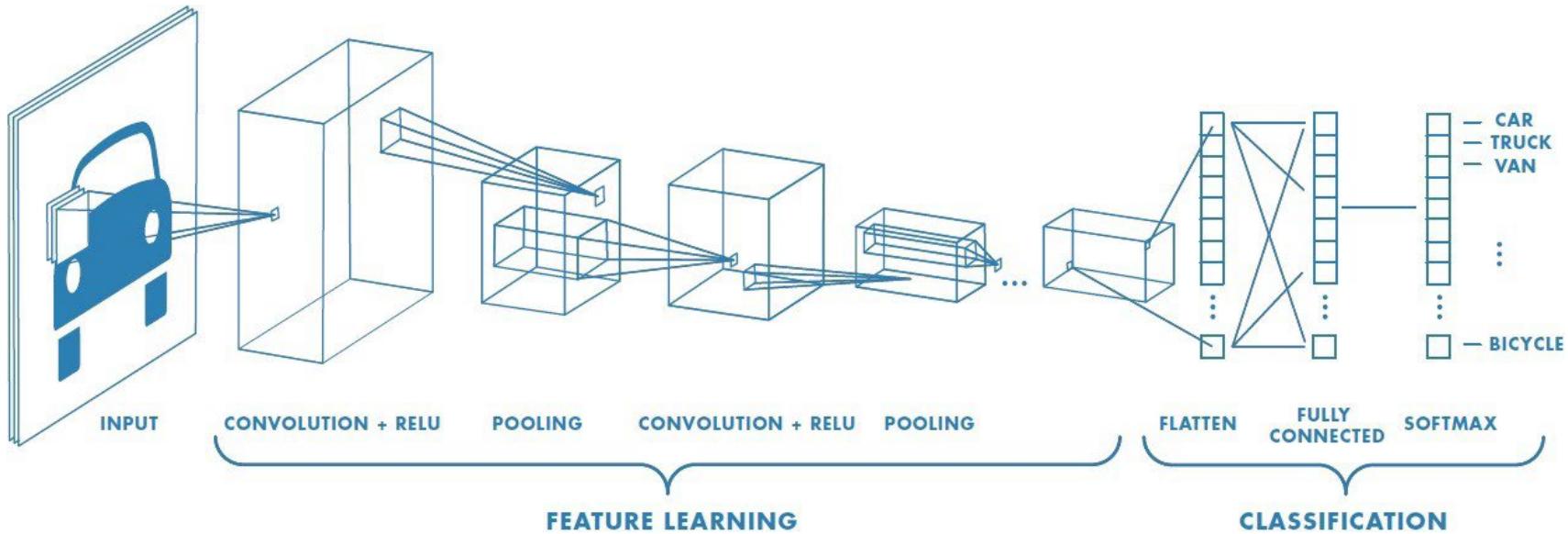
Esta operación reduce las dimensiones de la imagen dividiéndola en áreas y sustituyendo cada una de ellas por un único valor, que suele ser el máximo, la suma o el promedio de los valores del área.

Se utilizan capas de muestreo (*pooling, downsampling o subsampling*) para reducir la cantidad de datos con los que se trabaja y, por tanto, el tamaño de la red, conservando la información.



# Capas totalmente conectadas y Flatten

Tras las capas de convolución y muestreo, se utilizan un conjunto de capas totalmente conectadas para aprender los patrones resultantes de las operaciones previas.



La capa *Flatten* convierte una capa tridimensional de la red en un vector unidimensional para ajustarse a la entrada de la capa totalmente conectada de clasificación.

Por ejemplo, un tensor  $5 \times 5 \times 2$  se convertiría en un vector de tamaño 50.

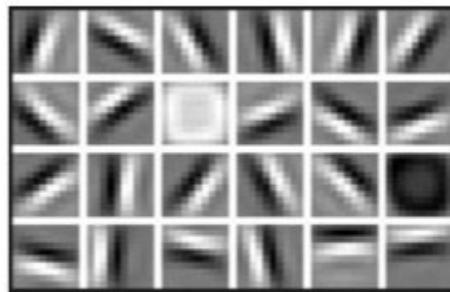
# CNN: parámetros e hiperparámetros

Además de los hiperparámetros propios del MLP final (tamaño del lote, tasa de aprendizaje, número de capas, número de unidades, función de activación, etc.), existe una serie de hiperparámetros adicionales en las CNN:

- Número de capas de convolución y *pooling*
- Función de activación
- Número y dimensiones de los filtros a aplicar en las capas de convolución.
- Paso (*stride*)
- Tipo de muestreo y *stride* de las capas de pooling.
- ...

El proceso de entrenamiento, además de ajustar los pesos de las neuronas del MLP final, también debe aprender el valor de los contenidos de los filtros de las fases de convolución.

# Redes neuronales convolucionales (CNN)



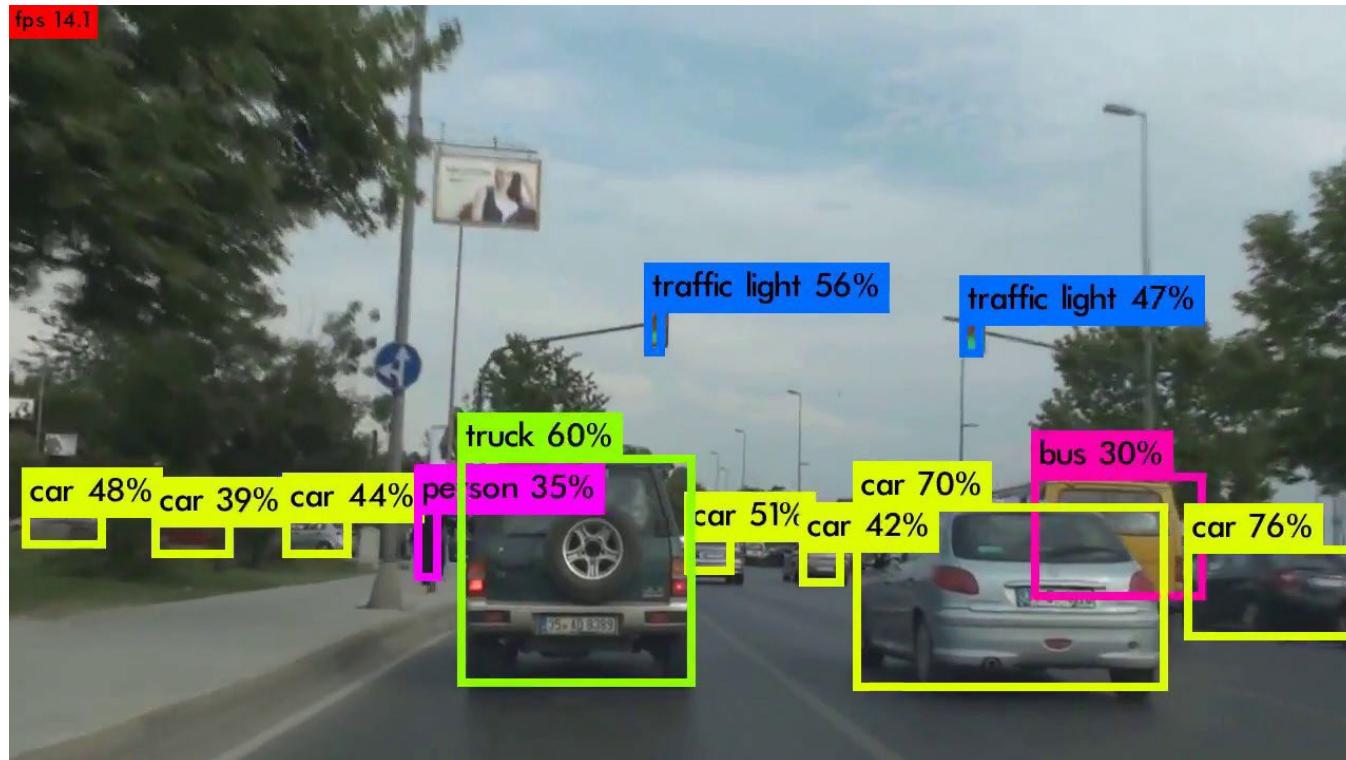
First Layer Representation



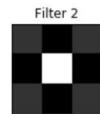
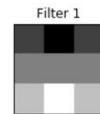
Second Layer Representation



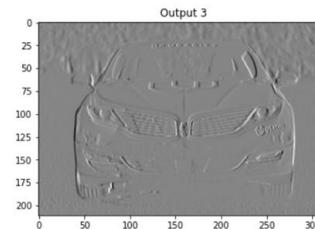
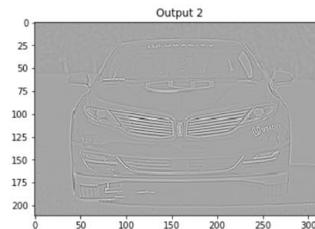
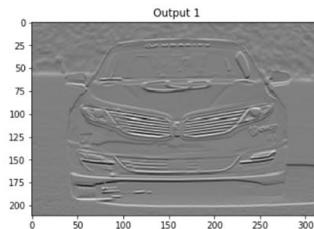
Third Layer Representation



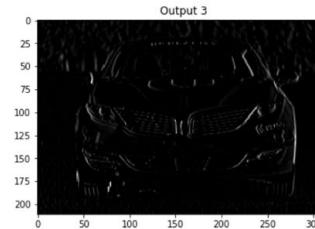
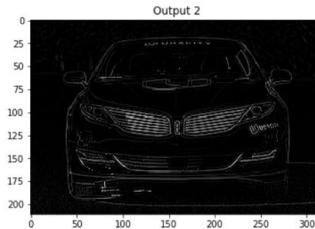
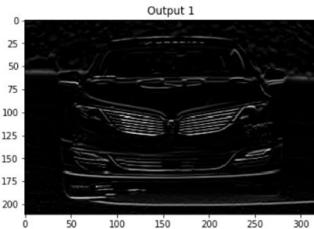
# Redes neuronales convolucionales



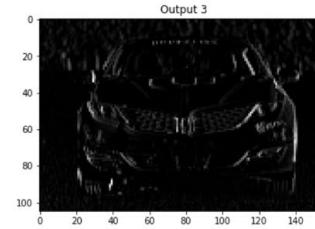
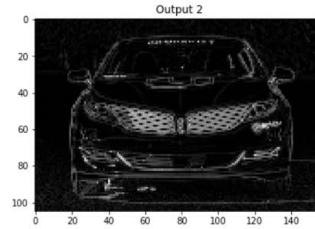
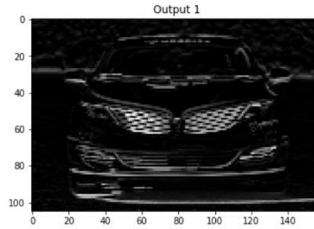
convolutional  
layer



+ ReLu  
(activation)



maxpooling  
layer



# Programación de una CNN

El proceso de construcción y entrenamiento de una CNN es parecido al de una MLP:

1. Redimensionar los datos de entrada usando el método `reshape()`.
2. Para clasificación multclase, aplicar la codificación *one-hot* usando la función `to_categorical()`.
3. El *dataset* se convierte a un array de Numpy.
4. Construir el modelo y añadir las capas necesarias:
  - Capa convolucional: **Conv2D()**
  - Capas de muestreo (pooling): **MaxPooling2D()**, **AveragePooling2D...**
  - Capas de "aplanamiento": **Flatten()**.
  - Capas totalmente conectadas: **Dense()**
  - Otras capas MLP: **Dropout()**, **BatchNorm()**, etc.
5. Compilar el modelo: `compile()`
6. Entrenar el modelo con `fit()`
7. Generar predicciones con `predict()`.

# Programación de una CNN

```
modelo = Sequential()
modelo.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1),
                 activation='relu', input_shape=dims_entrada))
modelo.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
modelo.add(Conv2D(64, (5, 5), activation='relu'))
modelo.add(MaxPooling2D(pool_size=(2, 2)))
modelo.add(Flatten())

modelo.add(Dense(1000, activation='relu'))
modelo.add(Dense(num_clases, activation='softmax'))

modelo.compile(loss=keras.losses.categorical_crossentropy,
                optimizer=keras.optimizers.SGD(lr=0.01),
                metrics=['accuracy'])

modelo.fit(x_entr, y_entr, batch_size=tam_lote, epochs=repeticiones,
            verbose=1, validation_data=(x_val, y_val),
            callbacks=[history])

evaluacion = modelo.evaluate(x_val, y_val, verbose=0)
print('Pérdidas en validación:', evaluacion[0])
print('Exactitud (accuracy) en validación:', evaluacion[1]))
```

# Aumento de datos (*data augmentation*)

- *Image augmentation*: aplicación de transformaciones a las imágenes originales para aumentar el *dataset* de entrenamiento.
- Transformaciones: rotaciones, reflexiones, desplazamientos, etc.
- Obviamente, las versiones modificadas de cada imagen corresponden a la misma clase.
- El *dataset* resultante tiene más variedad, lo que le permitirá generalizar mejor



# *Image augmentation* con Keras

- La clase **ImageDataGenerator** de Keras realiza *Image augmentation* de forma sencilla.
- Proporciona muchas transformaciones diferentes: cambio de brillo, rotación, estandarización, desplazamientos, etc.
- Esta clase trabaja en tiempo real: genera las nuevas imágenes durante el proceso de entrenamiento.
- En cada iteración (*epoch*), **ImageDataGenerator** devuelve imágenes distintas, sin añadirlas al *dataset*.
- Consume muy poca memoria, ya que carga las imágenes por lotes, en lugar de todo el *dataset* a la vez.
- Creación e inicialización del **ImageDataGenerator**:

```
datagen = ImageDataGenerator(rotation_range=10,  
                             width_shift_range=0.2,  
                             height_shift_range=0.2,  
                             zoom_range=0.2,  
                             horizontal_flip=True)
```

# *Image augmentation con Keras*

El método **flow\_from\_directory()** de **ImageDataGenerator** lee las imágenes directamente del directorio y las aumenta durante el entrenamiento.

Este método necesita que las imágenes de cada clase estén en su propio directorio, y todas ellas contenidas en la misma carpeta:

```
| -- train  
|   | -- cat  
|   |   | -- image001.jpg  
|   |   | -- image002.jpg  
|   |   | .....  
|   | -- dog  
|   |   | -- image011.jpg  
|   |   | -- image012.jpg  
|   | .....
```

```
gen_entr = datagen.flow_from_directory(  
    directory= 'dataset/train/',  
    target_size=(400, 400), # redimensionar a este tamaño  
    color_mode="rgb", # trabajar con rgb (color), 'grayscale' (gris)...  
    batch_size=1, # nº imgs. a extraer de cada carpeta para cada lote  
    class_mode="binary", # classes a predecir (tb: 'categorical')  
    seed=2020 # para que sea repetible  
)
```

# *Image augmentation con Keras*

El método `flow_from_dataframe()` de `ImageDataGenerator` se utiliza cuando todas las imágenes están en el mismo directorio y el nombre y su clase están en un DataFrame de Pandas.

Típicamente, existe un fichero csv con el nombre de cada imagen y la clase a la que pertenece. Este fichero se cargaría en un *DataFrame* de Pandas.

Las imágenes se aumentan durante el entrenamiento.

```
gen_entr = datagen.flow_from_dataframe(  
    dataframe=df_train, # data frame con la info de cada imagen  
    directory='dataset/imagenes/', # directorio con las imágenes  
    x_col="image_names", # columna del dataframe con el nombre de la imagen  
    y_col="emergency_or_not", # columna del df con la clase (str)  
    class_mode="binary", # binary o categorical  
    target_size=(200, 200),  
    batch_size=1,  
    rescale=1.0/255,  
    seed=1234  
)
```

# *Image augmentation con Keras*

El método **flow()** lee las imágenes de arrays de Numpy. Por ejemplo, cuando se carga el dataset cifar 10 de Keras:

```
(X_entr, y_ entr), (X_val, y_val) = cifar10.load_data()
```

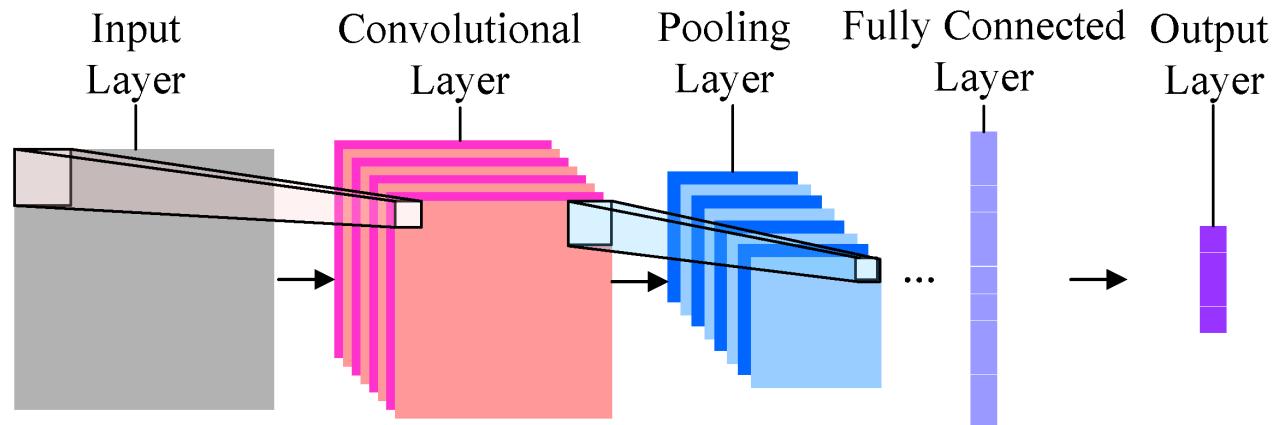
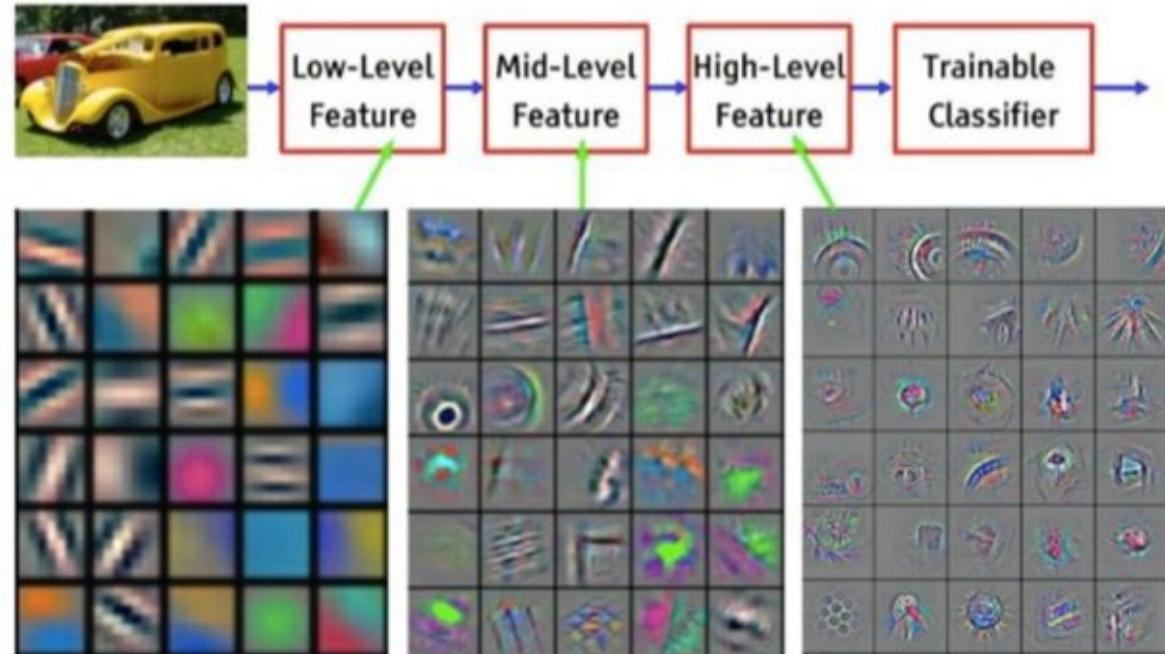
```
gen_entr = datagen.flow(X_entr, y_entr, batch_size=32, seed=1234)
```

- Los tres primeros argumentos son obligatorios.
- Las entradas (*x\_entr* en este ejemplo) debe tener las siguientes dimensiones:  
(*num\_ ejemplos, ancho, alto, canales*)
- Las clases (*y\_entr* en este ejemplo) debe tener las siguientes dimensiones:  
(*num\_ ejemplos, clase*)

# Aprendizaje por transferencia en CNNs

Algunas redes convolucionales necesitan días o semanas de entrenamiento.

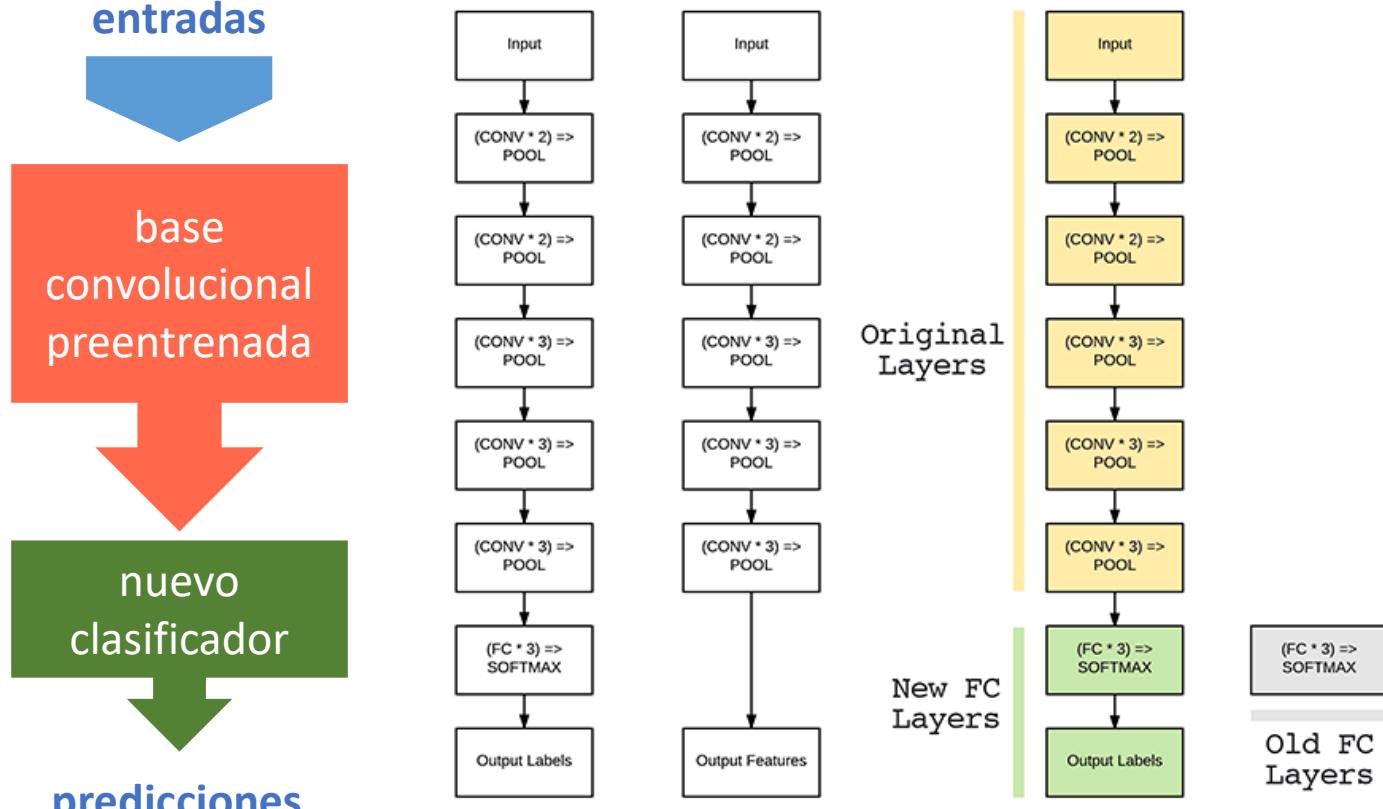
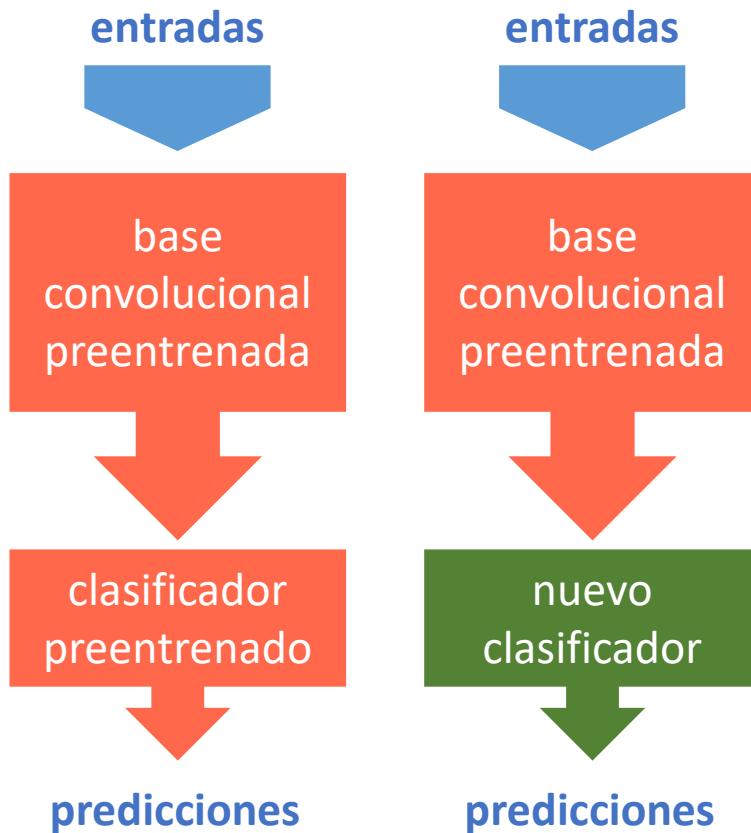
Las primeras capas convolucionales de las CNN extraen características básicas de la imagen (rectas, círculos, etc.) que son comunes a cualquier tipo de imagen.



# Aprendizaje por transferencia en CNNs

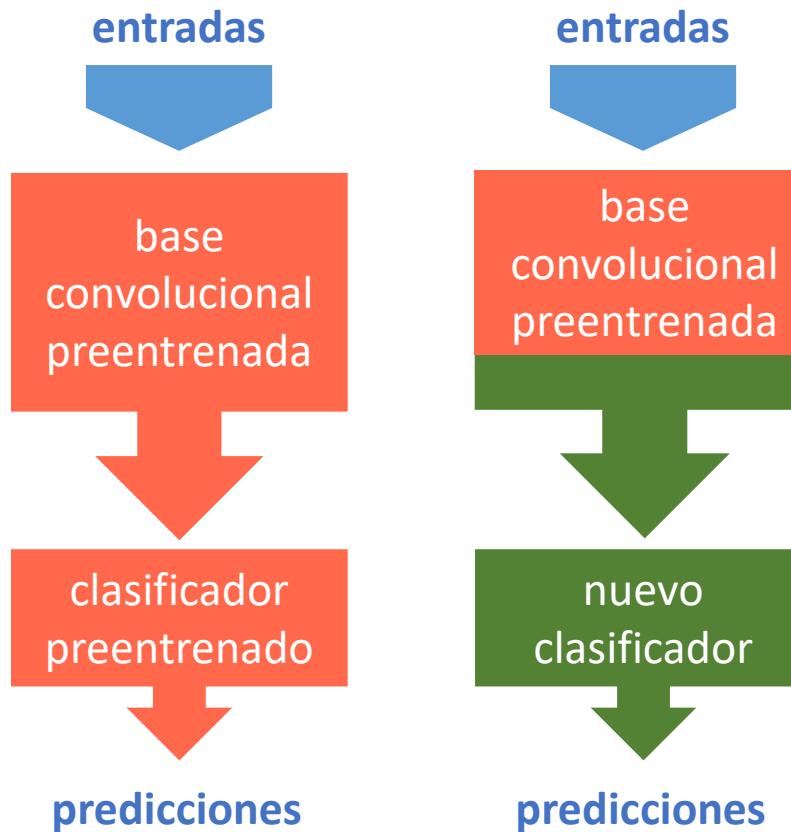
El aprendizaje por transferencia (*transfer learning*) consiste en partir de una red preentrenada y luego aplicar una de las dos siguientes estrategias:

- *Feature extraction*: se mantienen las primeras capas (base convolucional) sin cambios y se reentrenan las últimas capas (clasificador), que suelen ser capas densas (*fully connected*) y que son específicas para las imágenes que se quieren identificar:

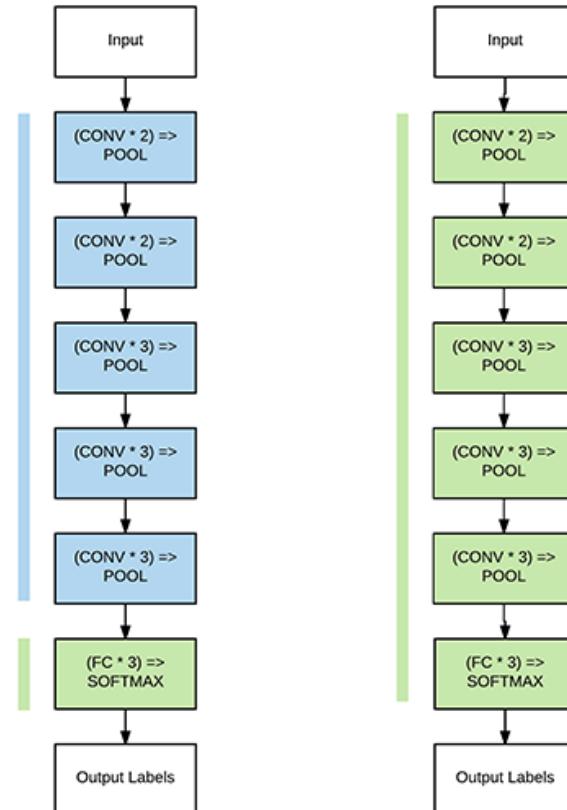


# Aprendizaje por transferencia en CNNs

- *Fine tuning*: además de añadir y entrenar un nuevo clasificador, también se entranan algunas (o todas) de las últimas capas de la base convolucional. Estas últimas se entranan con una tasa de aprendizaje baja para que los pesos no cambien demasiado.



*Fase 1: entrenar solo el clasificador*  
*Fase 2: entrenar el clasificador y todas o algunas de las últimas capas de la base*



# Aprendizaje por transferencia en Keras

Keras permiten aplicar esta técnica de manera sencilla:

- Dispone de modelos preentrenados que se pueden descargar.
- Permite indicar qué capas son entrenables (*trainable layers*) y cuales no (*frozen layers*).

Para la mayoría de las redes preentrenadas se ha utilizado el banco de imágenes ImageNet, que dispone de más de 14 millones de imágenes etiquetadas.

Entre los modelos preentrenados disponibles en Keras, tenemos: Xception, VGG16, VGG19, ResNet50, InceptionV3, InceptionResNetV2, MobileNet, MobileNetV2, DenseNet y NASNet.

```
from tensorflow.keras.applications.vgg16 import VGG16
```

Cuando se instancia un modelo, automáticamente se descargan los pesos:

```
modelo = ResNet50(weights='imagenet')
```

Con el argumento *include\_top=False* el modelo se descarga sin el clasificador final:

```
modelo=VGG16(weights='imagenet', include_top=False, input_shape=(150, 150, 3))
```

# *Feature extraction con Keras*

```
import pandas as pd
...
from keras.applications import VGG16
from keras import models
from keras import layers
from keras.preprocessing.image import ImageDataGenerator

# importa el modelo mobilenet sin la última capa (el clasificador)
base = VGG16(weights='imagenet', include_top=False, input_shape=(150, 150, 3))

modelo = models.Sequential()
modelo.add(base)
modelo.add(layers.Flatten())
modelo.add(layers.Dense(256, activation='relu'))
modelo.add(layers.Dense(1, activation='sigmoid'))
...
modelo.compile(...)
historia = model.fit(...)

...
```

# Ajuste fino de modelo preentrenado en Keras

```
# crea y entrena modelo como en feature extraction y a continuación, los siguientes pasos.  
  
# en la fase anterior de feature extraction se congelaron todas la capas convolucionales  
base.trainable = True # se descongelan las capas convolucionales  
  
N = 5 # número de capas que se mantienen congeladas  
  
for pos in range(N):  
    base.layers[pos].trainable = False # se congelan las primeras capas  
  
modelo.compile(...)  
  
historia = modelo.fit(...)
```

# Redes preentrenadas en Keras

Las principales arquitecturas de redes neuronales ya están construidas y preentrenadas en Keras: InceptionResNetV2, Xception, ResNet152V2, ResNet101V2, ResNet152, etc.

```
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.applications.resnet50 import decode_predictions
...
import numpy as np

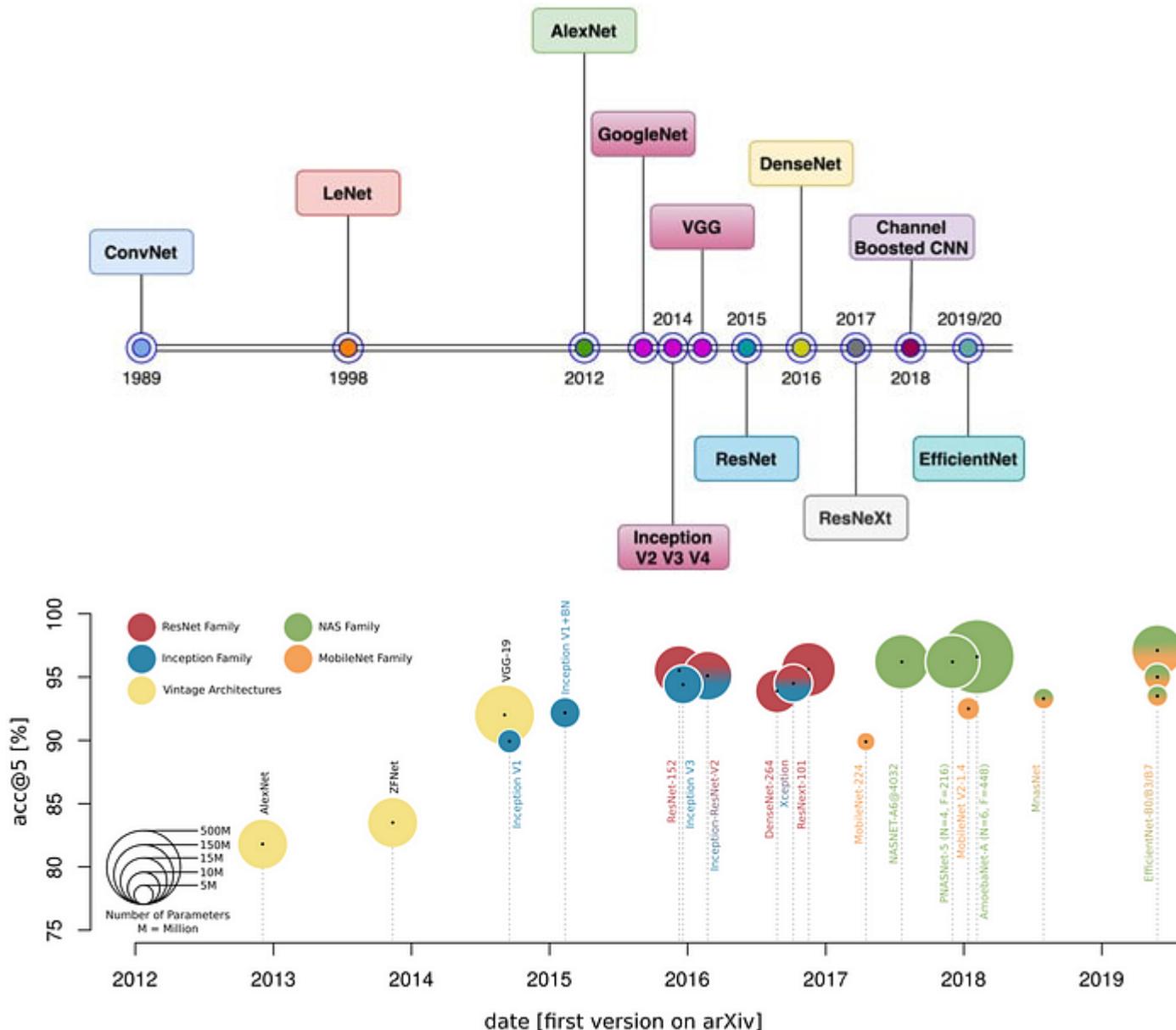
modelo = ResNet50(weights='imagenet') # imagenet/None (aleatorios)/ruta fichero pesos

fich_imagen = 'elephant.jpg'
img = image.load_img(fich_imagen, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x) # formatea la imagen según necesite esta arquitectura

predicciones = modelo.predict(x)
# resultados en una lista de tuplas (clase, descripción, probabilidad)

print('Predicción:', decode_predictions(predicciones, top=2)[0])
#[('n02504013', 'Indian_elephant', 0.82658225), ('n01871265', 'tusker', 0.1122357)]
```

# Redes neuronales convolucionales profundas



# Redes neuronales convolucionales profundas

## Layers

- conv 3×3** Convolutional operations, in red
- avg-pool 2×2** Pooling operations, in grey
- concat** Merge operations eg. concat, add in purple
- Dense layer, blue**

## Activation Functions

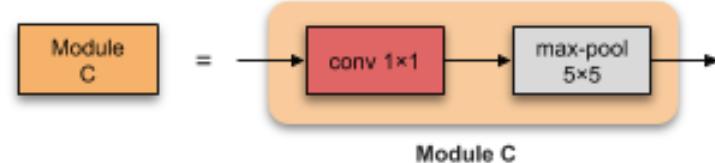
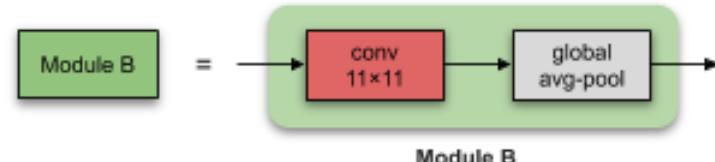
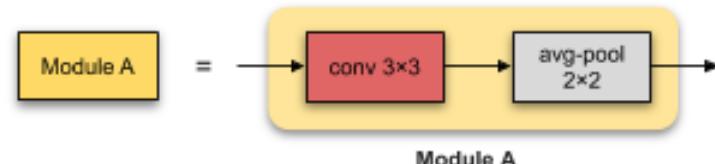
- T** Tanh
- R** ReLU

## Other Functions

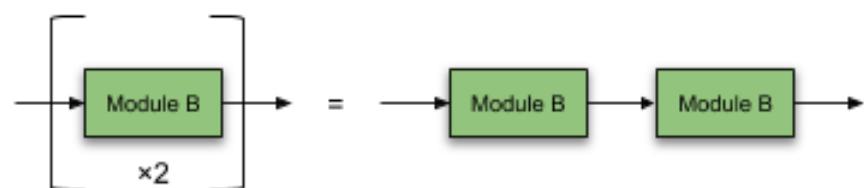
- B** Batch normalisation
- S** Softmax

## Modules/Blocks

Modules (groups of convolutional, pooling and merge operations), in **yellow**, **green**, or **orange**. The operations that make up these modules will also be shown.

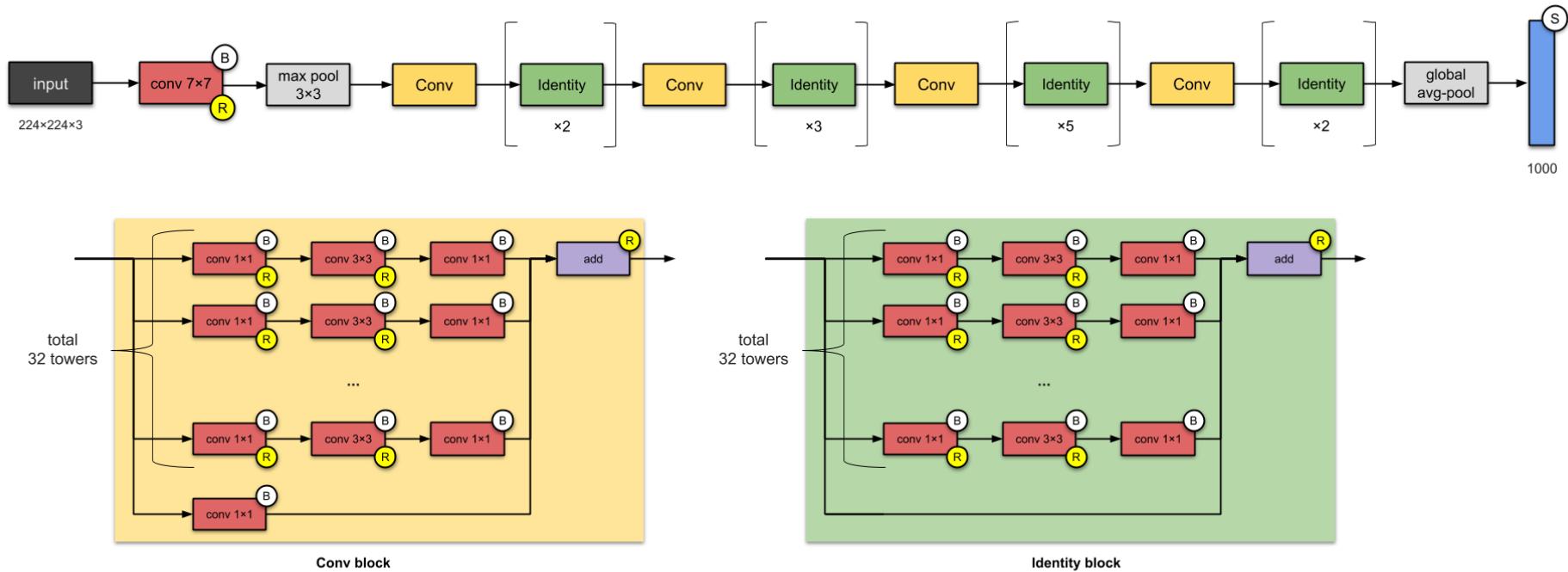


## Repeated layers or modules/blocks



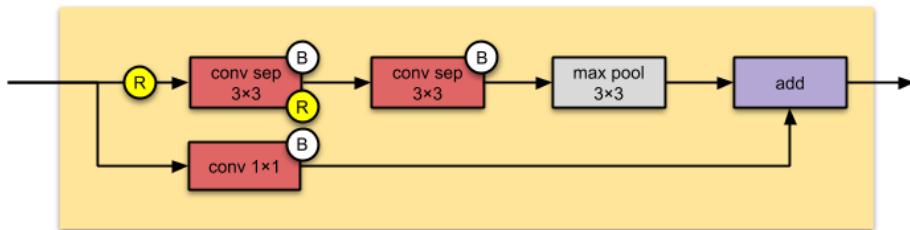
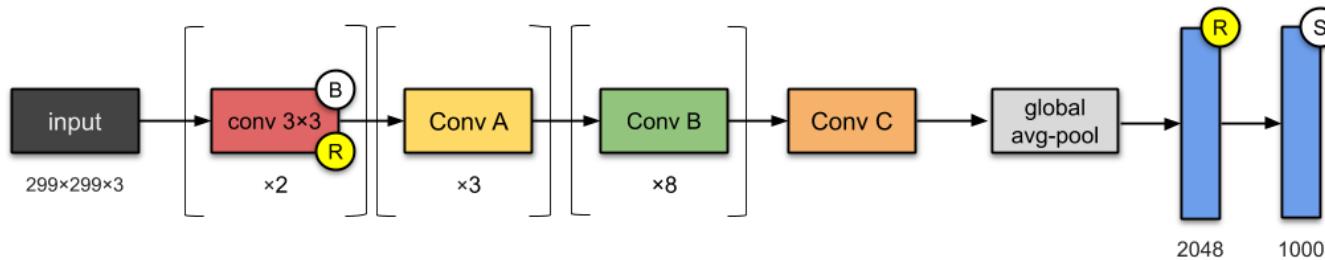
# Redes neuronales convolucionales profundas

ResNeXt-50

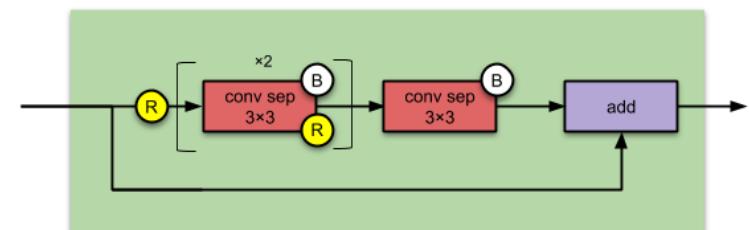


# Redes neuronales convolucionales profundas

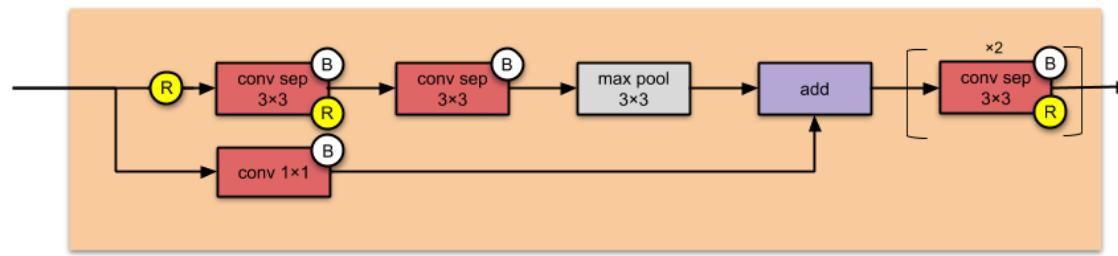
Xception



Conv A



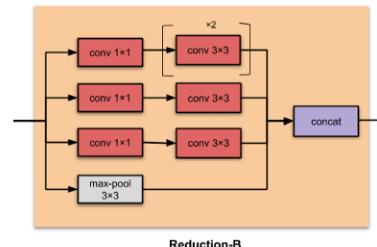
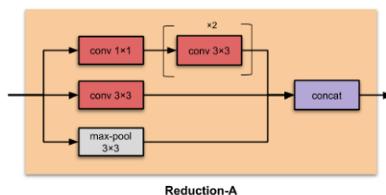
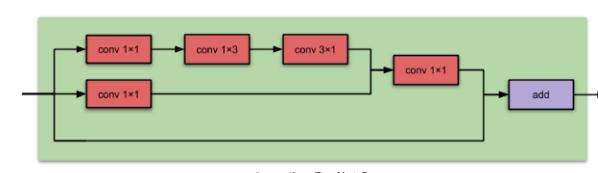
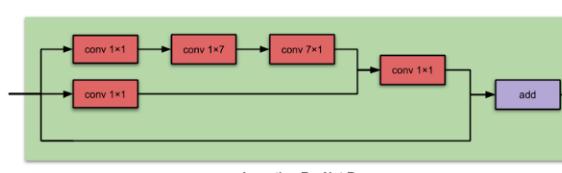
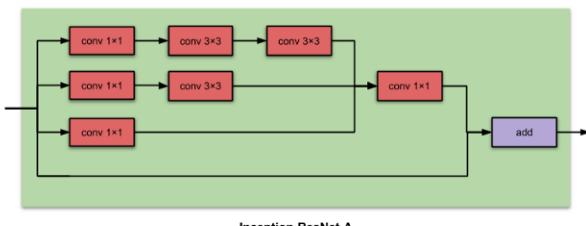
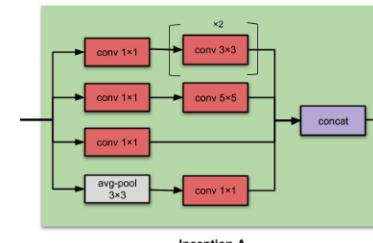
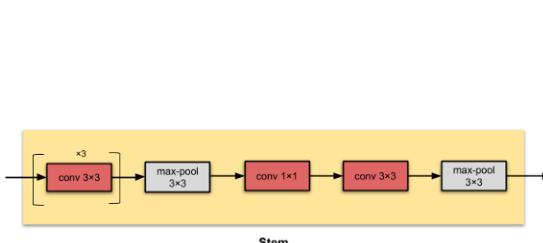
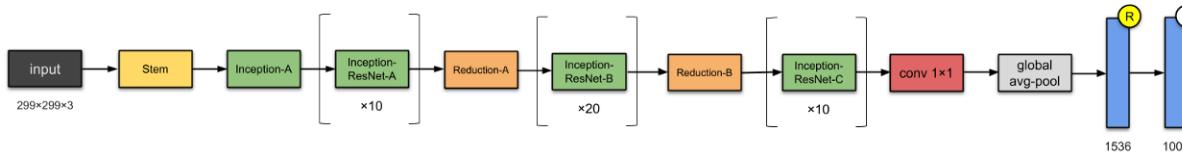
Conv B



Conv C

# Redes neuronales convolucionales profundas

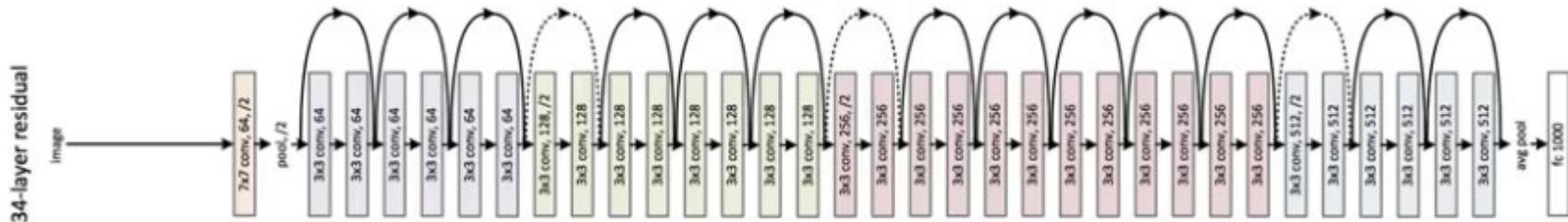
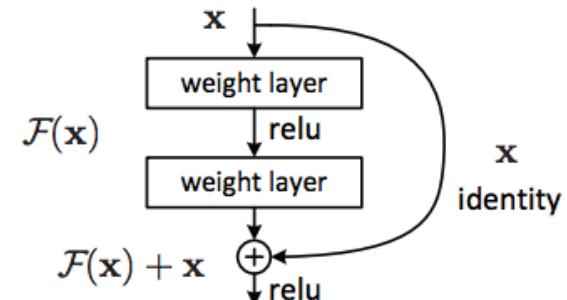
## Inception-ResNet-V2



# Redes convolucionales residuales (ResNets)

El entrenamiento de redes muy profundas puede presentar problemas de desvanecimiento o explosión del gradiente. Las redes residuales mejoran el rendimiento de las redes muy profundas usando el bloque residual:

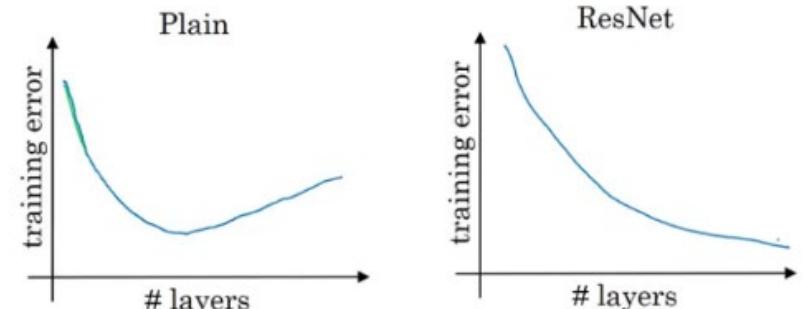
$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s \mathbf{x}.$$



La idea básica consiste en sumar la salida de la capa  $i$  a la entrada de la capa  $i+2$ .

Cuando los dos sumandos  $x$  y  $F(x)$  no tienen las mismas dimensiones, se puede solucionar con rellenos y convoluciones 1x1.

El beneficio de este tipo de redes es que, a diferencia de lo que ocurre con los MLP, el error siempre decrece con el número de capas.



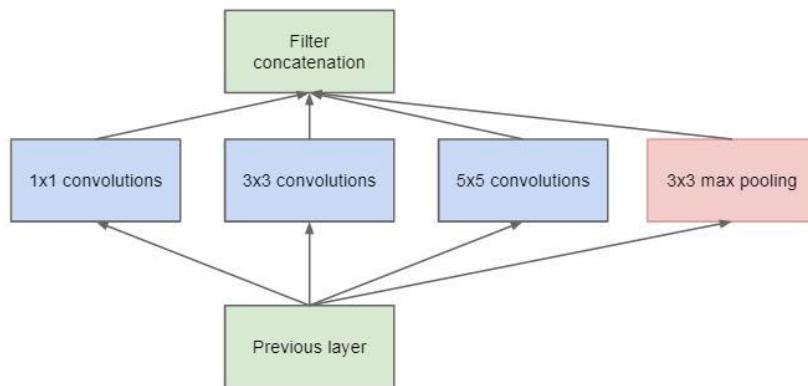
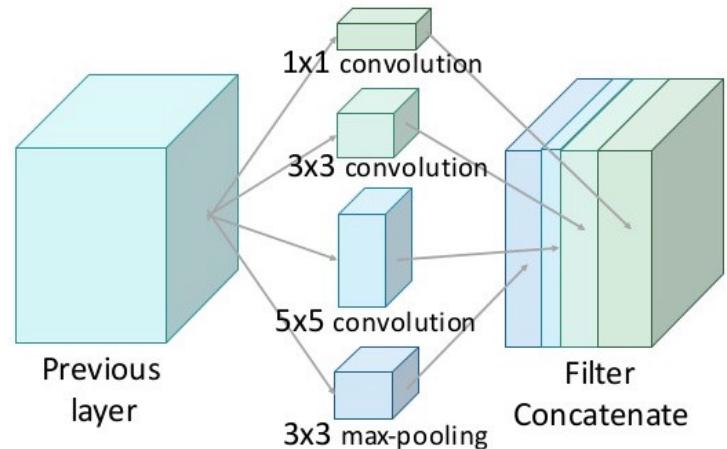
# CNNs: Inception

En una CNN tradicional, la salida de una convolución de  $5 \times 5$  extrae características diferentes de las de un  $3 \times 3$  o un máximo de agrupación o a un muestreo (*pooling*).

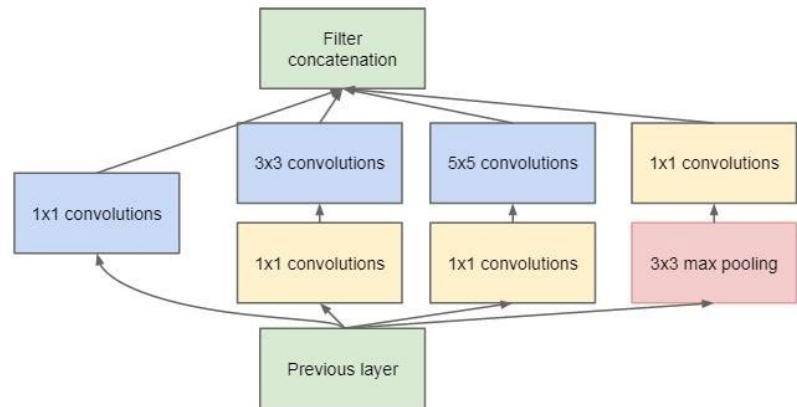
El modelo *inception* calcula varias y deja que el entrenamiento decida cuáles son más interesantes.

Este enfoque implica un incremento importante de la carga computacional, que se mitiga reduciendo la dimensionalidad por medio de convoluciones  $1 \times 1$ .

Inception Module



(a) Inception module, naïve version



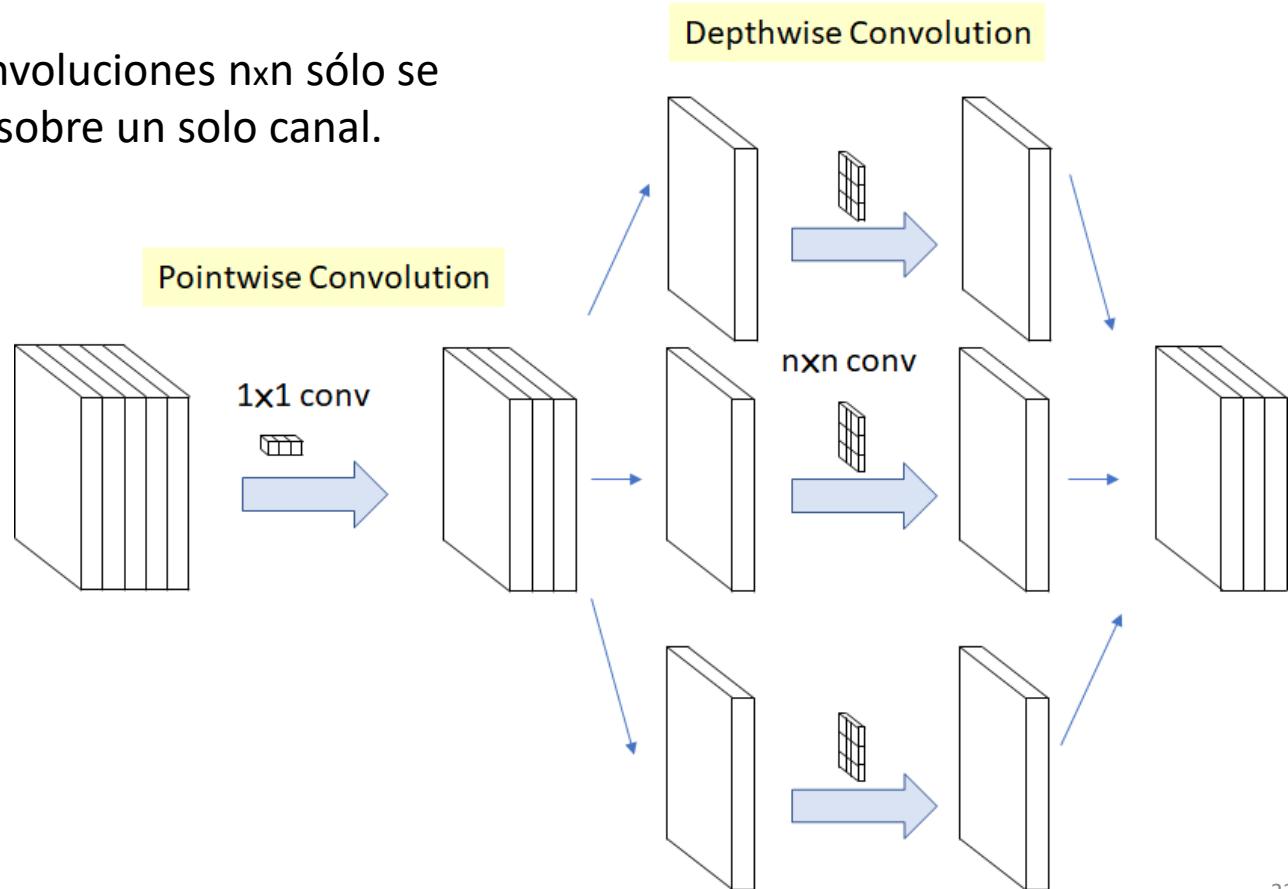
(b) Inception module with dimension reductions

# CNNs: Xception

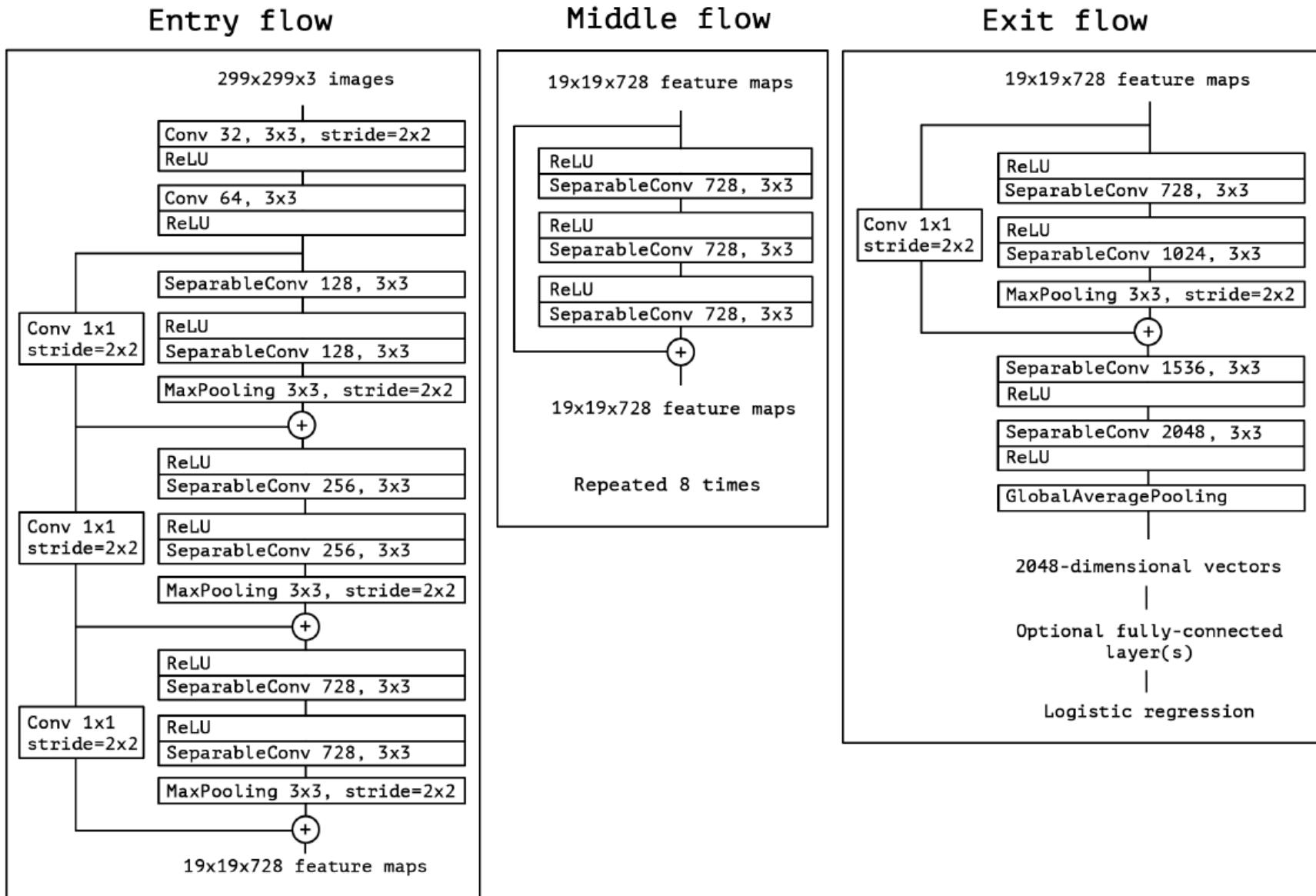
En el bloque básico de las redes Xception, se aplica la convolución  $1 \times 1$  a la entrada y sobre cada canal resultante se aplica una convolución o un muestreo (*pooling*).

En los modelos *inception*, las convoluciones  $n \times n$  se aplicaban tanto espacialmente (ancho y alto) como a los canales.

En este modelo, las convoluciones  $n \times n$  sólo se aplican espacialmente sobre un solo canal.

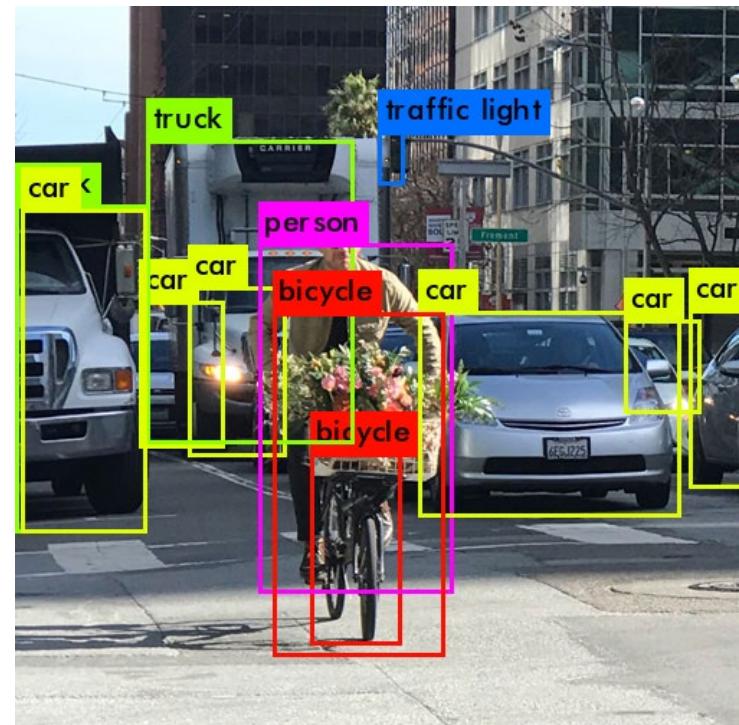


# CNNs: Arquitectura general de Xception



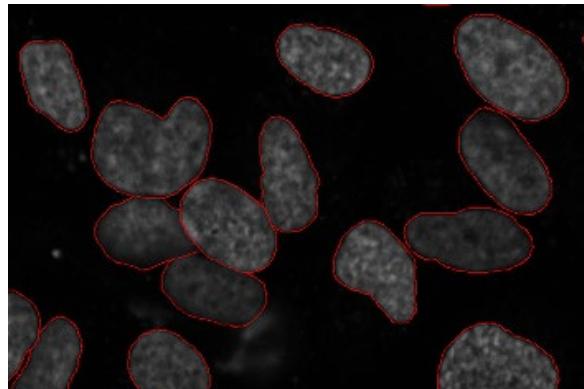
# Detección

- Los modelos anteriores eran útiles para tareas de **clasificación** (dígitos, por ejemplo).
- La **detección** consiste en identificar la presencia, la ubicación y el tipo de uno o más objetos en una imagen.
- Problema: los objetos de interés pueden estar en diferentes ubicaciones y presentar diferentes proporciones.
- Algoritmos:
  - R-CNN: Region-Based CNN:
    - Faster R-CNN
    - Mask R-CNN
  - YOLO: You Only Look Once

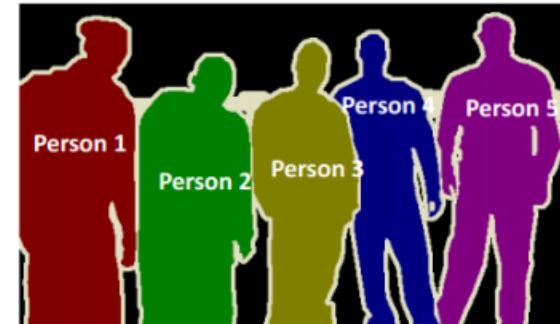


# Segmentación de imágenes

- La **segmentación de imágenes** es el proceso de asignación de una etiqueta a cada píxel de la imagen de forma que los píxeles que comparten la misma etiqueta también tendrán ciertas características visuales similares.
- Los dos tipos de segmentación existentes son la segmentación semántica y la segmentación de instancia:
  - La semántica agrupa todos los elementos del mismo tipo en una única región.
  - La de instancia genera una región para cada ocurrencia del tipo detectado.



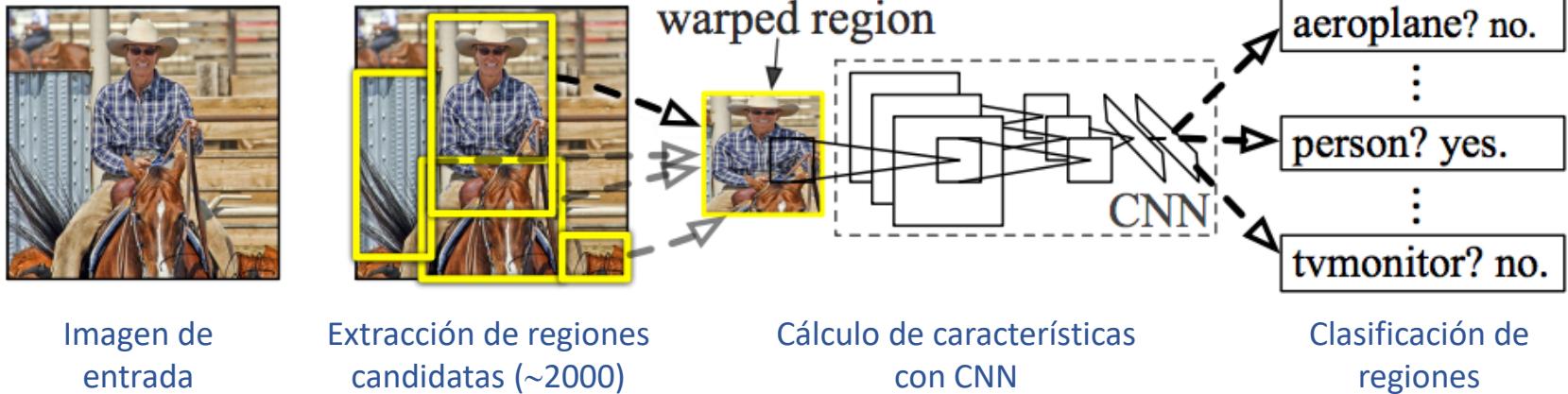
Semantic Segmentation



Instance Segmentation

# Detección con R-CNN (*Regions with CNN*)

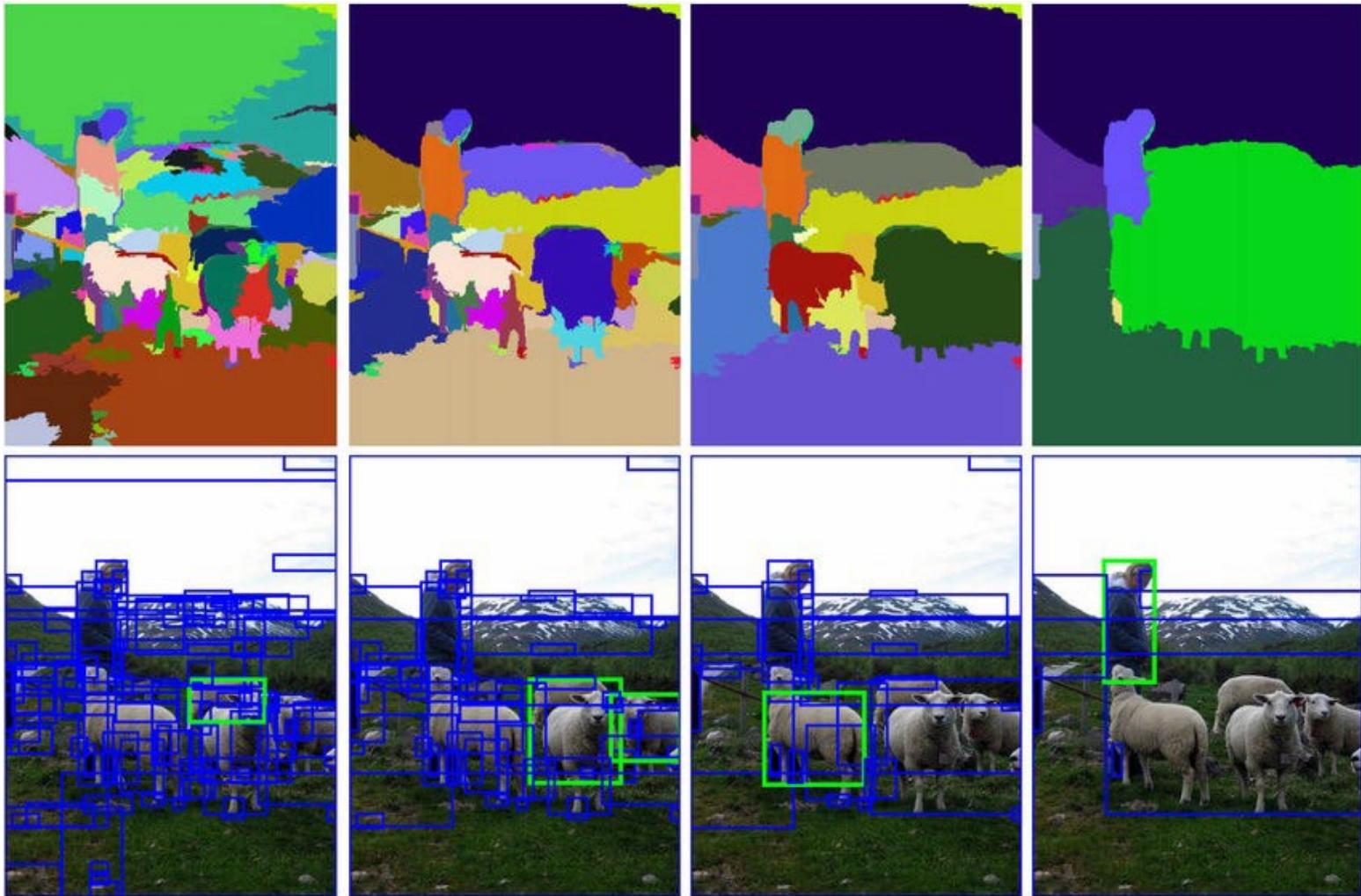
- Técnica compuesta por tres módulos:
  1. Propuesta de regiones: genera cuadros delimitadores (*bounding boxes*) candidatos.
  2. Extracción de características de cada región propuesta: usa CNN
  3. Clasificación de características: clasifica las características como una de las clases a detectar.



- La búsqueda de regiones candidatas se hace por medio de un algoritmo de visión llamado búsqueda selectiva (*selective search*). Su mayor inconveniente es que es fijo y no hay aprendizaje en esa etapa.
- Esta versión inicial de R-CNN es demasiado lenta para usarla en tiempo real porque implica aplicar una CNN sobre cada región candidata.

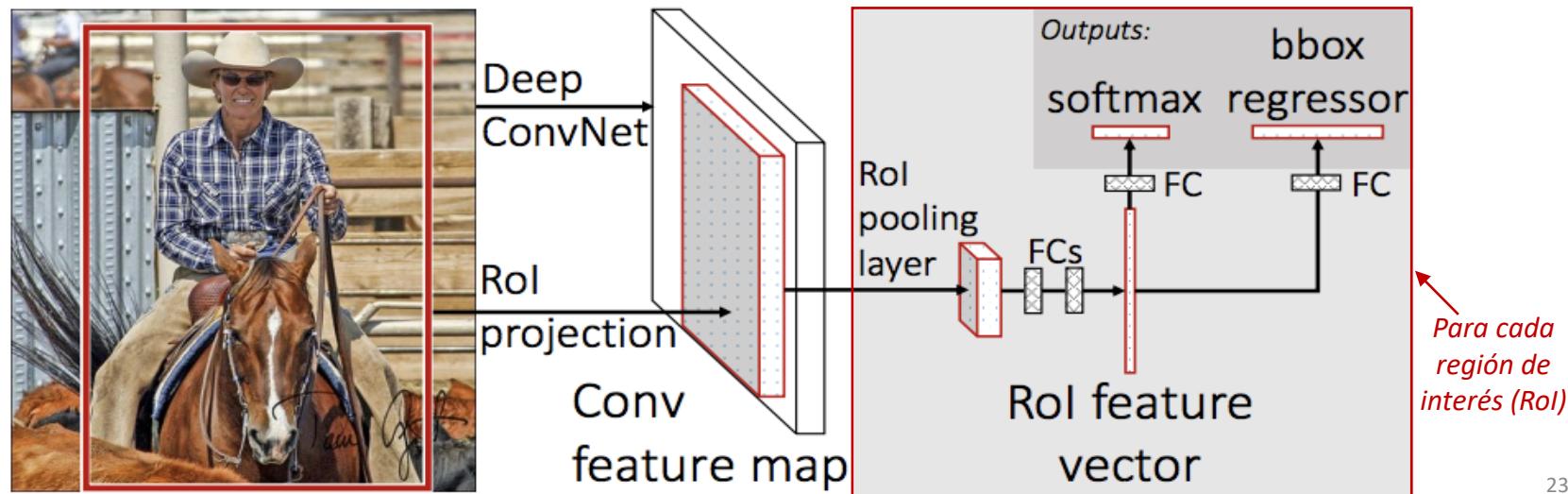
# Segmentación de imágenes

El algoritmo de búsqueda selectiva reduce las regiones combinando de forma recursiva las que son similares.



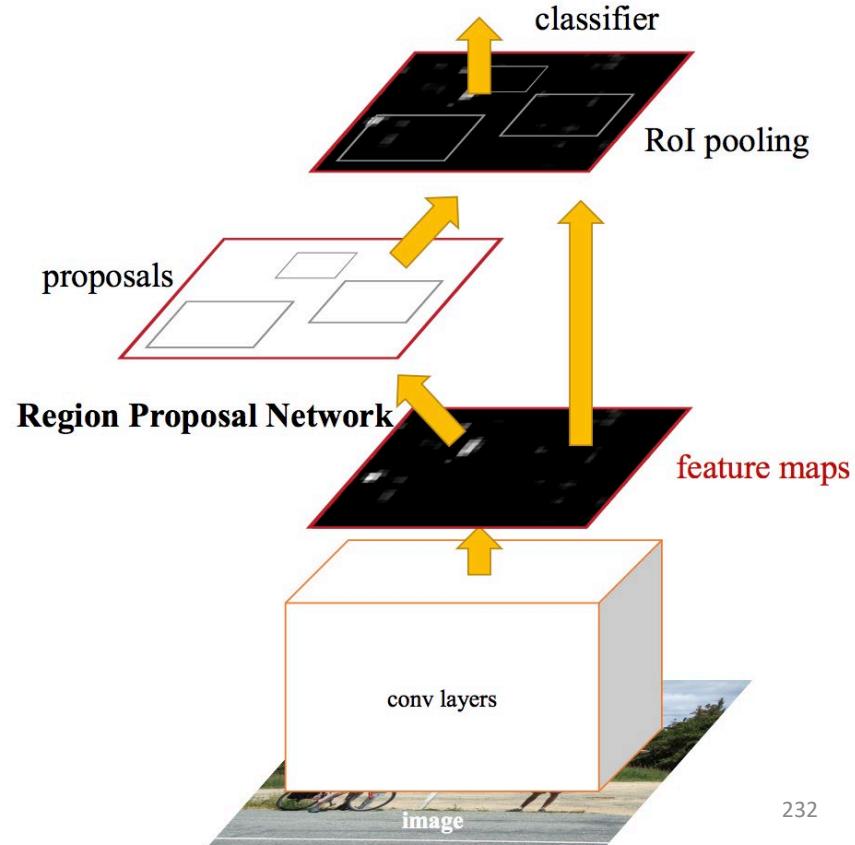
# Detección con Fast R-CNN

- Fast R-CNN evita procesar por separado cada una de las regiones candidatas.
- Se aplica el algoritmo de búsqueda selectiva de R-CNN para identificar las regiones de interés.
- A cada una de las regiones anteriores se le aplica la misma CNN para obtener sus mapas de características. Sin embargo, para cada región se aplica un clasificador diferente (últimas capas de la CNN).
- La reducción de tiempos se debe a que no hay que repetir los cálculos convolucionales para cada región.
- El sistema sigue siendo lento, ya que la selección de regiones de interés se realiza usando el mismo algoritmo de búsqueda selectiva que aplican las R-CNN.



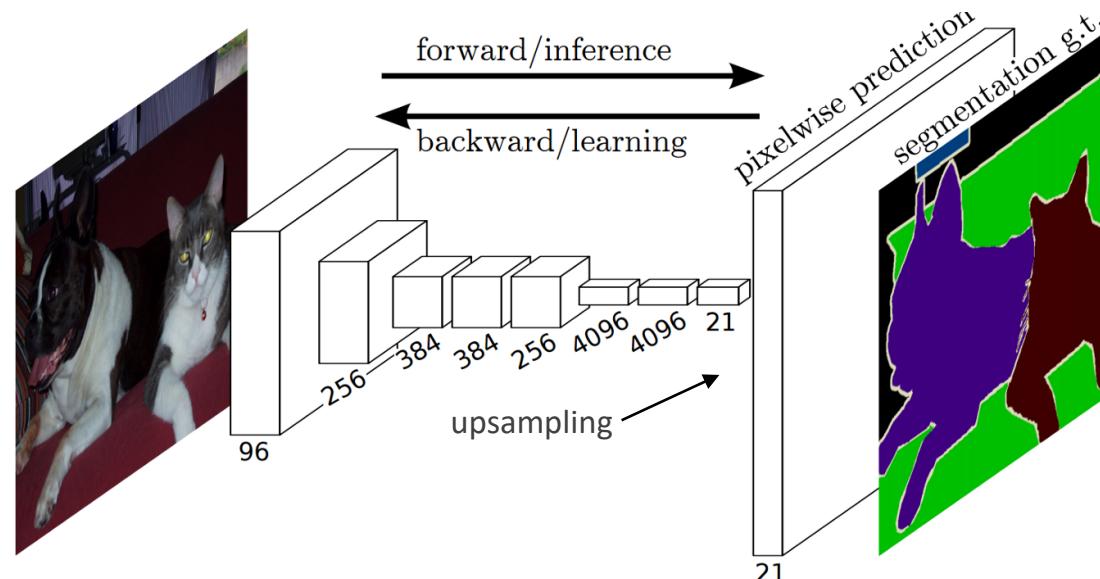
# Detección con Faster R-CNN

- Técnica derivada de R-CNN compuesta por tres módulos:
  1. Capas convolucionales que generan un mapa de características (*feature map*) de la imagen original. Este mapa va a ser sobre el que trabajen las dos redes posteriores: la red de propuesta de regiones informa a la segunda red de dónde debe prestar atención.
  2. Propuesta de regiones: se usa una red independiente para generar los cuadros delimitadores (*bounding boxes*) candidatos. Esta red trabaja sobre el mapa de características producidas por las capas convolucionales aplicadas a la imagen original.
  3. Red Fast R-CNN que extrae las características de las regiones propuestas y genera los cuadros delimitadores y las etiquetas de las clases.



# Red completamente convolucional (FCN)

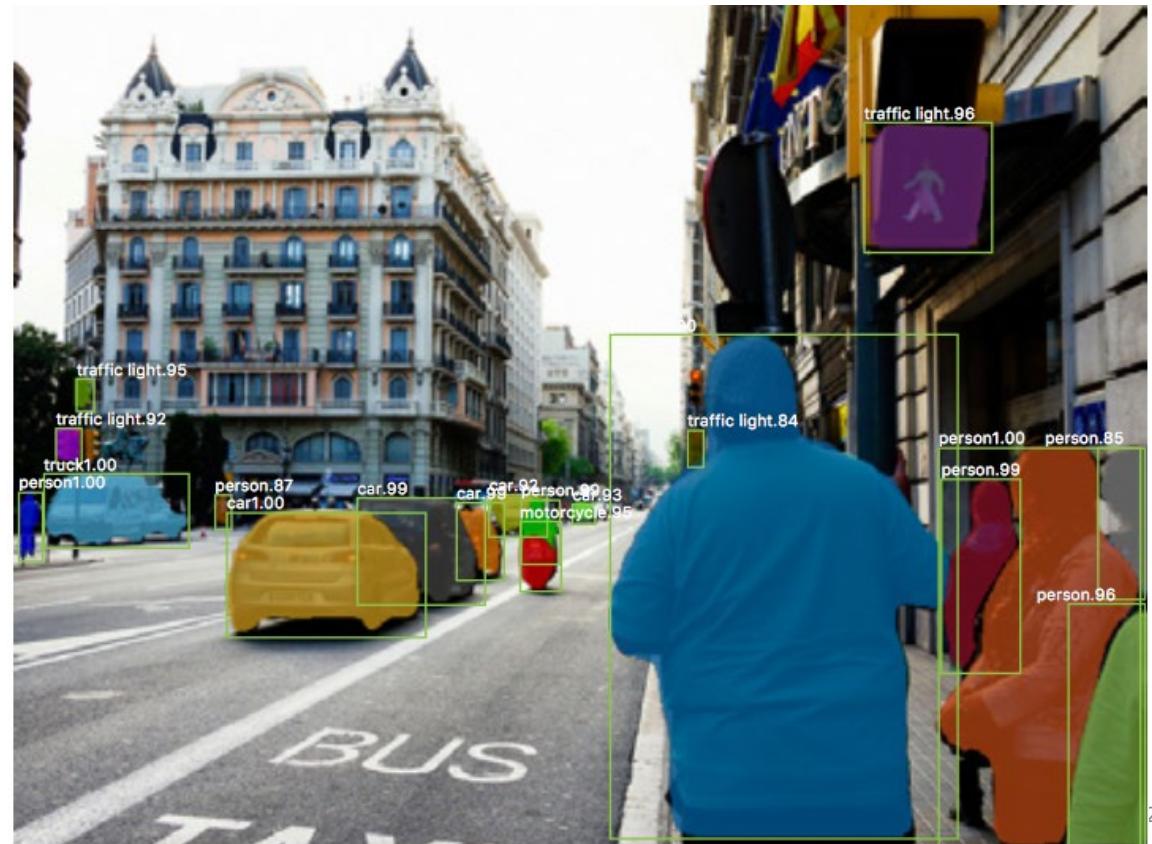
- Las redes completamente convolucionales (*Fully Convolutional Networks*) o FCN son un método muy típico de segmentación semántica.
  - Usa varios bloques de capas convolucionales y de muestreo (*pooling*) para reducir el tamaño de la imagen en un factor de 1/32.
  - Tras esto hace una predicción de clases con este nivel de granularidad.
  - Finalmente, usa capas de desconvolución y de desmuestreo para redimensionar la imagen al tamaño original.
  - No dispone de capas densas, reduciendo el número de parámetros a entrenar.



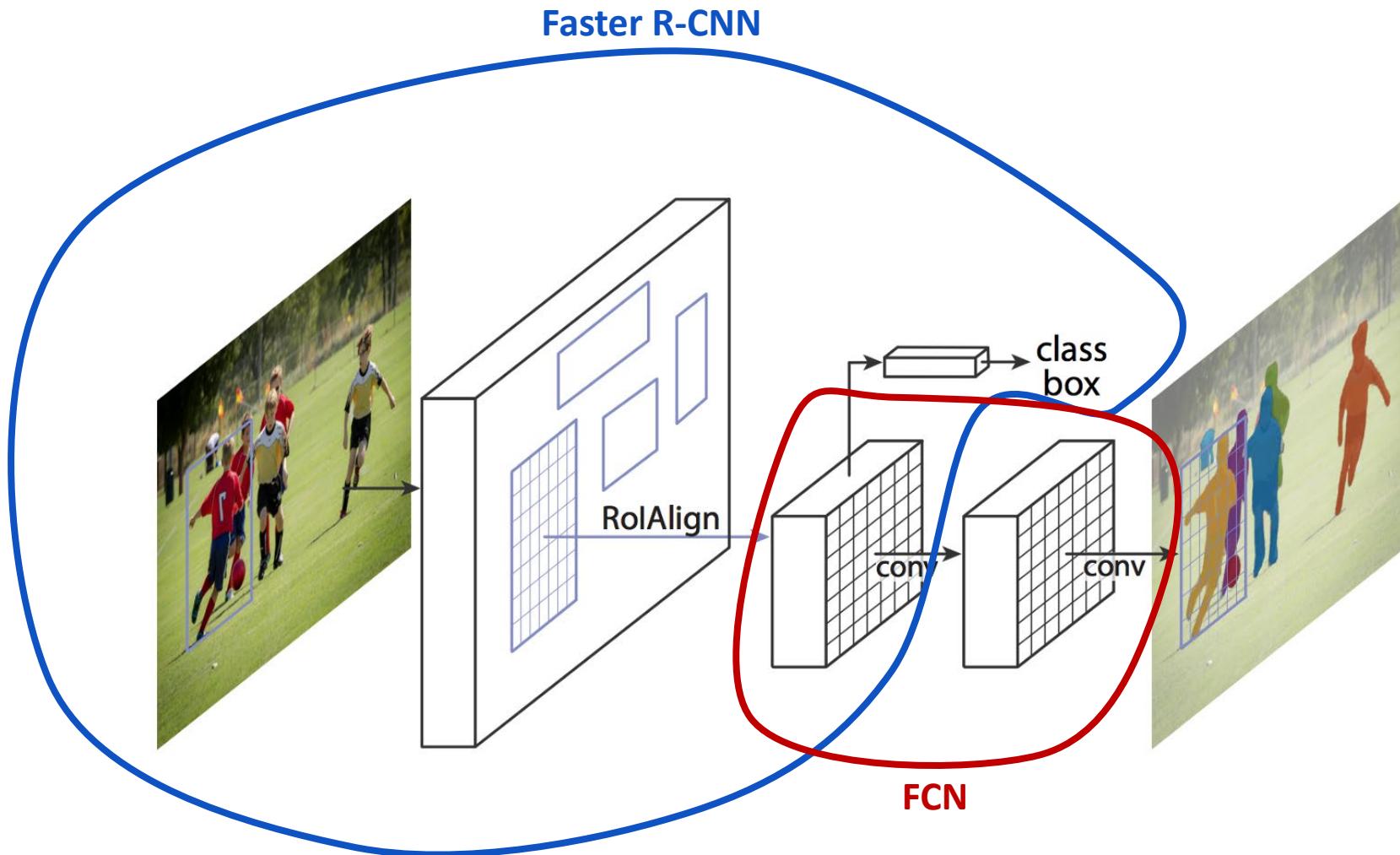
# Detección con Mask R-CNN

Mask R-CNN es conceptualmente simple:

- Faster R-CNN con dos salidas para cada objeto candidato, una etiqueta de clase y un desplazamiento de cuadro delimitador
- A esto agregamos una tercera rama que genera la máscara de objeto, que es una máscara binaria que indica los píxeles donde el objeto está en el cuadro delimitador.
- Pero la salida de máscara adicional es distinta de las salidas de clase y cuadro, lo que requiere la extracción de un diseño espacial mucho más fino de los objetos.
- Para hacer esta máscara R-CNN utiliza la red de convolución completa (FCN).



# Detección con Mask R-CNN

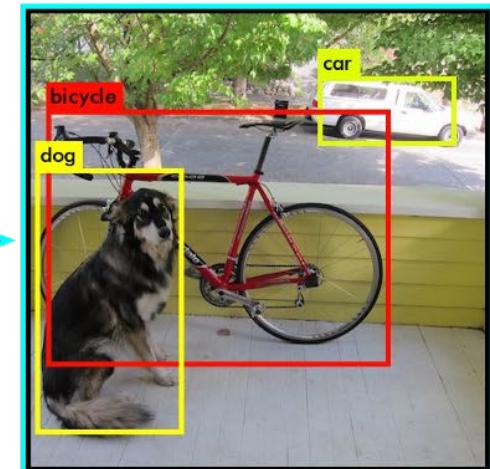
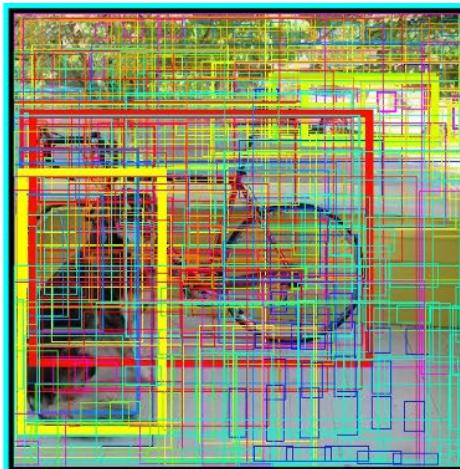
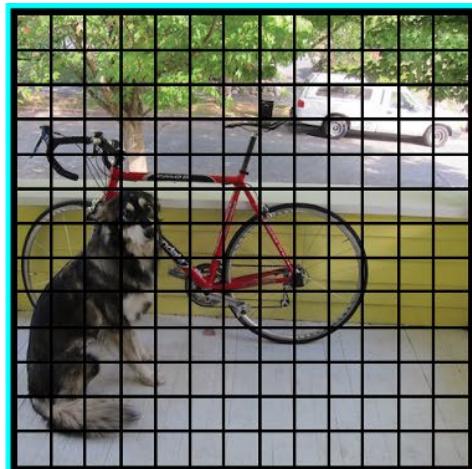


# Detección con YOLO

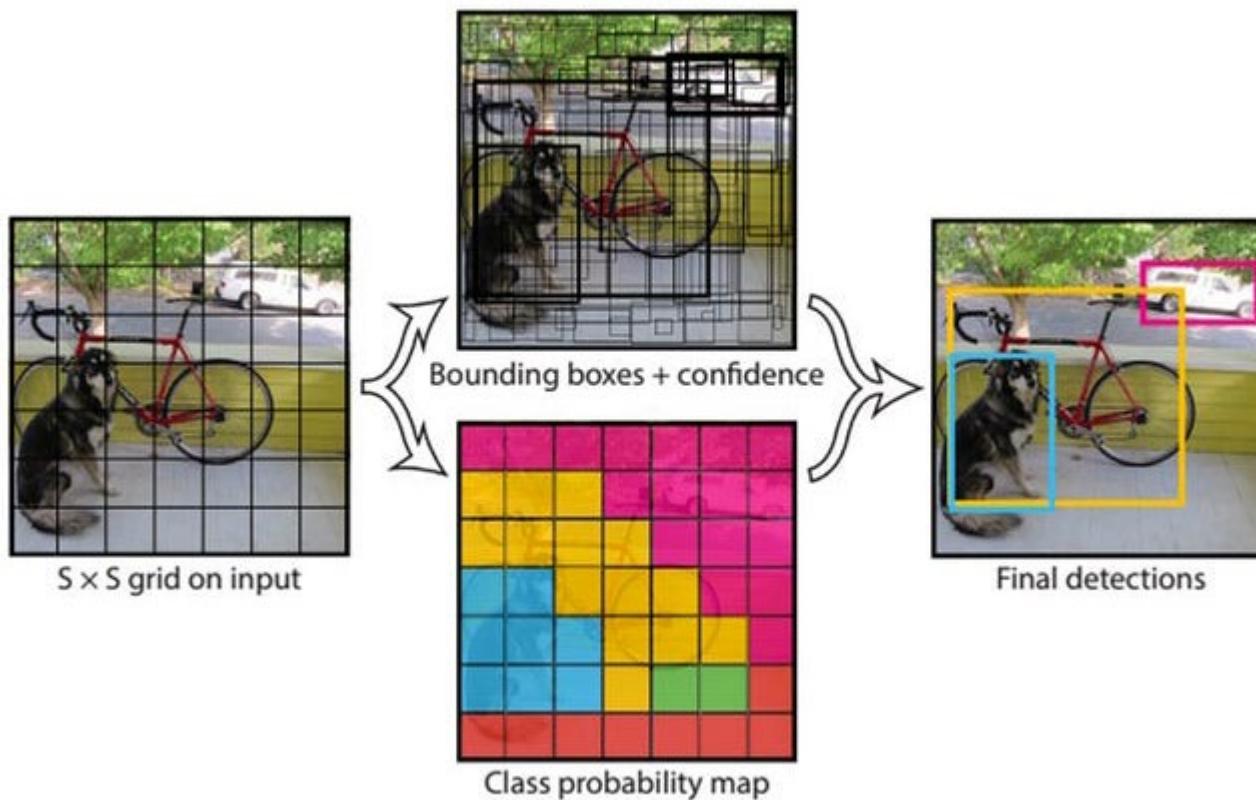
- YOLO (You Only Look Once) utiliza una única CNN para detectar objetos haciendo una sola pasada sobre la imagen.
- Ventaja: gran velocidad, que le permite detectar objetos en tiempo real en videos.
- Desventaja: menos precisión que la familia R-CNN.
- Funcionamiento:
  - Primero divide la imagen en una cuadrícula de  $S \times S$  celdas.
  - En cada una de las celdas predice  $N$  posibles cuadros delimitadores (*bounding boxes*) y calcula el nivel de certidumbre (probabilidad) de cada una de ellas.
  - De los  $S \times S \times N$  cuadros calculados se eliminan los de baja confianza (la mayoría)
  - A los cuadros restantes se les aplica un algoritmo que elimina los posibles objetos detectados por duplicado y dejar el más exacto.

Video 1

Video 2

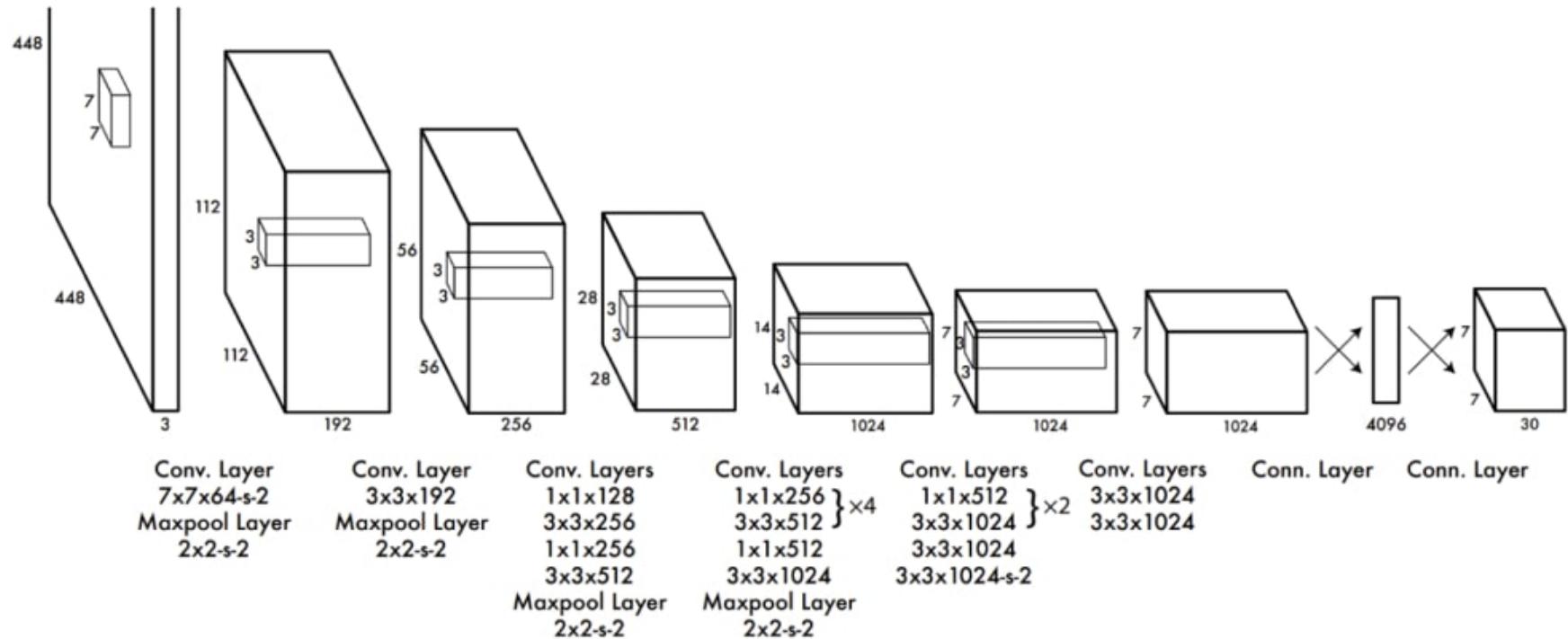


# Detección con YOLO



- Cada caja delimitadora es calculada por 5 neuronas: las coordenadas del centro ( $x,y$ ), el ancho, el alto y la fiabilidad de esa caja.
- Además, cada celda necesitará una neurona por cada clase a detectar (*softmax*).

# Detección con YOLO



# Redes Neuronales Recurrentes RNN

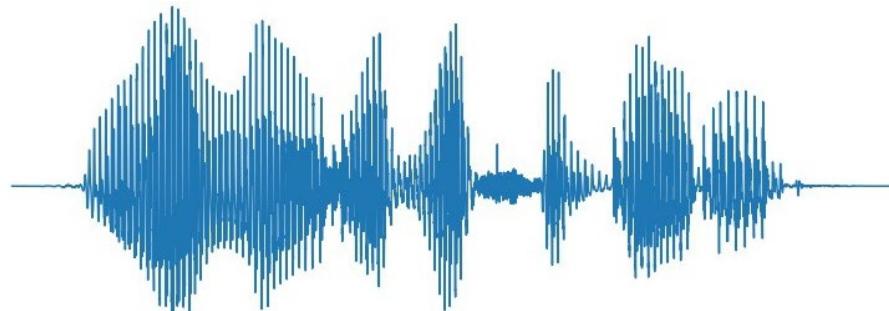
# *Datos secuenciales*

Frase: “Mañana va a hacer calor”

Señales médicas:



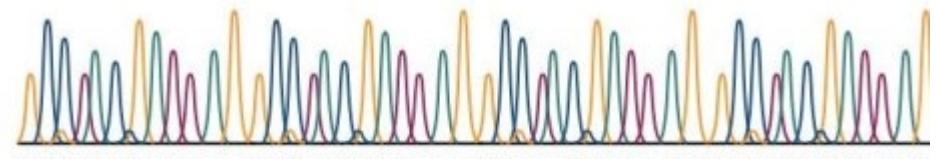
Voz:



Vídeo:



ADN:



# Aplicaciones de las RNN



Reconocimiento del habla (*speech recognition*)

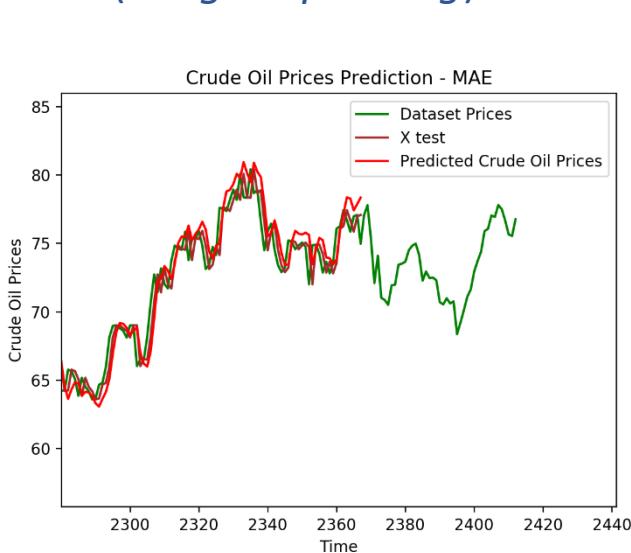


Señal de stop en una carretera

**Esto es divertido → これは楽しいです**

Interpretación de texto y traducción

- Generación de texto: generación de resúmenes, p. ej.
- Generación de vídeo.
- Interpretación de vídeo.
- Análisis de ADN.
- ...



Análisis de tendencias

# **Problemas específicos de las secuencias**

Hay problemas específicos de las secuencias que dificulta el trabajo de las redes neuronales tradicionales (MLPs):

- El tamaño de los datos es variable.
- El orden de los datos es variable.
- Hay información que se debe compartir a lo largo de toda la secuencia.

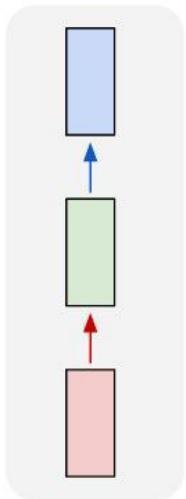
**Ejemplo generación de análisis de sentimiento:**

**Aunque el principio de la película fue malo, el resto fue muy bueno.**

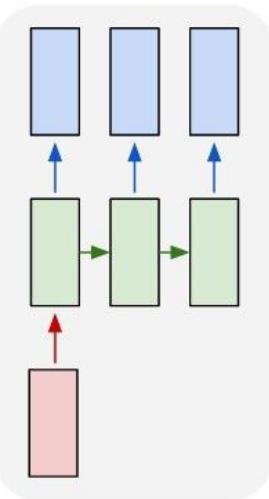
- El **tamaño** de las frases no es fijo.
- Si se intercambia el **orden** de las palabras “malo” y “bueno”, el significado global de la frase es el opuesto.
- La palabra “aunque” es fundamental para el análisis, aunque aparece muchas posiciones antes: dependencias a largo plazo (*long-term dependencies*).

# *Modelado de secuencias*

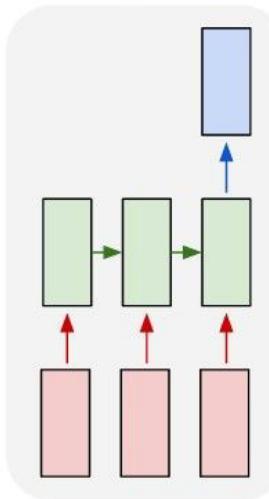
Una a una



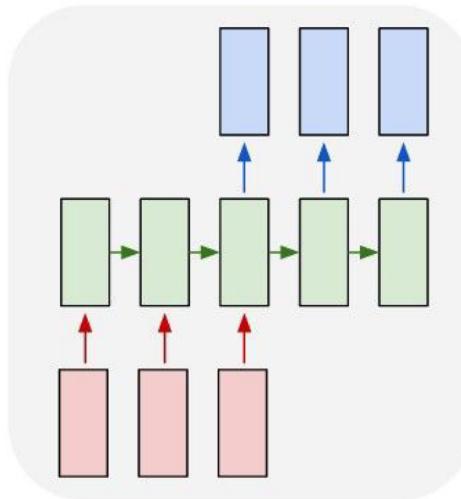
1 a N



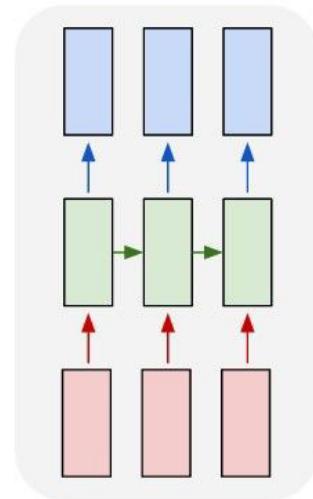
N a 1



N a N



N a N



MLPs

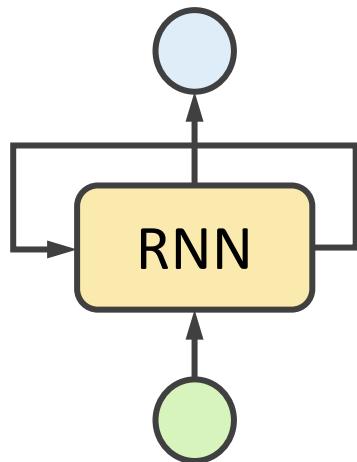
Descripción  
de imágenes

Clasificación de  
sentimiento

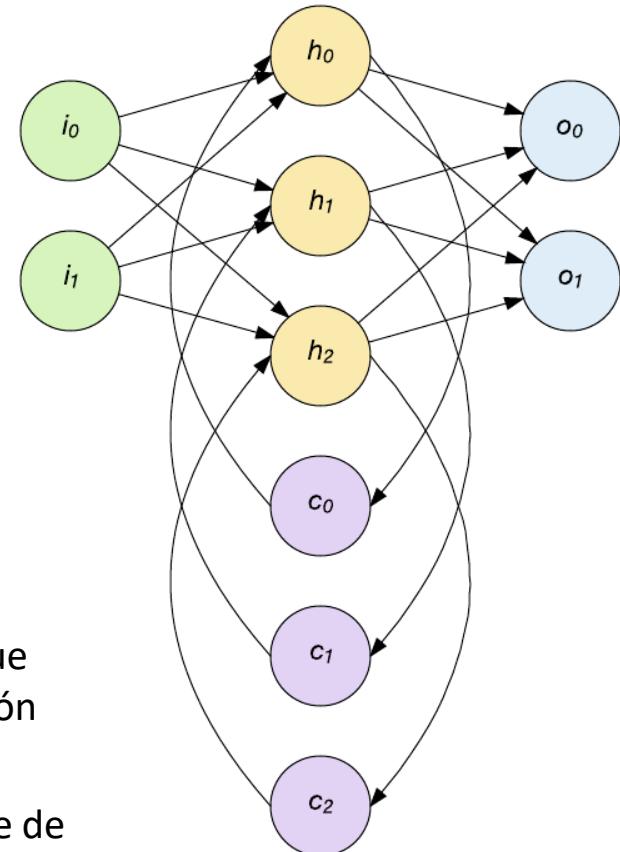
Traducción

Clasificación de  
vídeo por marcos

# Estructura de las RNN

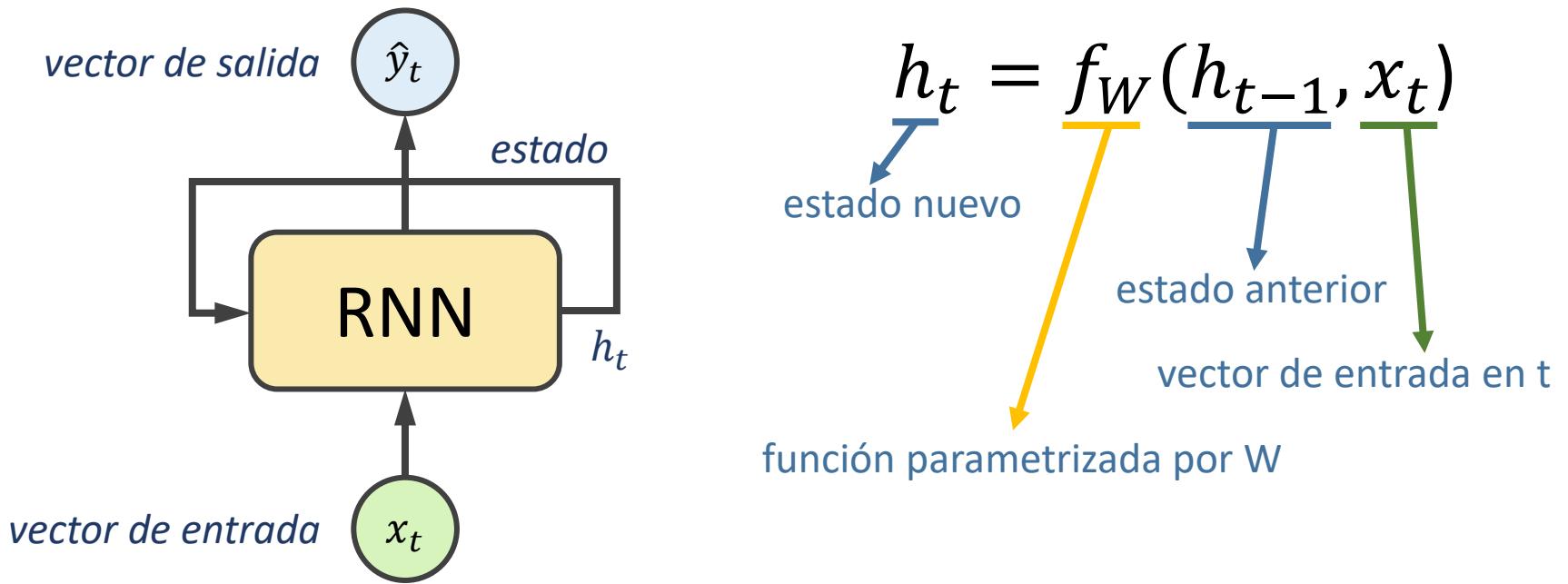


representación

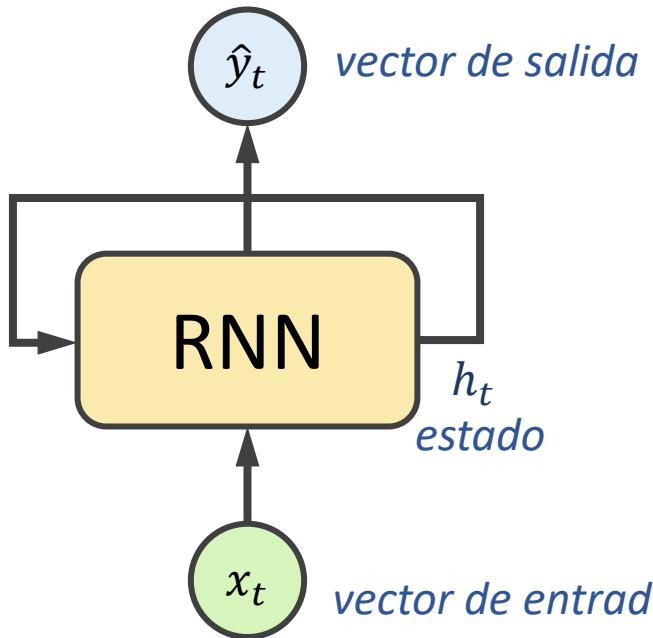


- Las Redes Neuronales Recurrentes son redes de neuronas que contienen bucles, permitiendo de esta forma, la memorización de la información.
- La entrada es un vector de valores distinto para cada instante de tiempo.
- Las RNN tienen la forma de una cadena de módulos que se repiten.
- En las RNN estándar, este modulo que se repite es una simple capa de neuronas con función de activación *tanh*.

# Relación de recurrencia en las RNN



# Relación de recurrencia en las RNN



$$h_t = f_W(h_{t-1}, x_t)$$

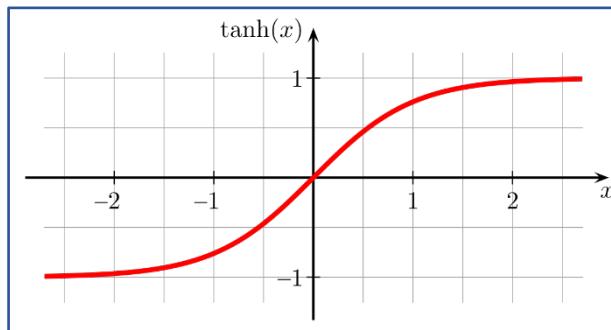
- Los pesos se comparten a lo largo del tiempo.
- La función de activación suele ser  $\tanh$  aunque ReLU suele dar buenos resultados.

Actualización del estado oculto:

$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$$

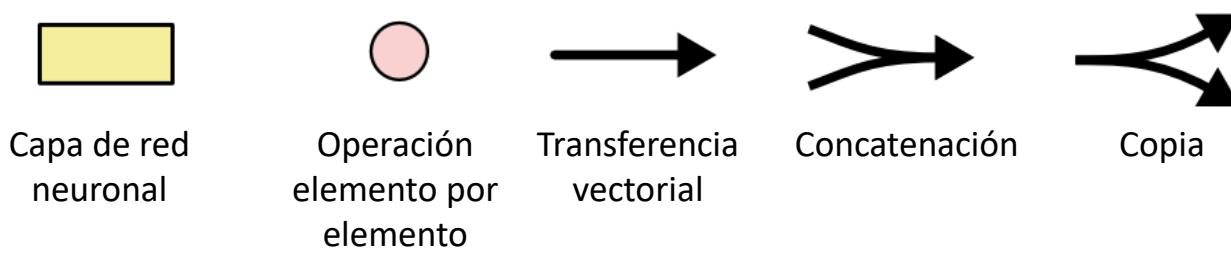
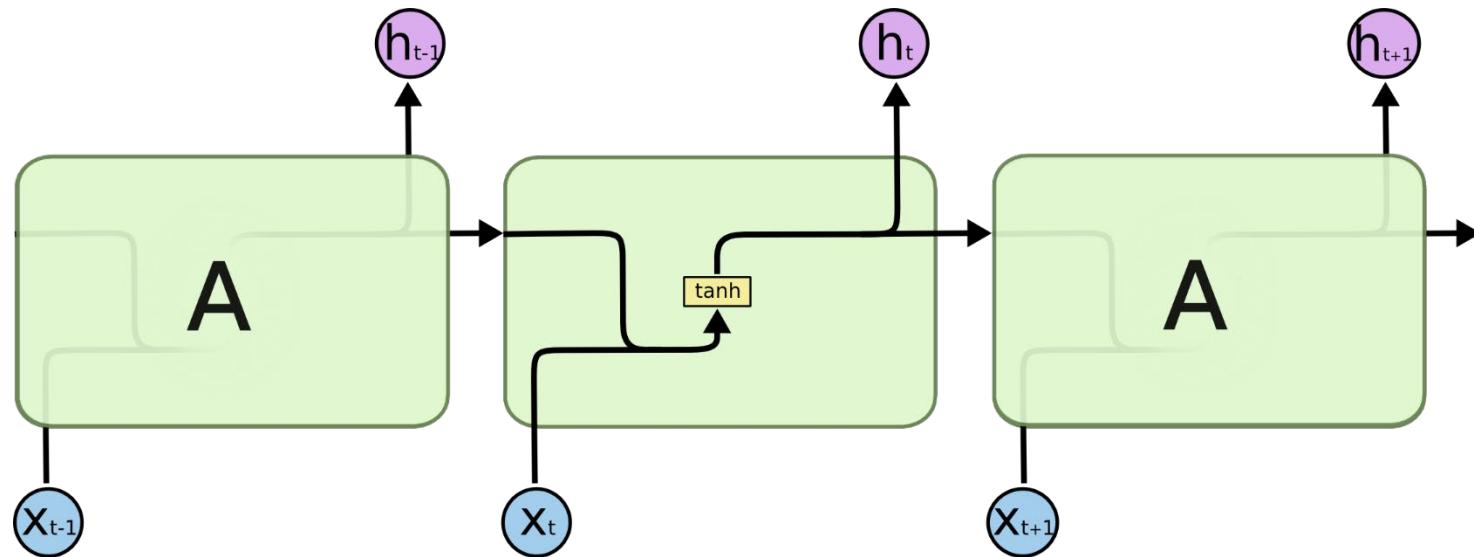
Vector de salida:

$$\hat{y}_t = W_{hy} \cdot h_t$$

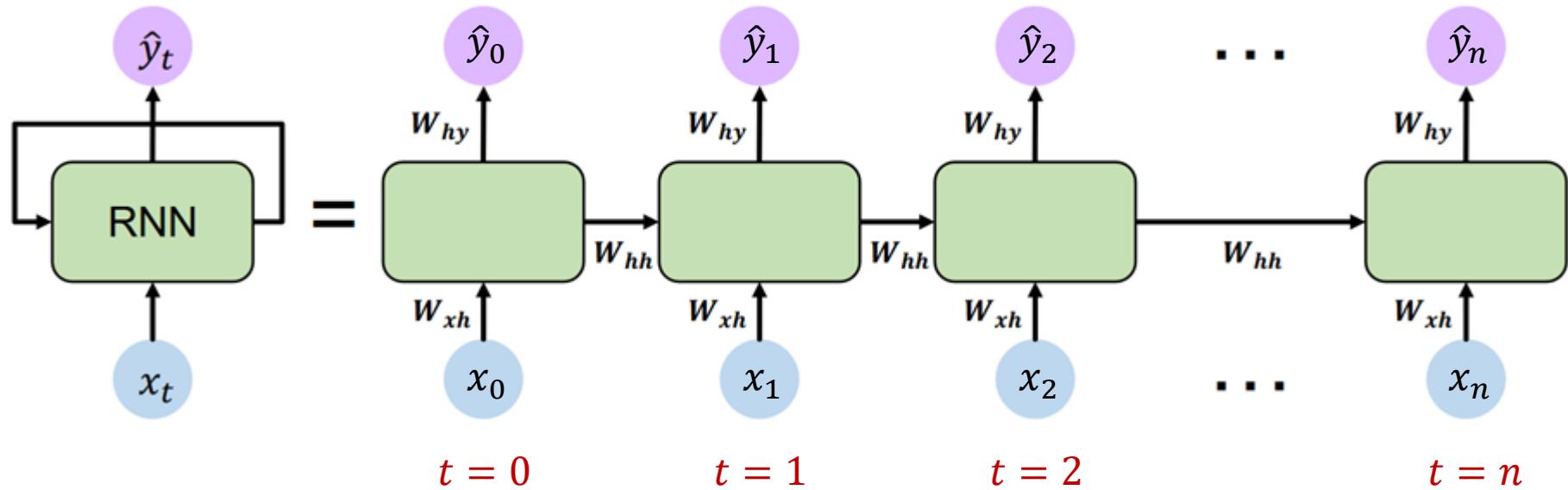


$\tanh$

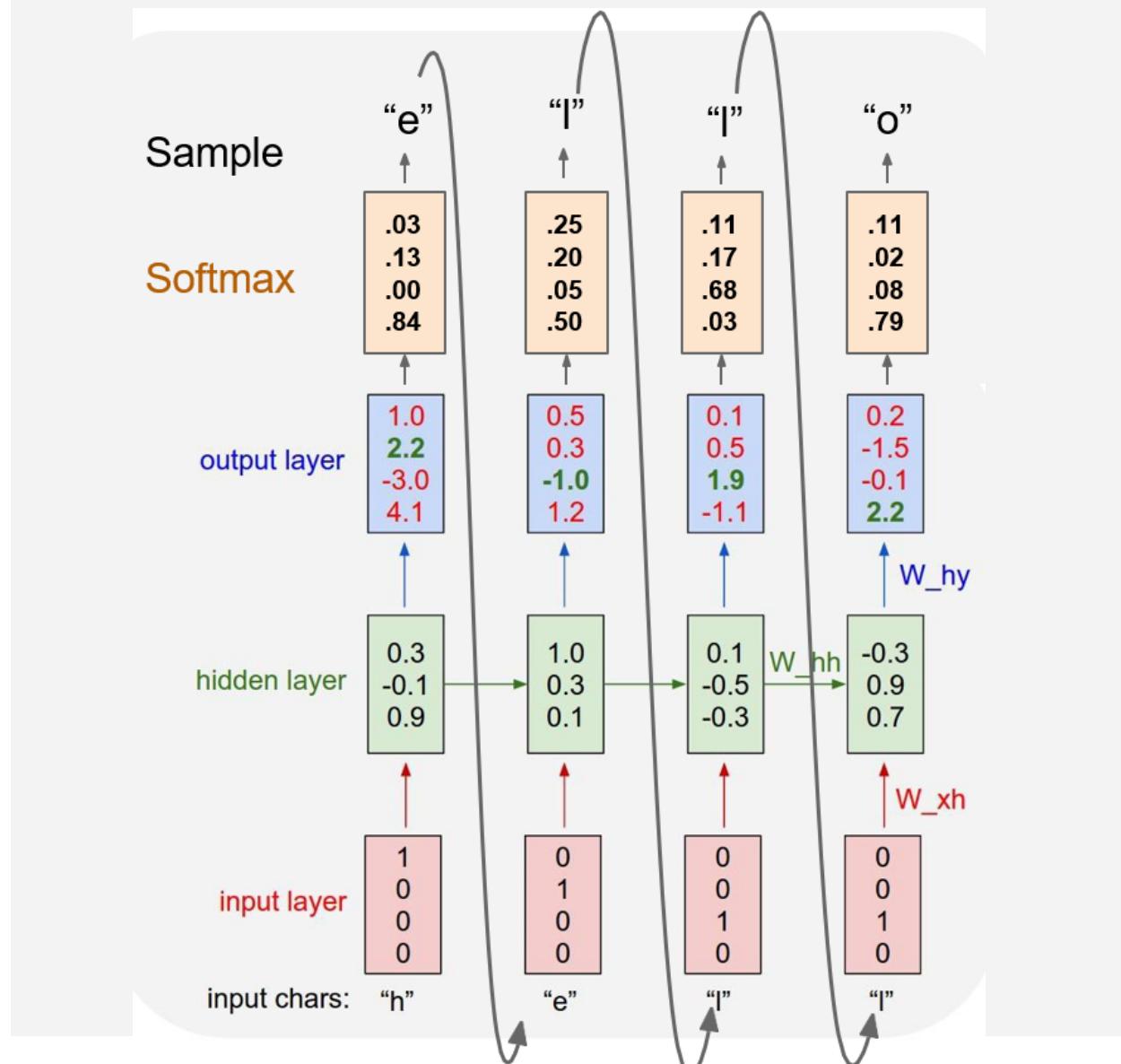
# Cálculo a través del tiempo



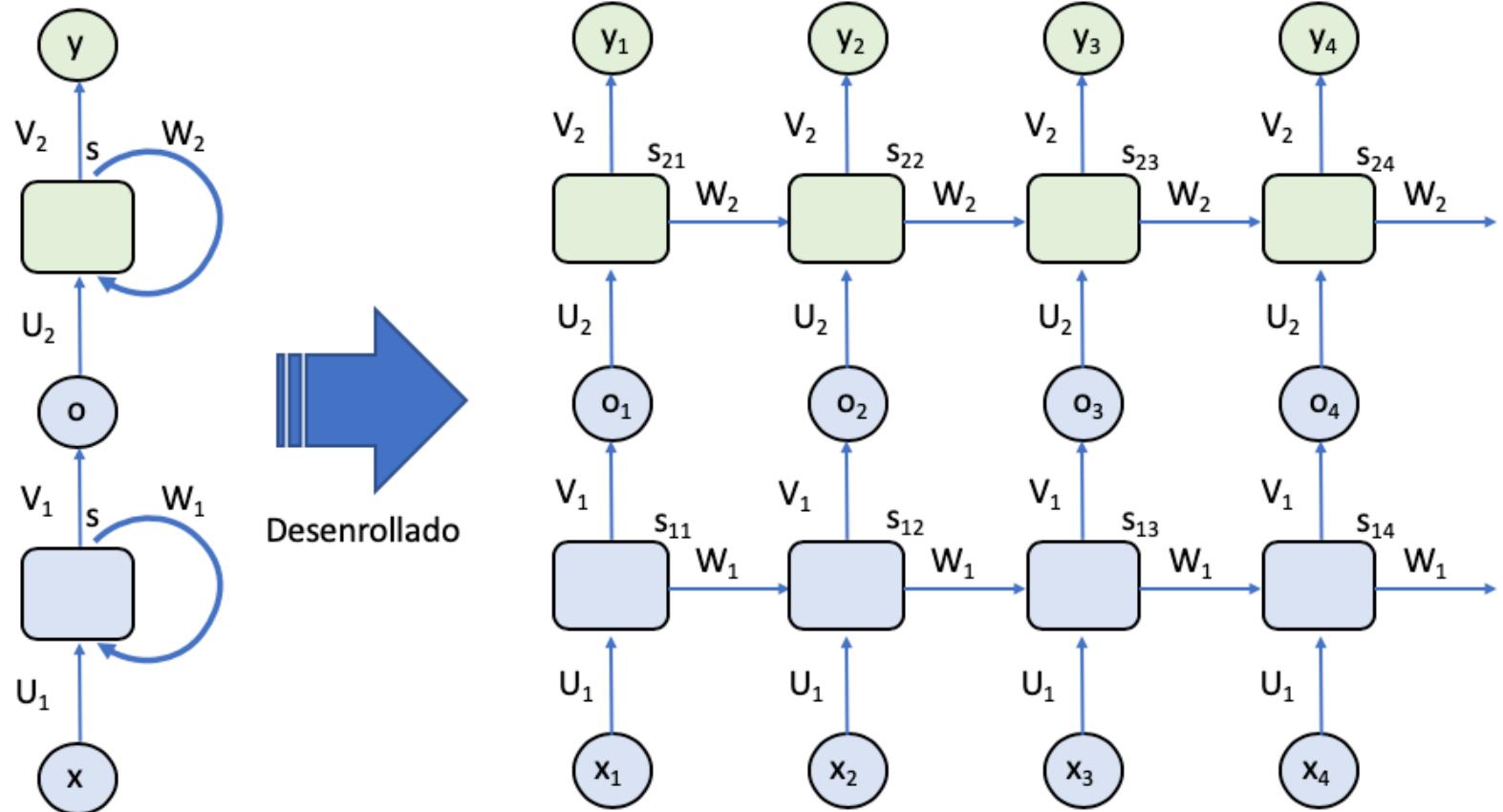
# Cálculo a través del tiempo



# Cálculo a través del tiempo



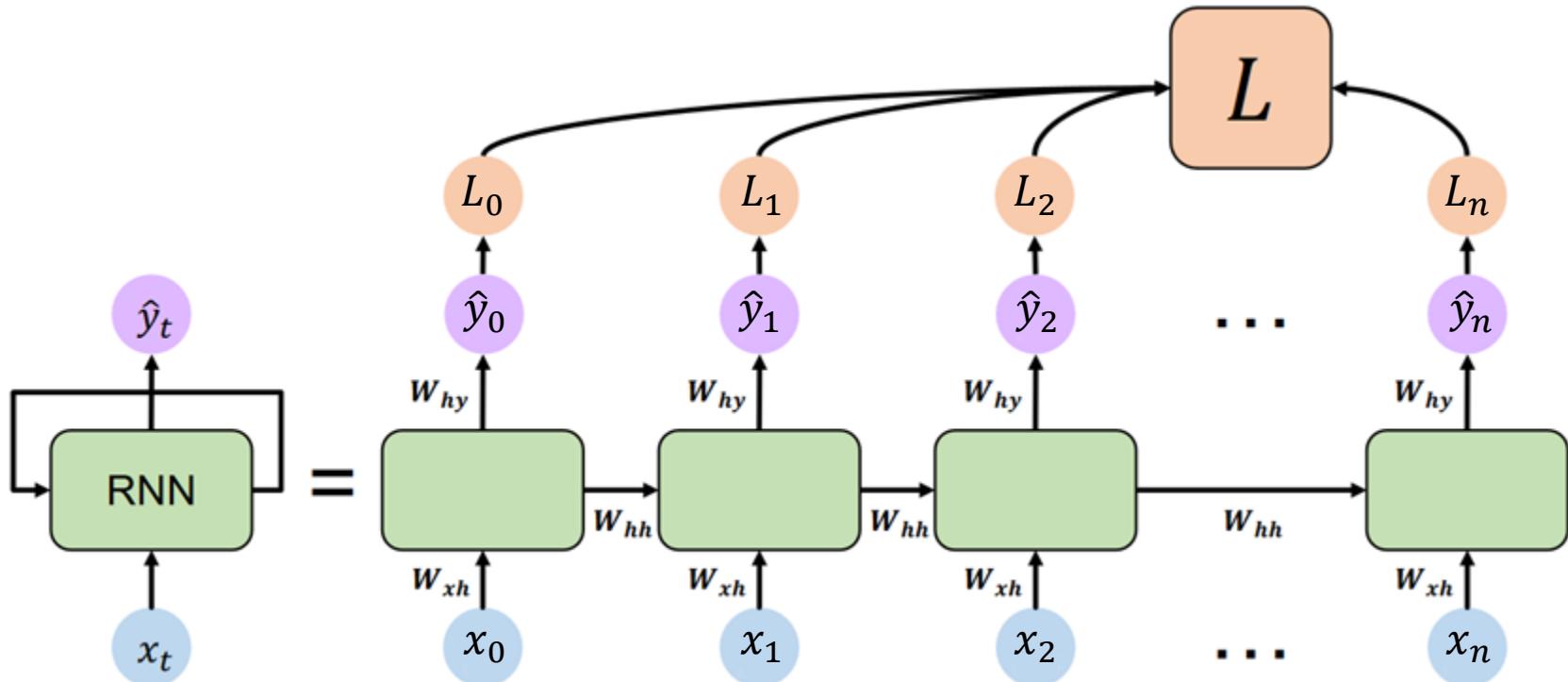
# RNNs apiladas (*Stack*)



# Pérdidas a través del tiempo

$$\text{Pérdidas} \rightarrow L = \sum_{t=0}^n L_t$$

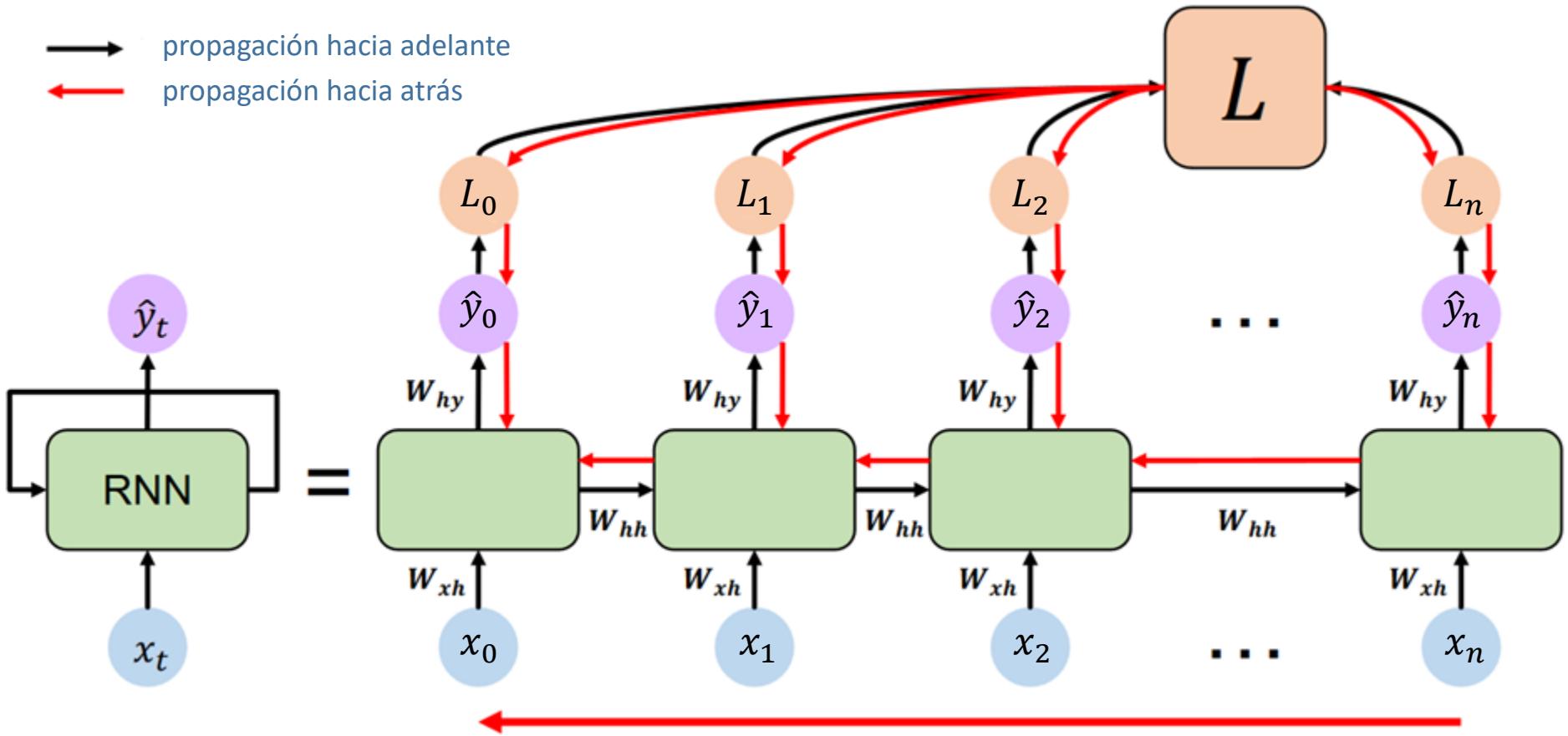
$$\text{Propagación hacia atrás normal} \rightarrow \frac{\partial L}{\partial W_{xh}} = \sum_{t=0}^n \frac{\partial L_t}{\partial W_{xh}}$$



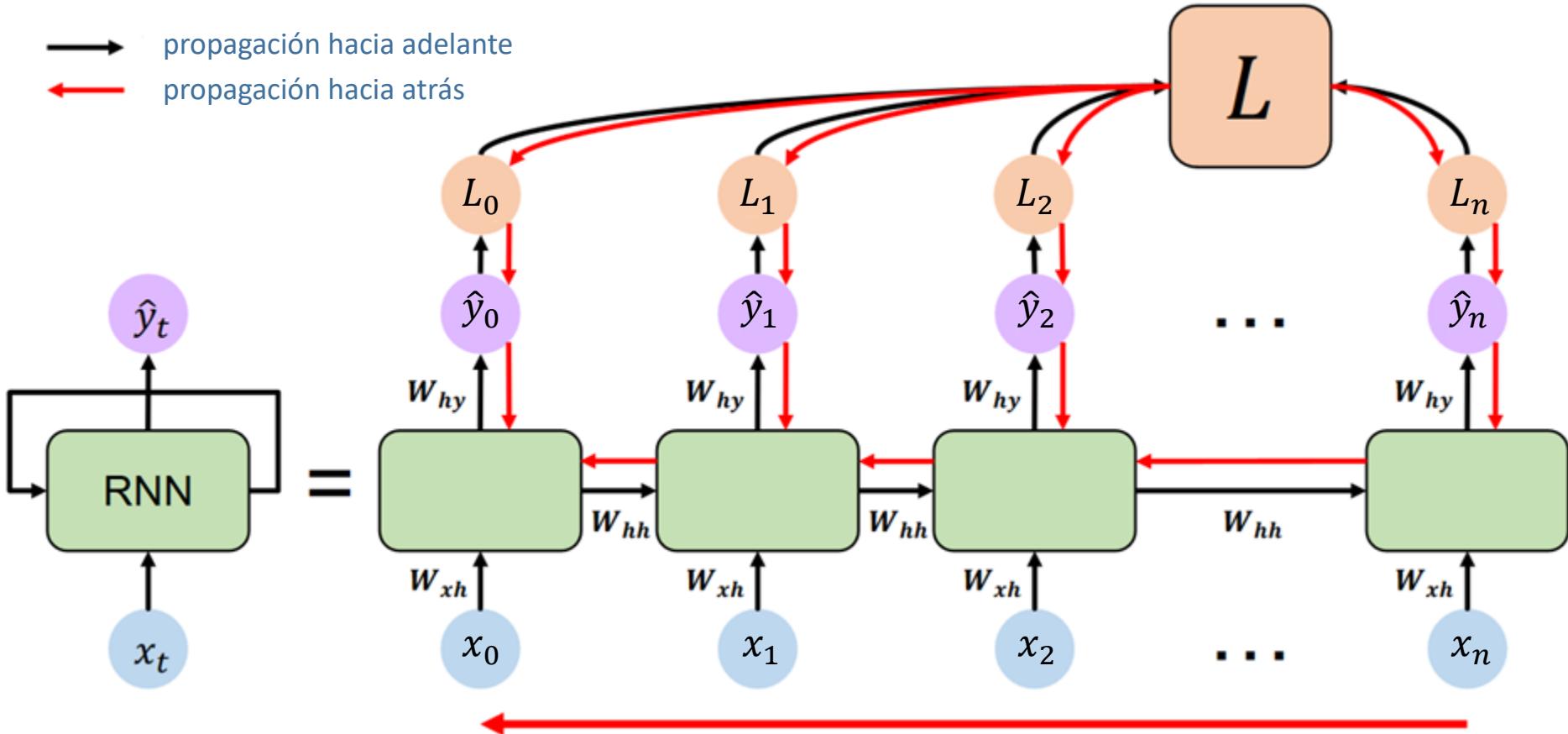
# Propagación hacia atrás a través del tiempo (BPTT)



propagación hacia adelante  
propagación hacia atrás



# Propagación hacia atrás a través del tiempo (BPTT)



- Los cálculos se “despliegan” a través de la secuencia para propagar las activations y calcular el gradiente.
- Habitualmente se limita el intervalo de tiempo para reducir la carga computacional (BPTT Truncada)

# Problemas de la BP<sub>TT</sub>

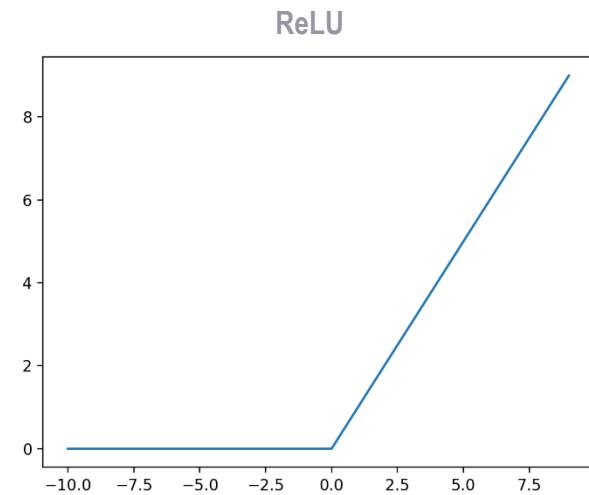
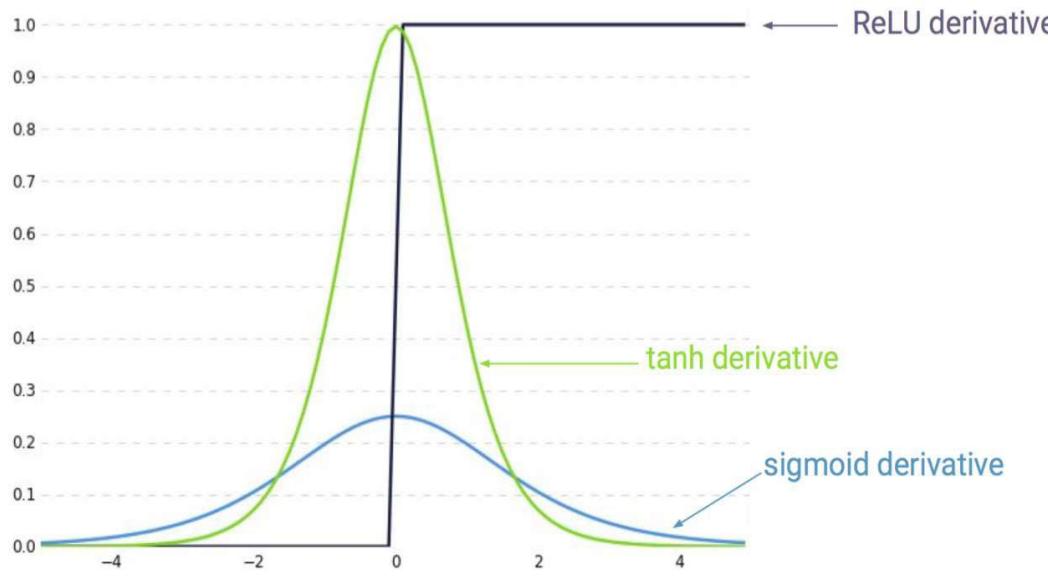
- ❑ Dependencia a largo plazo (*long-term dependencies*): Si una salida  $y_t$  depende de información muy atrás en el tiempo, el estado  $h_t$  debe mantener información a lo largo de muchos pasos.
- ❑ Debido a ello, los gradientes pueden multiplicarse muchas veces y podría ocurrir una de los dos problemas siguientes:
  - **Explosión de los gradientes**, cuando éstos son grandes (>1).
    - La consecuencia es que el algoritmo de aprendizaje diverge.
    - Se soluciona aplicando un recorte del gradiente (*gradient clipping*)
  - **Desvanecimiento de los gradientes**, cuando éstos son pequeños (<1).
    - La consecuencia es que el aprendizaje se ralentiza o incluso se detiene.
    - Se soluciona:
      - ✓ Introduciendo memoria: LSTM, GRU, etc.
      - ✓ Usando una función de activación diferente
      - ✓ Cambiando los valores de inicialización

# *Desvanecimiento del gradiente en BPTT*

Para evitar que la derivada de la función de activación haga que los gradientes se aproximen a cero, se utiliza ReLU.

Sin embargo en RNNs se usan poco porque no limitan los valores de salida.

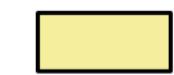
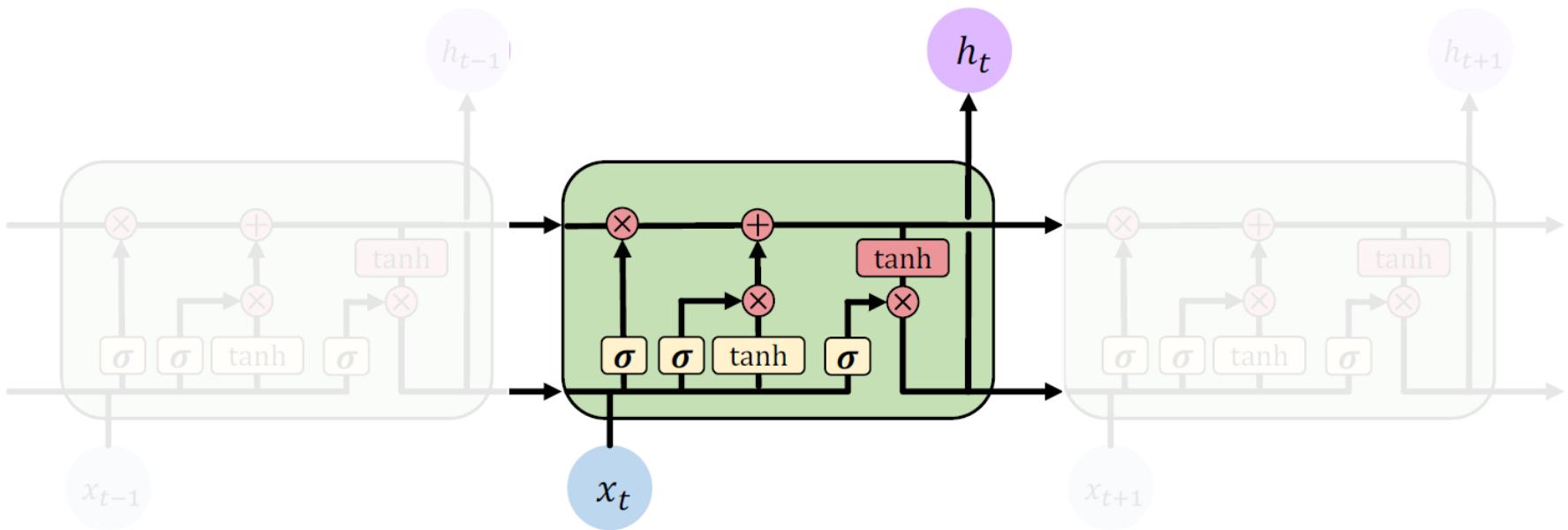
Tanh garantiza que las entradas están entre -1 y 1 a la vez que su segunda derivada tarda en llegar a cero.



# *Redes LSTM (Long Short Term Memory)*

- ❑ Las redes de memoria a corto/largo plazo (LSTM) son una variante de las RNN que pueden mantener dependencias de largo plazo.
- ❑ Están compuestas por celdas de memoria y puertas de control (*gates*) que controlan en qué medida la información modifica el estado y se transmite al siguiente paso.
- ❑ Disponen de dos vectores de estado:
  - **Estado oculto** (*hidden state*) que actúa como memoria a corto plazo.
    - Cambios (actualizaciones) grandes.
  - **Estado de la celda** (*cell state*) que actúa como memoria a largo plazo.
    - Cambios (actualizaciones) pequeños.
    - Este es el punto clave de las LSTM.
- ❑ Las puertas (*gates*) son las que controlan el flujo de información de un estado al siguiente.
- ❑ La información puede fluir entre las memorias de corto y largo plazo.

# Redes LSTM (Long Short Term Memory)



Capa de red  
neuronal



Operación  
elemento por  
elemento



Transferencia  
vectorial



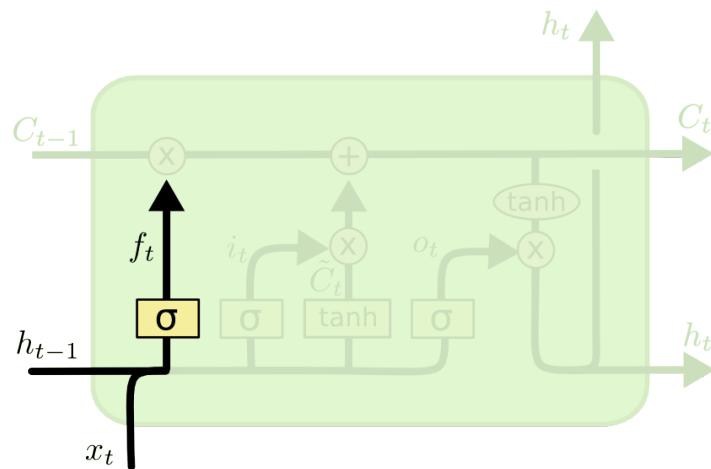
Concatenación



Copia

# Redes LSTM: Puerta de olvido

- La puerta de olvido (*forget gate layer*) es una capa sigmoide que decide qué información se descarta del estado de la celda.
- Genera un valor entre cero y uno para cada valor del estado de celda  $C_{t-1}$  a partir de  $h_{t-1}$  y  $x_t$ . Un resultado de 0 representa el olvido completo y de uno, el recuerdo completo.
- Si, por ejemplo, se trata de predecir la siguiente palabra de la frase “*El helado que comí ayer en el restaurante de tus padres estaba realmente ...*”, el estado de la celda incluiría el género del sujeto (“helado”) de forma que el adjetivo que generará la red concuerde con él.
- Cuando la red detecta un nuevo sujeto, olvida el género del anterior.

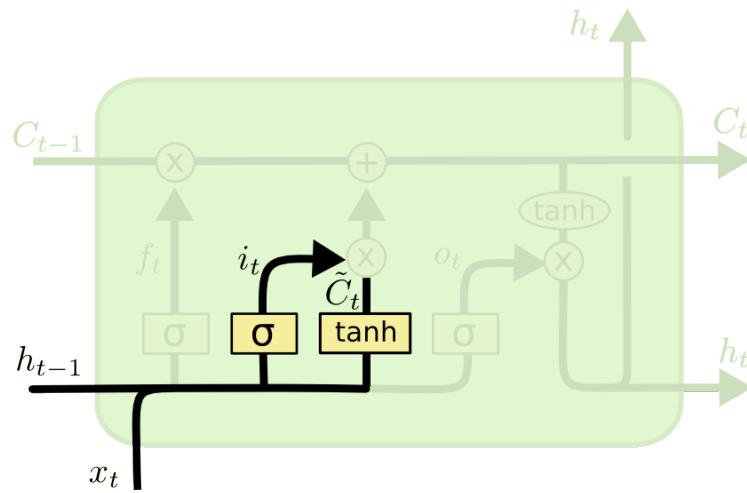


$$f_t = \sigma (W_f \cdot [h_{t-1}, x_t] + b_f)$$

Sesgo (bias): constante entrenable

# Redes LSTM: actualización del estado de la celda

- ❑ Tras gestionar el olvido en la estado de la celda, se decide qué información nueva se almacenará en él. Ese proceso se divide en dos etapas:
  1. La puerta de entrada (*input gate layer*) decide qué valores se actualizarán.
  2. Una capa *tanh* crea un vector de nuevos valores candidatos ( $\tilde{C}_t$ ) que podrían ser añadidos al estado.
  3. Se combinan los dos vectores de valores para crear la actualización del estado de la celda.
- ❑ En el ejemplo anterior, en esta etapa se añadiría al estado de la celda el género del nuevo sujeto para reemplazar el anterior que se debe olvidar.

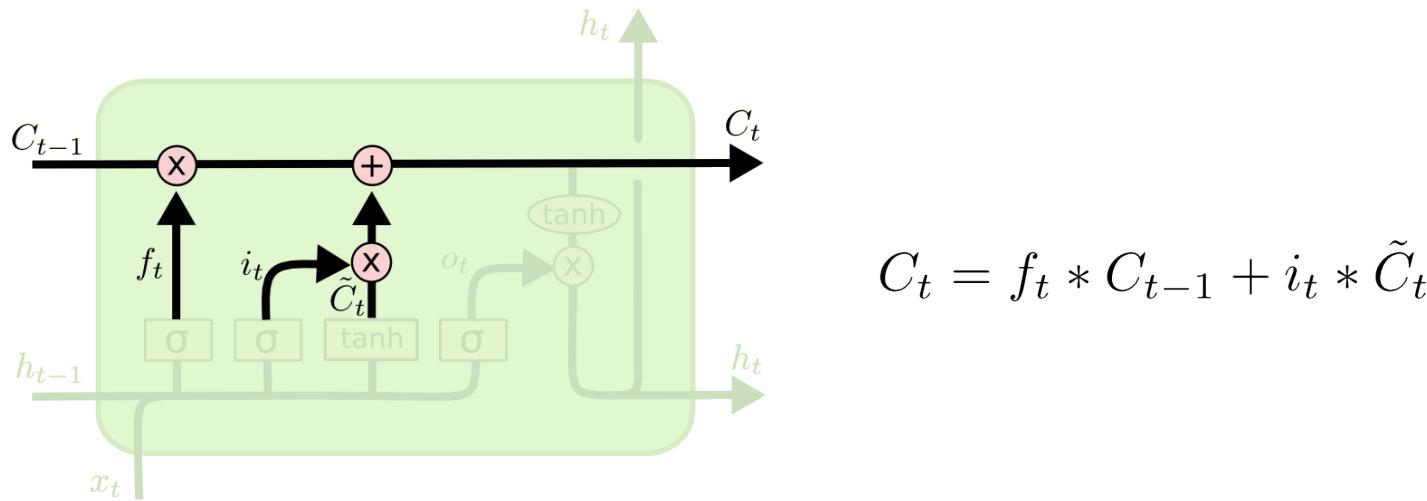


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

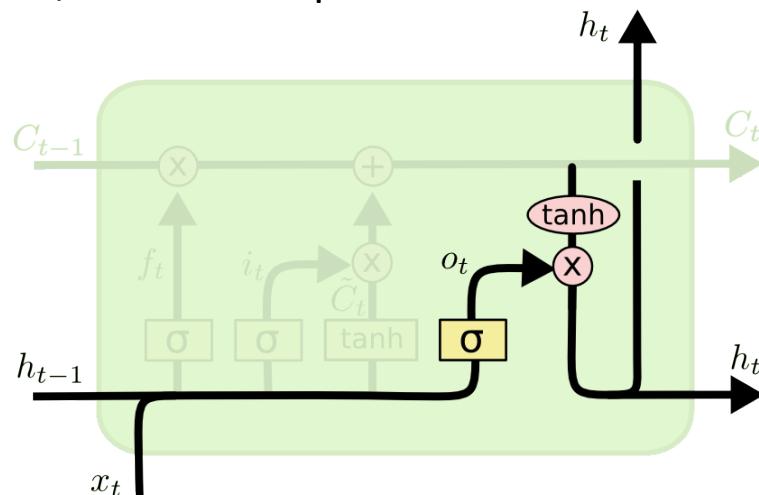
# Redes LSTM: actualización del estado de la celda

- El ultimo paso de esta segunda fase es la la actualización del estado de la celda anterior  $C_{t-1}$  para generar el nuevo estado,  $C_t$ . Para ello:
  1. Se multiplica el estado anterior por  $f_t$  para olvidar la información que corresponda.
  2. A lo anterior, se le suma  $i_t \cdot \tilde{C}_t$ . Estos son los nuevos valores candidatos, escalados por cuánto hemos decidido actualizar cada valor de estado.
- En nuestro ejemplo de predicción de la siguiente palabra, en este punto es donde se descartaría el género del sujeto anterior y se almacenaría el del nuevo sujeto.



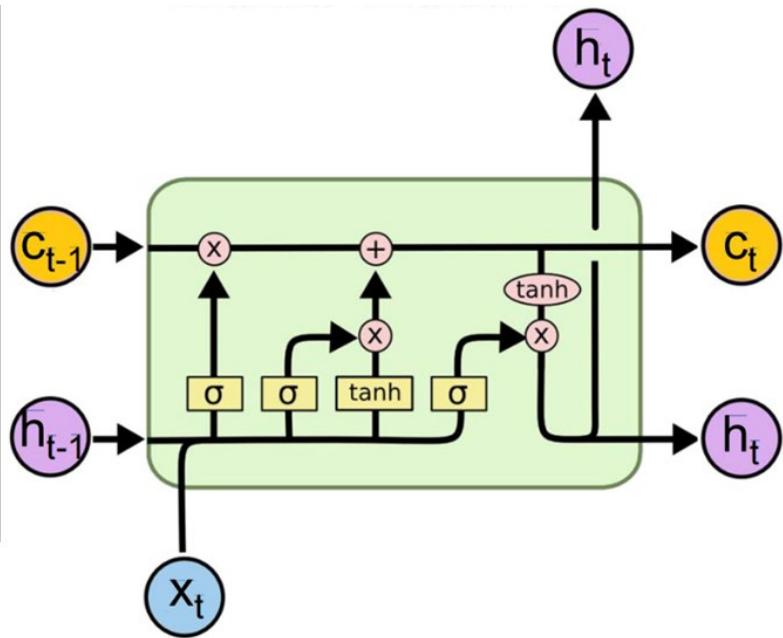
# Redes LSTM: cálculo de la salida

- La tercera y última fase consiste en generar la salida.
- Esta salida estará basada en una **versión filtrada** el estado de la celda. Para ello:
  1. Se aplica una capa sigmoide que decide qué partes del estado de la celda se sacará como salida.
  2. A continuación, pasamos el estado de la celda por una *tanh*, de modo que los valores estén entre -1 y 1 y se multiplica por la salida de la puerta sigmoide. De esta forma, sólo se genera como salida las partes que se hayan decidido.
- En nuestro ejemplo de predicción de la siguiente palabra, dado que se ha visto un nuevo sujeto, podría decidir sacar información relevante con respecto al verbo que viene a continuación. Por ejemplo, podría generar como salida si el sujeto es singular o plural, de forma que se supiera cómo se debería conjugar el verbo que vendría a continuación, en caso de que fuese así.



$$o_t = \sigma (W_o [ h_{t-1}, x_t ] + b_o)$$
$$h_t = o_t * \tanh (C_t)$$

# Redes LSTM: resumen



$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$$

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$

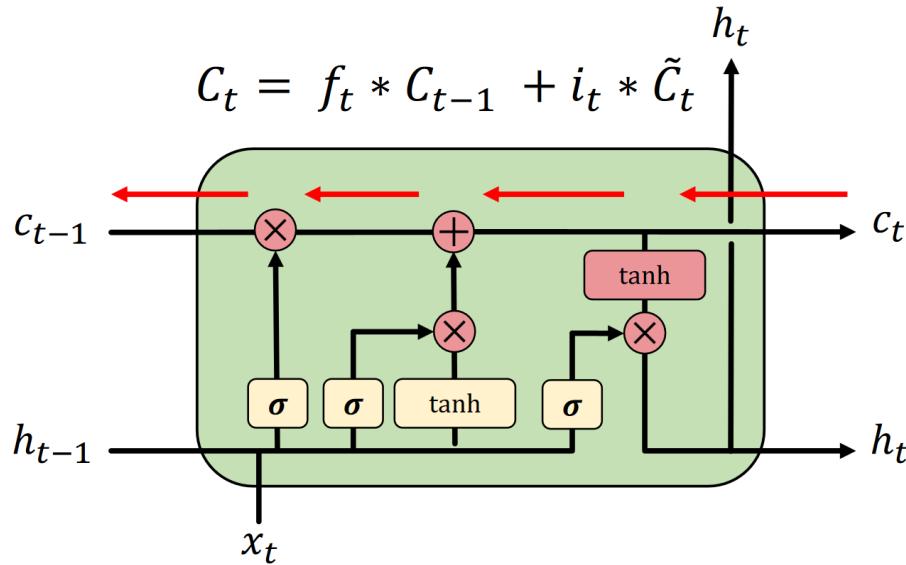
$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$$

$$g_t = \tanh(U_g x_t + W_g h_{t-1} + b_g)$$

$$c_t = i_t \odot g_t + f_t \odot c_{t-1}$$

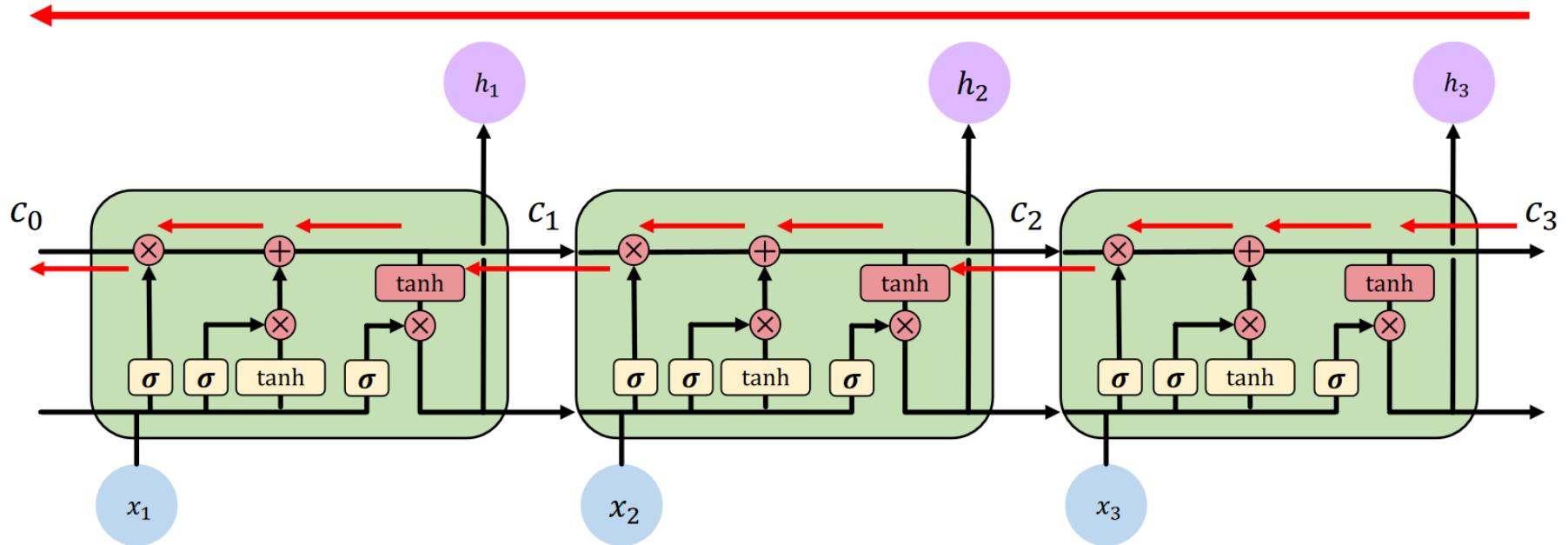
$$h_t = o_t \odot \tanh(c_t)$$

# Redes LSTM: flujo del gradiente



La propagación hacia atrás de  $C_t$  a  $C_{t-1}$  sólo utiliza multiplicaciones elemento por elemento. Al evitar multiplicaciones matriciales se evita el problema del desvanecimiento del gradiente.

# Redes LSTM: flujo del gradiente



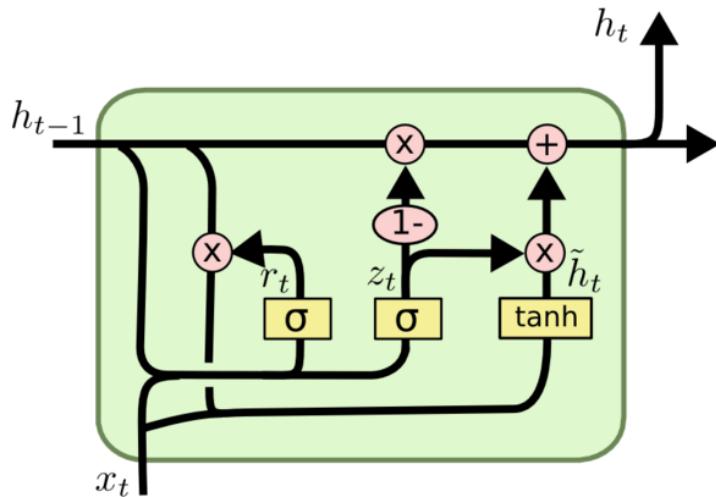
El gradiente fluye de forma ininterrumpida.

# Redes LSTM: conceptos clave

- ❑ Mantienen un estado de la celda independiente de la salida.
- ❑ Usa puertas para el control del flujo de información:
  - Olvida información irrelevante.
  - Actualiza el estado de la celda de forma selectiva.
  - La puerta de salida devuelve una versión filtrada del estado de la celda.
- ❑ La aplicación de la propagación hacia atrás no necesita multiplicaciones de matrices, porque el flujo de gradiente es continuo (sin interrupción)
- ❑ En Keras, las redes LSTM pueden tener estado (*stateful* LSTM) o no (*stateless* LSTM).
  - Sin estado (*stateless*): Borran la memoria tras cada lote (*batch*). Se usa cuando las predicciones de un lote son independientes del resto, como por ejemplo, en traducción de textos.
  - Con estado (*stateful*): Borran la memoria tras cada iteración (*epoch*). Se usan cuando la información de un lote está relacionada con el resto, como por ejemplo, en predicción de valores de cotización de acciones.

# Redes GRU (*Gated Recurrent Unit*)

- Son una modificación de las redes LSTM con menos parámetros.
- En estas redes, la salida y la memoria van por la misma ruta, por lo que carece de la puerta de salida.
- Tan solo dispone de dos puertas:
  - De actualización (*update gate*), similares a las puertas de olvido y entrada de las LSTM, decide que información descartar y qué información nueva añadir.
  - De borrado (*reset gate*), que decide cuánta información del pasado se olvida.
- Usa menos memoria y es más rápida que las redes LSTM, pero tienen un rendimiento peor cuando se trabaja con secuencias largas.



$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

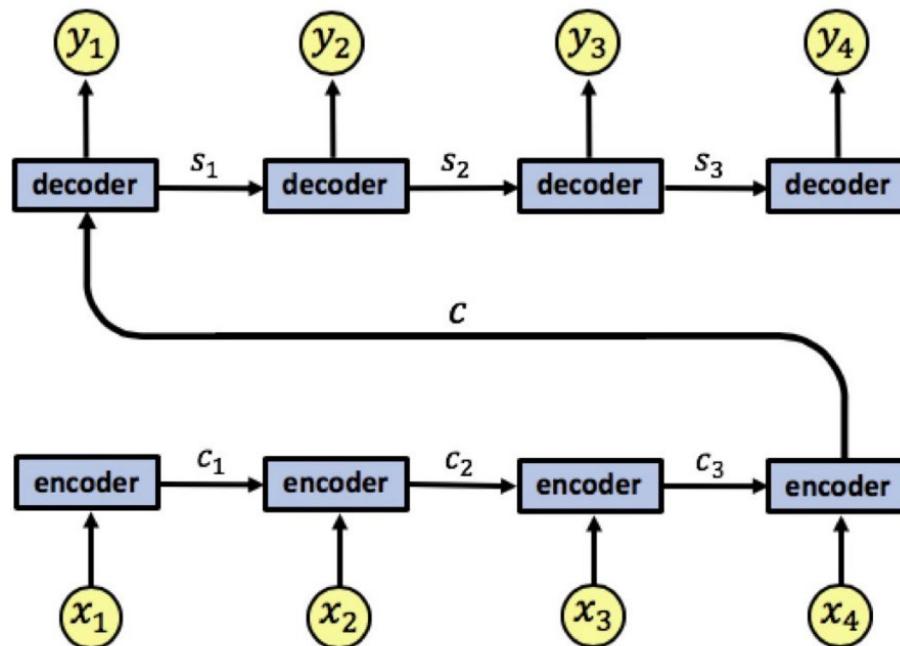
$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

$$h_t' = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t'$$

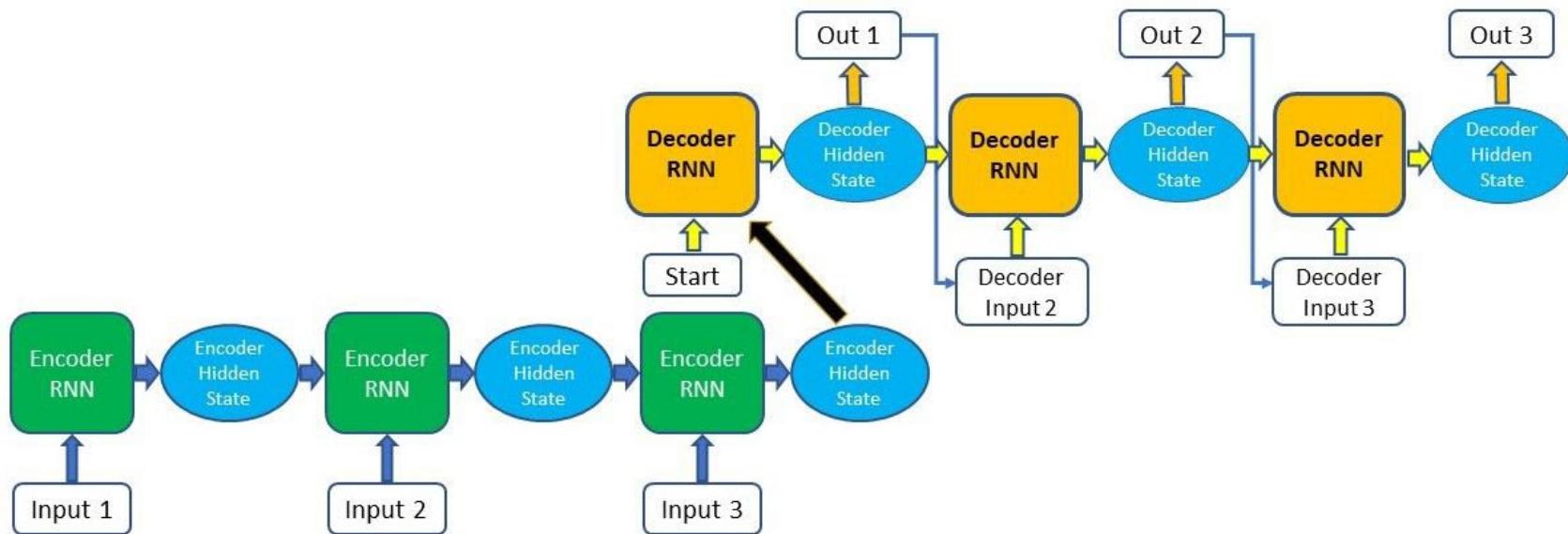
# Secuencia a secuencia

- El modelo **Seq2Seq** (*Sequence-to-Sequence*) se aplica a problemas de conversión de secuencia a secuencia, como la traducción automática o el reconocimiento de voz.
- Un modelo Seq2Seq contiene dos partes: un codificador y un descodificador, que generalmente son dos RNN/LSTM diferentes.
- La salida del codificador se utiliza como entrada del descodificador y éste último es responsable de emitir el resultado de traducción.



# Secuencia a secuencia: vector de contexto

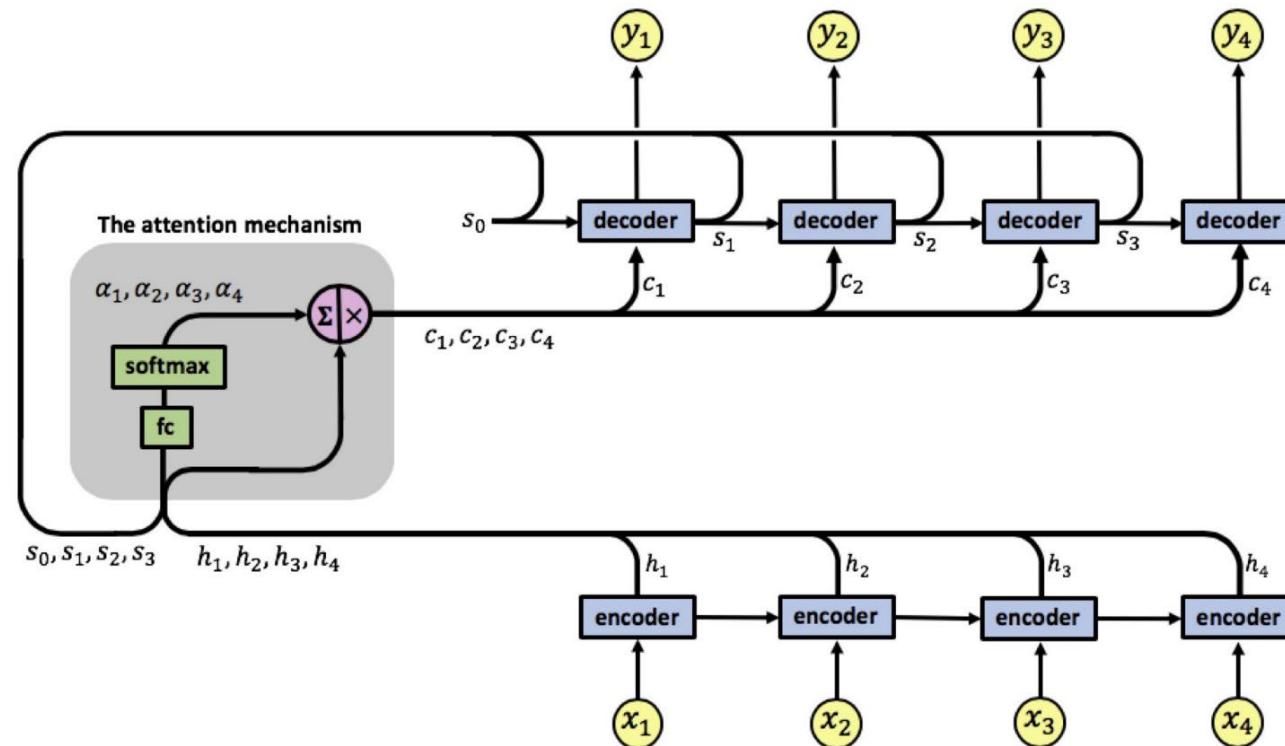
- El codificador codifica la secuencia de entrada (frase a traducir, por ejemplo) como un vector llamado vector de contexto.
- El decodificador toma el vector de contexto como entrada y genera la secuencia de salida (traducción, por ejemplo).



- El vector de contexto tiene que representar toda la información de la secuencia de entrada y esto es un problema, sobre todo en secuencias de entrada largas, debido al problema del desvanecimiento/explosión del gradiente.

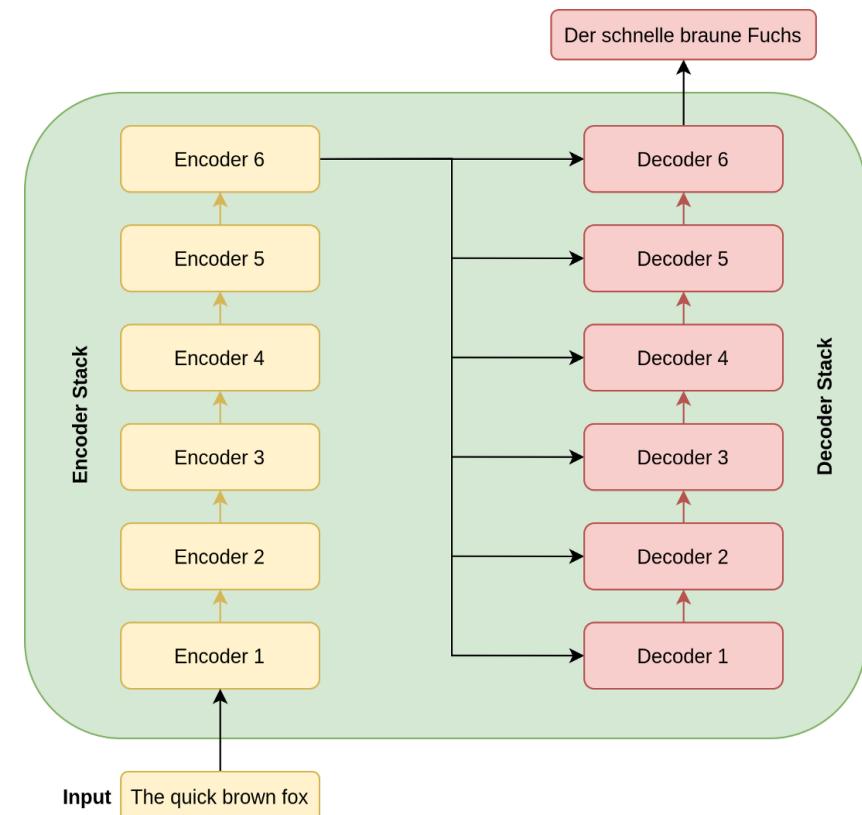
# Secuencia a secuencia: mecanismo de atención

- El mecanismo de atención soluciona el problema anterior porque permite que el descodificador tenga acceso a todos los estados pasados del codificador: en lugar de usar solo el último vector de contexto,  $h(t)$ , utiliza todos valores previos ( $h_{t-1}, h_{t-2} \dots$ ) ponderándolos de forma distinta.
- De esta forma, el descodificador se concentra en ciertas partes de la secuencia de entrada cuando se predice ciertas partes de la secuencia de salida.



# Secuencia a secuencia: *transformers*

- La arquitectura *Transformer* consiste básicamente, en un apilamiento de codificadores y descodificadores, sustituyendo las capas recurrentes (RNN, LSTM, GRU) por las denominadas **capas de atención**.
- Estas capas de atención codifican cada palabra de una frase en función del resto de la secuencia, permitiendo así introducir el contexto en la representación numérica del texto, motivo por el cual a los modelos basados en *Transformer* se les denomina también **embeddings contextuales**.
- Las capas descodificadoras, usan la información codificada por las capas codificadoras para generar la secuencia de salida.
- La arquitectura *Transformer* incluye otras innovaciones, como los **embeddings posicionales**, que permiten al algoritmo conocer la posición relativa de cada palabra del texto.



# Secuencia a secuencia: *transformers*

