

## Tema 7: Redes Neuronales

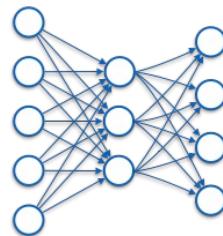
## Índice

- Introducción
- Neuronas artificiales
- Interpretación geométrica
- Arquitectura de la red. Perceptrón Multicapa (MLP)
- Diseño de la red
- Entrenamiento de la red
- *Backpropagation*
- Aprendizaje estocástico
- Ajuste de la red
- Otras Redes Neuronales Artificiales:
  - Redes Recurrentes
  - Redes Convolucionales
  - Transformers

# Introducción

**Sistema computacional** inspirado en redes neurales biológicas

Conjunto de **neuronas artificiales** conectadas entre si



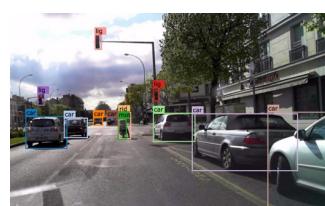
Son capaces de **aprender** presentándole una serie de ejemplos

Los ejemplos deben estar *etiquetados* con la salida esperada

Se van “*ajustando*” las conexiones entre neuronas

# Áreas de aplicación

Reconocimiento de imagen



Reconocimiento del habla

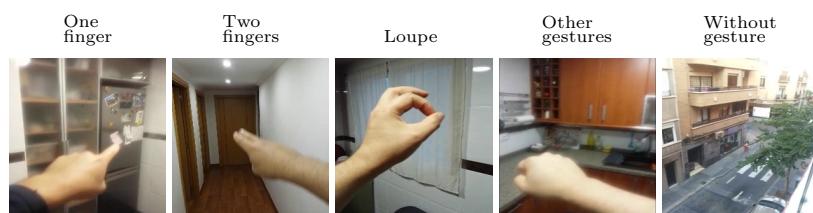
Procesamiento del lenguaje natural

Conducción autónoma

Diagnóstico médica

...

**Ejemplo:** Reconocimiento de gestos

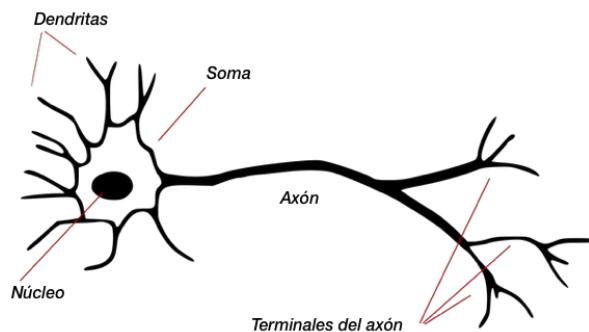


## Neuronas biológicas

**Dendritas:** Reciben entrada (potencial eléctrico) de otras neuronas

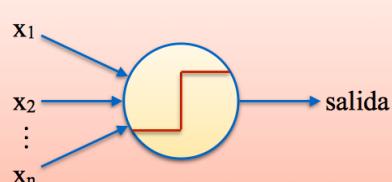
**Soma:** Integra las entradas. Es un dispositivo “*todo o nada*”, se activa si recibe suficiente potencial de entrada.

**Axón:** Transporta la salida a otras neuronas. La comunicación entre el axón de una neurona y las *dendritas* de otra se denomina *sinapsis*.

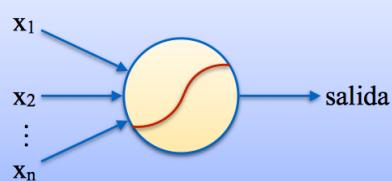


## Neuronas artificiales

### Perceptrón

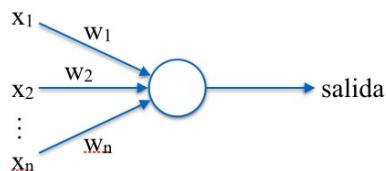


### Neurona sigmoidea



## Perceptrón

Neurona artificial que toma una serie de **entradas**  $x$  y produce una *salida*



El perceptrón **toma una decisión** (*salida*) ponderando ( $w$ ) una serie de factores ( $x$ )

$$\text{salida} = \begin{cases} 0 & \text{si } \sum_j w_j x_j \leq \text{umbral} \\ 1 & \text{si } \sum_j w_j x_j > \text{umbral} \end{cases}$$

## Perceptrón: Notación matricial

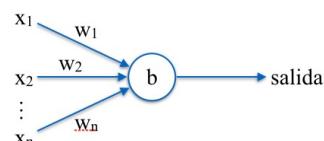
Podemos representar las entradas y pesos **como tuplas**

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

El **bias** ( $b$ ) nos indica lo fácil que es que el perceptrón “se dispare”  $b = -\text{umbral}$

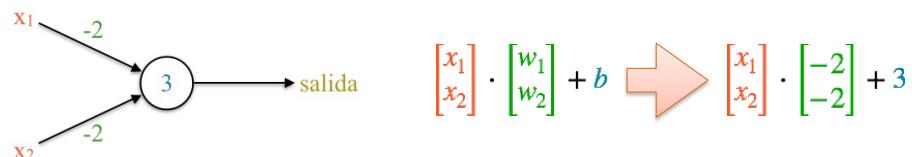
Podemos reescribir la **salida** como

$$\text{salida} = \begin{cases} 0 & \text{si } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{si } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$



## Ejemplo

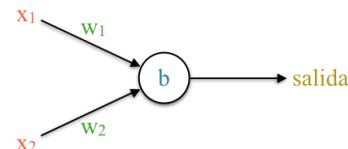
### Operación NAND



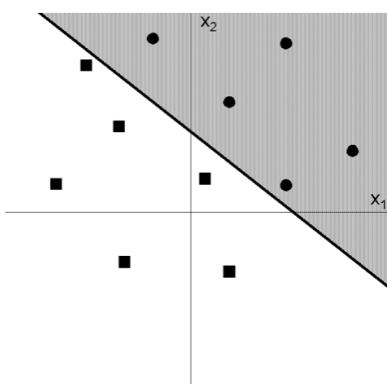
$x_1$	$x_2$	$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} -2 \\ -2 \end{bmatrix} + 3$	Salida
0	0	3	1
0	1	1	1
1	0	1	1
1	1	-1	0

- Podemos implementar cualquier operación lógica

### Interpretación geométrica

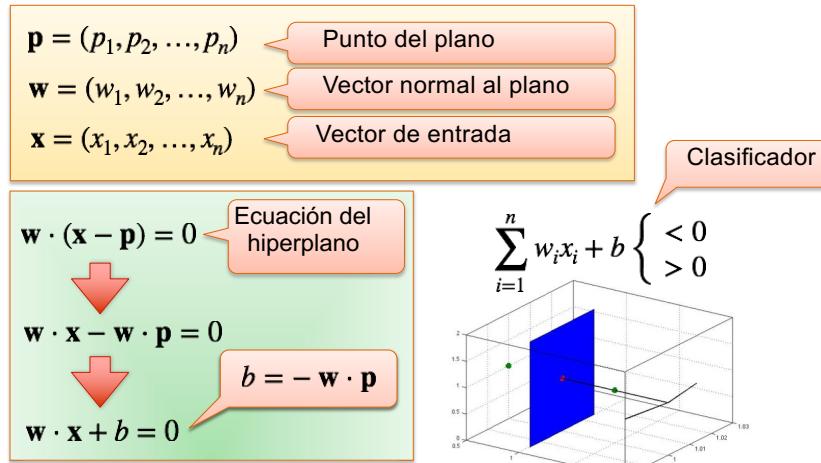


$w_1x_1 + w_2x_2 + b = 0$  Ecuación de la recta



## Interpretación geométrica

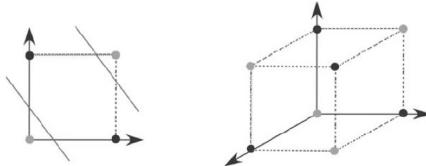
En el caso general de un perceptrón de  $n$  entradas, los datos se clasifican mediante un **hiperplano** de  $n$  dimensiones



## No-separabilidad lineal

Existen situaciones donde **un único hiperplano** no puede separar los datos

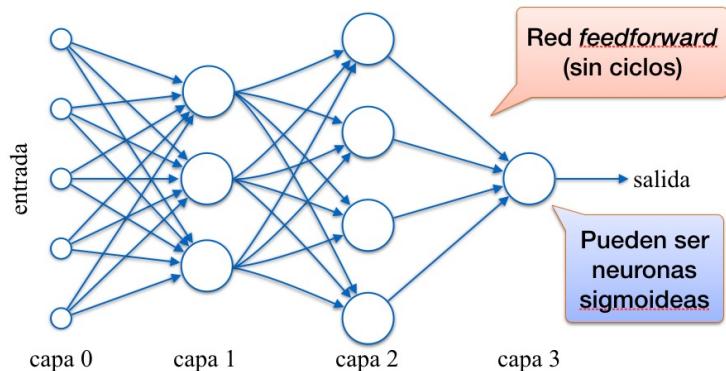
Por ejemplo cuando la frontera de decisión es curva



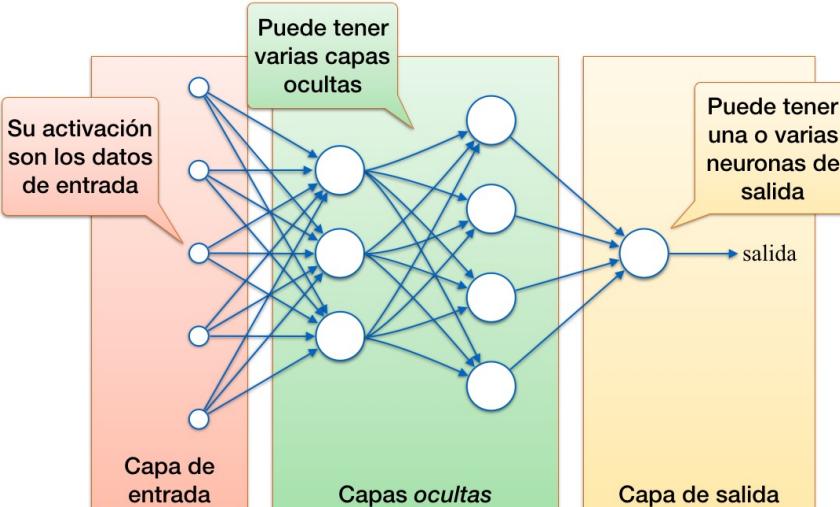
Podemos **combinar varios perceptrones** conectados entre si para implementar funciones más complejas

## Percepción Multicapa (MLP)

- Organizaremos las neuronas en forma de **capas**
  - La salida de las neuronas de una capa será la entrada de las de la siguiente
  - A mayor **número de capas**, podrá tomar decisiones **más complejas**



## Arquitectura de la red



## Tipos de redes

- **Shallow Networks**

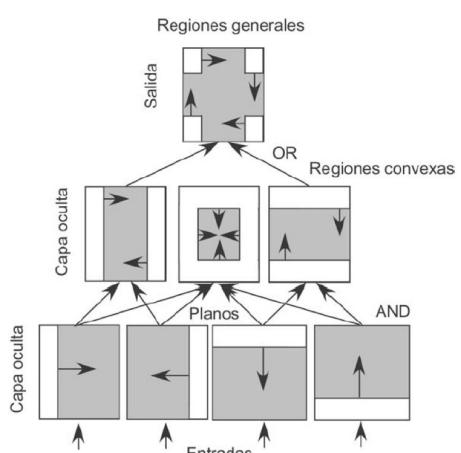
- Sólo una capa oculta

- **Deep Networks**

- Dos o más capas ocultas
- Habitualmente se entranan redes entre 5 y 10 capas
- Se entranan mediante **descenso por gradiente** y **back propagation**
- Nuevas técnicas han posibilitado el entranamiento de estas redes más complejas

## Interpretación geométrica

Se demuestra que un perceptrón con dos capas puede **aproximar cualquier función**



## Diseño de la red

- El diseño de las capas de entrada y salida suele ser directo

- Capa de entrada

- Tenemos en cuenta cómo podemos descomponer los datos que recibimos como diferentes neuronas de entrada

- Capa de salida

- Tenemos en cuenta cómo podemos codificar el resultado que queremos obtener como salida

- El diseño de las **capas ocultas** no es trivial

## Ejemplo: MNIST

**Base de datos MNIST** (<http://yann.lecun.com/exdb/mnist/>)

Reconocimiento de dígitos manuscritos del 0 al 9

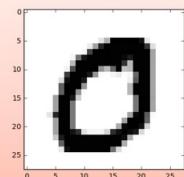
## Ejemplo: MNIST

- **Capa de entrada**

- Imágenes de 28x28 píxeles
- Niveles de gris de [0, 1]



- **784 neuronas** de entrada con valores [0, 1]



- **Capa de salida**

- Número del 0 al 10



- **10 neuronas** con valores {0, 1}



- **¿Cómo ajustamos los pesos?**

¡Puede ser entrenada!

## Entrenamiento

- No le decimos a la máquina **cómo** resolver un problema
  - La máquina **lo aprende** a partir de observar ejemplos
- Necesitamos entrenar la red con un **gran número de ejemplos** para ajustar los pesos que produzcan la salida deseada
- Dividiremos la base de datos en **dos conjuntos**:

- **Conjunto de entrenamiento (training)**

- Ejemplos con los que ajustamos los pesos de la red

- **Conjunto de prueba (test)**

- Ejemplos para validar los resultados se clasificación de la red

## Ejemplos de la base de datos

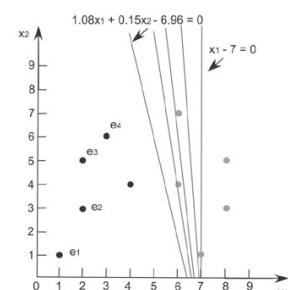
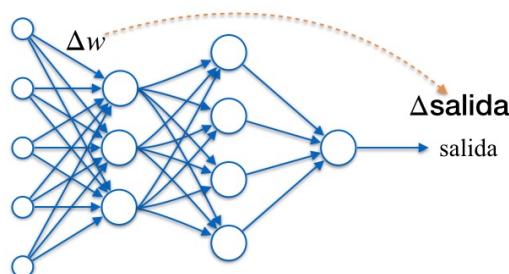
Para cada ejemplo tendremos una **entrada  $x$**  y una **salida esperada  $y(x)$**

Por ejemplo, en el caso del reconocimiento de dígitos la salida esperada para diferentes ejemplos podría ser:

$x$	$y(x)$
	$[1,0,0,0,0,0,0,0,0]^T$
	$[0,1,0,0,0,0,0,0,0]^T$
	$[0,0,0,0,0,1,0,0,0,0]^T$

## ¿Cómo entrenamos la red?

- **Idea:** Supongamos que una pequeña modificación en un peso provoca una pequeña modificación de la salida



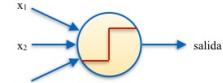
- Podemos ir **modificando los pesos y biases** de forma que nos acerque a la salida deseada
- **Problema:** La salida del perceptrón no se modifica poco a poco, es un escalón

## Neurona sigmoidea

- Definimos:  $z = w \cdot x + b$

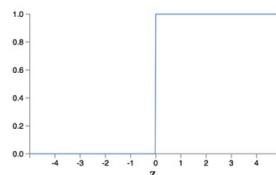
Entrada ponderada de la neurona

### Perceptrón



Función de activación

$$f(z) = \begin{cases} 0 & \text{si } z \leq 0 \\ 1 & \text{si } z > 0 \end{cases}$$

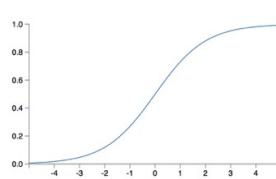


### Neurona sigmoidea



Función de activación

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



## Propiedades de la neurona sigmoidea

Con valores de  $z$  de gran magnitud equivale a la función escalón

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



$$z \ll 0 : \lim_{z \rightarrow -\infty} \sigma(z) = 0$$

$$z \gg 0 : \lim_{z \rightarrow \infty} \sigma(z) = 1$$

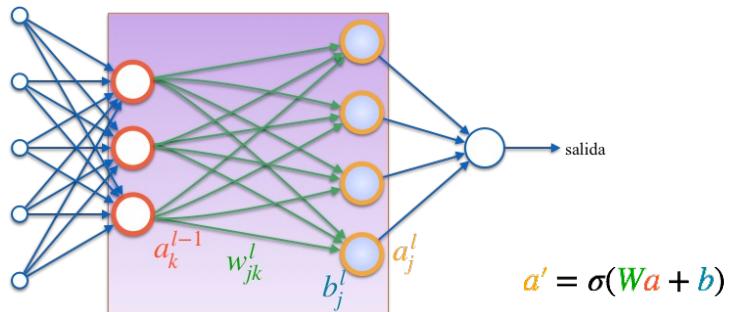
Podemos ver la función *sigmoidea* como un **escalón suavizado**

**Pequeños cambios** de los pesos  $\Delta w$  y del bias  $\Delta b$  producen pequeños cambios en la salida

$$\Delta \text{salida} \approx \sum_j \frac{\partial \text{salida}}{\partial w_j} \Delta w_j + \frac{\partial \text{salida}}{\partial b} \Delta b$$



## Cálculo de la activación en cada capa

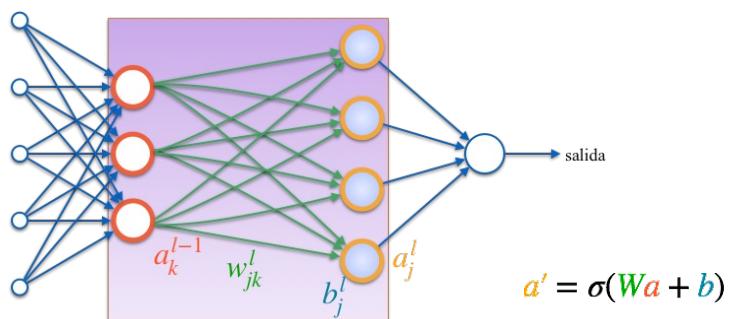


$$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad a' = \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \\ a'_4 \end{bmatrix}$$

25

25

## Cálculo de la activación en cada capa

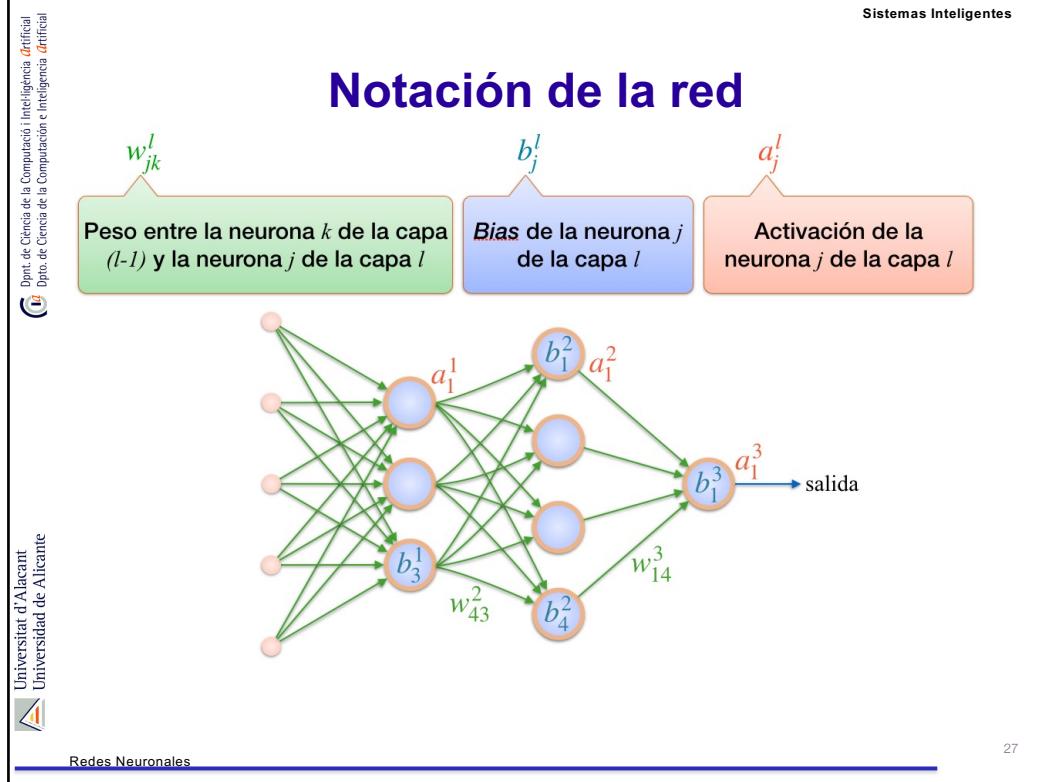


$$\begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \\ a'_4 \end{bmatrix} = \sigma \left( \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \right)$$

26

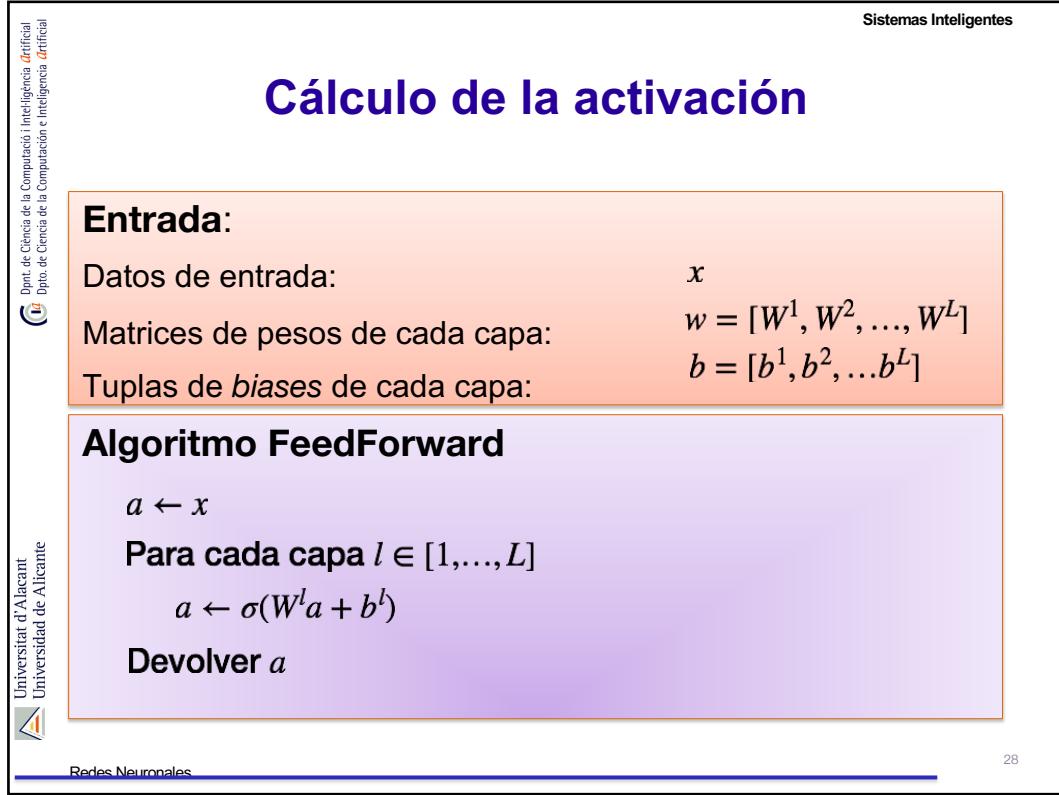
26

## Notación de la red

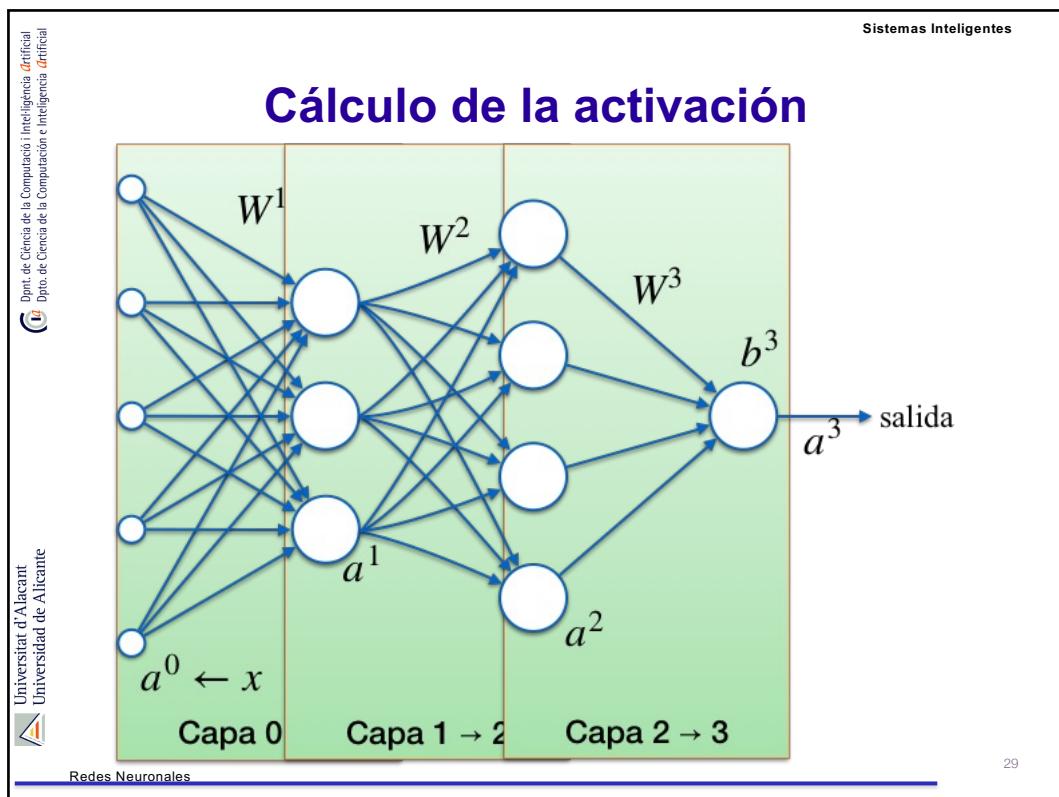


27

## Cálculo de la activación



28



Sistemas Inteligentes

## Función de coste

Debemos buscar los **pesos  $w$**  y **biases  $b$**  de toda la red que produzcan las salidas deseadas.

Dado:

- $a$ : **salida real** de la red para la entrada  $x$ , pesos  $w$  y biases  $b$
- $y(x)$ : **salida esperada** de la red para la entrada  $x$
- $n$ : **Número total de ejemplos** de entrenamiento (*training*)

Podemos definir una **función de coste  $C$**  que evalúe el error cuadrático medio (MSE) de clasificación de la red:

$$C(w, b) = \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

**Debemos minimizar su valor**

Dip. de Ciencia de la Computación i Inteligencia Artificial  
Dpto. de Ciencia de la Computación e Inteligencia Artificial

Universitat d'Alacant  
Universidad de Alicante

Redes Neuronales

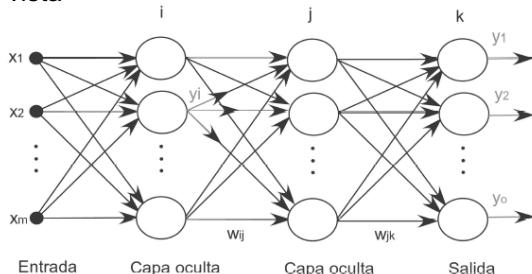
30

## Backpropagation: explicación heurística

- Supongamos que al clasificar un ejemplo una neurona de la última capa tiene una salida  $y_k$ , siendo la deseada  $d_k$
- Dicha neurona es responsable de un error

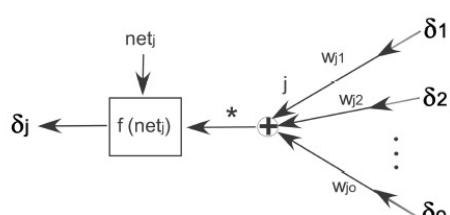
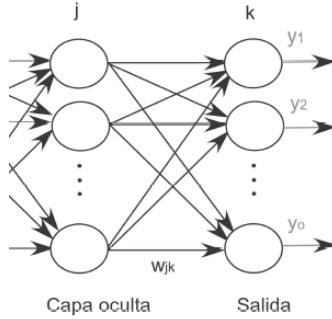
$$\delta_k = (d_k - y_k) f'(net_k),$$

- La regla de actualización de los pesos de la última capa será similar a la regla delta ya vista



## Error (delta) en capas intermedias

- Una neurona de una capa intermedia contribuye en los  $\delta$  de las de la capa siguiente
- Por tanto, para calcular su  $\delta$  necesitamos estos



## Backpropagation: algoritmo

- Se aplica para cada ejemplo del conj. de entrenamiento.  
Se itera hasta que el error baje de un umbral

- Fases:

- Hacia delante: cálculo de la salida de la red (los  $y_k$ ), Cálculo de los  $\delta$  en la última capa
- Hacia atrás. Cálculo de los  $\delta$  de la capa en función de los de la siguiente
- Finalmente, actualización de los pesos de todas las capas

```

Algoritmo BACKPROPAGATION(red ejemplos,  $\eta$ ) {
     $\{w_{ij}\} \leftarrow$  INICIALIZAR;
    Mientras  $\neg$  CONVERGENCIA(red) Hacer {
         $e \leftarrow$  SELECCIONAREJEMPLO(ejemplos);
         $\{y_k\} \leftarrow$  FORWARD(e);
         $\{d_k\} \leftarrow$  DESEADAS(e);
        Para cada  $n_k \in$  CAPA(red, k) Hacer {
             $\delta_k = (d_k - y_k)f'(net_k);$ 
        }
        Para  $j = k - 1$  hasta 1 Hacer {
            Para  $n_j \in$  CAPA(red, j) Hacer {
                 $\delta_j = f'(net_j) \sum_{j+1} \delta_{k+1} w_{j(k+1)};$ 
            }
            Para  $j = k$  hasta 1 Hacer {
                 $w_{(j-1)j} = w_{(j-1)j} + \eta \delta_j y_{(j-1)};$ 
            }
            red  $\leftarrow$  ACTUALIZARRED( $\{w_{ij}\}$ );
        }
        Devolver red;
    }
}

```

## Perceptrón Multicapa (MLP)

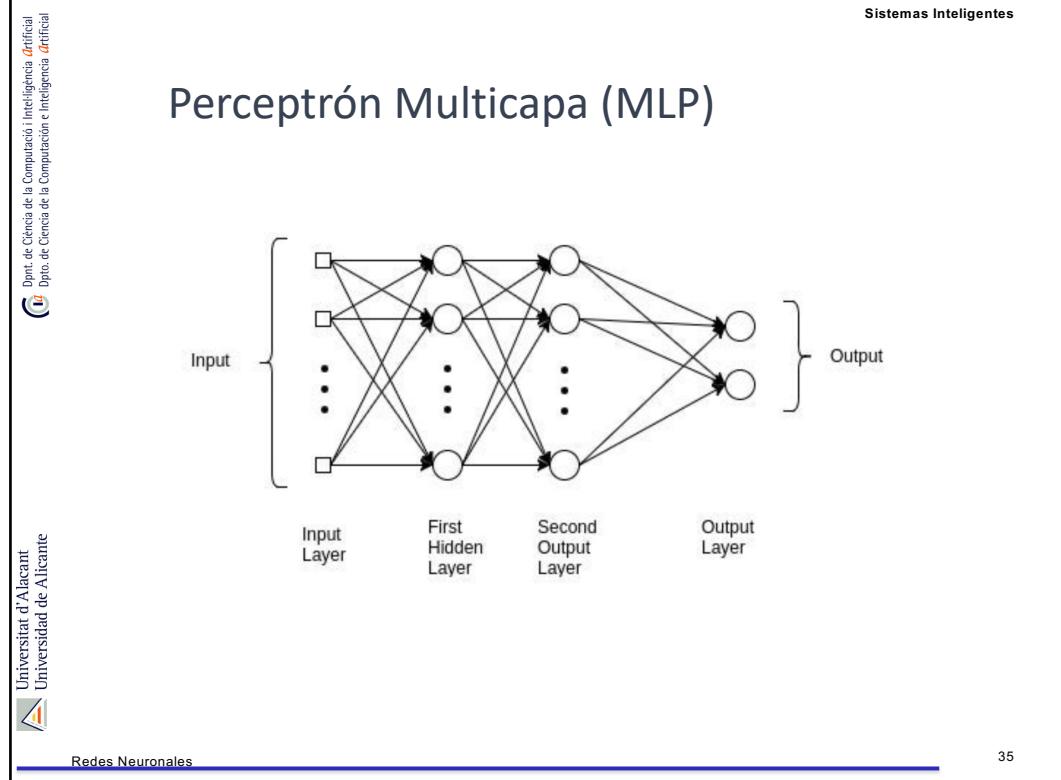
Un perceptrón multicapa (MLP) es un tipo de red neuronal artificial organizada en múltiples capas. Los MLP están compuestos por al menos tres capas de nodos o neuronas:

- 1.Capa de entrada: Recibe las señales de entrada.
- 2.Capas ocultas: Realizan transformaciones no lineales de las entradas. Puede haber una o más capas ocultas.
- 3.Capa de salida: Proporciona la respuesta o salida de la red.

Cada nodo en una capa está conectado a todos los nodos de la capa siguiente mediante conexiones ponderadas. Cada conexión tiene un peso asociado que se ajusta durante el entrenamiento de la red. Además, cada nodo (excepto los de la capa de entrada) tiene una función de activación, que suele ser no lineal, para introducir no linealidades en el modelo. Esto permite que los MLP puedan aproximar funciones complejas y resolver problemas que no son linealmente separables.

El proceso de ajuste de los pesos se realiza a través de un procedimiento de entrenamiento, típicamente usando el algoritmo de retropropagación junto con un algoritmo de optimización como el descenso del gradiente. Durante el entrenamiento, se minimiza una función de pérdida que mide la diferencia entre las salidas predichas por la red y las salidas reales esperadas.

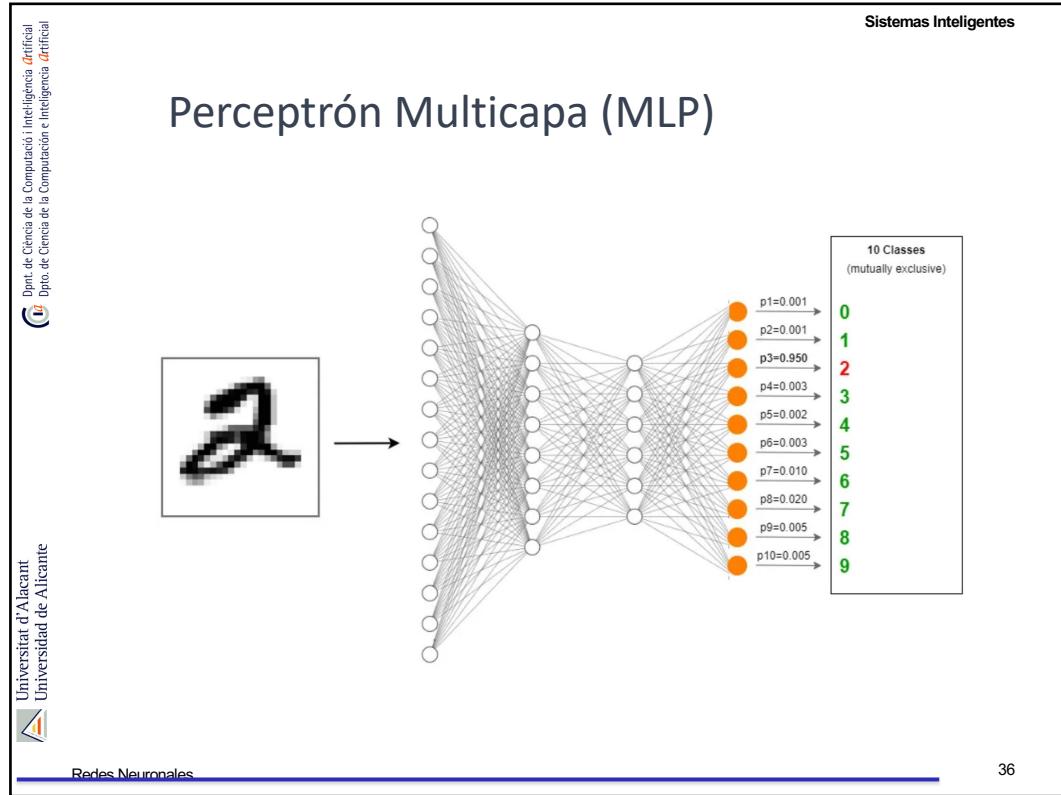
## Perceptrón Multicapa (MLP)



35

35

## Perceptrón Multicapa (MLP)



36

36

## Perceptrón Multicapa (MLP) Utilizando keras

```

print(f"MLP")

# Importar las librerías necesarias
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.utils import to_categorical

# Cargar y procesar los datos
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalizar los valores de los pixeles en el rango [0, 1]
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# Convertir las etiquetas en representación one-hot
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Definir el modelo MLP
model = Sequential()
model.add(Flatten(input_shape=(28, 28))) # Aplanar la entrada 28x28 a un vector de 784
model.add(Dense(512, activation='relu')) # Capa oculta con 512 neuronas y función de activación ReLU
model.add(Dropout(0.2)) # Dropout del 20% para prevenir el sobreajuste
model.add(Dense(512, activation='relu')) # Otra capa oculta con 512 neuronas
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax')) # Capa de salida con 10 neuronas (una por dígito) y función de activación softmax

# Compilar el modelo
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Entrenar el modelo
model.fit(x_train, y_train, batch_size=128, epochs=10, verbose=1, validation_data=(x_test, y_test))

# Evaluar el rendimiento del modelo
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

```

## Redes Recurrentes (RNN)

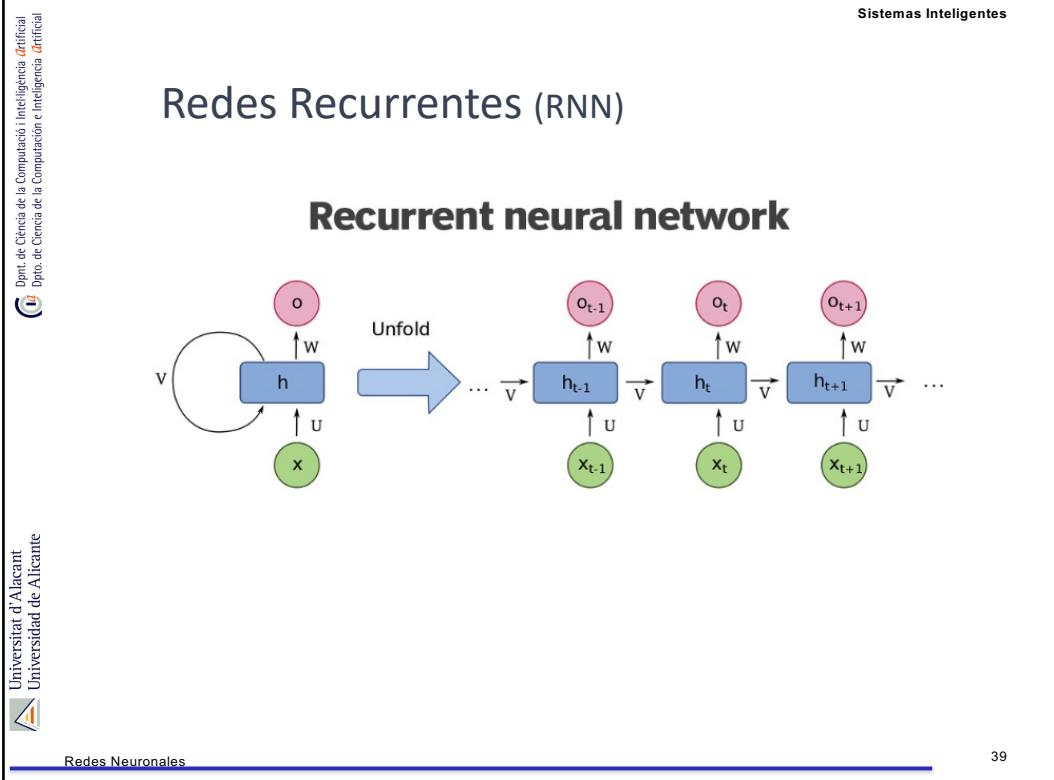
Una red neuronal recurrente (RNN) es un tipo de red neuronal artificial que se utiliza para procesar datos secuenciales. A diferencia de las redes neuronales feedforward, en las que la información fluye en una sola dirección, en las RNN la información fluye en bucle, es decir, la salida en un momento dado se utiliza como entrada en el siguiente momento.

Estas características hacen que las RNN sean especialmente buenas para tareas que involucran secuencias de datos. Por ejemplo, pueden utilizarse para:

- 1. Reconocimiento de voz:** Convertir el audio hablado en texto.
- 2. Generación de texto:** Crear texto coherente y gramaticalmente correcto.
- 3. Traducción automática:** Traducir texto de un idioma a otro.
- 4. Análisis de sentimientos:** Determinar si un texto tiene una connotación positiva o negativa.
- 5. Predicción de series temporales:** Prever tendencias futuras basándose en datos históricos.

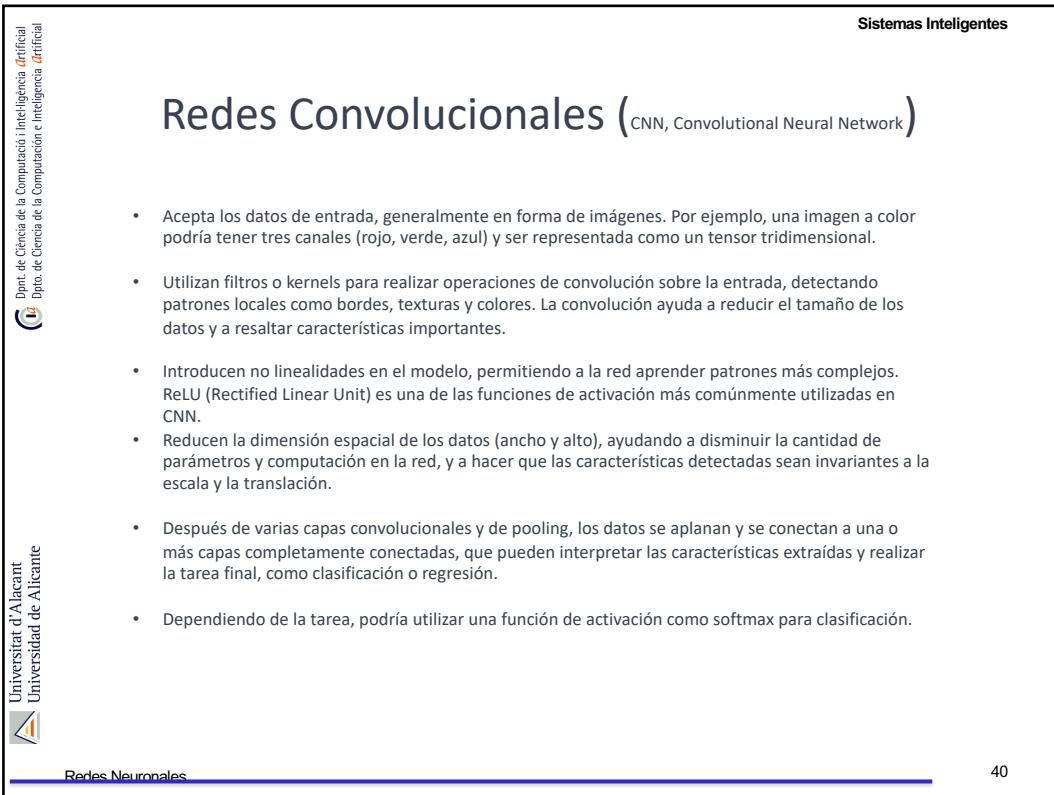
## Redes Recurrentes (RNN)

### Recurrent neural network



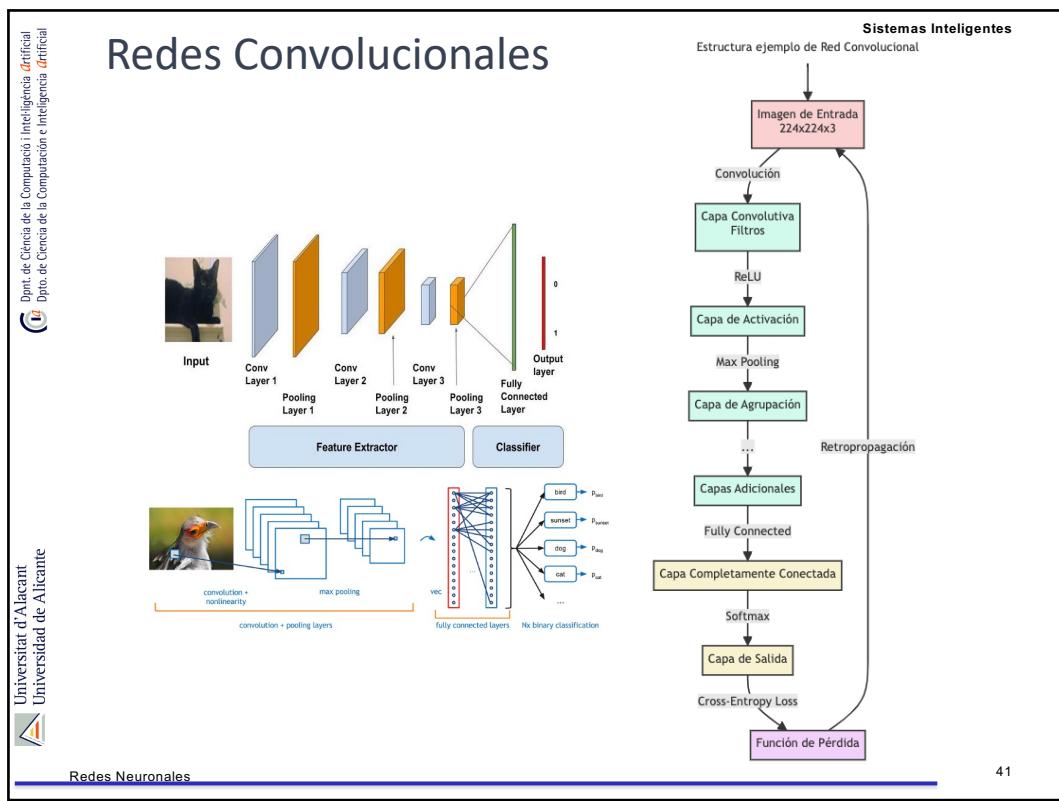
39

39

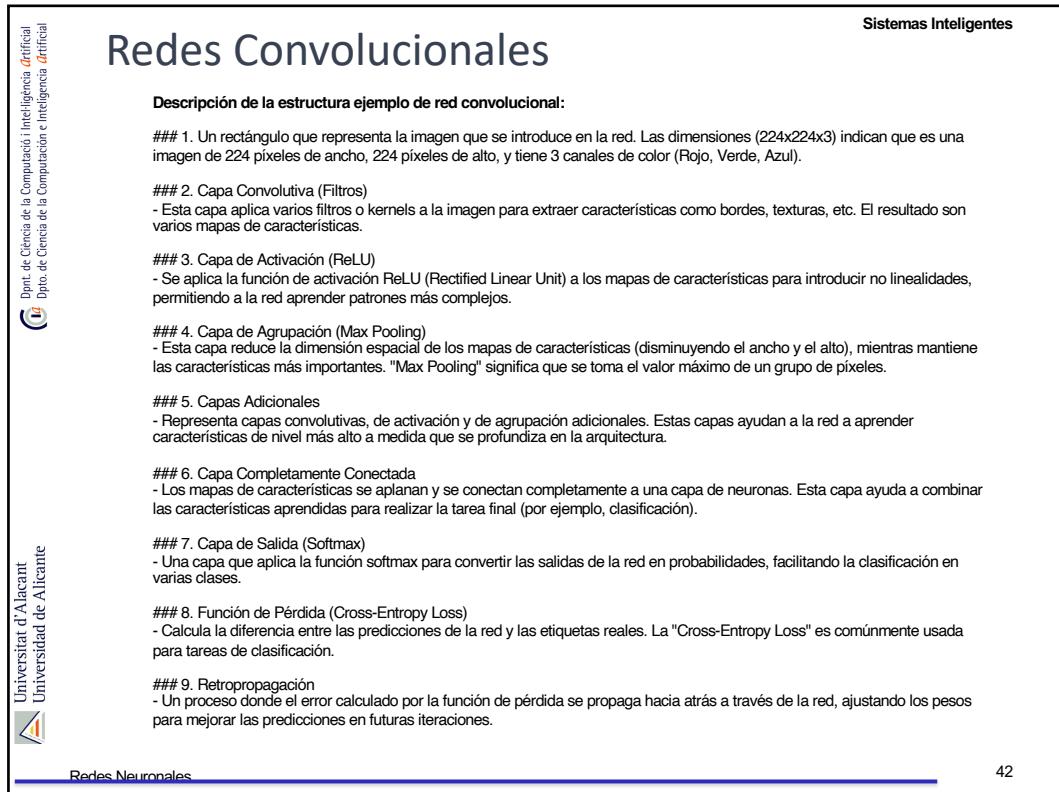


40

40



41



42

# Transformers

"Transformer" en LLM se refiere a un tipo de arquitectura de red neuronal desarrollada por investigadores de Google en 2017, que ha demostrado ser muy efectiva para una amplia variedad de tareas de procesamiento del lenguaje natural (PLN), incluyendo la traducción automática, el resumen de texto y la generación de texto. Los "LLM" se refieren a los Modelos de Lenguaje a Gran Escala.

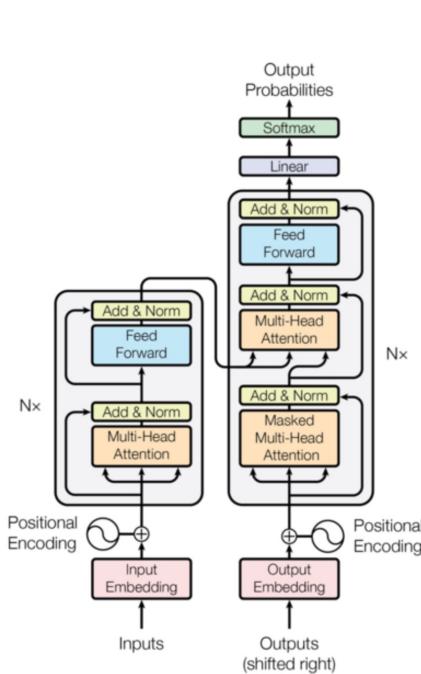
**1. Codificación de Posición:** Dado que los Transformers no tienen una noción inherente del orden de las palabras en una secuencia, utilizan codificaciones de posición para darle al modelo información sobre la posición de cada palabra en una secuencia.

**2. Atención de Múltiples Cabezas:** Esta es una de las innovaciones clave de los Transformers. Permite al modelo prestar atención a diferentes partes de la entrada para cada palabra en la salida, ayudando a capturar mejor las dependencias a larga distancia y las relaciones entre palabras.

**3. Capas de Atención:** Estas capas utilizan la atención de múltiples cabezas para transformar la entrada en una representación más rica, que luego se puede usar para tareas específicas.

**4. Capas de Feedforward:** Despues de las capas de atención, hay capas de feedforward que realizan transformaciones adicionales en la representación.

**5. Normalización de Capa y Conexiones Residuales:** Estas ayudan a estabilizar el entrenamiento y permiten que los gradientes fluyan más fácilmente a través de la red durante el retropropagación.



La Arquitectura de los Transformers

Sistemas Inteligentes

## Transformers

¿Cómo Funcionan en los LLM?:

En el contexto de los LLM, los Transformers se utilizan para capturar las complejidades y sutilezas del lenguaje humano. Estos modelos son entrenados en enormes cantidades de texto para aprender patrones, relaciones semánticas, gramática, y otros aspectos del lenguaje.



1. **Entrenamiento:** Los LLM basados en Transformers son entrenados para predecir la siguiente palabra en una secuencia dada el contexto de las palabras anteriores. Esto se hace mediante el ajuste de los pesos de la red para minimizar la diferencia entre las predicciones del modelo y las palabras reales.
2. **Generación de Texto:** Una vez entrenados, estos modelos pueden ser utilizados para generar texto, completar oraciones, responder preguntas, y realizar muchas otras tareas de PLN. Utilizan la atención para enfocarse en diferentes partes del texto de entrada y generar una respuesta con discurso coherente. **Problema: presentan alucinaciones.**
3. **Fine-Tuning:** Aunque los LLM pueden ser muy poderosos, a menudo necesitan ser afinados en un conjunto de datos más pequeño y específico para realizar bien tareas específicas.



Redes Neuronales

45

45

Sistemas Inteligentes

## Bibliografía

Escolano et al. Inteligencia Artificial. Thomson-Paraninfo 2003. Capítulo 4.



Mitchell, Machine Learning. McGraw Hill, Computer Science Series. 1997

Reed, Marks, Neural Smithing. MIT Press, CA Mass 1999

Neural Networks and Deep Learning. Libro digital:  
<http://neuralnetworksanddeeplearning.com/index.html>

Neural Networks. Canal de YouTube 3blue1brown  
<https://youtu.be/aircAruvnKk>



Redes Neuronales

46

46