

# Redes neuronales artificiales

---

Aprendizaje profundo

Departamento de Sistemas Informáticos

E.T.S.I. de Sistemas Informáticos - UPM

2 de marzo de 2024



# Conceptos generales (I)

---

Las redes neuronales (ANN, del inglés *artificial neural networks*) son un modelo computacional inspirado en el funcionamiento del cerebro humano

- Están formadas por un conjunto de nodos (neuronas) interconectados
- Cada neurona recibe una serie de entradas, las procesa y produce una salida
- Las conexiones tienen un peso que modifica la influencia de la entrada en la salida
- **La red se entrena ajustando los pesos de las conexiones para minimizar el error de la salida**

La misma red neuronal puede adaptarse para diferentes tipos de entrada (imágenes, texto, numéricos) y obtener buenos resultados

# Conceptos generales (II)

---

El proceso de funcionamiento de una red neuronal es el siguiente

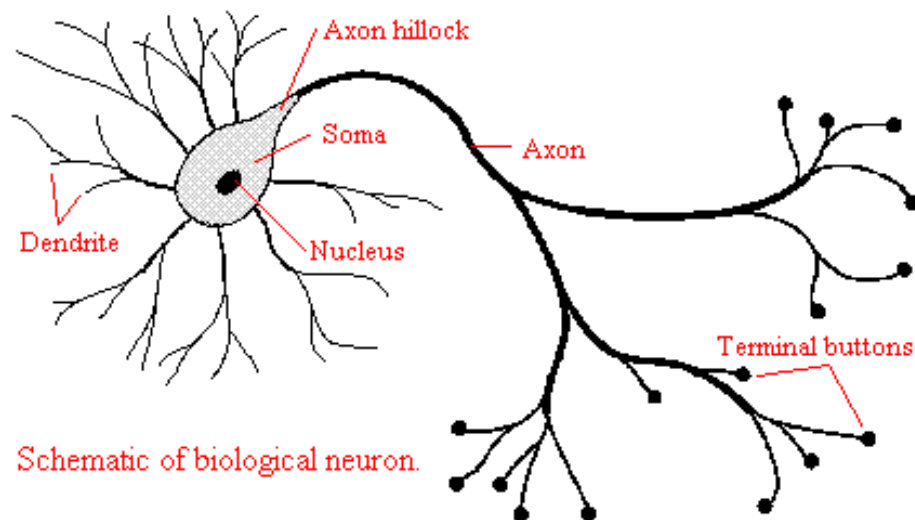
1. Introducimos unos datos de entrada en la red
2. Realizamos la predicción, obteniendo la salida
  - También inferencia, propagación hacia adelante o, simplemente, propagación
3. Si estamos en el proceso del entrenamiento de la red, además:
  - i. Comparamos la predicción con la salida esperada, viendo el error
    - Esto sólo en el caso de un esquema de entrenamiento supervisado
  - ii. Ajustamos los parámetros internos para tratar de minimizar ese error y mejorar las futuras predicciones

Los datos se almacenan como vectores y matrices, por eso es útil el hardware destinado a cálculos matriciales como las GPU

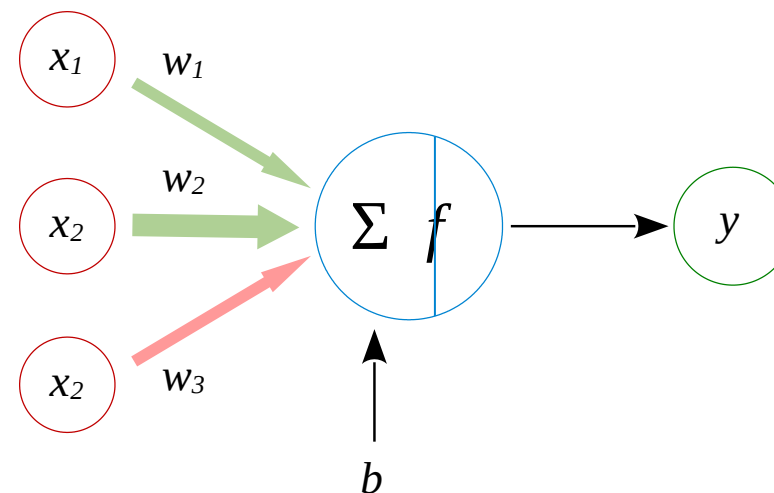
# Neurona artificial o **perceptrón**

Modelo matemático que **simula el comportamiento de una neurona biológica**

1. Toma las entradas  $x_1, x_2, \dots, x_n$ , cada una con su peso  $w_1, w_2, \dots, w_n$
2. Aplica una función de activación a la suma ponderada de las entradas y los pesos



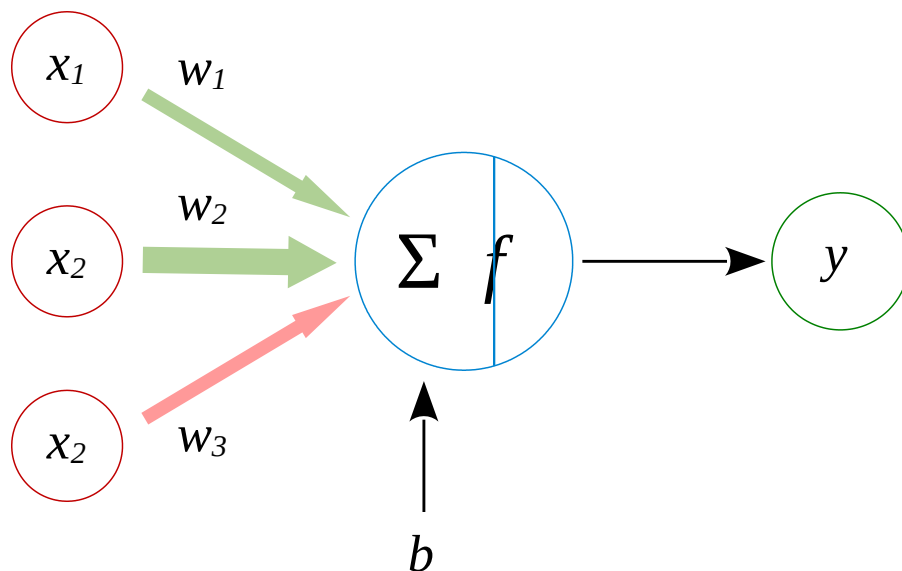
**Figura 1.** Esquema de neurona biológica.



**Figura 2.** Esquema de neurona artificial.

# Inferencia

Obtención de la salida a partir de la suma ponderada de las entradas y sus pesos



**Figura 3.** La salida se obtiene en **función** de la suma ponderada de las entradas.

Dos formas de obtener la salida:

## 1. Escalar

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

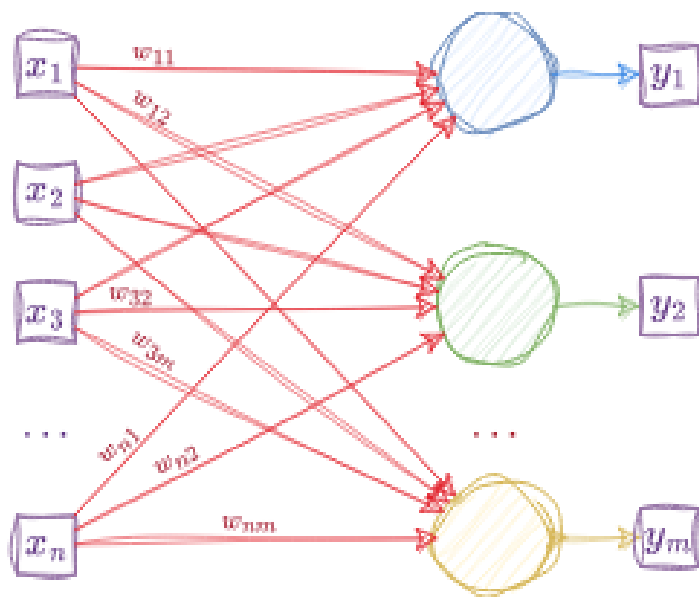
## 2. Forma vectorial

$$\hat{y} = f(WX)$$

Ambas formas **son equivalentes**

# Múltiples salidas (I)

Varias salidas para una misma entrada es equivalente a tener varias neuronas



**Figura 4.** Múltiples salidas para una misma entrada es, en esencia, varios perceptrones.

Las entradas siguen siendo un vector  $X$

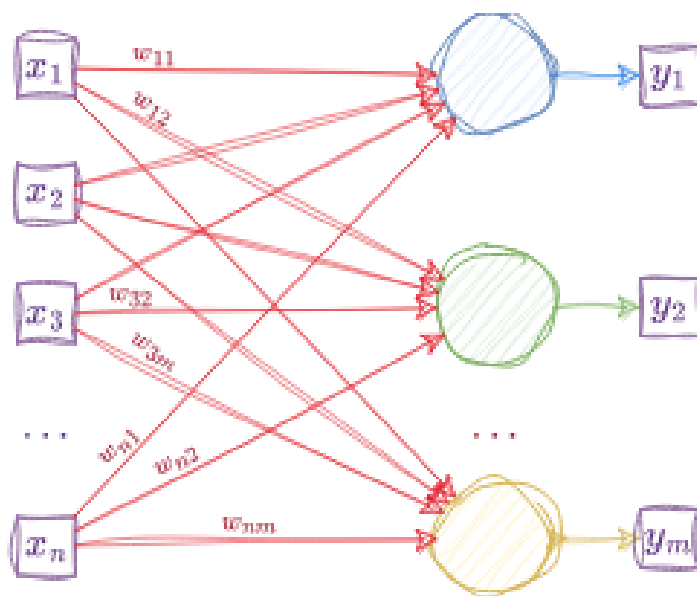
$$X = [x_1 \quad x_2 \quad \dots \quad x_n]$$

Los pesos de la red serán una **matriz**  $W$

$$W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix}$$

# Múltiples salidas (II)

Varias salidas para una misma entrada es equivalente a tener varias neuronas



**Figura 4.** Múltiples salidas para una misma entrada es, en esencia, varios perceptrones.

La salida será

$$\hat{y} = f(WX) = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_m \end{bmatrix}$$

Es decir, una inferencia por neurona

- Esta estructura se conoce como **capa**
- Cobrará importancia más adelante

# Perceptrón

**Notebook:** [Perceptrón simple y perceptrón multicapa.ipynb](#)



# Funciones de activación (i)

Son quienes determinan la salida de la neurona; algunos ejemplos son:

## Funciones escalón

- Heaviside:  $f(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$
- Signo:  $f(x) = \begin{cases} -1 & \text{si } x < 0 \\ 0 & \text{si } x = 0 \\ 1 & \text{si } x > 0 \end{cases}$

## Funciones lineales

- Identidad:  $f(x) = x$
- Lineal:  $f(x) = ax + b$

## Funciones sigmoide

- Logística:  $f(x) = \frac{1}{1+e^{-x}}$
- Tan. hiperbólica:  $f(x) = \tanh(x)$

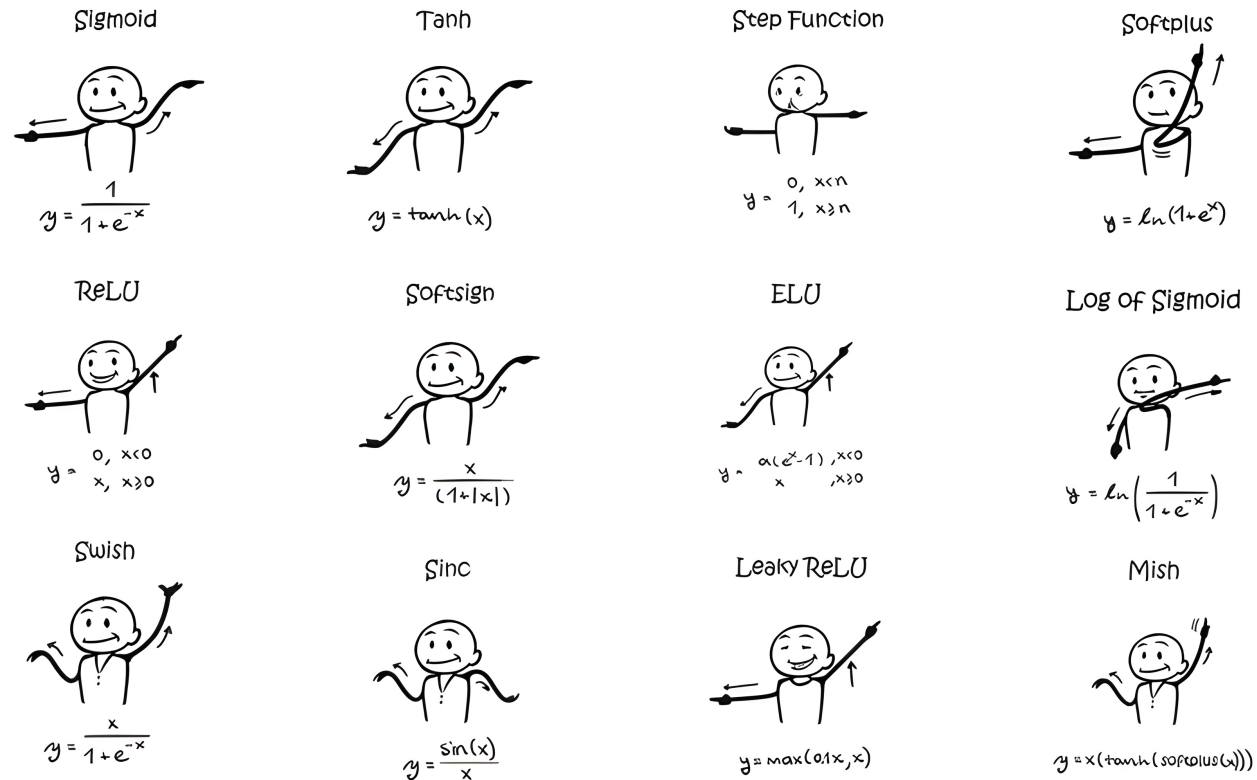
## Funciones rectificadas

- ReLU:  $f(x) = \max(0, x)$
- Leaky ReLU:  $f(x) = \max(\alpha x, x)$

## Funciones de activación suaves

- Softmax:  $f(x) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$

# Funciones de activación (II)



**Figura 5.** Algunas funciones de activación comunes.

# Estimación de error o *loss*

---

El error de la red se mide con una función de pérdida (*loss function*)

- Mide la diferencia entre la salida de la red y la salida esperada
- El objetivo es minimizar esta función
- Algunas funciones de pérdida comunes son:
  - **Error cuadrático medio:**  $\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
  - **Entropía cruzada:**  $-\sum_{i=1}^n y_i \log(\hat{y}_i)$
  - **Error absoluto medio:**  $\frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$
- La elección de la función de pérdida depende del problema a resolver
- En general, se busca que sea derivable para poder aplicar algoritmos de optimización

# Métricas

**Notebook:** [Métricas para problemas de clasificación y regresión.ipynb](#)

# Entrenamiento de redes neuronales

# ¿En qué consiste el entrenamiento?

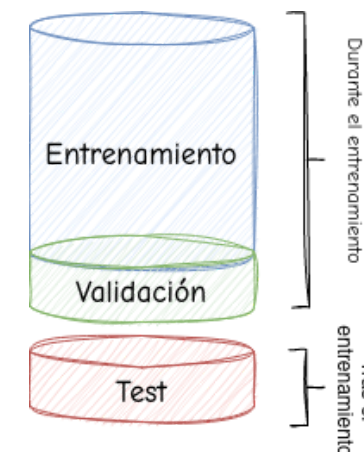
Consiste en ajustar los pesos de las conexiones para minimizar el error

Para aumentar su fiabilidad, se suele realizar una división de tres conjuntos:

1. **Entrenamiento:** Ajuste de parámetros (pesos)
2. **Validación:** Ajuste de hiperparámetros
3. **Test:** Validación una vez finalizado el entrenamiento

El conjunto de validación se usa **durante el entrenamiento**

- El conjunto de test se utiliza **al final del entrenamiento**

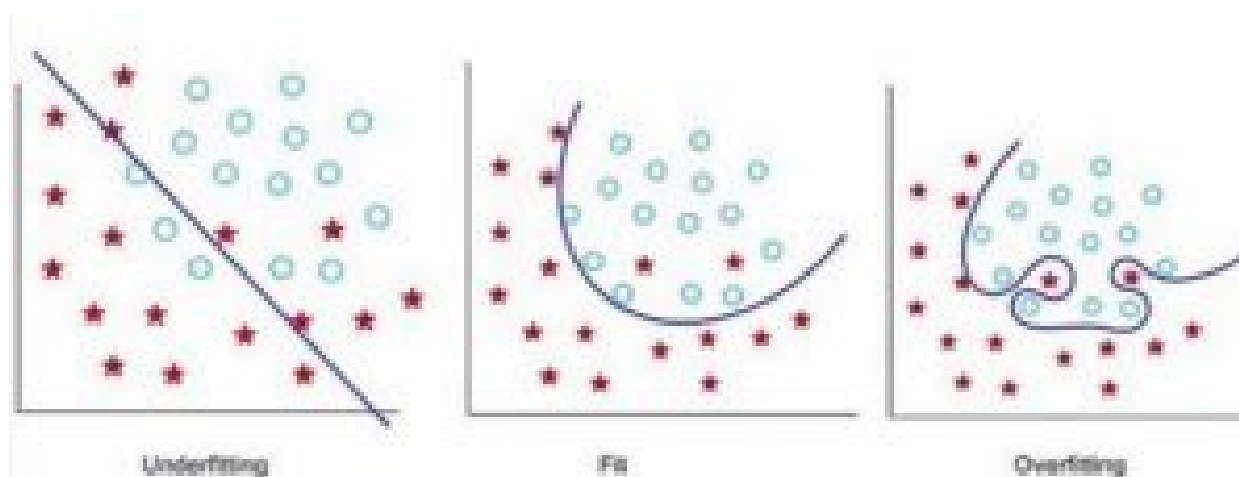


**Figura 6.** Diferentes conjuntos de datos.

# Problemas de ajuste (I)

Compromiso sesgo-varianza (*bias-variance tradeoff*): Dos métricas que ayudan a evaluar el comportamiento de un modelo:

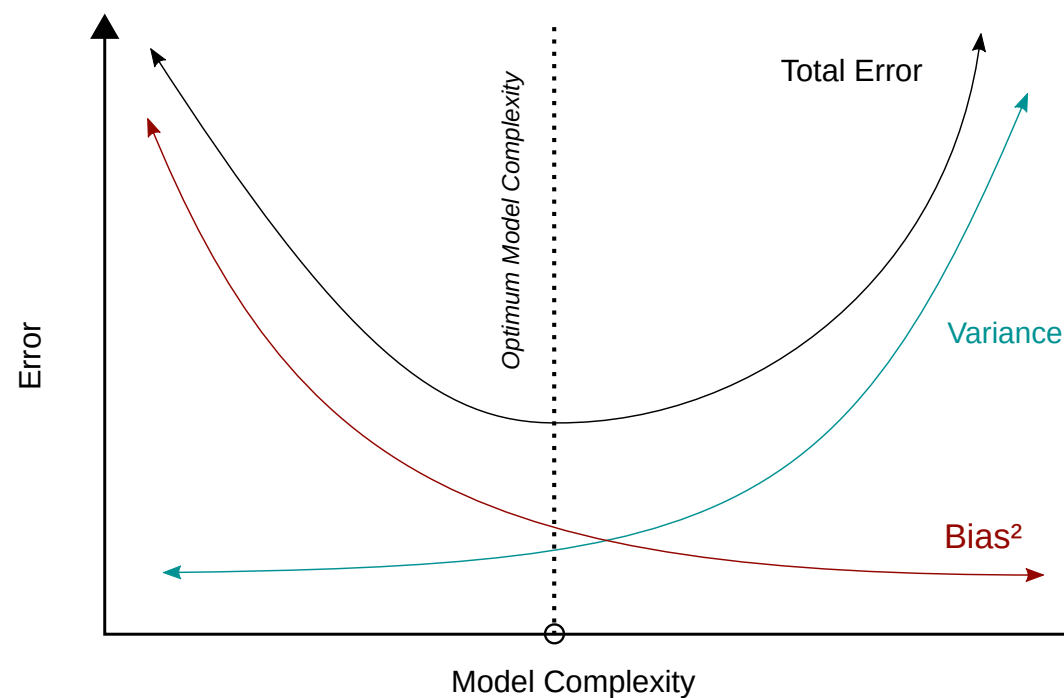
- **Sesgo** (*bias*): Error por suposiciones incorrectas en el modelo
- **Varianza** (*variance*): Error por la sensibilidad del modelo a variaciones en los datos



**Figura 7.** Ilustración del compromiso >bias-variance. Fuente: [Data Science Central](#)

# Problemas de ajuste (II)

**Sesgo y varianza** son dos fuentes de error que afectan a los modelos de ML



**Figura 8.** Sesgo y varianza en función de la complejidad del modelo. Fuente: [Wikipedia](#)



# Problemas de ajuste (III)

---

No posible minimizar **el sesgo y la varianza a la vez**

## Sesgo alto

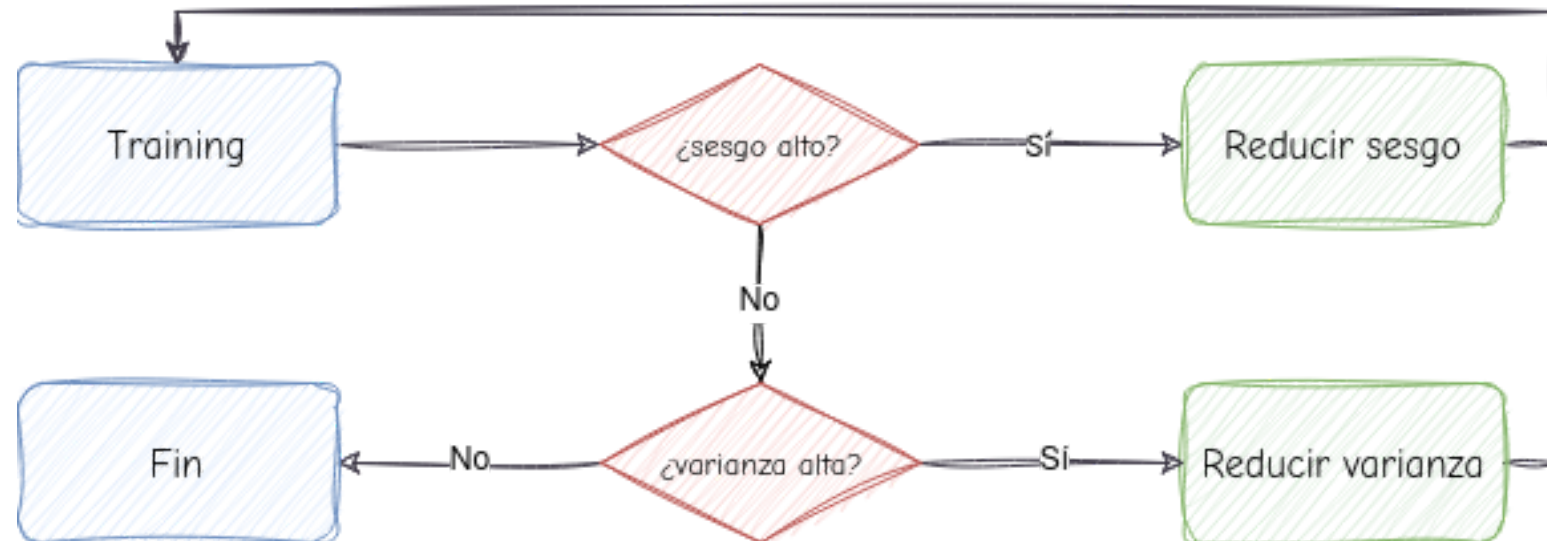
- Subajuste (*underfitting*)
- **Sobresimplificación** del problema
- *Losses* demasiado altos
- No captura la tendencia de los datos

## Varianza alta

- Sobreajuste (*overfitting*)
- **Sobrecomplicación** del problema
- *Dataset* demasiado ruidos
- Demasiada complejidad

# Compromiso sesgo-varianza en el entrenamiento (I)

El objetivo del entrenamiento es encontrar un **equilibrio entre sesgo y varianza**



**Figura 8.** Proceso de entrenamiento de un modelo teniendo en cuenta el compromiso sesgo-varianza.

# Compromiso sesgo-varianza en el entrenamiento (II)

---

## ¿Cómo reducimos sesgo?

- Más entrenamiento
- Cambiar de arquitectura
- Aumentar complejidad del modelo
  - Añadir neuronas
  - Añadir capas
- ...

## ¿Cómo reducimos varianza?

- Aumentar el *dataset*
- Cambiar de arquitectura
- Regularización
  - *Dropout*
  - *L1, L2, ...*
- Menos entrenamiento
- ...

# Técnicas de regularización

# Regularización

---

Es el proceso por el cual se **evita el sobreajuste** de un modelo

- Regularización  $\mathcal{L}1$  y  $\mathcal{L}2$
- Decaimiento de pesos
- *Dropout*
- *Batch normalization*
- *Data augmentation*
- *Early stopping*

# Regularización $\mathcal{L}1$

---

La idea de estas técnicas es reducir el valor de los parámetros para que sean pequeños

- Introduce una **penalización** a la **función de corte**  $\mathcal{L}$ , añadiendo a su valor el valor absoluto de los parámetros ( $\omega$ )

$$\mathcal{L}1(X, \omega) = L(X, \omega) + \lambda \sum |\omega|$$

- $\lambda$  es un parámetro que controla la fuerza de la regularización

$\mathcal{L}1$  «empuja» el valor de los parámetros hacia valores muy pequeños pequeños

- Hay que tener cuidado que podemos anular valores de entrada a la red.
- Se puede ver como una suerte de selección de variables automática

# Regularización $\mathcal{L}2$

---

Es una técnica muy parecida a la anterior donde los parámetros usan el cuadrado de los parámetros en lugar del valor absoluto

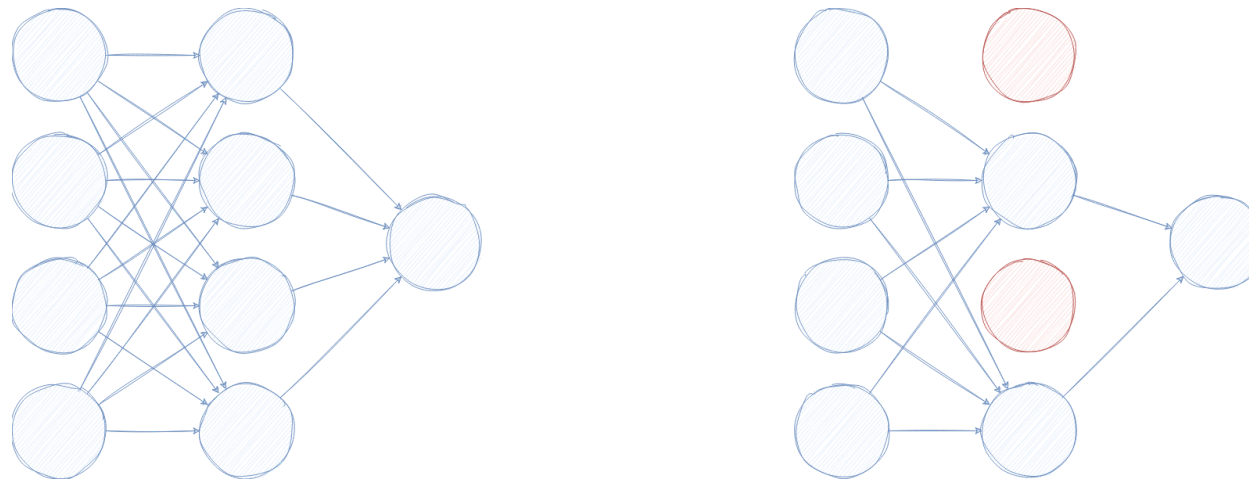
$$\mathcal{L}2(X, \omega) = L(X, \omega) + \lambda \sum \omega_i^2$$

- Mediante el parámetro  $\lambda$  podemos ajustar la regularización

El resultado de  $\mathcal{L}1$  y  $\mathcal{L}2$  es una mejor generalización (hasta cierto punto)

# Dropout

Esta técnica **desactiva neuronas** de la red **durante el entrenamiento** de forma aleatoria



**Figura 9.** Ejemplo de actuación del dropout durante un entrenamiento.

Deactivando neuronas, la red se **obliga a aprender** de forma más robusta

- Esto es, repartiendo el conocimiento de todos los ejemplos entre todas las neuronas



# Dropout

**Notebook:** [Regularización dropout.ipynb](#)

# ***Batch normalization***

---

Es una técnica que **normaliza** las salidas de las neuronas añadiendo una capa extra entre las neuronas y la función de activación

- Ocasiona que el rango de la entrada escale fácilmente hasta el rango de salida, lo que ayudará 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.

**Cuidado:** No siempre beneficia a nuestra red, hay estudios que describen una mayor tendencia a la aparición de problemas de desvanecimiento del gradiente

# ***Data augmentation***

---

Consiste en aumentar el tamaño del dataset de entrenamiento de forma sintética

- En realidad no es un algoritmo de regularización, sino una estrategia
- Al aumentar el tamaño del conjunto de datos, se reduce el riesgo de sobreajuste

Los métodos más comunes para dataset de imágenes son:

- Voltar la imagen en horizontal / vertical
- Rotar la imagen X grados.
- Recortar, añadir relleno, redimensionar, ...
- Introducir ruido, defectos, ...

En la actualidad disponemos de modelos de DL capaces de generar datos sintéticos más fieles a la realidad

# Problemas del gradiente

# ¿Qué pasa con el gradiente?

---

El gradiente es la herramienta principal para ajustar los pesos de la red

$$W_{t+1} = W_t - \alpha \frac{\partial L}{\partial W_t}$$

1. Al entrenar, usamos algoritmos de descenso del gradiente para minimizar el error
2. En  $n$  capas, el gradiente se propaga hacia atrás, ajustando los pesos de la red
3. Dependiendo del valor de estos gradientes, esta acumulación puede hacer los valores se descontrolen:
  - Derivadas pequeñas → El gradiente **disminuye** exponencialmente (**desvanecimiento del gradiente**)
  - Derivadas grandes → El gradiente **aumenta** exponencialmente (**explosión del gradiente**)

Afortunadamente, hay soluciones para estos problemas

# Desvanecimiento y explosión del gradiente

---

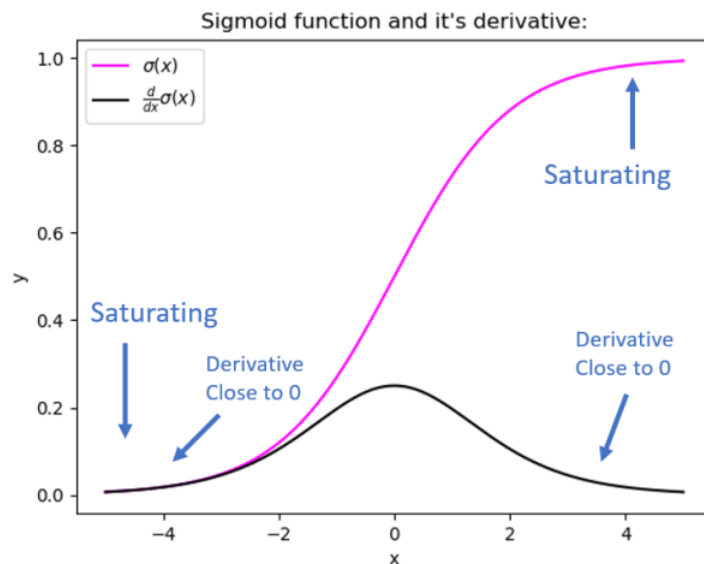
Están directamente influenciados por el número de capas que posee una red

1. **Desvanecimiento del gradiente:** Los pesos de las primeras capas no se actualizan
2. **Explosión del gradiente:** Los pesos de las primeras capas se actualizan en exceso

En la retropropagación se propagan los errores desde la salida hasta la entrada

- La acumulación de valores puede hacer que las derivadas de las funciones de activación se descontrolen

# Desvanecimiento del gradiente (*vanishing gradient*) (I)



**Figura 11.** En sus extremos la derivada de la sigmoide es casi 0. Fuente: [neptune.ai](https://neptune.ai).

Ocurre al usar funciones de activación con derivada **muy pequeña**

- Tangente hiperbólica y sigmoide

Se prefieren **ReLU** o **Leaky ReLU**:

- Son más estables y no sufren de saturación
- La derivada de ReLU y LeakyReLU es 1 si  $x > 0$
- Ambas siguen siendo no lineales

Se identifica cuando el *loss* no disminuye o lo hace muy lentamente

# Desvanecimiento del gradiente (*vanishing gradient*) (II)

---

Algunas posibles soluciones para resolver este problema son las siguientes:

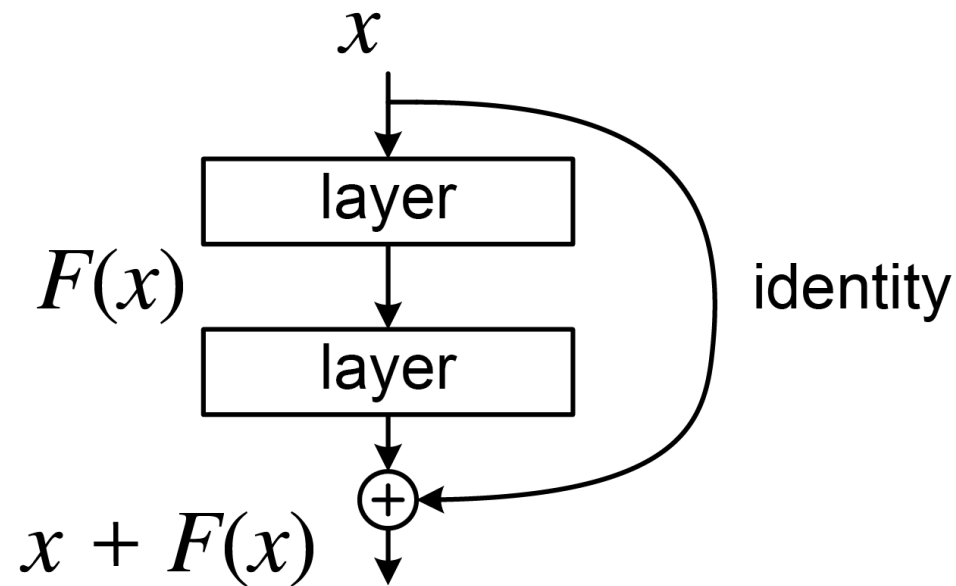
- Reducir la cantidad de capas, propagando así menos errores
- Elección cuidadosa de los pesos de inicialización de la red
- Usar funciones de activación alternativas, como ReLU o Leaky ReLU
- Uso de arquitecturas como las redes residuales<sup>1</sup>

---

<sup>1</sup> He, K., Zhang, X., Ren, S., & Sun, J. (2016). *Deep residual learning for image recognition*. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).



# Desvanecimiento del gradiente (*vanishing gradient*) (III)



**Figura 12.** Bloque residual de una red residual, donde la entrada se salta una capa para propagar el error evitando.

Fuente: [Wikipedia](#).

# Explosión del gradiente (*exploding gradient*)

---

Ocurre cuando las derivadas de las funciones de activación son **muy grandes**



Ilustración de la explosión del gradiente

***Figura 13.** Ilustración de la explosión del gradiente.*

Se identifica con valores de *loss* de NaN o muy exagerados

# Ejercicios sugeridos

---

- Comparación de rendimiento de modelos
- Clasificación de diabetes

# Licencia

Esta obra está licenciada bajo una licencia **Creative Commons  
Atribución-NoComercial-CompartirIgual 4.0 Internacional**.

Puedes encontrar su código en el siguiente enlace:

<https://github.com/etsisi/Aprendizaje-profundo>