

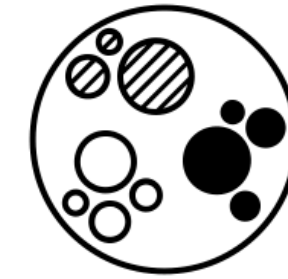
# INTRODUCCIÓN AL DL



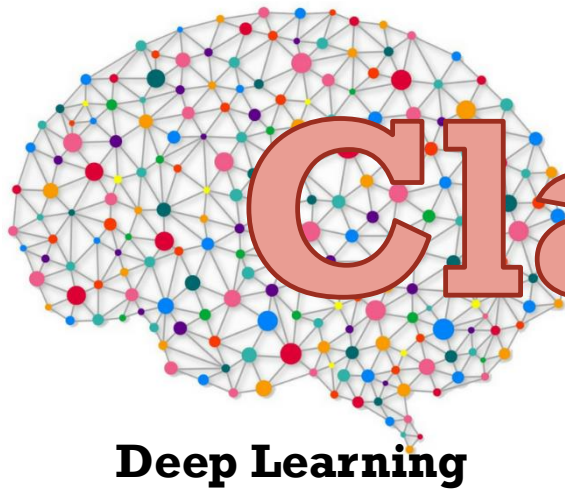
Javier Diaz, PhD



# AGENDA



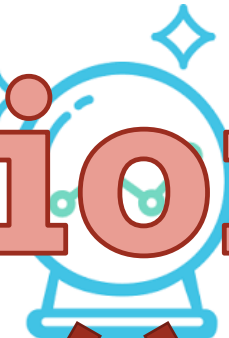
**Aprendizaje  
no supervisado**



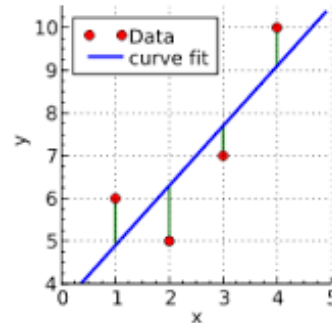
**Deep Learning**

# Clase anterior

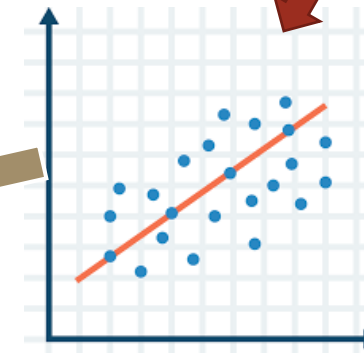
**Machine Learning  
(Aprendizaje  
Automático)**



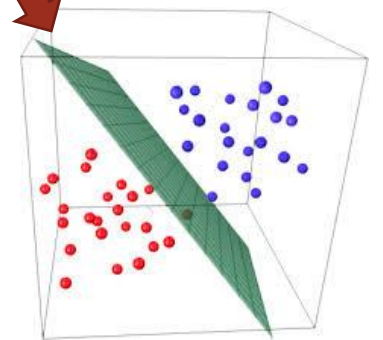
**Aprendizaje  
supervisado**



**Mínimos  
cuadrados  
ordinarios**



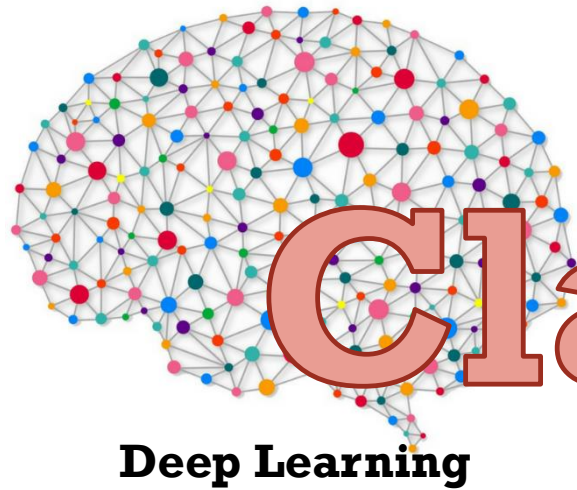
**Regresión**



**Clasificación**



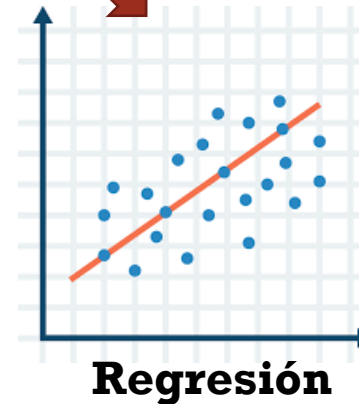
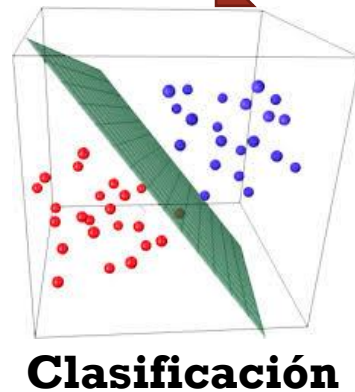
# AGENDA



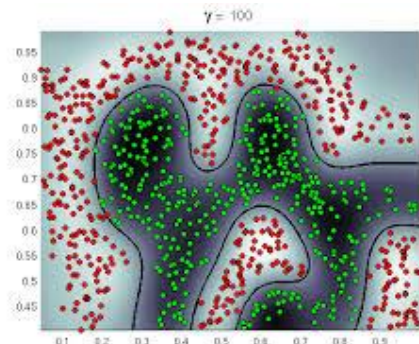
# Clase anterior

Aprendizaje  
supervisado

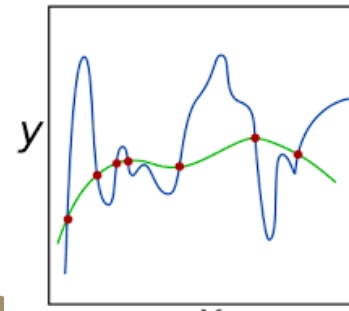
Descenso de  
gradiente



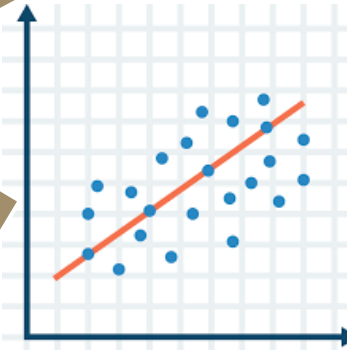
# AGENDA



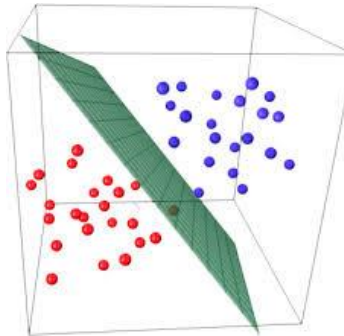
**Aprendizaje  
supervisado**



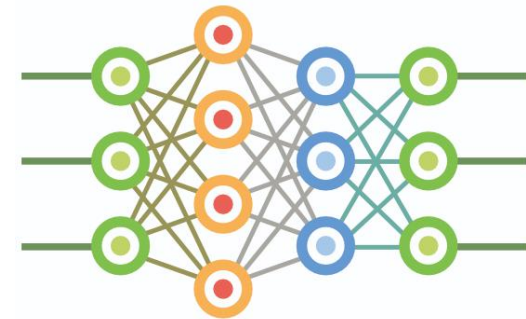
**Regularización**



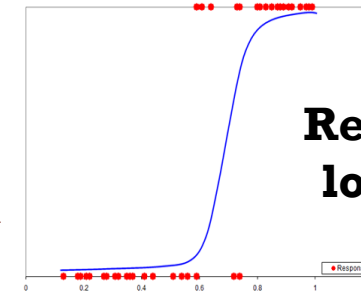
**Regresión**



**Clasificación**



**Red Neuronal  
Artificial (ANN)**

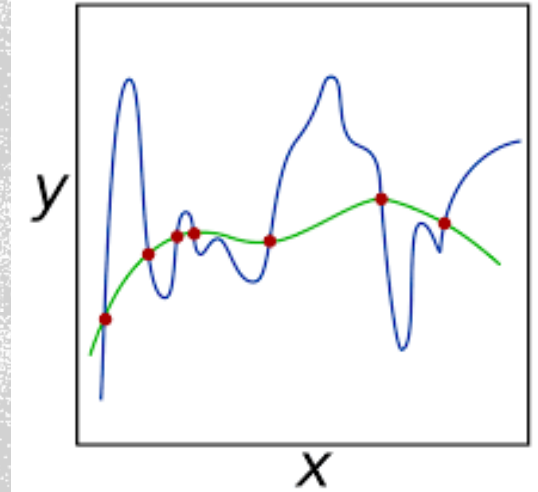


**Regresión  
logística**

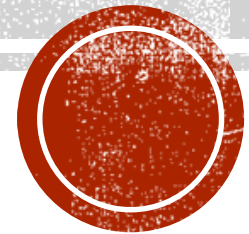




# REGULARIZACIÓN: RIDGE Y LASSO



Aplicación a la regresión lineal



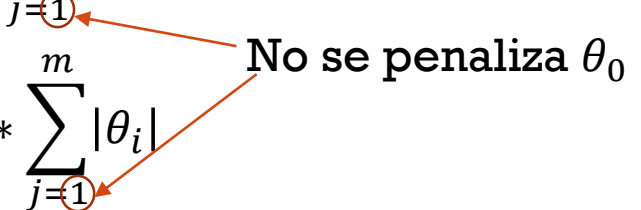
# REGULARIZACIÓN DE LA REGRESIÓN

- **Técnicas de regularización:** penalizan la magnitud de los coeficientes de las variables independientes al mismo tiempo que se trata de minimizar los errores de predicción.
- Se quiere minimizar la **complejidad** de los modelos
  - Disminuir el sobre-aprendizaje de los datos de entrenamiento de parte del modelo
  - Controlar los requerimientos computacionales de tener muchas variables independientes (big data)

- Se cambia la función de costo a minimizar utilizando normas L1 y L2:

- **Ridge (L2):** 
$$J(\Theta) = \sum_{i=1}^n (\theta_0 + \theta_1 x_1 + \dots + \theta_m x_m - y_i)^2 + \frac{\lambda}{2} * \sum_{j=1}^m \theta_j^2$$
- **Lasso (L1):** 
$$J(\Theta) = \sum_{i=1}^n (\theta_0 + \theta_1 x_1 + \dots + \theta_m x_m - y_i)^2 + \lambda * \sum_{j=1}^m |\theta_j|$$

No se penaliza  $\theta_0$





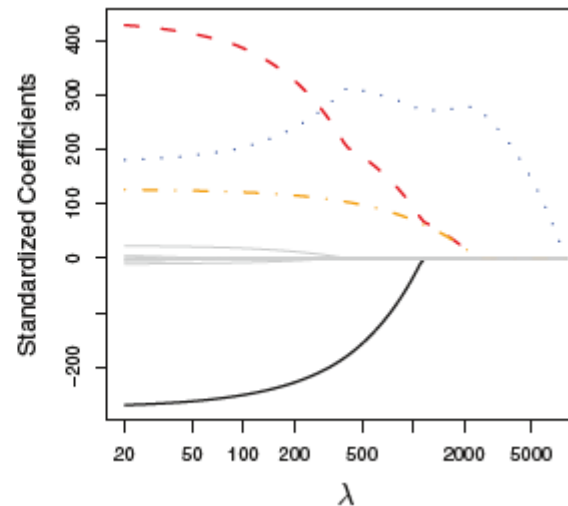
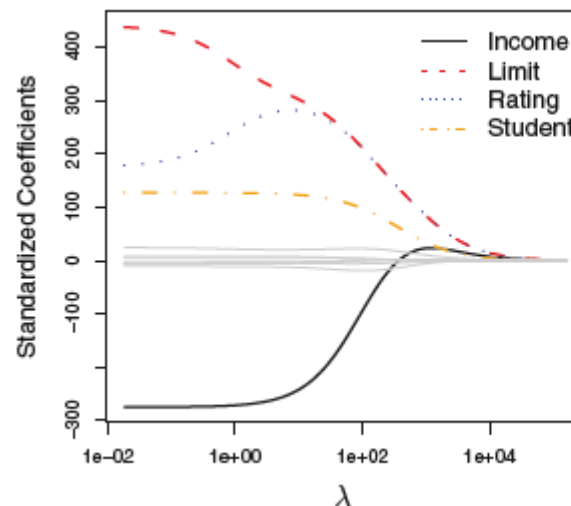
# REGULARIZACIÓN DE LA REGRESIÓN

- El parámetro  $\lambda$  sirve para controlar el impacto relativo de los dos términos en la función de costo (el que minimiza el error, y el que limita la complejidad)
  - Cuando  $\lambda = 0$ , la penalidad no tiene efecto
  - Entre más grande  $\lambda$ , los coeficientes obtenidos van a ser más pequeños
  - Seleccionar un valor de  $\lambda$  es crítico; se usa Cross-Validation
- Se actualiza la actualización de los coeficientes en el algoritmo de descenso de gradiente.
  - Ridge:  $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{j=1}^m \left( \theta_0 + \theta_1 x_1^{(j)} + \dots + \theta_n x_n^{(j)} - y^{(j)} \right) * x_j^{(j)} + \frac{\lambda}{m} \theta_j$
  - Lasso: [muy complicado, incluye definición de subgradiente, dado que la función de valor absoluto no es derivable]
- ElasticNet: modelo de regresión con regularización que combina las penalizaciones de Ridge y Lasso



# REGULARIZACIÓN DE LA REGRESIÓN

- Ridge controla la magnitud de los coeficientes de los atributos
  - Todos los atributos predictores están presentes en el modelo
  - El modelo resultante es difícil de interpretar si hay muchas variables predictivas
- Lasso hace desaparecer los coeficientes menos importantes
  - Método de selección de atributos, forzando los coeficientes de los atributos eliminados a 0, si  $\lambda$  es suficientemente grande





# REGULARIZACIÓN DE LA REGRESIÓN

## Consideraciones:

- Ridge:
  - Correlación de las variables independientes: funciona bien, pues aunque incluya todas las variables sus coeficientes van a distribuirse considerando sus correlaciones
  - Usada para prevenir el overfitting
- Lasso:
  - Correlación de las variables independientes:
    - Escoge arbitrariamente cualquiera de las variables altamente correlacionadas, poniendo en 0 los coeficientes de las demás.
    - Puede haber problemas en caso de términos polinomiales que puedan desaparecer al estar correlacionados entre ellos.
  - Produce modelos mas simples, robustos con respecto al overfitting, y fáciles de interpretar.
  - Usada como método de selección de atributos a considerar cuando hay miles de variables independientes → produce modelos “dispersos” (sparse)

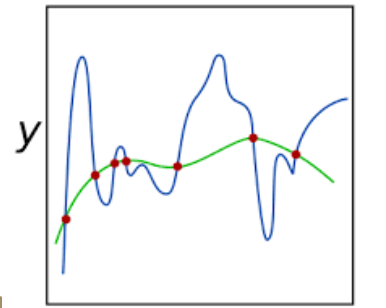


# TALLER LASSO Y RIDGE

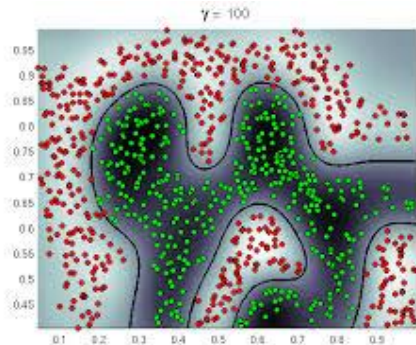
Desarrollar el taller de regresión lineal sin y con regularización (Ridge y Lasso).



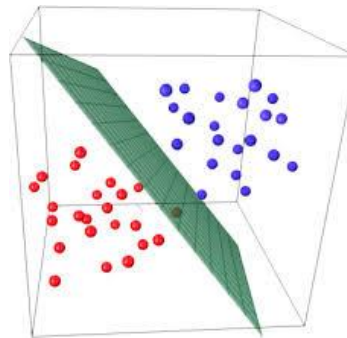
# AGENDA



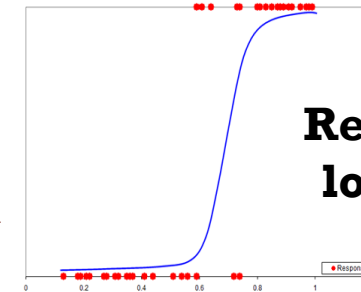
**Regularización**



**Aprendizaje  
supervisado**



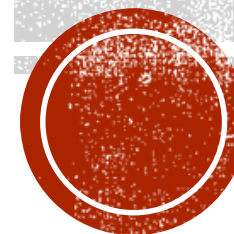
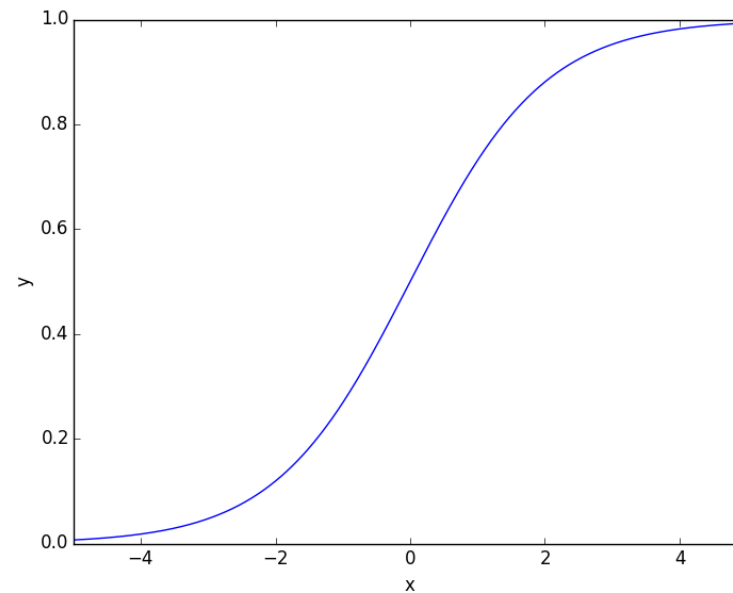
**Clasificación**



**Regresión  
logística**



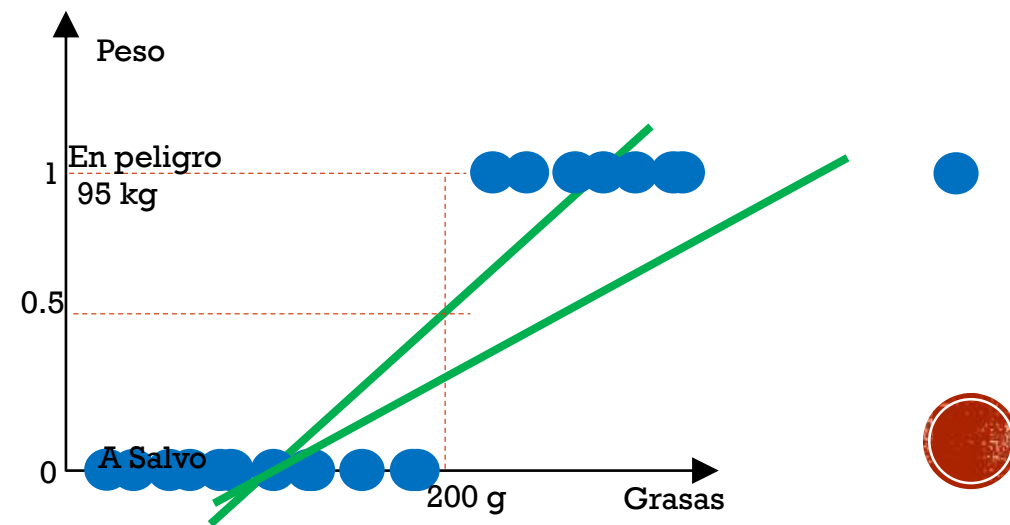
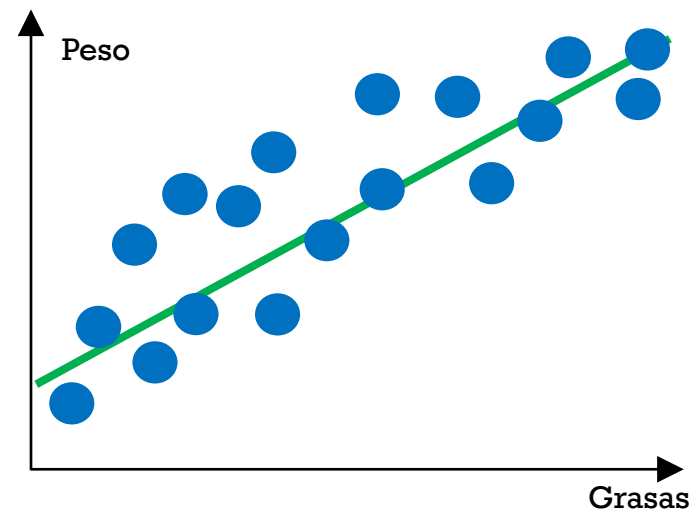
# REGRESIÓN LOGÍSTICA



# ¿REGRESIÓN LINEAL PARA CLASIFICACIÓN?

Ejemplo: tenemos los datos que relacionan la cantidad de grasas consumidas y el peso de las personas → Regresión

- Si un doctor estima que mas de 95kg implica riesgo de diabetes, el problema se convierte en uno de clasificación: 0=a salvo, 1=en peligro
- Una regresión lineal podría ayudar a estimar el límite sobre el cual se estaría en peligro de diabetes
- Pero no se puede interpretar sus predicciones como probabilidades (valores no están en  $[0;1]$ )
- Y no sería muy robusto...





# REGRESIÓN LOGÍSTICA

- Algoritmo de **clasificación**, no de **regresión**
- Se obtiene la probabilidad de cada categoría  $k$  posible para la variable objetivo, dados los valores de las variables independientes:

$$P(y^{(i)} = y_k | x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)})$$

- Parte de la idea de la regresión lineal, cuyo resultado es modificado para poder obtener una salida **binaria**: sólo permite distinguir entre 2 clases.
  - Churn vs. Stay
  - Compra vs. No compra
  - Cliente valioso vs. Cliente no valioso
- Se agrega una transformación del resultado de la regresión lineal a partir de una función de distribución acumulativa logística, también conocida como función **logit** o **sigmoide**.

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



# REGRESIÓN LOGÍSTICA

- El modelo pasa de:

$$h_{\Theta}(X) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$$

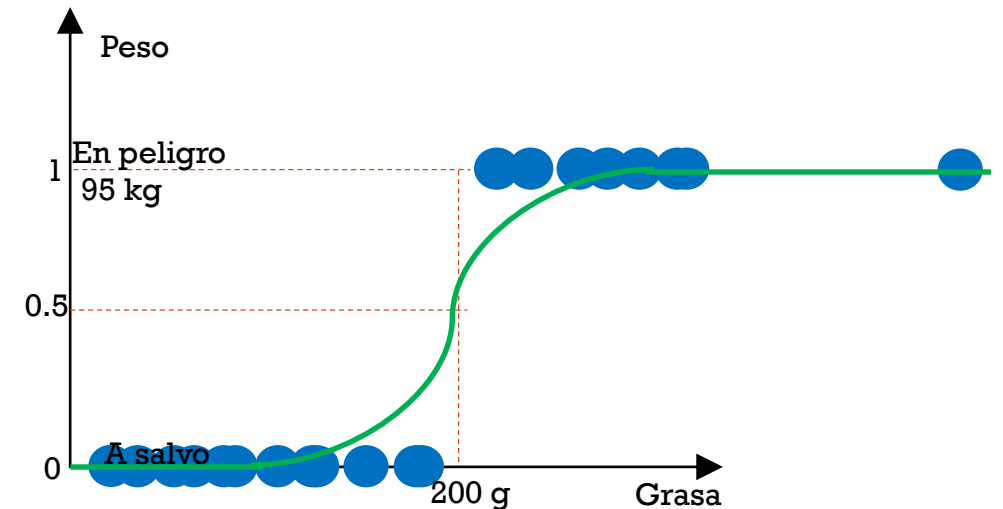
a

$$h_{\Theta}(X) = \mathbf{f(z)} = \mathbf{f}(\theta_0 + \theta_1 x_1 + \dots + \theta_n x_n),$$

con  $\max(f(z))=1$  y  $\min(f(z))=0$

- $\mathbf{f(z)}$  es la función **sigmoide** o **logística**
- Se pueden interpretar los valores de  $\mathbf{f(z)}$  como **probabilidades** de que una instancia con atributos  $\mathbf{X}$  pertenezca a la clase  $Y=1$ ,  $p_1(X)$
- $\log(\text{odds}) = \log\left(\frac{p_1(X)}{1-p_1(X)}\right) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$

$$h_{\Theta}(X) = p_1(X) = \frac{1}{1 + e^{-\Theta^T X}}$$



# REGRESIÓN LOGÍSTICA

- Comportamiento:
  - Si  $y=1$ , queremos que  $h_{\theta}(X) \approx 1$ , que  $\theta^T X \gg 0$
  - Si  $y=0$ , queremos que  $h_{\theta}(X) \approx 0$ , que  $\theta^T X \ll 0$
- Predicción: se establece un valor de umbral, por ejemplo 0.5
  - Predecir clase 1 si  $h_{\theta}(X) \geq 0.5$ , cuando  $\theta^T X \geq 0$
  - Predecir clase 0 de otra manera
- Se pueden establecer un umbral diferente si se quiere ser mas o menos robusto en la calificación
- Es necesario establecer si los  $\theta_i$  encontrados por la regresión logística son o no significativos (valores p)

$$h_{\theta}(X) = p_{1(X)} = \frac{1}{1 + e^{-\theta^T X}}$$

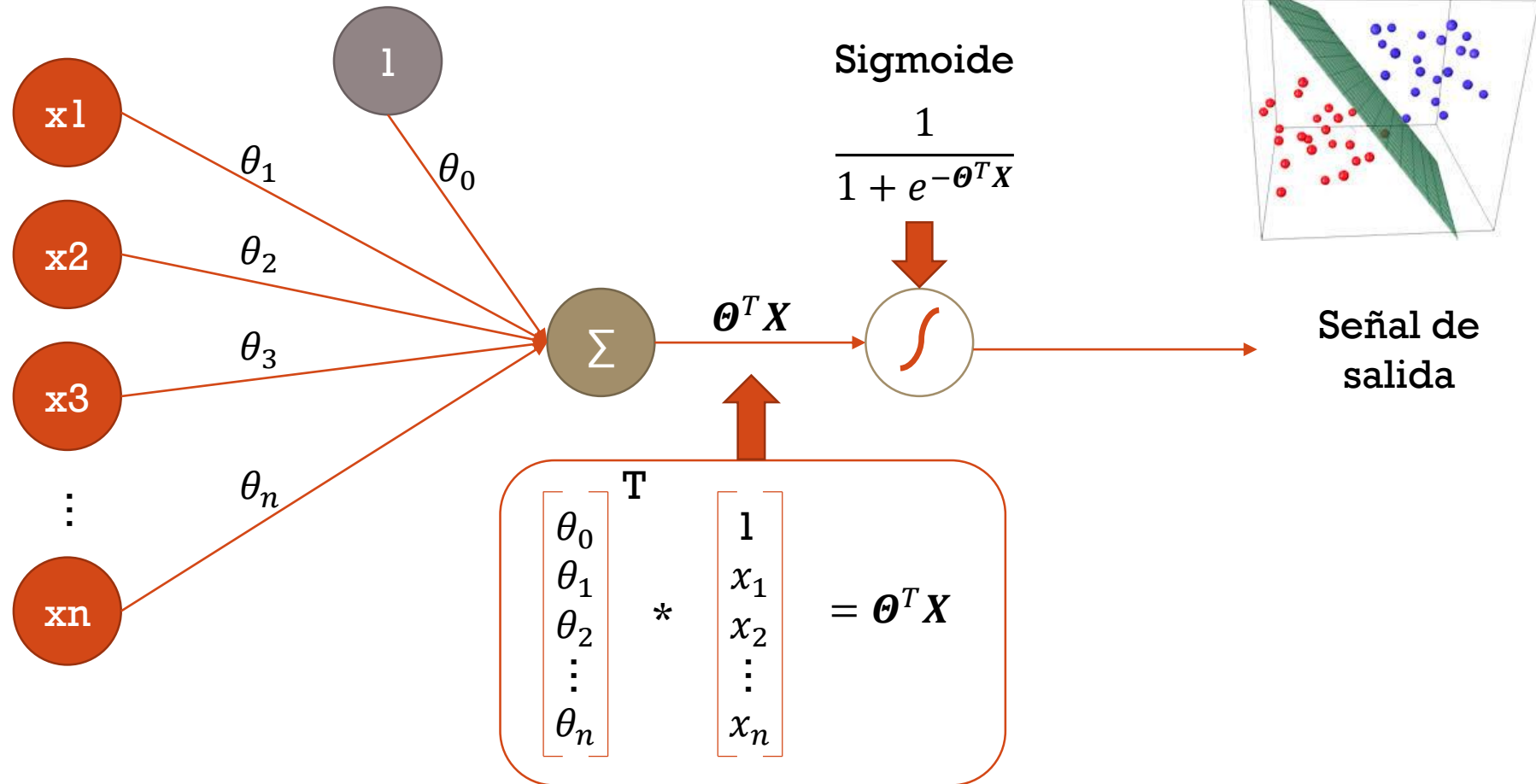
$$odds = \frac{p_{1(X)}}{1 - p_{1(X)}} = \frac{\frac{1}{1 + e^{-\theta^T X}}}{\left(1 - \frac{1}{1 + e^{-\theta^T X}}\right)}$$

$$odds = \frac{\frac{1}{1 + e^{-\theta^T X}}}{\frac{e^{-\theta^T X}}{1 + e^{-\theta^T X}}} = e^{\theta^T X}$$

$$\log(odds) = \log\left(\frac{p_{1(X)}}{1 - p_{1(X)}}\right) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$$

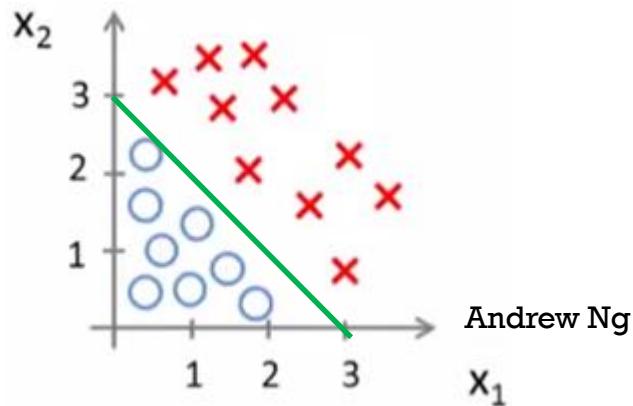


# REGRESIÓN LOGÍSTICA



# REGRESIÓN LOGÍSTICA

- El algoritmo de regresión logística determina una frontera de decisión lineal

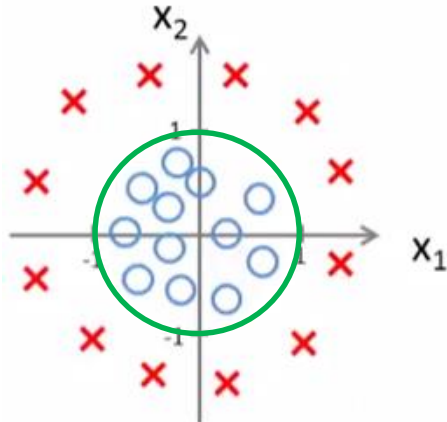


$$h_{\theta}(X) = f(-3 + x_1 + x_2)$$

Predecir la clase roja de cruz cuando:

- $h_{\theta}(X) \geq 0.5$
- $f(-3 + x_1 + x_2) \geq 0,5$

- Para fronteras de decisión no lineales: usar polinomios de un mayor orden



$$h_{\theta}(X) = f(-1 + x_1^2 + x_2^2)$$

Predecir la clase roja de cruz cuando:

- $h_{\theta}(X) \geq 0.5$
- $f(-1 + x_1^2 + x_2^2) \geq 0,5$





# DESCENSO DE GRADIENTE

- **Función de costo o de pérdida (loss) J a minimizar para la regresión logística**

- Podríamos utilizar la función de costo de la regresión lineal múltiple:

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^m (\theta_0 + \theta_1 x_1^{(i)} + \dots + \theta_n x_n^{(i)} - y^{(i)})^2$$

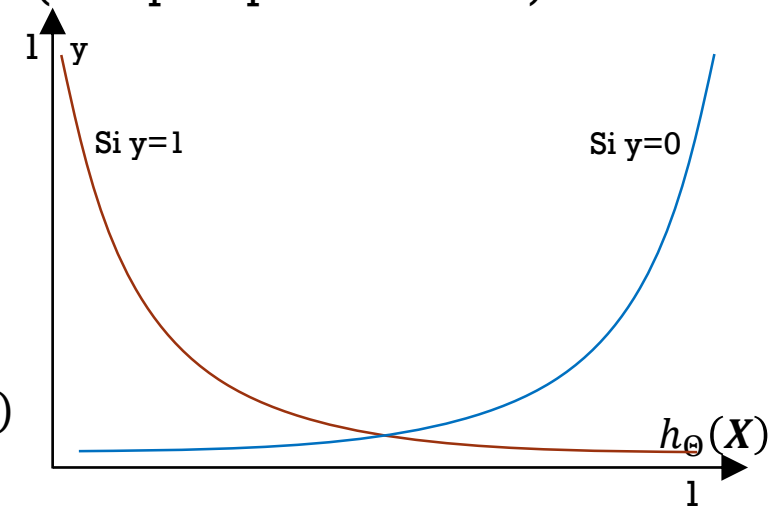
→ Pero para la regresión logística sería una función no convexa (múltiple óptimos locales)

- Se utiliza la siguiente función de costo llamada **categorical cross-entropy** para cada punto:

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{m} \sum_{i=1}^m \text{Costo}(h_{\Theta}(X^{(i)}), y^{(i)})$$

$$\text{Costo}(h_{\Theta}(X), y) = \begin{cases} -\log(h_{\Theta}(X)) & , \text{si } y = 1 \\ -\log(1 - h_{\Theta}(X)) & , \text{si } y = 0 \end{cases}$$

$$\text{Costo}(h_{\Theta}(X), y) = -y * \log(h_{\Theta}(X)) - (1 - y) \log(1 - h_{\Theta}(X))$$



# DESCENSO DE GRADIENTE

▪ Solución para la regresión lineal múltiple:

- Para la intersección:  $\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\Theta}(X) - y^{(i)})$
- Para los coeficientes:  $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\Theta}(X) - y^{(i)}) * x_j^{(i)}$

→ **Regla de actualización idéntica a la de regresión lineal!**

→ **Aunque no realmente, pues la definición de  $h_{\Theta}(X)$  ha cambiado:**

$$h_{\Theta}(X) = p_1(X) = \frac{1}{1 + e^{-\Theta^T X}}$$



# REGULARIZACIÓN DE LA REGRESIÓN LOGÍSTICA

- Al igual que con la regresión lineal, se quiere minimizar la complejidad de los modelos
  - Disminuir la posibilidad de sobre-aprendizaje del modelo
  - Controlar los requerimientos computacionales de tener muchas variables independientes (big data)

- Se cambia la función de costo a minimizar

- **Ridge:**

$$J(\Theta) = \sum_{i=1}^n (h_{\Theta}(X) - y_i) + \frac{\lambda}{2} * \sum_{j=1}^m \theta_j^2$$

- **Lasso:**

$$J(\Theta) = \sum_{i=1}^n (h_{\Theta}(X) - y_i) + \lambda * \sum_{j=1}^m |\theta_j|$$

No se penaliza  $\theta_0$

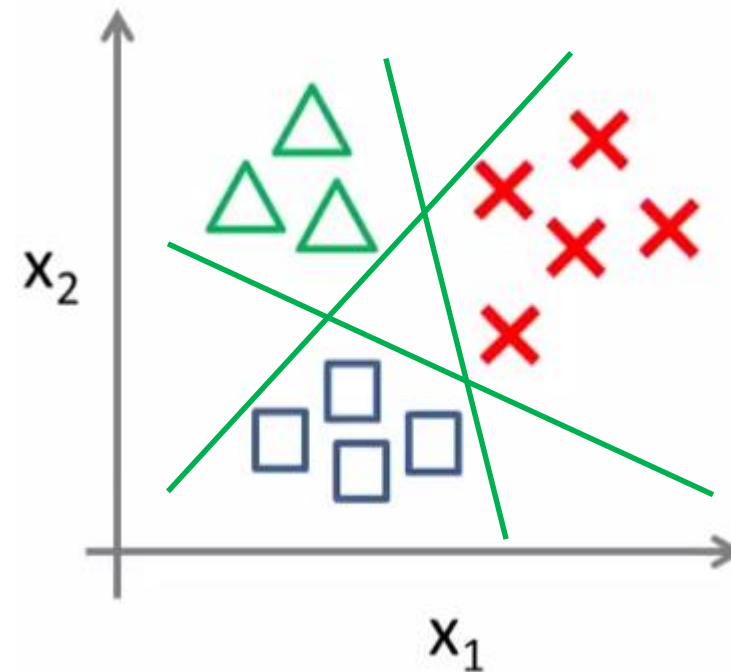
- Con respecto a la regresión lineal, no cambian las funciones de costo, ni las de actualización de los parámetros del algoritmo de descenso de gradiente



# REGRESIÓN LOGÍSTICA

¿Qué se puede hacer si se tienen más de 2 clases?

- Para problemas de clasificación con más de 2 clases, es necesario utilizar una aproximación de **1 vs. todos**
- Un clasificador por regresión logística es necesaria para cada clase
- Para una nueva instancia, la clase con la mayor probabilidad en su propio modelo es predicha



Andrew Ng

→ También se puede hacer regresión logística **multinomial** con la función **softmax**



# SOFTMAX

- La función Softmax permite tomar un conjunto de scores de clasificación y convertirlos en probabilidades.

$$s(y^{(i)}) = \frac{e^{y^{(i)}}}{\sum_j e^{y^{(j)}}} = \frac{e^{\theta^T X^{(i)}}}{\sum_j e^{\theta^T X^{(j)}}}$$

- Se puede crear un clasificador logístico basado en softmax y no en la función sigmoide, entrenando los parámetros de una combinación lineal de predictores siguiendo un descenso de gradiente, a partir de una función de costo
- La función sigmoide calcula una sola salida, mientras que la softmax calcula múltiples valores intermedios que son luego normalizados.

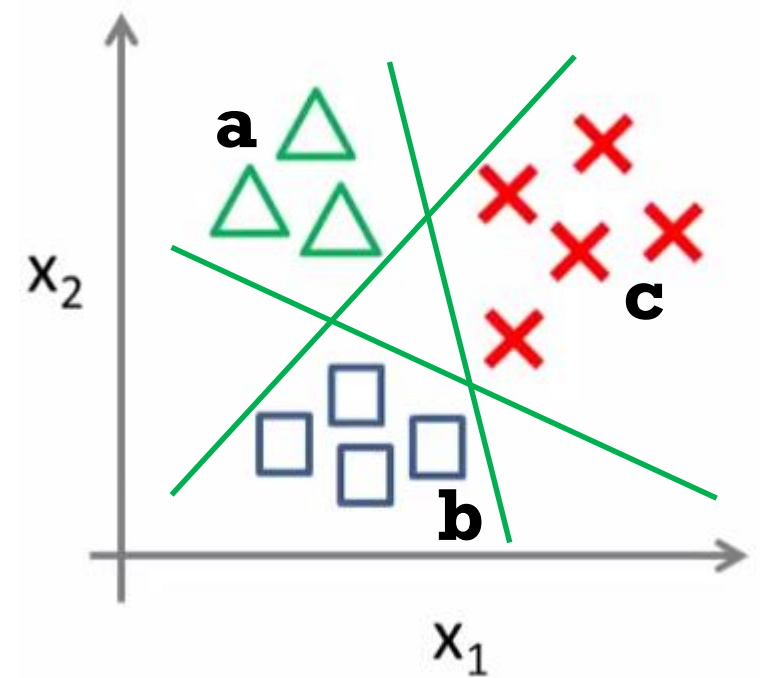
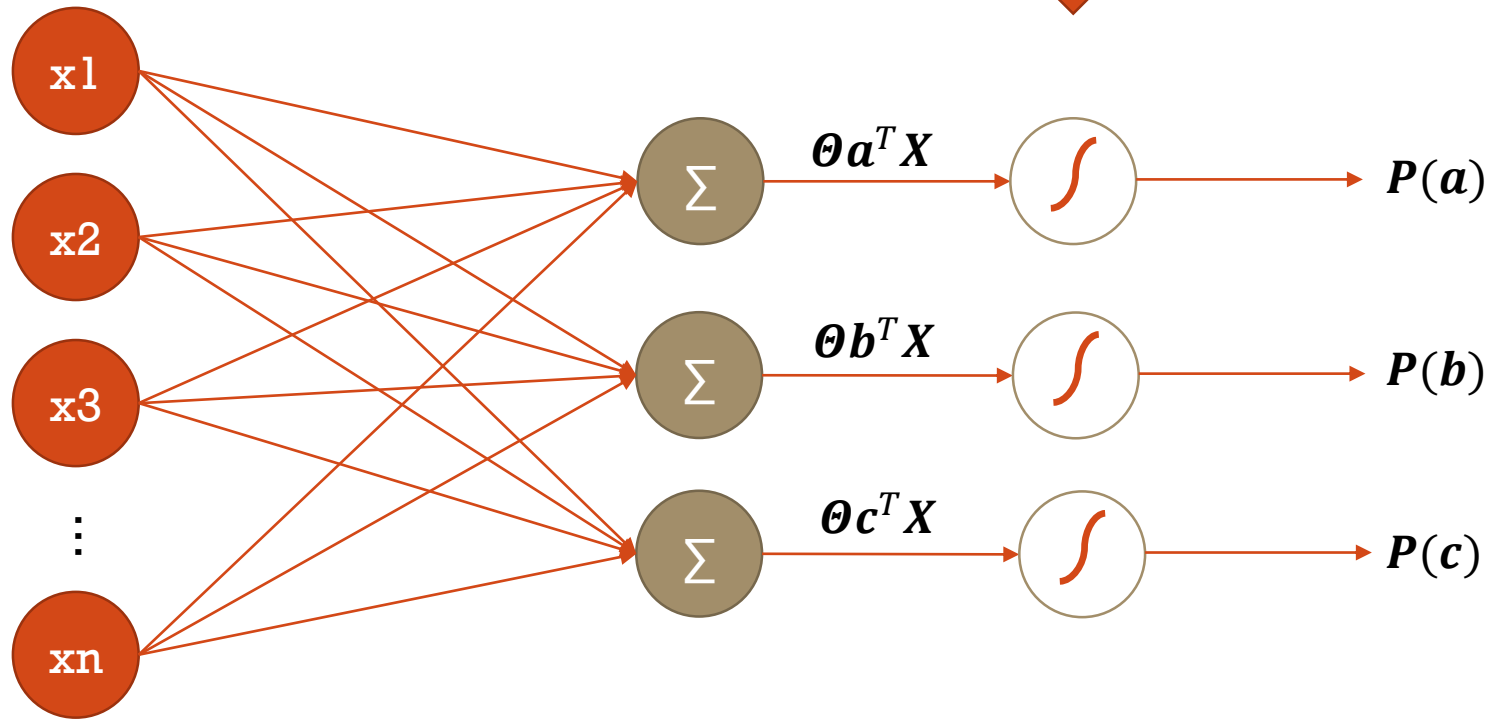




# SOFTMAX

Softmax

$$\frac{e^{\theta^T X^{(i)}}}{\sum_j e^{\theta^T X^{(j)}}}$$



# SOFTMAX

- Nótese en los siguientes ejemplos el efecto no proporcional de las diferencias entre los resultados dadas las magnitudes de los puntajes de entrada a la transformación del Softmax

<code>softmax([3.0, 1.0, 0.2])</code>	<code>=</code>	<code>[0.8360188 0.11314284 0.05083836]</code>
<code>softmax([3.0, 2.9, 2.8])</code>	<code>=</code>	<code>[0.3671654 0.33222499 0.30060961]</code>
<code>softmax([3.0, 0.2, 0.1])</code>	<code>=</code>	<code>[0.89619123 0.05449744 0.04931133]</code>
<code>softmax([3.0, 0.02, 0.01])</code>	<code>=</code>	<code>[0.908199 0.04613 0.045671]</code>
<code>softmax([3.0, 0.0002, 0.0001])</code>	<code>=</code>	<code>[0.90943064 0.04528694 0.04528241]</code>
<code>softmax([3.0, 0.00000002, 0.00000001])</code>	<code>=</code>	<code>[0.909443 0.0452785 0.0452785]</code>
<code>softmax([30.0, 0.00000002, 0.00000001])</code>	<code>=</code>	<code>[1.00000000e+00 9.35762316e-14 9.35762306e-14]</code>



# SOFTMAX

- Se trata de una generalización de la función sigmoide para mas de dos clases (la función sigmoide es un caso particular de la softmax)
  - Multinomial logistic regression

$$\text{Sigmoide}(x) = \frac{1}{1 + e^{-x}} = \frac{1}{1 + \frac{1}{e^x}} = \frac{1}{\frac{e^x + 1}{e^x}} = \frac{e^x}{e^x + 1} = \frac{e^x}{e^x + e^0} = \text{Softmax}(x)$$



# SOFTMAX

- La función de costo de la regresión multinomial es la generalización de la de la regresión logística binaria (entropía cruzada multinomial) :

$$L_{Cross-entropy}(\hat{y}, y) = - \sum_{i=0}^{k \text{ clases}} y_i \log(\hat{y}_i)$$

- Para un conjunto de registros  $m$ , se saca el promedio de los valores
- En cuanto a la derivada parcial a utilizar para la actualización de los parámetros, también es una generalización de la de sigmoide



# REGRESIÓN LOGÍSTICA

- Consideraciones

- No hay parámetros a afinar, solo las variables independientes a considerar.
- Se puede utilizar descenso de gradiente para encontrar los parámetros (mismas ecuaciones de actualización de parámetros que para regresión lineal, cambiando la función de predicción)
- Se puede utilizar los mismos métodos de regularización que con la regresión lineal (ridge, lasso)



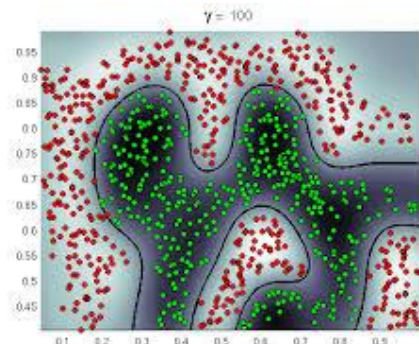
# TALLER REGRESIÓN LOGÍSTICA

Desarrollar el taller de regresión logística sobre el dataset “credit default”.

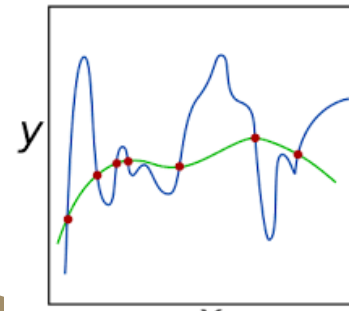




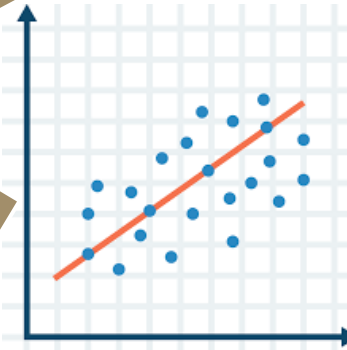
# AGENDA



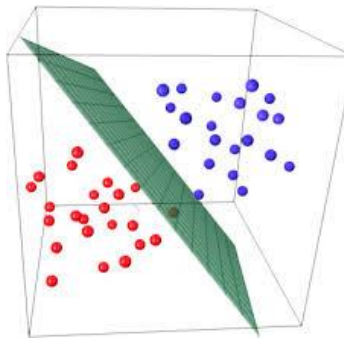
**Aprendizaje  
supervisado**



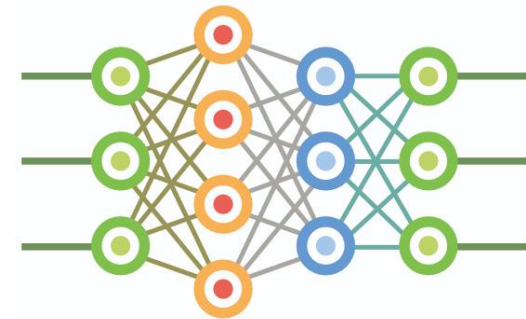
**Regularización**



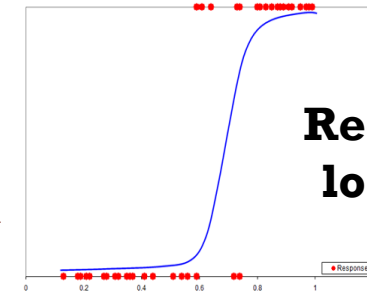
**Regresión**



**Clasificación**



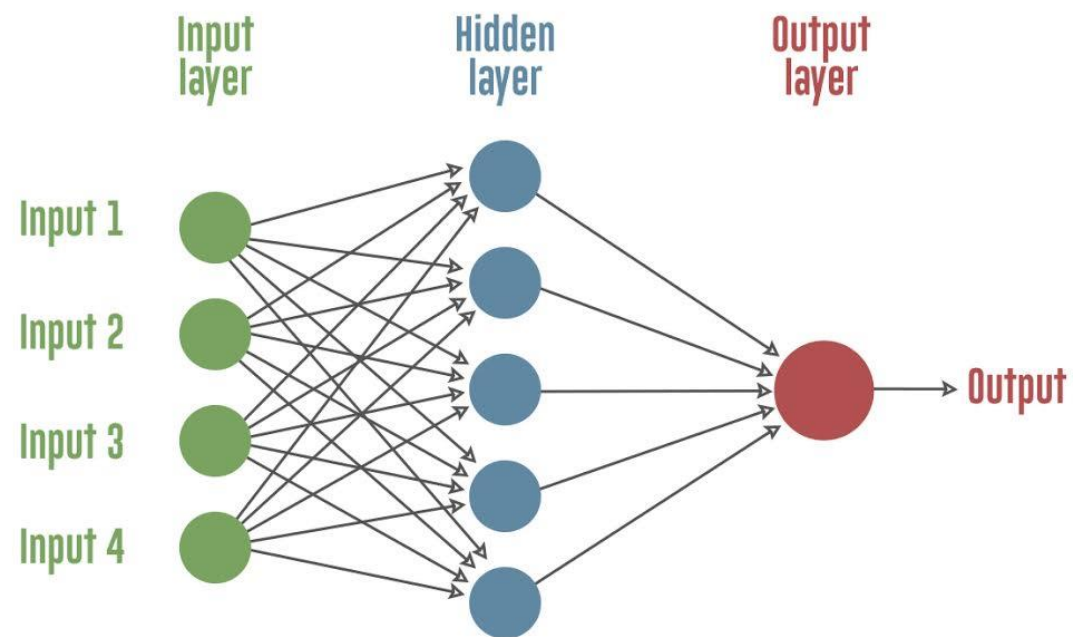
**Red Neuronal  
Artificial (ANN)**



**Regresión  
logística**

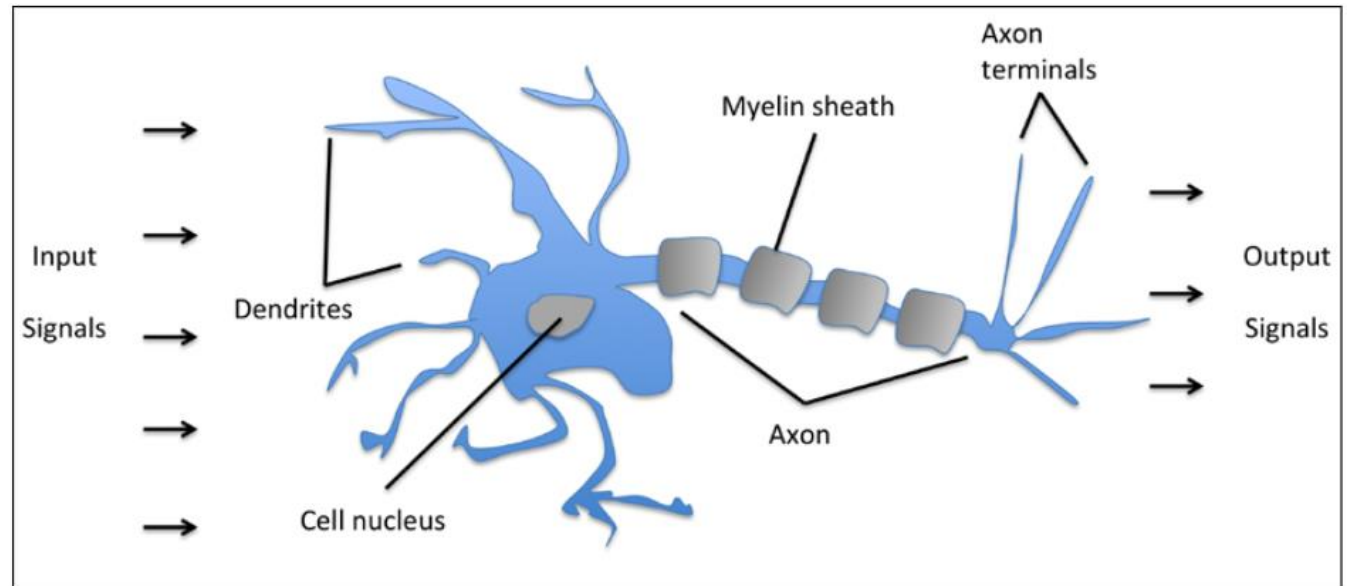


# REDES NEURONALES ARTIFICIALES



# REDES NEURONALES

- Modelos de aprendizaje automático **bio-inspirados**: tratan de modelar cómo funciona el cerebro → 1943 (McCulloch & Pitts), transmisión de señales eléctricas y químicas
- Simplificación. Una neurona:
  - recibe **múltiples** señales de entrada,
  - que se **acumulan** en el cuerpo de la neurona
  - emiten una señal binaria que evalúa el sobrepaso de un **umbral**
  - Se conectan a otras neuronas a partir de sinapsis entre axones y dendritas

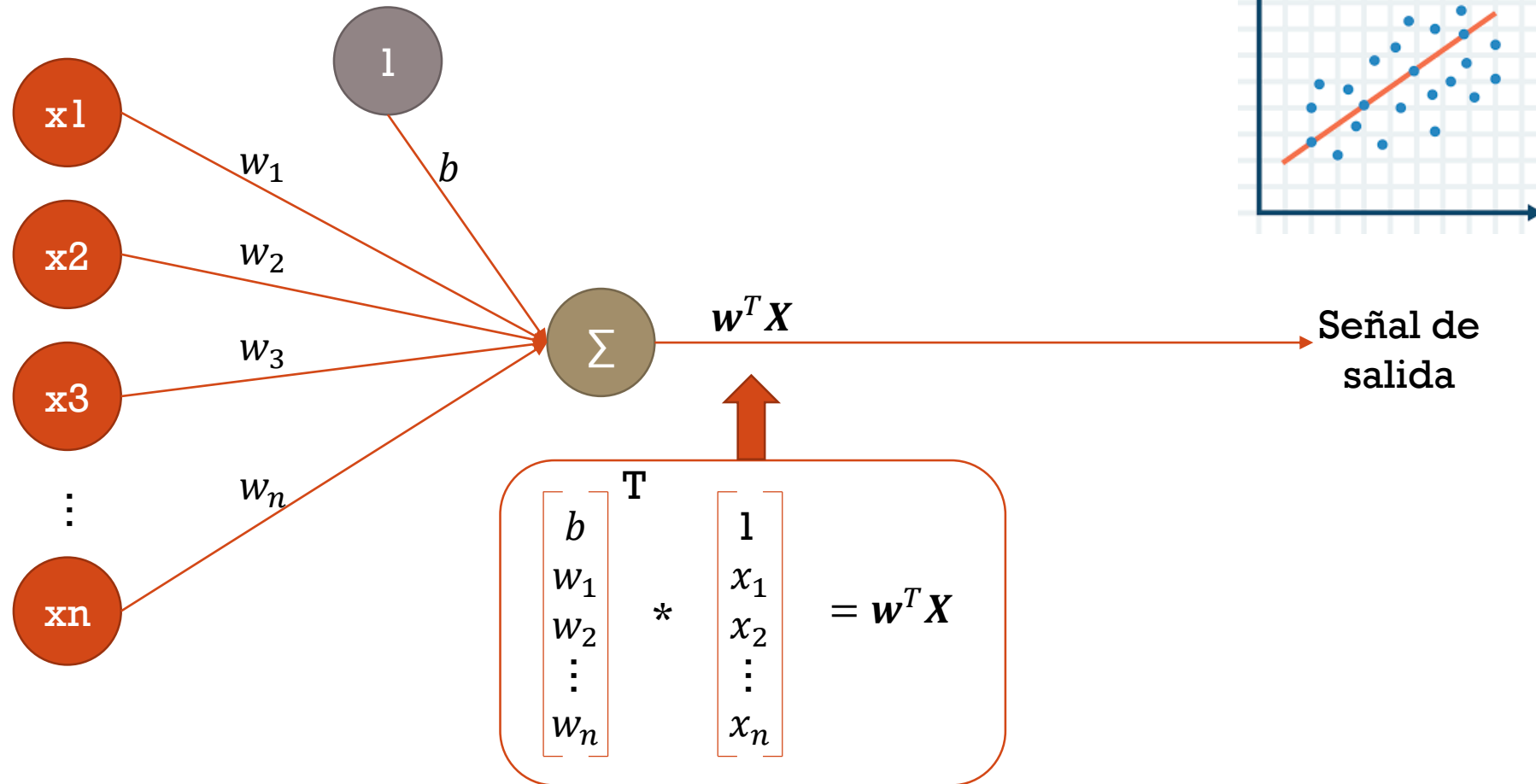


Python Machine Learning, 2015

**¿cómo se parece esto a una regresión?**



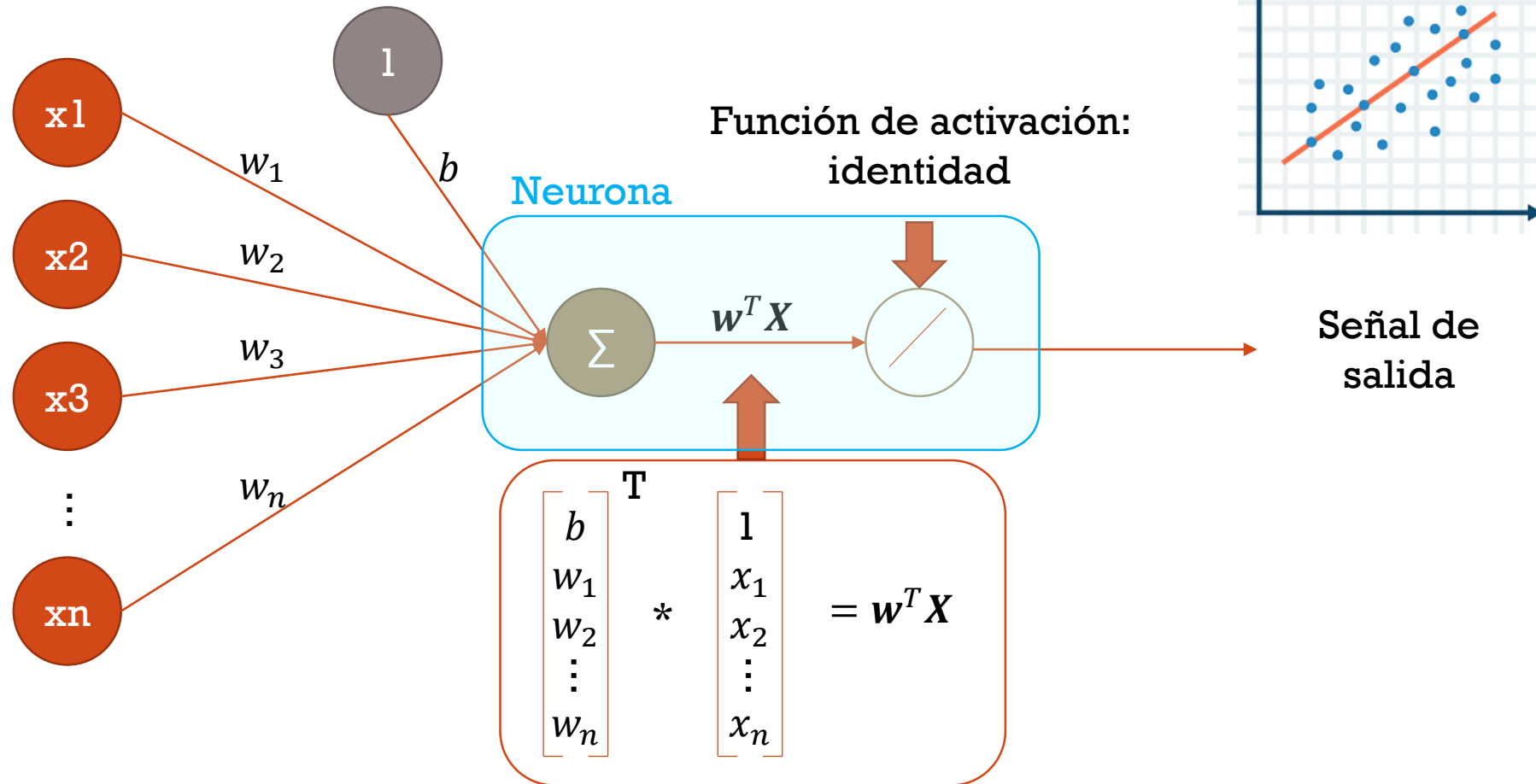
# REGRESIÓN LINEAL



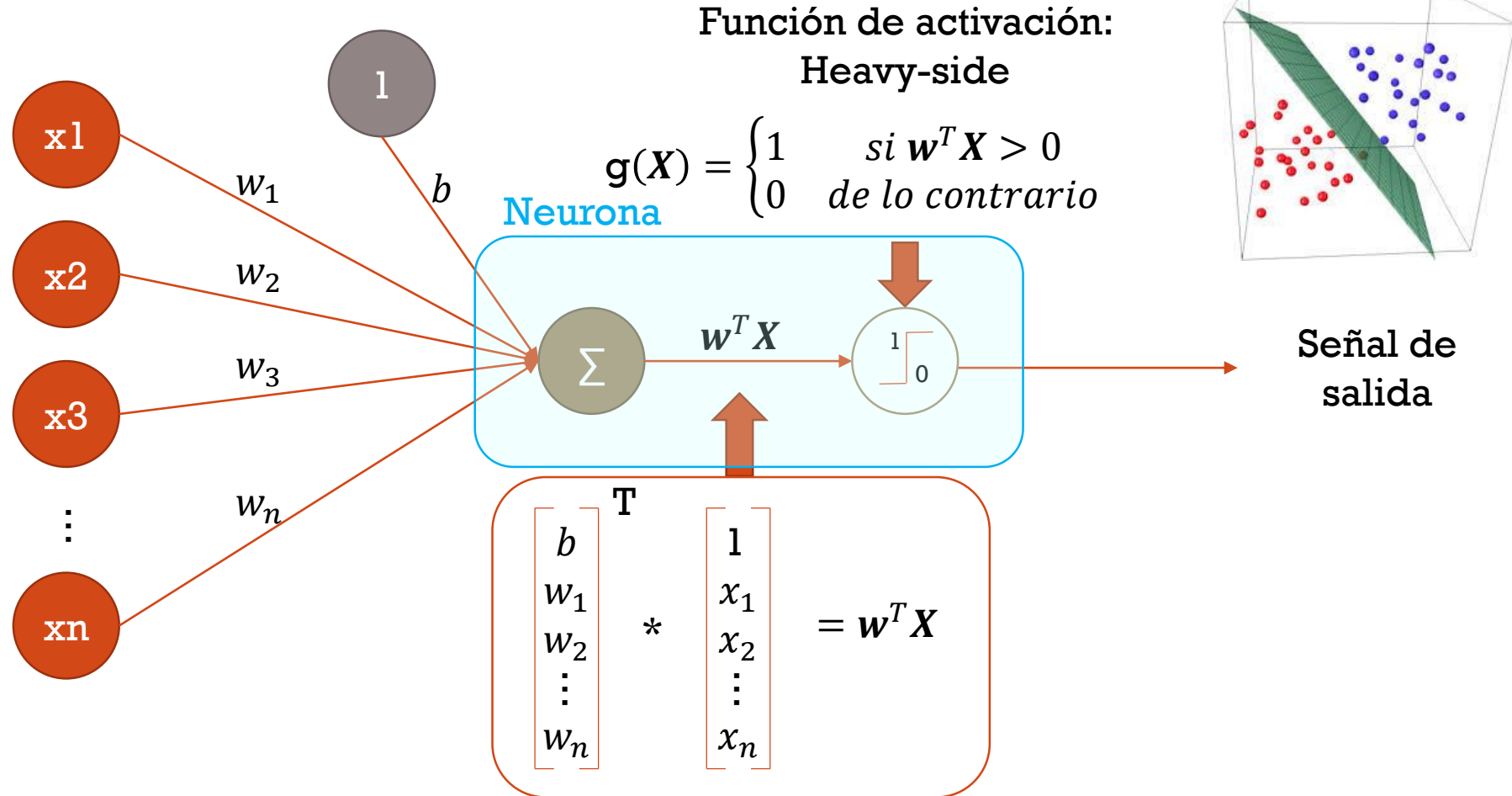
Utilizaremos  $w_i = \theta_i, i > 0$ , para referirnos a los coeficientes asociados al  $i$ -ésimo input, y  $b = \theta_0$  para el sesgo



# REGRESIÓN LINEAL



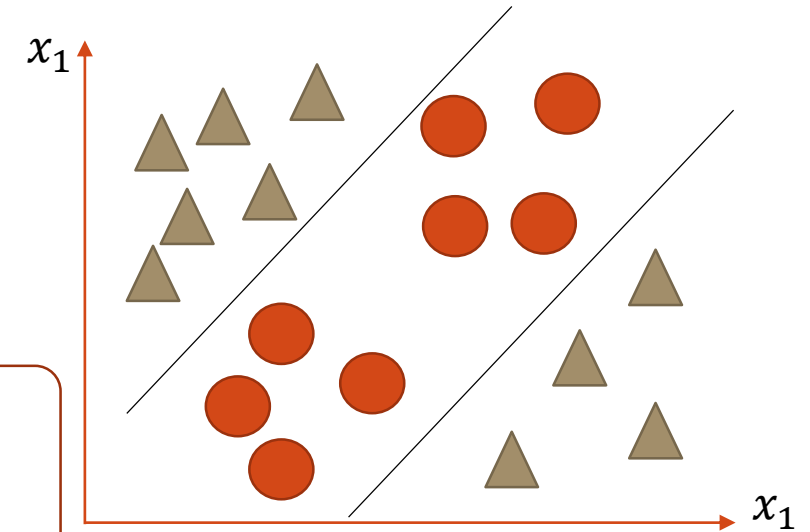
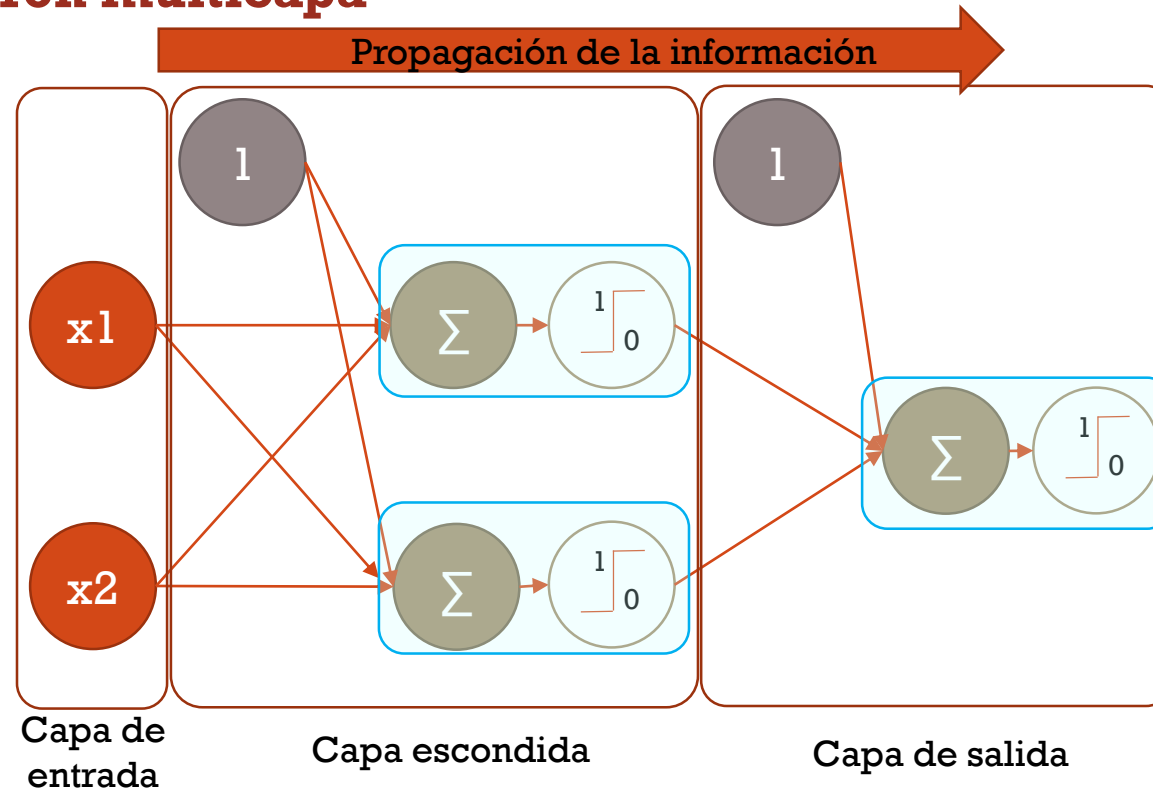
# PERCEPTRÓN, 1957 (ROSENBLATT)



# PERCEPTRÓN, 1957

- Imposibilidad de tratar casos no linealmente separables, e.g. XOR, Minsky et Papert, 1969

→ **Perceptrón multicapa**

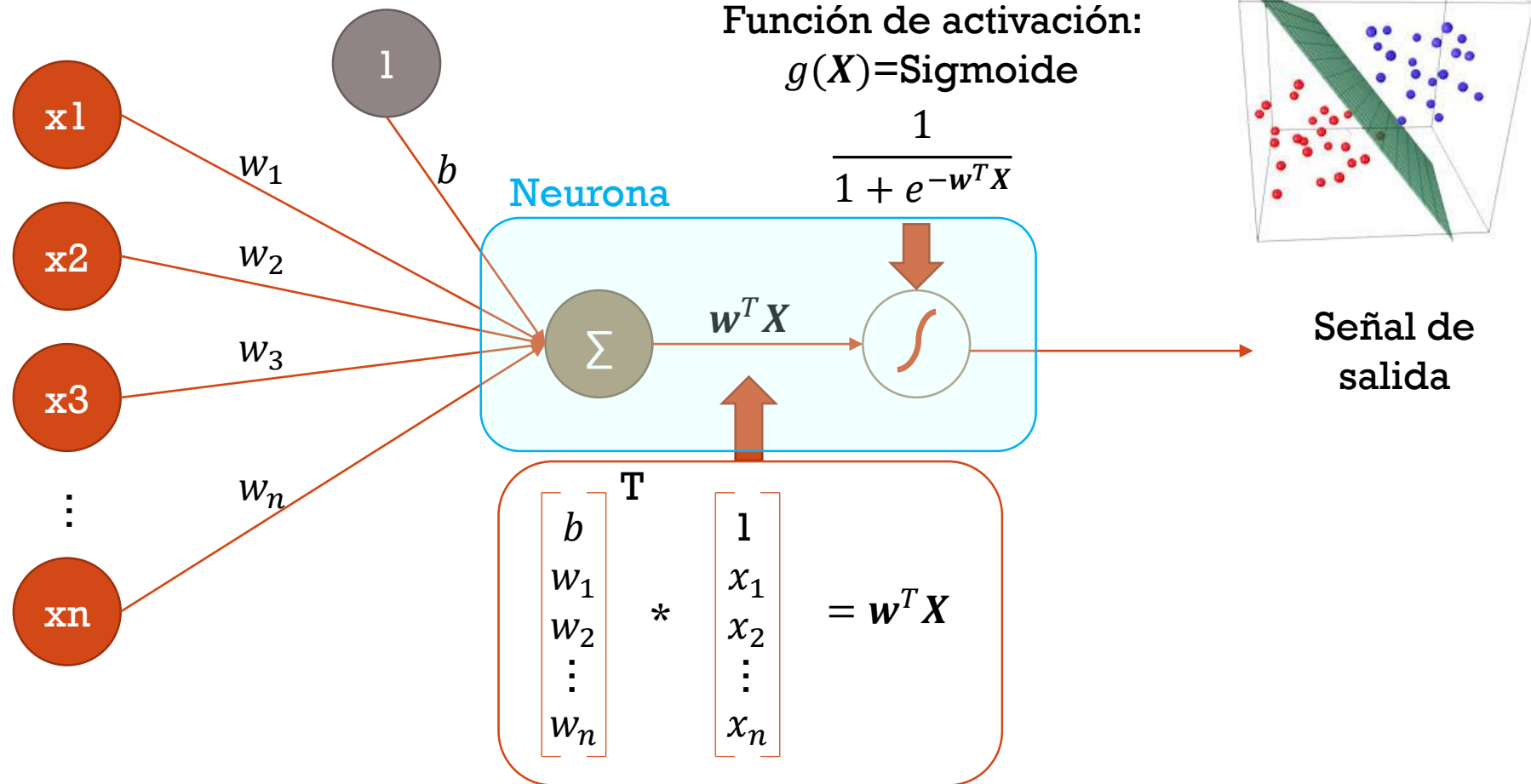


**La señal de salida es el resultado de 2 fases:**  
La agregación lineal y  
la aplicación de la  
función de activación





# REGRESIÓN LOGÍSTICA



# REDES NEURONALES

Una red neuronal se distingue por:

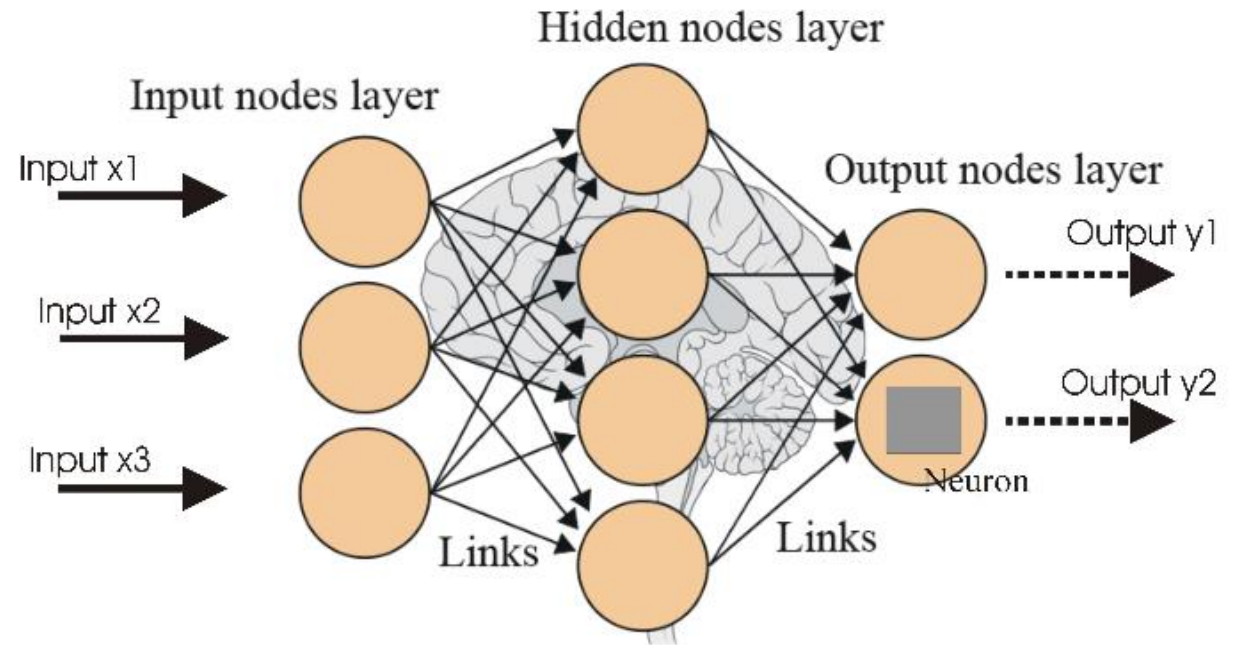
- La **topología de arquitectura de red**: describe el número de neuronas en cada una de las capas y la manera como se conectan entre ellas
- La **función de activación**: transforma la combinación de los inputs en una sola señal de salida a ser comunicada a las siguientes neuronas
- El **algoritmo de entrenamiento**: especifica como los pesos de las conexiones se establecen de tal manera que se cohíba o incite la activación de las neuronas en proporción de las señales de entrada.



# REDES NEURONALES

**Topología:** determina la complejidad de las tareas que se pueden aprender

- Número de capas y como se conectan entre ellas. Deep Learning se refiere a la profundidad de las capas.
- **Número de neuronas en cada capa:** salvo por la capa de entrada y salida, depende de la complejidad del problema y calidad de los datos. Cuidado con **overfitting**.
- Dirección del envío de la información
  - Feedforward: hacia adelante
  - Recurrent: se permite retorno (short term memory)

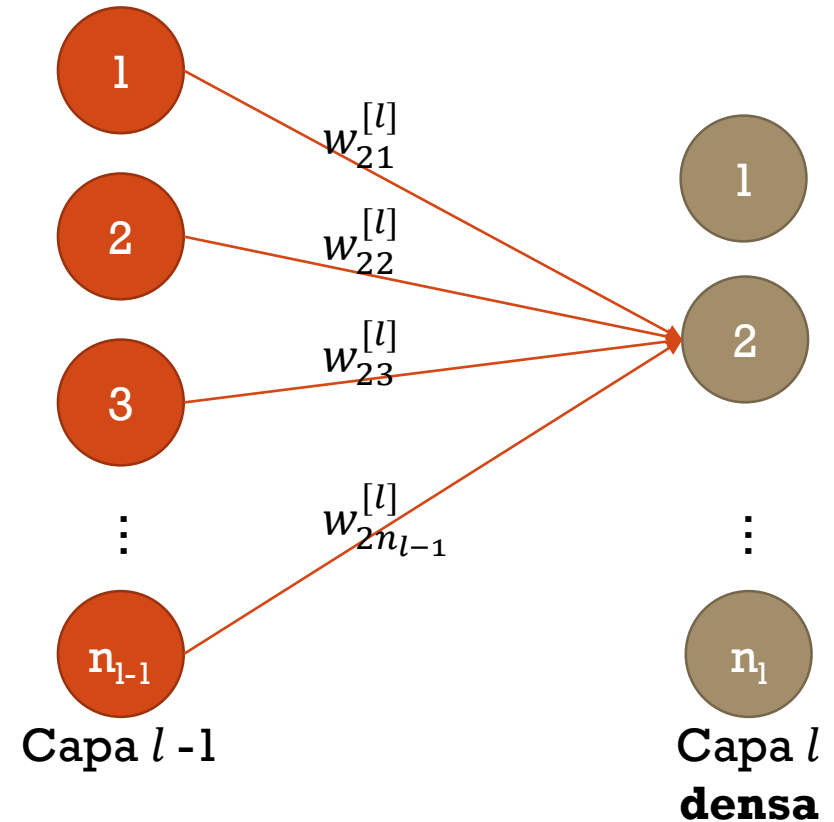


[www.analyticsvidhya.com/wp-content/uploads/2016/08/Artificial-Intelligence-Neural-Network-Nodes.jpg](http://www.analyticsvidhya.com/wp-content/uploads/2016/08/Artificial-Intelligence-Neural-Network-Nodes.jpg)



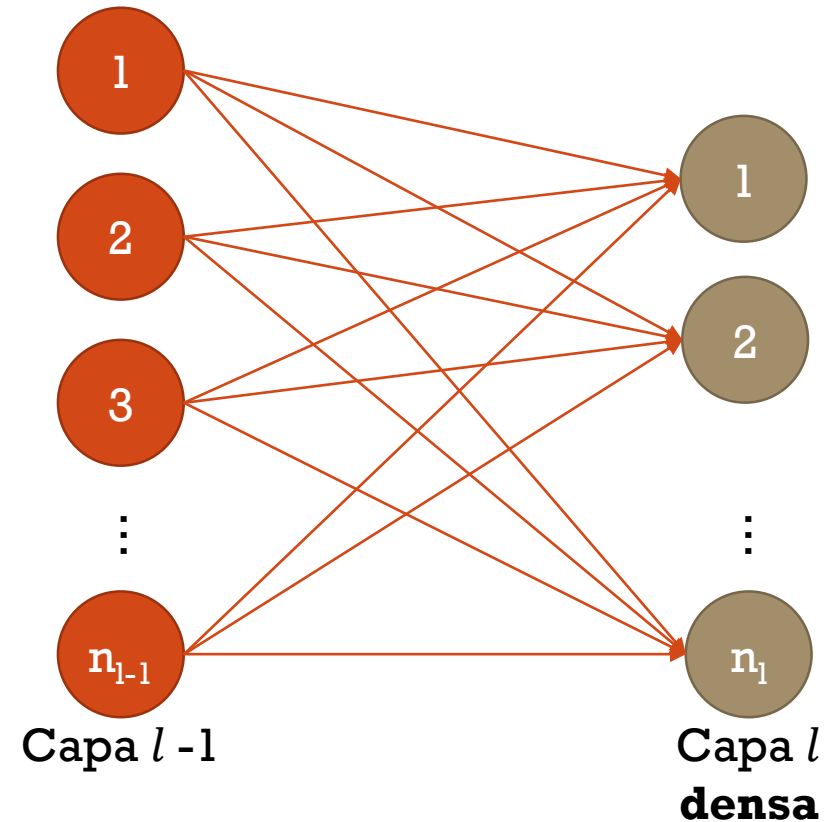
# REDES NEURONALES — ARQUITECTURAS DE CAPAS COMUNES

- En una capa **densa** o **fully connected**, cada una de sus neuronas están conectadas con todas las neuronas de la capa anterior (son las que se usan en las redes neuronales tradicionales).



# REDES NEURONALES — ARQUITECTURAS DE CAPAS COMUNES

- En una capa **densa** o **fully connected**, cada una de sus neuronas están conectadas con todas las neuronas de la capa anterior (son las que se usan en las redes neuronales tradicionales).
- Una capa **convolucional** captura relaciones espaciales de la capa anterior (se utiliza mucho con entradas imágenes).
- Una capa puede ser **recurrente**, al considerar como inputs en un siguiente paso de cálculo sus propias salidas de pasos anteriores (se utiliza en señales de audio, secuencias, lenguaje natural, etc.).



# REDES NEURONALES

Las más parecidas a la realidad biológica

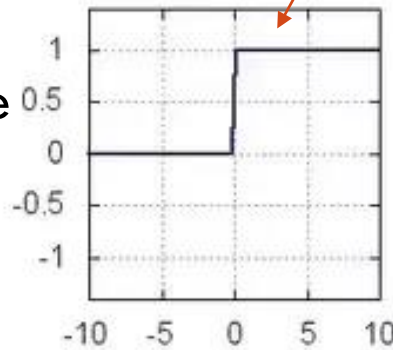
Muy usada en Deep Learning.  
Capas intermedias, rapidez

**Funciones de activación:**  
mecanismo de procesamiento de la información entrante que permite la propagación de la señal en la red buscando una no linealidad.

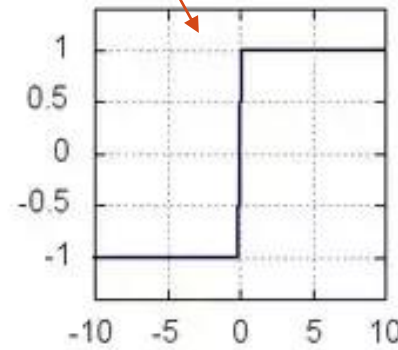
Se prefieren las que tengan buenas propiedades matemáticas (simples, derivables)

Para evitar problemas de aprendizaje, se acostumbra **normalizar las entradas**, para que los valores se encuentren en los rangos dinámicos de las funciones de activación.

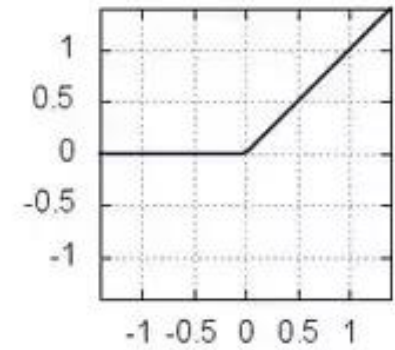
0/1 step



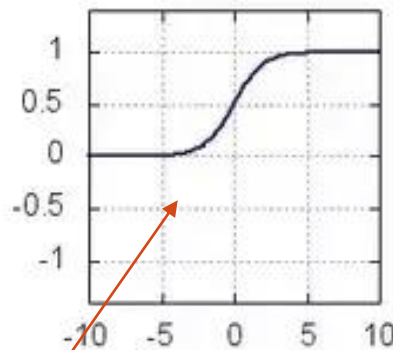
-1/+1 step



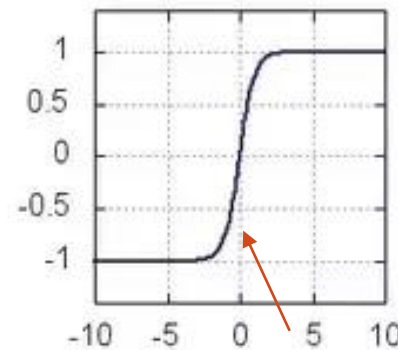
relu:  $\max(0, x)$



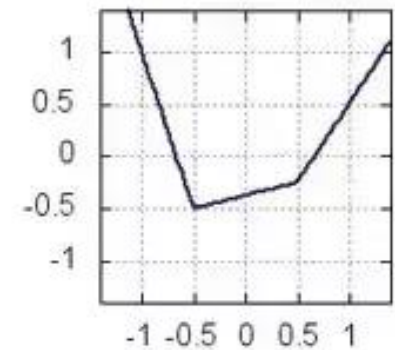
sigmoid:  $1/(1+e^{-x})$



tanh:  $(e^x - e^{-x}) / (e^x + e^{-x})$



maxout



Regresión logística,  
capa de salida binaria

Muy usada, funciona mejor  
que la sigmoide en las  
capas escondidas

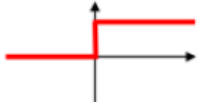
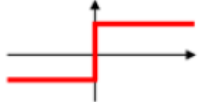

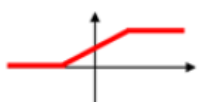

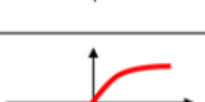


# REDES NEURONALES

## Funciones de activación:

Se busca la no linealidad en las funciones de activación de las capas intermedias, de lo contrario tener 1 capa o varias correspondería se podría reducir a una simple combinación lineal de los datos.

La función de activación **lineal** solo se utiliza en la capa de salida, en problemas de **regresión**.

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

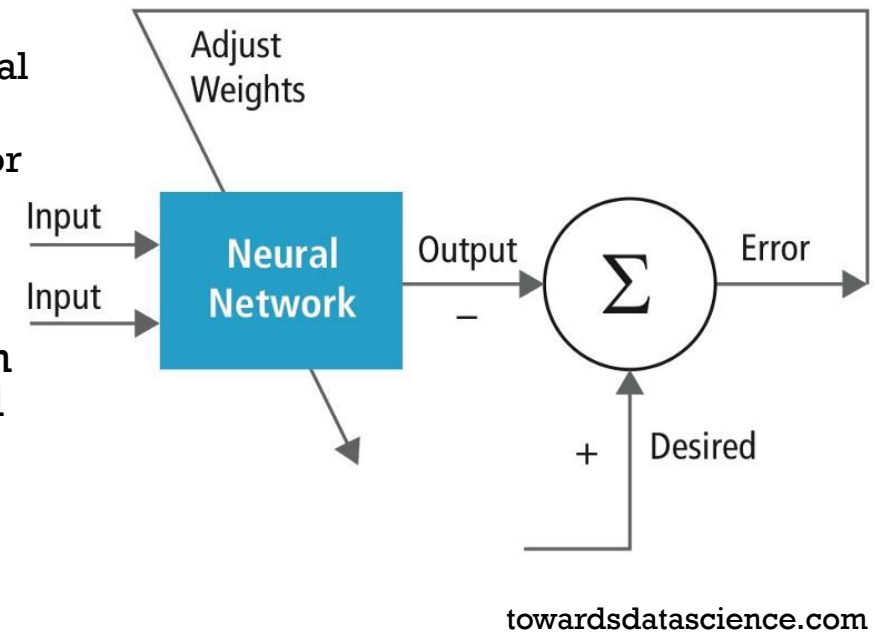




# REDES NEURONALES — APRENDIZAJE

**Back-propagation:** el algoritmo más común para entrenar una red neuronal feed-forward de múltiples capas, 1986 (Hinton).

- 2 fases
  - **Forward:** para cada ejemplo propagar la información hasta llegar al final, calcular el error de predicción.
  - **Backward:** modificar los pesos de la capa inmediatamente anterior de tal manera que se reduzcan los errores. Continuar el proceso con las capas anteriores.
- Basado en la propagación del error y el **descenso de gradiente** (derivadas parciales de las funciones de activación que van en la dirección de la reducción del error). Necesidad de que las funciones de activación sean derivables.
- Computacionalmente intensivo
- Influencia de la inicialización aleatoria de las neuronas
- Tasa de aprendizaje a establecer



# REDES NEURONALES — APRENDIZAJE

## Representación vectorial de una red neuronal

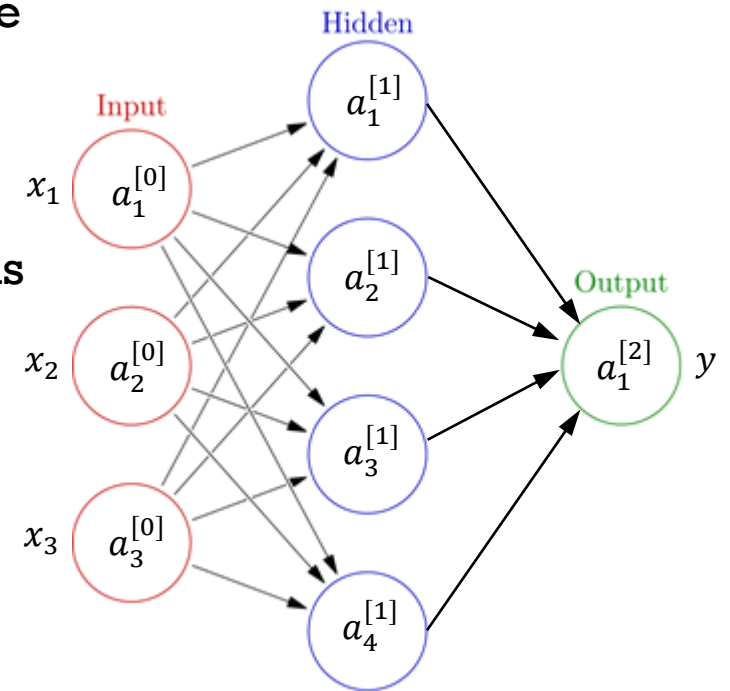
- Permite remplazar ciclos costosos en los cálculos de feed-forward y back-propagation (importante en Big Data) por multiplicaciones matriciales
- GPUs diseñadas para ejecutar operaciones matriciales
- <https://towardsdatascience.com/why-you-should-forget-for-loop-for-data-science-code-and-embrace-vectorization-696632622d5f>
- La **inicialización** de los pesos asociados a las relaciones entre las neuronas de capas subsecuentes (los  $w$ ) debe ser **aleatoria**. Si los pesos se inicializan en 0, los gradientes afectarían de la misma manera los pesos, actualizándolos de tal manera que serían siendo simétricas. Los sesgos (los  $b$ ) si se pueden inicializar en 0.
- Los pesos (los  $w$ ) deben ser “pequeños”, para garantizar que los gradientes de las funciones de activación (e.g. sigmoide) sean significativos y el aprendizaje no sea lento. La magnitud de los pesos debe ser inversamente proporcional a la profundidad de la red.



# REDES NEURONALES — APRENDIZAJE

## Representación vectorial

- $a_{[k]}^{[l]}$ , los resultados de la función de activación de la  $k$ -ésima neurona de la capa  $l$ ,  $\mathbf{a}^{[l]}$  es un array que agrupa las activaciones de todas las neuronas de la capa  $l$ . No se cuentan las activaciones de la primera capa,  $\mathbf{a}^{[0]} = \mathbf{x}$ , pues no tiene parámetros asociados.
- Análogamente a las activaciones, tenemos los pesos (matriz  $\mathbf{W}^{[l]}$ ) de las y los sesgos de las capas (array  $\mathbf{b}^{[l]}$ , no se muestra en la imagen)
  - $\mathbf{W}^{[l]}$  es una matriz de tantas filas como neuronas tiene la capa  $l$ , y con tantas columnas como neuronas de entrada tiene la capa  $l$ .
  - $\mathbf{b}^{[l]}$  es un vector de tantas filas como neuronas tiene la capa  $l$  y una columna
- Cada capa de neuronas pasa por dos fases
  - La fase de agregación:  $\mathbf{z}^{[l]} = \mathbf{W}^{[l]} * \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$
  - La función de activación:  $\mathbf{a}^{[l]} = \mathbf{g}(\mathbf{z}^{[l]})$



(Las matrices se notan en mayúscula y los vectores en minúscula)



# REDES NEURONALES — APRENDIZAJE

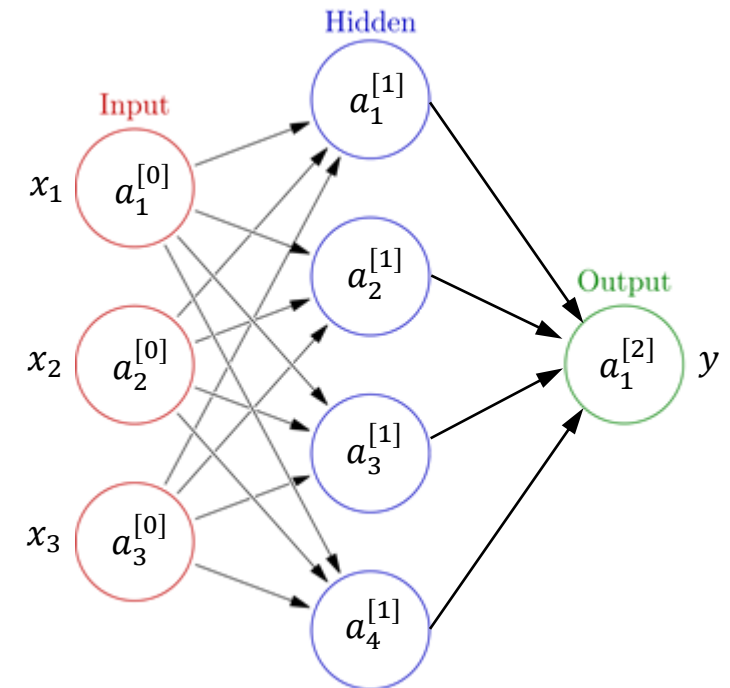
## Feed Forward

- Por ejemplo, ¿cómo sería el cálculo del resultado de la aplicación de la función de activación  $a^{[1]}$  de la capa 1 y  $a^{[2]}$  de la capa 2?

$$\mathbf{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = g \left( \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \right) = g \left( \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & w_{23}^{[1]} \\ w_{31}^{[1]} & w_{32}^{[1]} & w_{33}^{[1]} \\ w_{41}^{[1]} & w_{42}^{[1]} & w_{43}^{[1]} \end{bmatrix} \begin{bmatrix} a_1^{[0]} \\ a_2^{[0]} \\ a_3^{[0]} \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} \right)$$

$$z_1^{[1]} = w_{11}^{[1]} * a_1^{[0]} + w_{12}^{[1]} * a_2^{[0]} + w_{13}^{[1]} * a_3^{[0]} + b_1^{[1]}$$

$$y = \mathbf{a}^{[2]} = \begin{bmatrix} a_1^{[2]} \end{bmatrix} = g \left( \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & w_{13}^{[1]} & w_{14}^{[1]} \end{bmatrix} \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \end{bmatrix} \right)$$



# REDES NEURONALES — APRENDIZAJE

**Feed Forward para múltiples registros.** El mismo proceso se generaliza:

- En vez de tener un vector  $x^{(i)}$  con los valores de un registro, vamos a tener una matriz  $X$  con los vectores columnares del conjunto de instancias de aprendizaje, donde el número de columnas es  $m$  (el número de registros) y el número de filas es el número de inputs.
- La primera capa de neuronas pasa de producir un vector  $z^{[1](i)}$  para cada registro  $i$  a producir una matriz  $Z^{[1]} = W^{[1]}X + b^{[1]}$  para todos los registros.
- Igualmente pasamos de un vector de activación  $a^{[1](i)}$  para cada registro a una matriz  $A^{[1]} = g(Z^{[1]})$
- Y así sucesivamente para cada capa siguiente. Dada una capa  $[l]$  con  $n$  neuronas y  $k$  de entrada:

$$A^{[l]} = \begin{matrix} \xrightarrow{\text{instancias}} \\ \begin{bmatrix} a_1^{[l](1)} & a_1^{[l](2)} & \dots & a_1^{[l](m)} \\ a_2^{[l](1)} & a_2^{[l](2)} & \dots & a_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_n^{[l](1)} & a_n^{[l](2)} & \dots & a_n^{[l](m)} \end{bmatrix} \end{matrix} \begin{matrix} \downarrow \text{Neuronas de la capa } l \\ \end{matrix} = g(Z^{[l]}) = g \left( \begin{matrix} \downarrow \text{Neuronas de la capa } l \\ \begin{bmatrix} w_{11}^{[l]} & w_{12}^{[l]} & \dots & w_{1k}^{[l]} \\ w_{21}^{[l]} & w_{22}^{[l]} & \dots & w_{2k}^{[l]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}^{[l]} & w_{n2}^{[l]} & \dots & w_{nk}^{[l]} \end{bmatrix} \begin{matrix} \xrightarrow{\text{instancias}} \\ \begin{bmatrix} a_1^{[l-1](1)} & a_1^{[l-1](2)} & \dots & a_1^{[l-1](m)} \\ a_2^{[l-1](1)} & a_2^{[l-1](2)} & \dots & a_2^{[l-1](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_k^{[l-1](1)} & a_k^{[l-1](2)} & \dots & a_k^{[l-1](m)} \end{bmatrix} \end{matrix} + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_n^{[l]} \end{bmatrix} \right)$$



# REDES NEURONALES — APRENDIZAJE

**Función de costo.** Una vez hemos llegado a la predicción final en la capa de salida, se calcula el error de predicción de la red (costo  $J$ ), basado en los errores individuales de cada predicción (loss  $L$ , función dependiente del tipo de función de activación), para poder propagarlo a las capas anteriores:

$$J(\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \dots, \mathbf{W}^{[nCapas]}, \mathbf{b}^{[nCapas]}) = \frac{1}{m} \sum_{i=1}^m L(\hat{Y}, Y)$$



# TALLER ANN: FFWD DE XOR CON NUMPY

Para el problema de XOR con una sencilla red de una capa con 3 entradas, una capa escondida con 4 neuronas y una capa con 1 neurona de salida, utilizando funciones sigmoides:

- Desarrollar la función **feedForward(X, w1, w2, b1, b2)** que calcula el vector con las predicciones para el conjunto de registros de entrada. Retorna las activaciones de cada capa para cada registro de entrada para la capa escondida (a1) y para la capa de salida (a2=y\_estimado)
- Desarrollar la función **costoGlobal(y\_est, y)** que calcula el valor objetivo de minimización del aprendizaje, que recibe los vectores con las probabilidades calculadas por la predicción, los labels reales, y retorna el costo global





# REDES NEURONALES — APRENDIZAJE

## Back-propagation.

Para aprender los parámetros utilizamos descenso de gradiente, calculando entonces las derivadas parciales de la función de costo con respecto a cada parámetro de cada capa.

Se empieza por una inicialización aleatoria de los valores de los parámetros, que se irán actualizando a través de varias iteraciones según una tasa de aprendizaje  $\alpha$  aplicada al gradiente correspondiente:

$$\mathbf{W}^{[l]} := \mathbf{W}^{[l]} - \alpha \frac{\partial J}{\partial \mathbf{W}^{[l]}} \quad \mathbf{b}^{[l]} := \mathbf{b}^{[l]} - \alpha \frac{\partial J}{\partial \mathbf{b}^{[l]}}$$

Todo va a depender de las funciones de activación y sus derivadas parciales, veamos cuales serían los gradientes.



# REDES NEURONALES — APRENDIZAJE

**Ejemplo para neuronas logísticas.** En el caso de una función de activación sigmoide  $g(Z) = \sigma(Z)$ , el descenso de gradiente se hace a partir de la derivada  $\sigma'(Z)$

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

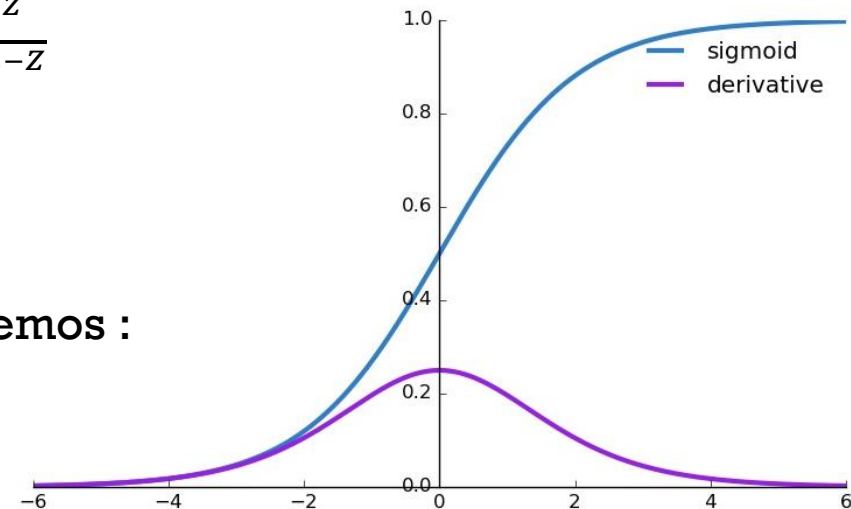
**Demostrar que  $\sigma'(Z) = \sigma(Z) \cdot (1 - \sigma(Z))$**

$$\begin{aligned}\sigma'(Z) &= -(1 + e^{-Z})^{-2}(-e^{-Z}) = \frac{e^{-Z}}{(1+e^{-Z})^2} = \frac{1}{1+e^{-Z}} \cdot \frac{e^{-Z}}{1+e^{-Z}} \\ &= \frac{1}{1+e^{-Z}} \cdot \frac{1+e^{-Z}-1}{1+e^{-Z}} = \frac{1}{1+e^{-Z}} \cdot \left(1 - \frac{1}{1+e^{-Z}}\right) \\ &= \sigma(Z) \cdot (1 - \sigma(Z))\end{aligned}$$

Para las activaciones sigmoides de una capa  $l$ , tenemos :

$$A^{[l]} = g(\mathbf{Z}^{[l]}) = \sigma(\mathbf{Z}^{[l]})$$

$$A'^{[l]} = A^{[l]} \cdot (1 - A^{[l]})$$



→ Para hacer el back-propagation, vamos a guardar las activaciones calculadas en el feed-forward

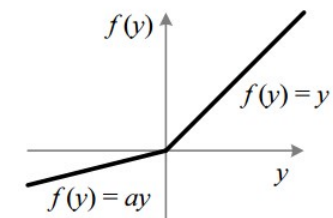
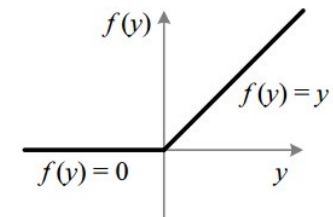
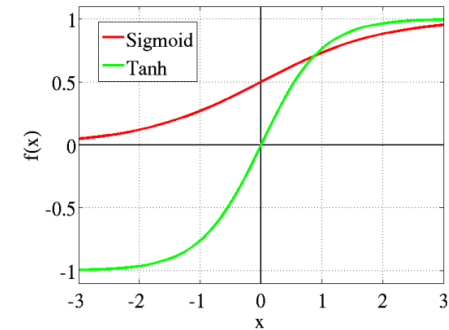


# REDES NEURONALES — APRENDIZAJE

## Gradientes para las activaciones más comunes en DL

Encontramos que para las funciones de activación más comunes, los gradientes se definen en función del cálculo de las activaciones previamente computados durante la fase de feed-forward:

- Sigmoide.  $a(Z) = \frac{1}{1+e^{-Z}}$   $a'(Z) = a(Z) \cdot (1 - a(Z))$
- Tanh.  $a(Z) = \frac{e^Z - e^{-Z}}{e^Z + e^{-Z}}$   $a'(Z) = 1 - a^2(Z)$
- ReLU.  $a(Z) = \max(0, Z)$   $a'(Z) = \begin{cases} 0, & \text{si } Z < 0 \\ 1, & \text{si } Z > 0 \\ \text{indefinido}, & \text{si } Z = 0 \end{cases}$
- Leaky ReLU.  $a(Z) = \max(0.01 * Z, Z)$   $a'(Z) = \begin{cases} 0.01, & \text{si } Z < 0 \\ 1, & \text{si } Z > 0 \\ \text{indefinido}, & \text{si } Z = 0 \end{cases}$



# REDES NEURONALES — APRENDIZAJE

**Back-propagation.** Ilustremos el proceso con un caso de clasificación binaria con una neurona sigmoide en la capa final y una sola capa escondida.

$$\begin{array}{l}
 A^{[1]} \rightarrow \\
 W^{[2]} \rightarrow \\
 b^{[2]} \rightarrow
 \end{array}
 \begin{array}{l}
 \rightarrow \\
 \rightarrow \\
 \rightarrow
 \end{array}
 Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \longrightarrow A^{[2]} = \sigma(Z^{[2]}) \longrightarrow$$

$$\begin{array}{l}
 \frac{\partial Z^{[2]}}{\partial b^{[2]}} = 1 \quad \frac{\partial Z^{[2]}}{\partial A^{[1]}} = W^{[2]} \quad \frac{\partial A^{[2]}}{\partial Z^{[2]}} = \sigma'(Z^{[2]}) = \sigma(Z^{[2]}) (1 - \sigma(Z^{[2]}))
 \end{array}$$

$$\begin{array}{l}
 L(\hat{Y}, Y) = L(A^{[2]}, Y) = -Y * \log(A^{[2]}) - (1 - Y) * \log(1 - A^{[2]}) \\
 \frac{\partial L}{\partial A^{[2]}} = -\frac{Y}{A^{[2]}} + \frac{1 - Y}{1 - A^{[2]}}
 \end{array}$$

Durante la fase feed forward se calculan los valores intermedios de  $A^{[1]}$ ,  $Z^{[2]}$  y  $A^{[2]}$ , que aplicados a las derivadas parciales y gracias a la regla de la cadena permiten encontrar los valores del gradiente de la función de pérdida utilizados para la actualización de los parámetros de la capa  $W^{[2]}$  y  $b^{[2]}$ .

$$\left. \begin{array}{l}
 \frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\
 \frac{\partial L}{\partial b^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial b^{[2]}}
 \end{array} \right\}$$



# REDES NEURONALES — APRENDIZAJE

**Back-propagation.** Ilustremos el proceso con un caso de clasificación binaria con una neurona sigmoide en la capa final, un solo registro y una sola capa escondida (parámetros  $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$ ).

Forward pass:

$$\mathbf{a}^{[1]}, \mathbf{W}^{[2]}, \mathbf{b}^{[2]} \rightarrow \mathbf{z}^{[2]} = \mathbf{W}^{[2]} \mathbf{a}^{[1]} + \mathbf{b}^{[2]} \rightarrow a^{[2]} = \sigma(\mathbf{z}^{[2]}) \rightarrow L(\hat{y}, y) = L(a^{[2]}, y) = -y * \log(a^{[2]}) - (1 - y) * \log(1 - a^{[2]})$$

Backward pass (Gradients):

$$\frac{\partial L}{\partial \mathbf{W}^{[2]}} = \frac{\partial L}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{W}^{[2]}} = (a^{[2]} - y) \mathbf{a}^{[1]}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[2]}} = \frac{\partial L}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{b}^{[2]}} = (a^{[2]} - y)$$

$$\frac{\partial a^{[2]}}{\partial \mathbf{z}^{[2]}} = \sigma'(\mathbf{z}^{[2]}) = a^{[2]}(1 - a^{[2]})$$

$$\frac{\partial L}{\partial \mathbf{z}^{[2]}} = \frac{\partial L}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial \mathbf{z}^{[2]}} = a^{[2]} - y$$

$$\frac{\partial L}{\partial a^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1 - y}{1 - a^{[2]}} = \frac{a^{[2]} - y}{a^{[2]}(1 - a^{[2]})}$$

El mismo proceso se realiza para la capa intermedia, donde  $A^{[0]} = X$ .

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{z}^{[1]}} &= \frac{\partial L}{\partial \mathbf{a}^{[2]}} \cdot \frac{\partial \mathbf{a}^{[2]}}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{a}^{[1]}} \cdot \frac{\partial \mathbf{a}^{[1]}}{\partial \mathbf{z}^{[1]}} \\ &= (\mathbf{a}^{[2]} - \mathbf{y}) \cdot \mathbf{W}^{[2]} \cdot \sigma'(\mathbf{z}^{[1]})\end{aligned}$$



# REDES NEURONALES — APRENDIZAJE

- Un resultado importante en cuanto al gradiente de la **última capa** con respecto a la combinación lineal  $Z$  entrante es que no es necesario considerar el gradiente de la función de activación si está es la función sigmoide o softmax, pues tenemos directamente:

$$\frac{\partial L}{\partial z^{[N]}} = \frac{\partial L}{\partial a^{[N]}} \cdot \frac{\partial a^{[N]}}{\partial z^{[N]}} = a^{[N]} - y = \hat{y} - y$$



# TALLER ANN: BACKPROP DE XOR CON NUMPY

Para el problema de XOR con una sencilla red de una capa con 3 entradas, una capa escondida con 4 neuronas y una capa con 1 neurona de salida, utilizando funciones sigmoides y sin considerar sesgos en las neuronas ( $b = 0$ ):

- Desarrollar la función **backProp(X, y, w1, w2, b1, b2, a2, a1)** que calcula el vector con las predicciones para el conjunto de registros de entrada. Retorna las activaciones de cada capa para los pesos y sesgos de la capa escondida ( $dw1$ ,  $db1$ ) y de la capa de salida ( $dw2$ ,  $db2$ )
- Desarrollar la función **entrenarRed(epocas, lr, X, y, w1, b1, w2, b2, a1, a2)** del ciclo de entrenamiento para un número específico de épocas, utilizando una tasa de aprendizaje definida.



# TAREA ANN: MNIST CON NUMPY

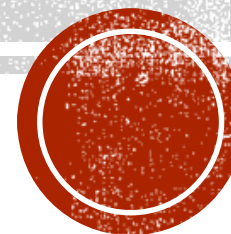
Para el problema de MNIST se define una red neuronal con 1 capa de entrada de  $28 \times 28 = 784$  neuronas de entrada, 1 capa escondida de 512 neuronas y una capa de salida de 10 neuronas con una función de activación softmax. Deben completar el código faltante, teniendo en cuenta para que funcione con una función de activación ReLU o tanh (dejar el código para tanh y dejar en comentario las líneas de ReLU en las funciones **feedForward** y **backProp**). Completar los códigos de las funciones:

- funciones **softmax** (no es necesaria la función gradiente, pues softmax solo se puede utilizar en la capa de salida, y generaliza la simplificación de softmax), **tanh**, **tanhGrad**, **relu**, **reluGrad**
- **initParams()**
- **costoGlobal(y\_est, y)**
- **backProp(X, y, w1, w2, b1, b2, a2, a1)**
- **entrenarRed(epocas, lr, X, y, w1, b1, w2, b2, a1, a2)**





# GRACIAS



# REFERENCIAS

- *Neural Networks and Deep Learning*, Andrew Ng, Coursera, 2017
- *Learning Tensor Flow*, Tom Hope, Yehezkel S. Resheff & Itay Lieder, 2017 O'Reilly
- *Machine Learning with TensorFlow*, Nishant Shukla, 2018 Manning
- *TensorFlow for Deep Learning*, Bharath Ramsundar & Reza Bosagh Zadeh, 2018, O'Reilly
- *Introduction to Statistical Learning with Applications in R (ISLR)*, G. James, D. Witten, T. Hastie & R. Tibshirani, Springer, 2014
- *Python Machine Learning (2nd ed.)*, Sebastian Raschka & Vahid Mirjalili, Packt, 2017

