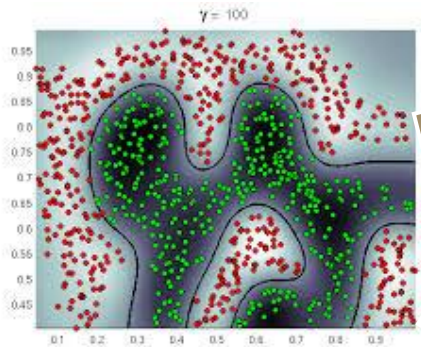
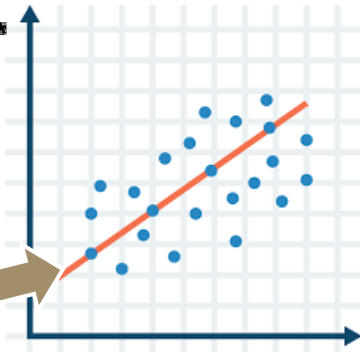


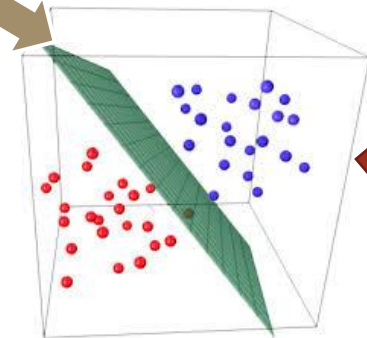
# AGENDA



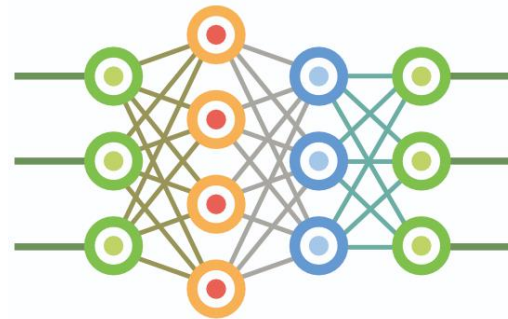
**Aprendizaje  
supervisado**



**Regresión  
(OLS)**



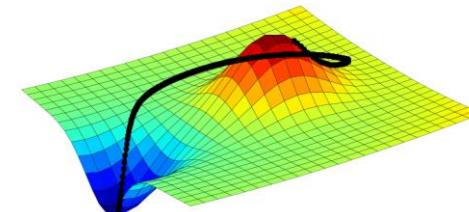
**Clasificación  
(Reg. Logística)**



**Redes Neuronales  
Artificial (ANN)**

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}}$$

**Back-Propagation**

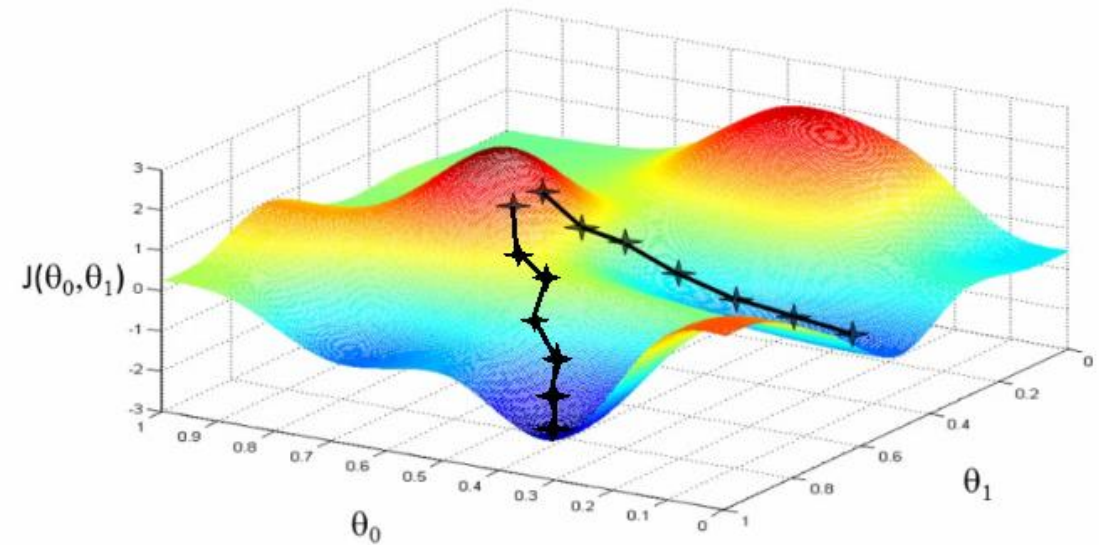


**Descenso de  
gradiente**



# DESCENSO DE GRADIENTE

Aplicación a la regresión lineal



# DESCENSO DE GRADIENTE

- Ilustraremos el proceso de descenso de gradiente para la **regresión lineal**.
- **Función de costo o de pérdida (loss) J**: función objetivo que se debe optimizar durante el proceso de aprendizaje, a partir del ajuste de los parámetros del modelo.
  - Varía en función de los parámetros  $\theta_i$  del modelo.
  - En el caso de la regresión lineal múltiple:

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

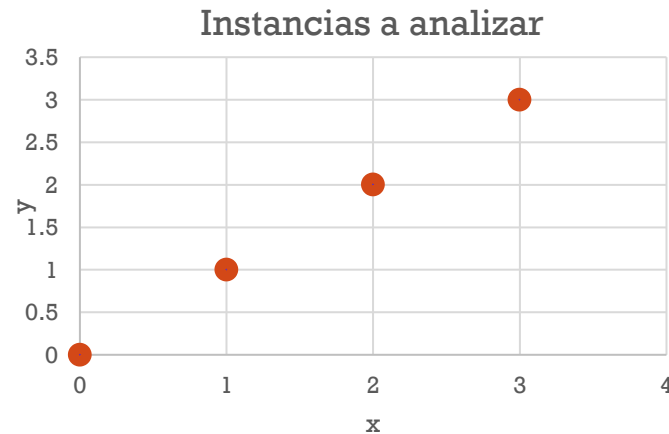
→ Cada combinación de parámetros  $\theta_0, \theta_1, \dots, \theta_n$  corresponde a un modelo lineal diferente



# DESCENSO DE GRADIENTE

- **Ejemplo:** tenemos los datos siguientes.

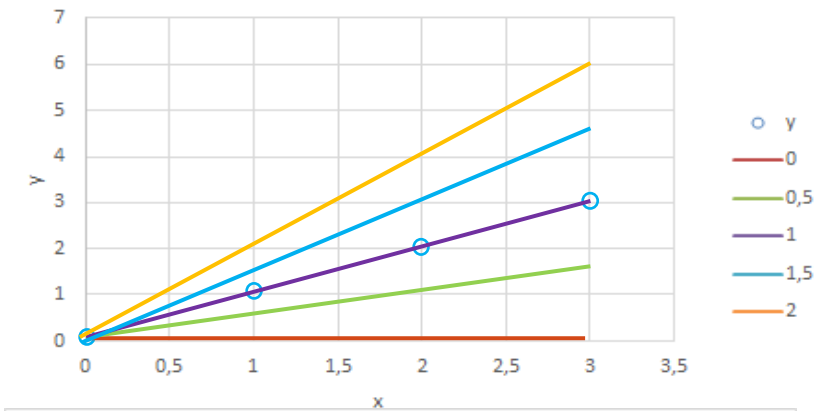
- ¿Cuáles son los valores de  $\theta_0$  y  $\theta_1$ ?



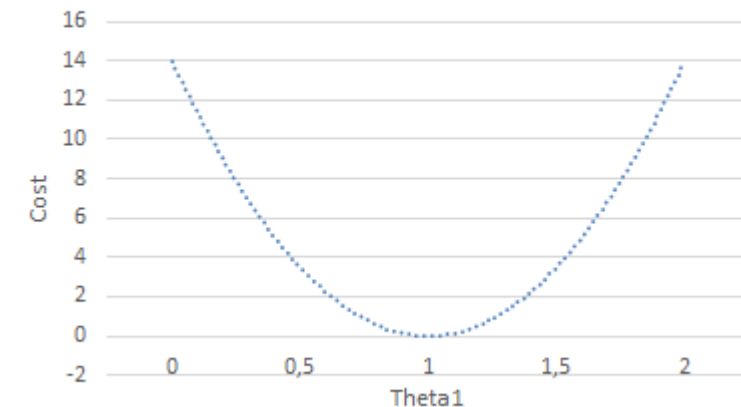
- $\theta_0 = 0$  y  $\theta_1 = 1$
- Partiendo del hecho de que sabemos que  $\theta_0 = 0$ , evaluemos la función de costo para  $\theta_1$ , haciendo variar su valor entre  $\{0, 0.5, 1, 1.5, 2\}$

		Valores de $\theta_1$									
		0	0,5	1	1,5	2	0	0,5	1	1,5	2
x	y	y est. = $\theta_1 \cdot x$					residuo = $(\theta_1 \cdot x - y)^2$				
0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0,5	1	1,5	2	1	0,25	0	0,25	1
2	2	0	1	2	3	4	4	1	0	1	4
3	3	0	1,5	3	4,5	6	9	2,25	0	2,25	9
J							14	3,5	0	3,5	14

Soluciones evaluadas

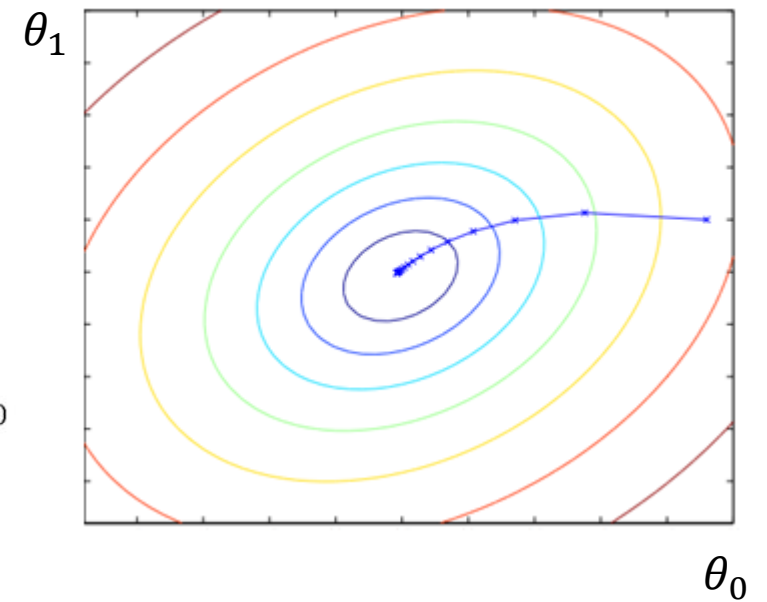
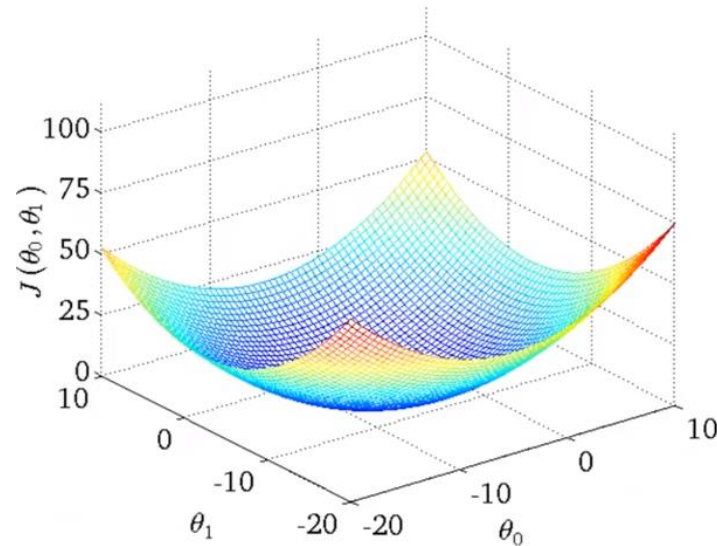
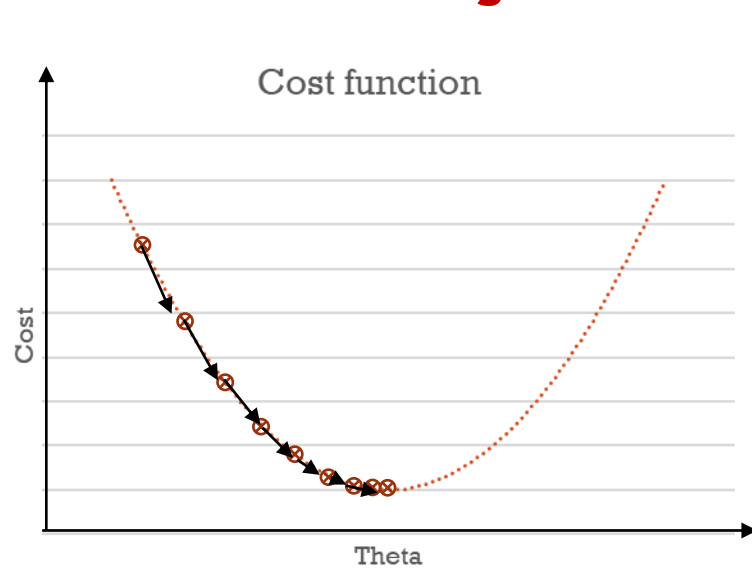


Función de costo con respecto a  $\theta_1$



# DESCENSO DE GRADIENTE

## Descenso de gradiente



# DESCENSO DE GRADIENTE

- Algoritmo:
  1. Escoger aleatoriamente valores para cada parámetro  $\theta_i$ .
  2. Actualizar todos los  $\theta_i$  **simultáneamente**:  $\theta_i := \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta)$ . Donde  $J(\theta)$  es la **función de costo** que deseamos minimizar. Nos basamos en las derivadas parciales para encontrar la dirección que se debe seguir para actualizar los parámetros de tal manera que se minimice la función de costo.
  3. Parar cuando se llegue a convergencia (mínimo del costo)
- $\alpha$  es el **learning rate (taza de aprendizaje)** y controla el nivel de actualización de los parámetros. Es importante no escoger un  $\alpha$  ni muy pequeño, ni muy grande (como veremos más adelante)
- No hay garantía de llegar a un mínimo **global**, puede que se alcance un mínimo **local**



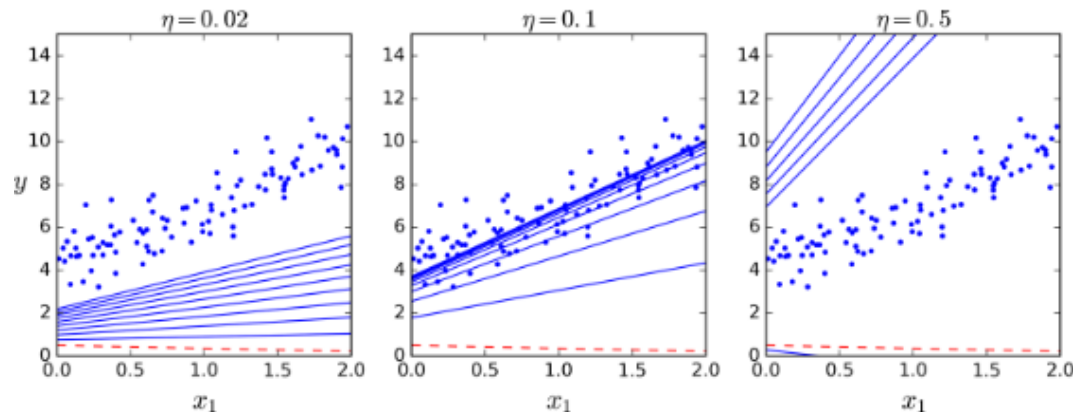
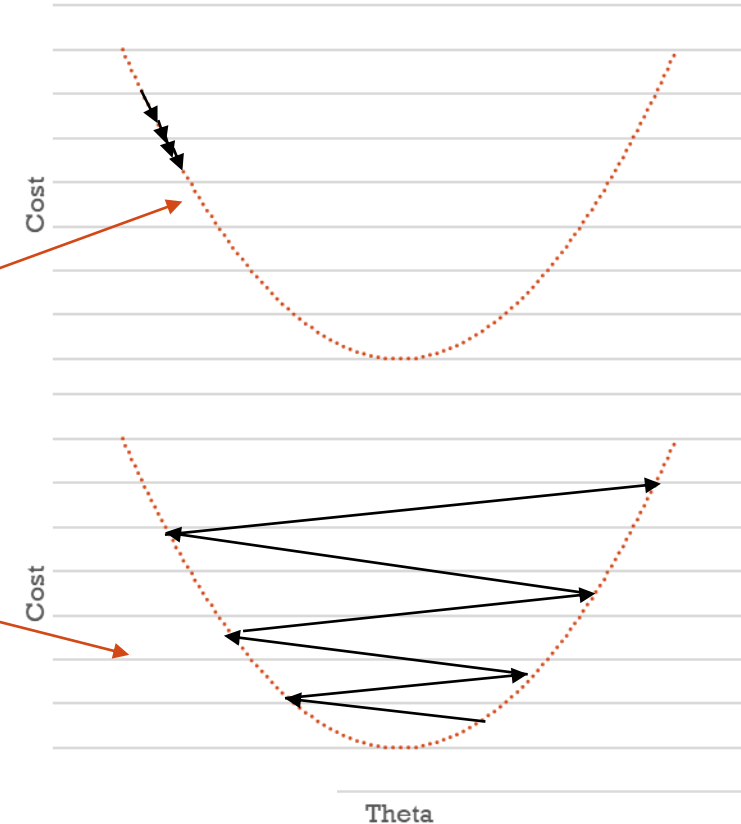


# DESCENSO DE GRADIENTE

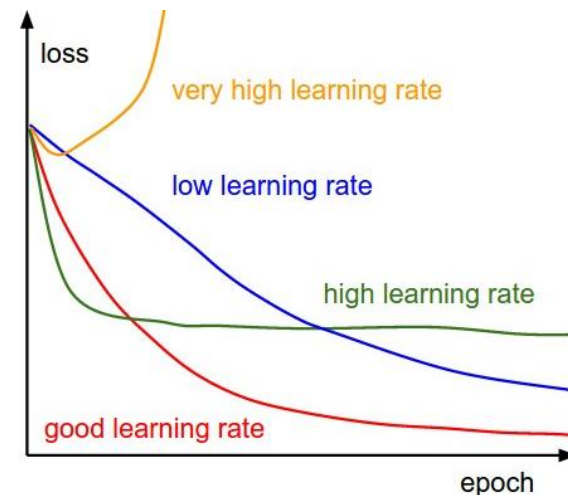
## Escogencia de la tasa de aprendizaje $\alpha$ :

- Si  $\alpha$  muy pequeño: demorado llegar a convergencia
- Si  $\alpha$  muy grande: demorado llegar a convergencia, peligro de divergencia
- El costo debe decrecer siempre en cada iteración; en caso contrario, se debe reducir el valor de  $\alpha$
- Se debe intentar de manera empírica con varios valores de  $\alpha$ : 0.001, 0.01, 0.1, 1.

Cost function



Géron, 2017



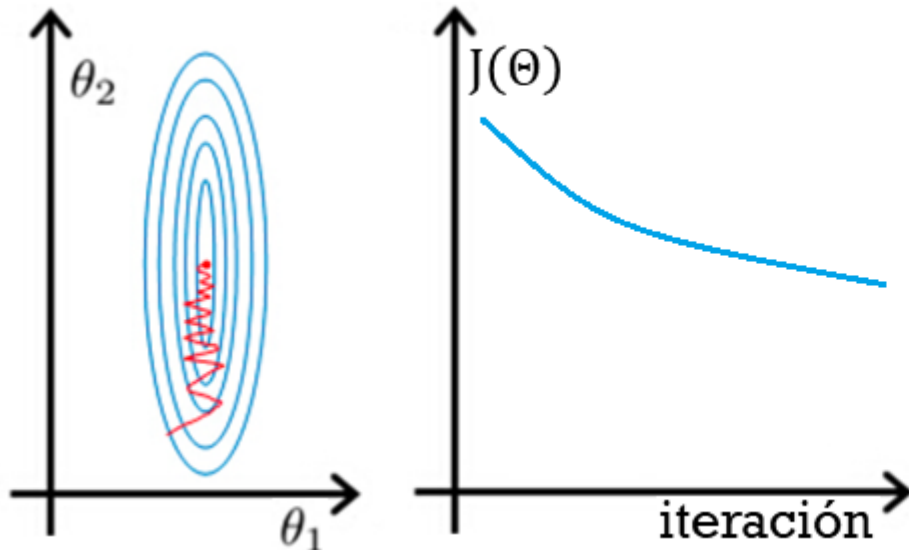
<http://cs231n.github.io/neural-networks-3/>



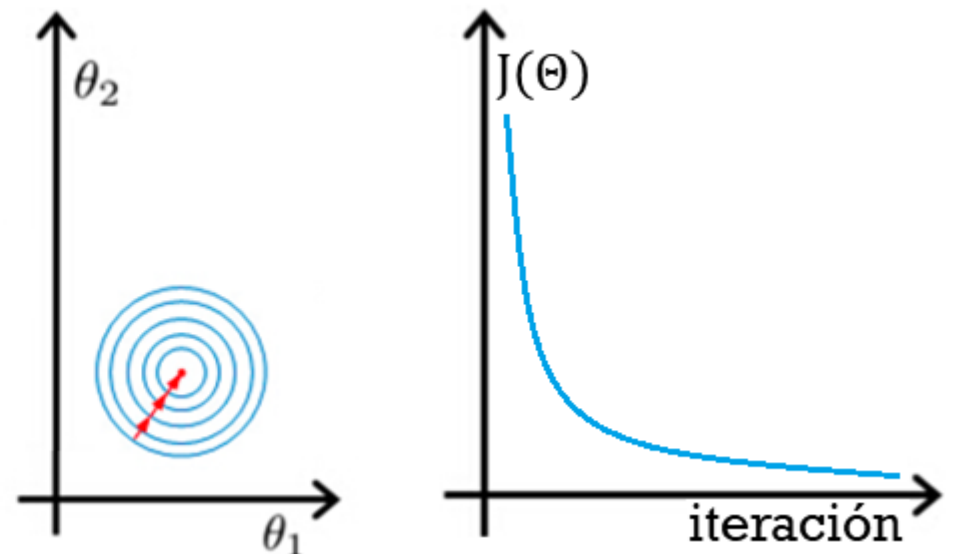
# DESCENSO DE GRADIENTE

## Feature Scaling

- Si escalas muy diferentes: demorado llegar a convergencia, sobre influencia de las derivadas parciales de las variables de mayor escala iteración  
→ **Normalización**



- Si misma escala: influencia igual de las derivadas parciales de todos los parámetros
- También ayuda que los datos estén centrados en 0

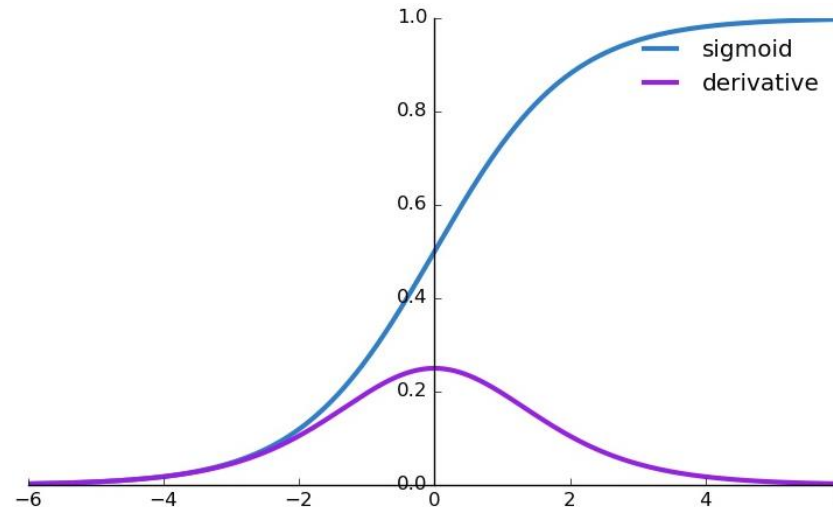




# DESCENSO DE GRADIENTE

## Gradiente explosivo o desvaneciente

- Si se usan funciones de activación como la sigmoide o la tangente hiperbólica, los valores muy pequeños o muy grandes tienen una pendiente muy baja, por lo que la optimización puede ser extremadamente lenta → Uso de ReLU
- **Batch normalization:** estandarizar los datos de entrada a cada función de activación de una capa para evitar valores extremos, teniendo en cuenta un subconjunto de instancias de entrenamiento (batch).

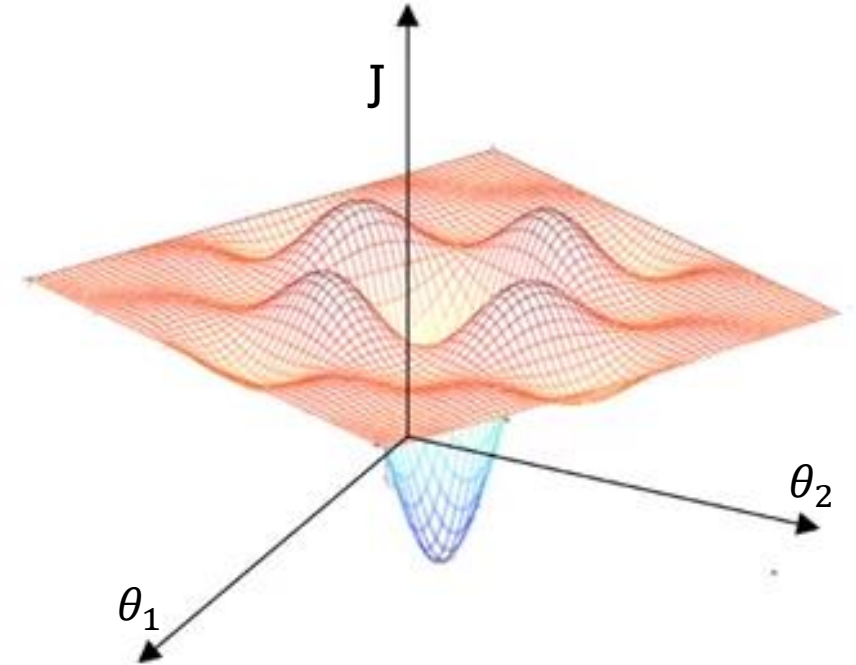


# DESCENSO DE GRADIENTE

## ¿Problema de óptimos locales?

Se creería que uno de los problemas principales del gradient descent en el contexto del DL es el de encontrarse con una función de costo con muchos óptimos locales.

Para que haya un óptimo local, todas las derivadas parciales con respecto a cada uno de los parámetros (e.g. 1 millón) tienen que ser convexas en ese punto, lo que no es tan probable.



Andrew Ng, 2017



# DESCENSO DE GRADIENTE

## Repaso de derivadas

- $\frac{\partial}{\partial x} c = 0$ , por ejemplo,  $\frac{\partial}{\partial x} 4 = 0$
- $\frac{\partial}{\partial x_1} (c_1 * x_1 + c_2 * x_2) = c_1$ , por ejemplo,  $\frac{\partial}{\partial x} (3 * x + 4) = 3$
- $\frac{\partial}{\partial x} (x^p) = p * x^{p-1}$ , por ejemplo,  $\frac{\partial}{\partial x} (x^2) = 2 * x$ ,  $\frac{\partial}{\partial x} \left(\frac{1}{x}\right) = \frac{\partial}{\partial x} (x^{-1}) = -x^{-2} = -\frac{1}{x^2}$
- $\frac{\partial}{\partial x} (e^x) = e^x$ ,  $\frac{\partial}{\partial x} (\log(x)) = \frac{1}{x}$

## Regla de la cadena (*chain rule*), para composiciones de funciones

- $\frac{\partial}{\partial x} (f(g(x))) = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$ ,  
por ejemplo,  $\frac{\partial}{\partial x} (3 * x + y)^2 = \frac{\partial}{\partial (3*x+y)} (3 * x + y)^2 * \frac{\partial}{\partial x} (3 * x + y) = 2 * (3 * x + y) * 3$



# DESCENSO DE GRADIENTE

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^m (\theta_0 + \theta_1 x_1^{(i)} + \dots + \theta_n x_n^{(i)} - y^{(i)})^2$$
$$\left. \begin{array}{l} \frac{\partial}{\partial \theta_0} J(\Theta) ?? \\ \frac{\partial}{\partial \theta_1} J(\Theta) ?? \end{array} \right\} \theta_i := \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\Theta)$$

→ **Desarrollar analíticamente las soluciones**

▪ Solución para la regresión lineal múltiple:

- Para la intercepción:  $\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (\theta_0 + \theta_1 x_1^{(i)} + \dots + \theta_n x_n^{(i)} - y^{(i)})$
- Para los coeficientes:  $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (\theta_0 + \theta_1 x_1^{(i)} + \dots + \theta_n x_n^{(i)} - y^{(i)}) * x_j^{(i)}$



# TALLER GRADIENT DESCENT

Desarrollar el taller de gradient descent para la regresión lineal utilizando la librería numpy



# 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** (aunque se debe prevenir la saturación del gradiente). 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, aunque también aleatoriamente.
- 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.

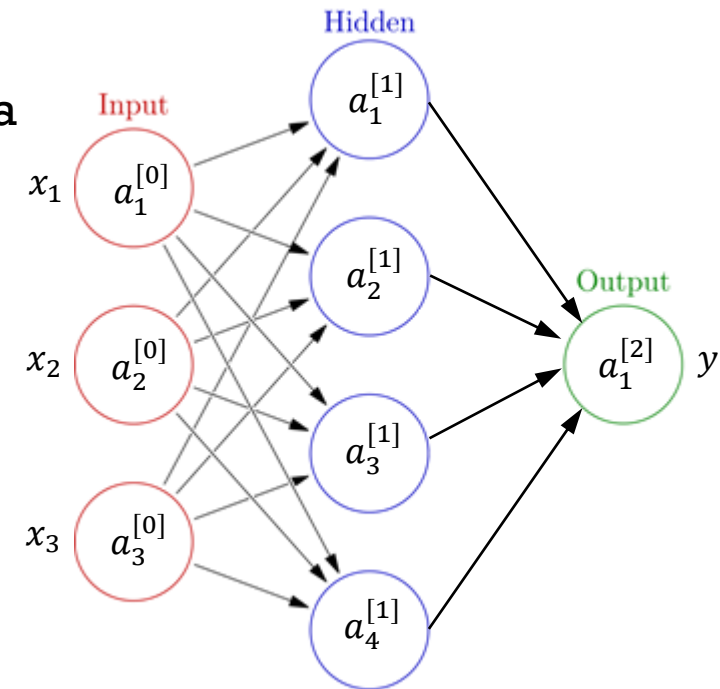




# REDES NEURONALES — APRENDIZAJE

## 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  $\mathbf{W}^{[l]}$ ) 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 la capa  $L-1$  (entradas 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:  $\mathbf{z}^{[l]} = \mathbf{W}^{[l]} * \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$ , que se conoce como “**net input**”
  - 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

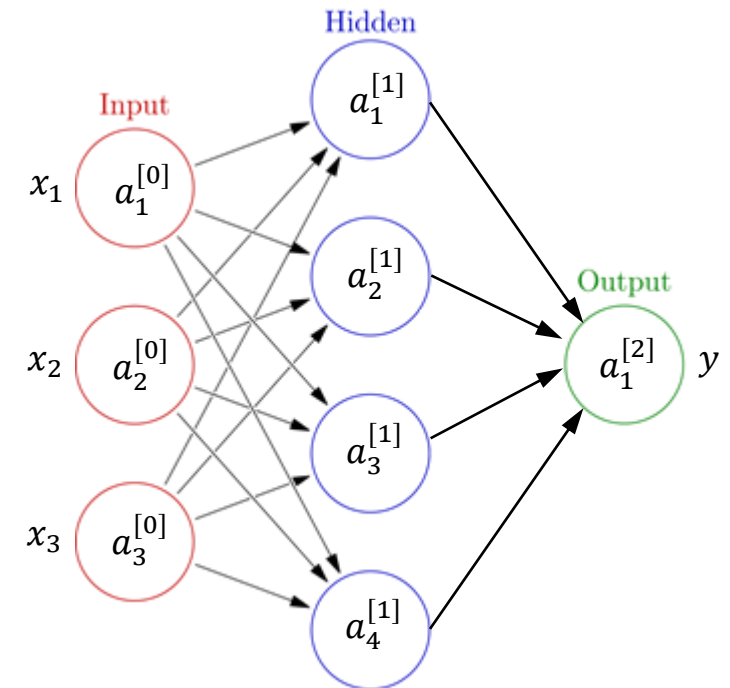
## Feed Forward

- 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?

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

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

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



# 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:

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



# 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 y del contexto), para poder propagarlo a las capas anteriores:

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

Binary cross-entropy loss:  $L(\hat{y}, y) = -\frac{1}{N} \sum_i^N (y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))$

Categorical cross-entropy loss:  $L(\hat{y}, y) = -\sum_i^N p(y_i) * \log(p(\hat{y}_i))$

Mean square loss:  $L(\hat{y}, y) = \frac{1}{N} \sum_i^N (\hat{y}_i - y_i)^2$

Mean square logarithmic loss:  $L(\hat{y}, y) = \frac{1}{N} \sum_i^N (\log(\hat{y}_i + 1) - \log(y_i + 1))^2$



# 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 ( $a2=y_{\text{estimado}}$ )
- 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:

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

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

Finalmente, se compara el resultado del modelo con la etiqueta real utilizando una **función de pérdida**, que dependerá del tipo de modelo y del tipo de salida (logits, probabilidades, predicción numérica). Para clasificación se utilizan binary cross-entropy, categorical cross-entropy, negative log likelihood (NLL). Para regresión se utiliza MSE sobretodo.





# 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  $\sigma'(Z)$

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

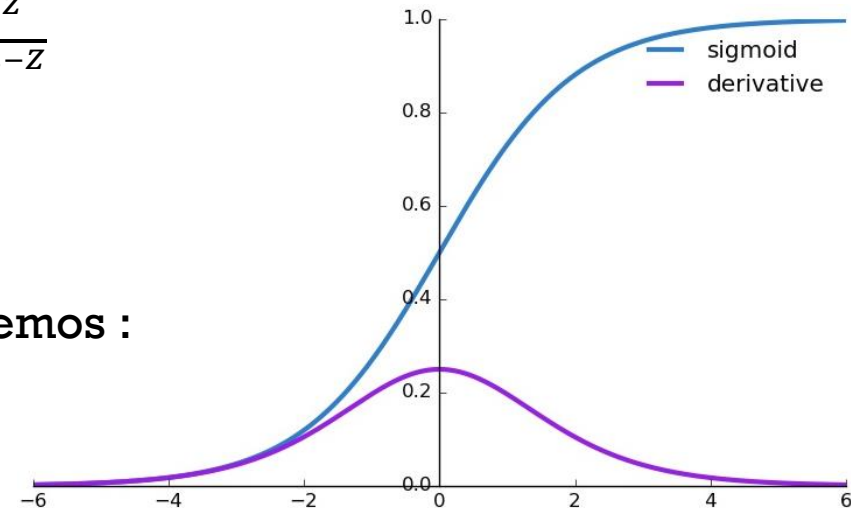
**Demostrar que  $\sigma'(Z) = \sigma(Z) \cdot (1 - \sigma(Z))$**

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

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

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

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



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

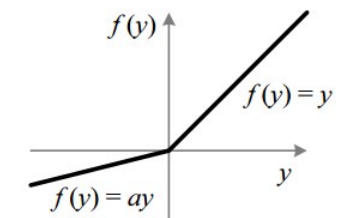
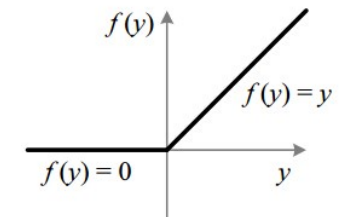
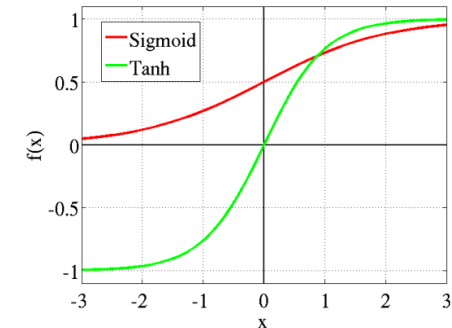


# REDES NEURONALES — APRENDIZAJE

## 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.  $a(Z) = \max(0.01 * Z, Z)$   $a'(Z) = \begin{cases} 0.01, & \text{si } Z < 0 \\ 1, & \text{si } Z > 0 \\ \text{indefinido}, & \text{si } Z = 0 \end{cases}$



[www.towardsdatascience.com](http://www.towardsdatascience.com)



# 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.

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

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

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

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  $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$ ).

Diagram illustrating the forward and backward passes for the second layer of a neural network.

**Forward Pass:**

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

**Backward Pass (Gradient Calculation):**

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

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

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

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

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

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.





# REFERENCIAS

- *Introduction to Statistical Learning with Applications in R (ISLR)*, G. James, D. Witten, T. Hastie & R. Tibshirani, 2014, Springer
- *Python Data Science Handbook*, Jake VanderPlas, 2017, O'Reilly
- *The Elements of Statistical Learning*, T. Hastie & R. Tibshirani, 2009, Springer
- *Python Machine Learning (2nd ed.)*, Sebastian Raschka, 2017, Packt

