

# Aprendizaje Automático

**Clase 9:**

Regresión Logística  
Entropía Cruzada  
Redes Neuronales  
Backpropagation

# Repaso

# Regresión

## Objetivo del aprendizaje supervisado (caso regresión)

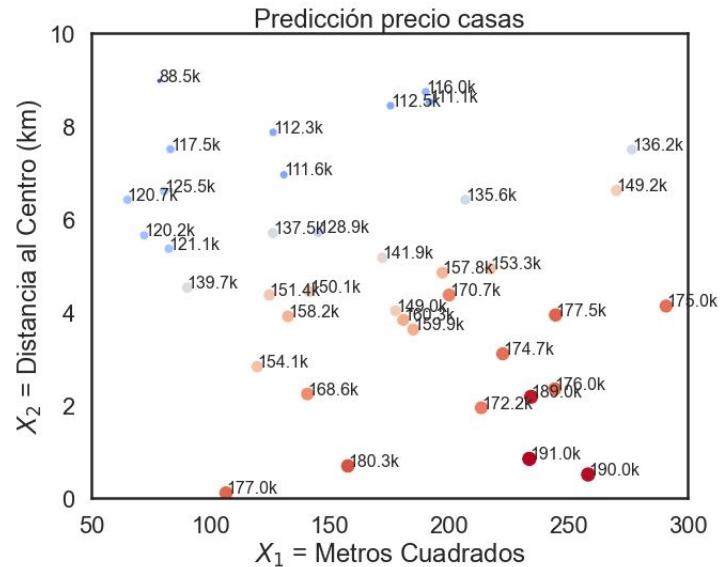
Estimar la **función determinista  $f(\mathbf{X})$**  que determina la relación  $\mathbf{X} \rightarrow \mathbf{Y}$ :

Es decir, estimar  $f$  tal que  $\mathbf{Y} = f(\mathbf{X}) + \varepsilon$  a través de un modelo  $\hat{h}_D(\mathbf{X})$ . En donde  $\varepsilon$  es el error irreducible.

En donde  $\mathbf{Y} \in \mathbb{R}$ .

### Ejemplo

Estimar el  $\mathbf{Y}$  = **precio de una casa** según  
 $X_1$  = los metros cuadrados cubiertos,  
 $X_2$  = su distancia al centro de la ciudad.



# MSE como función de costo

Una métrica muy utilizada en problemas de regresión.

$$\text{MSE}_{X,y} = \frac{1}{n} \sum_{i=1}^n (\hat{h}(x^{(i)}) - y^{(i)})^2$$

Vimos que para regresión lineal suponemos

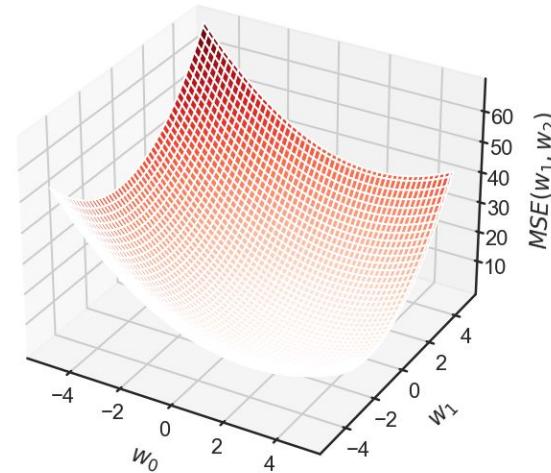
$$Y = w_0 + w_1 X_1 + w_2 X_2 + \dots + w_p X_p + \varepsilon$$

Es decir,  $\mathbf{h}$  ya entrenado tendrá la siguiente forma:

$$\hat{h}_{w_0, \dots, w_p}(x^{(i)}) = w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \dots + w_p x_p^{(i)}$$

$$\begin{aligned}\text{MSE}_{X,y} &= \frac{1}{n} \sum_{i=1}^n (\hat{h}_{w_0, w_1, \dots, w_p}(x^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{n} \sum_{i=1}^n (w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \dots + w_p x_p^{(i)} - y^{(i)})^2\end{aligned}$$

Pensando al MSE como **una función de costo que depende de los** parámetros de un modelo  $\text{MSE}_{X,y}(w)$ .



“Entrenar el modelo”: encontrar los  $w$  que minimicen la siguiente expresión

$$\text{MSE}_{X,y}(w) = \frac{1}{n} \sum_{i=1}^n (\hat{h}_{w_0, w_1, \dots, w_p}(x^{(i)}) - y^{(i)})^2$$

# Minimizando el MSE(w)

Para encontrar los pesos óptimos  $w_0, w_1, \dots, w_p$  según un dataset, podemos considerar  $\text{MSE}(w)$  como la función que deseamos optimizar.

Vimos cómo utilizar Descenso por el Gradiente, seteando la función de costo J de la siguiente manera:

$$(a) J_{X,y}(w) = \text{MSE}_{X,y}(w) = \frac{1}{n} \sum_{i=1}^n (\hat{h}_w(x^{(i)}) - y^{(i)})^2$$

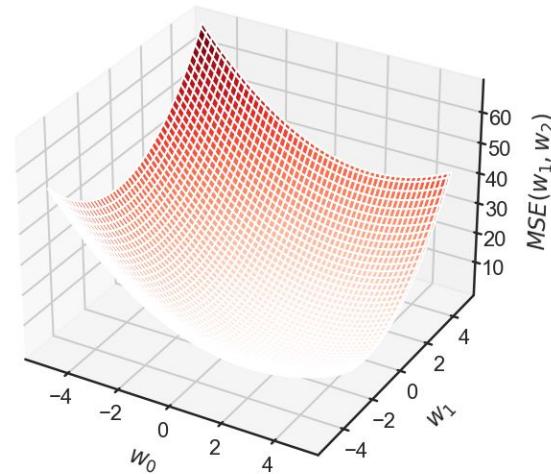
$$(b) \nabla_w J_{X,y}(w) = \nabla_w \text{MSE}_{X,y}(w) = \frac{2}{n} \sum_{i=1}^n (\hat{h}_w(x^{(i)}) - y^{(i)}) * x^{(i)}$$

En donde  $X, y$  son  $X_{\text{train}}, y_{\text{train}}$  (o un mini batch)

Preguntas para pensar.

- ¿La expresión (a) representa el  $\text{MSE}_{X,y}$  para cualquier modelo?
- ¿La expresión (b) representa el  $\nabla \text{MSE}_{X,y}$  siempre? (o es  $\nabla \text{MSE}_{X,y}^{\text{reg-lineal}}$ )?
- ¿Es  $\text{MSE}(w)$  una función convexa, o sólo  $\text{MSE}_{X,y}^{\text{reg-lineal}}(w)$ ?

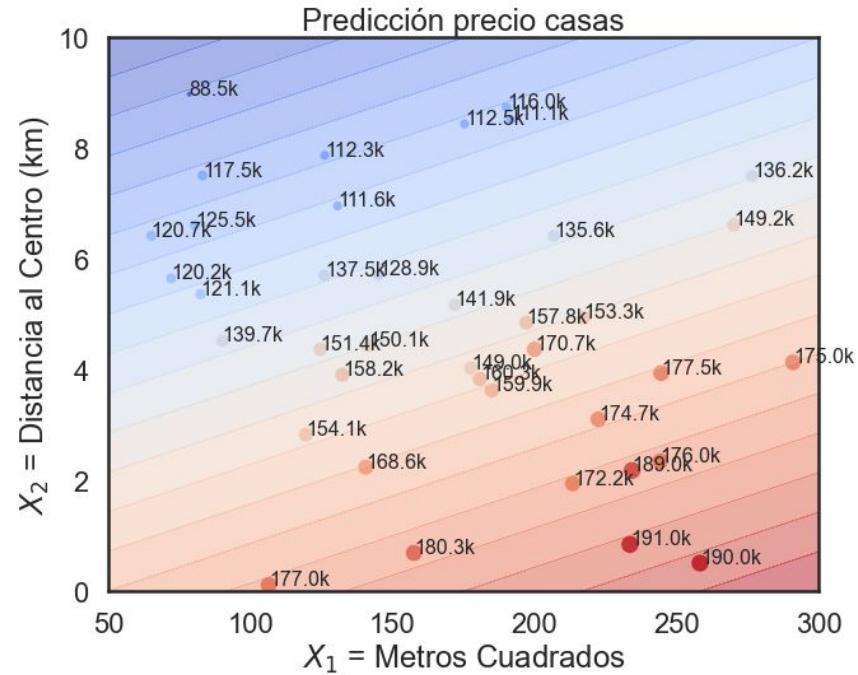
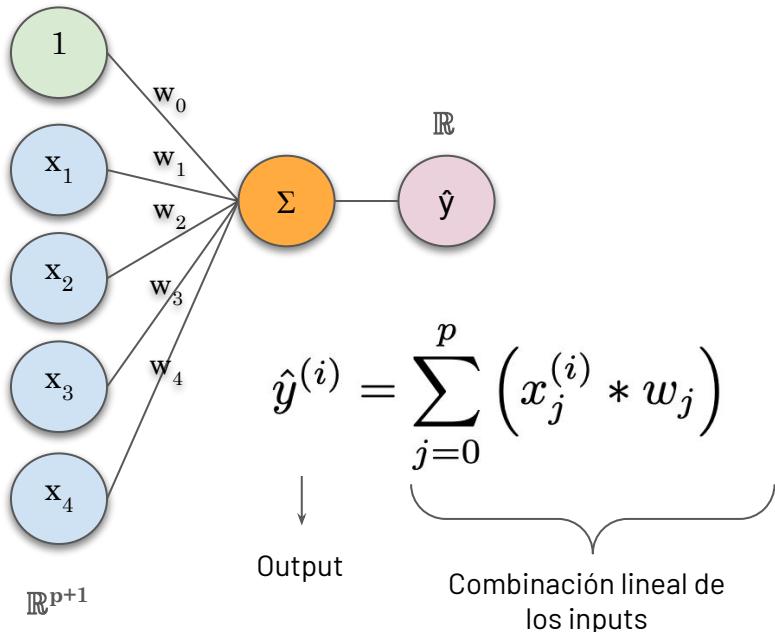
“Entrenar el modelo”: encontrar los  $w$  que minimicen la siguiente función



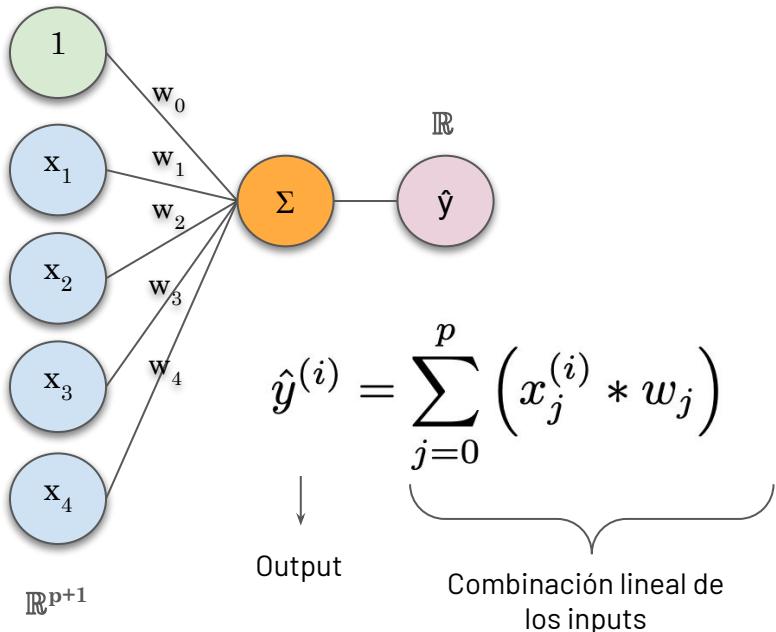
# Regresión Logística

# Otra visualización para regresión lineal

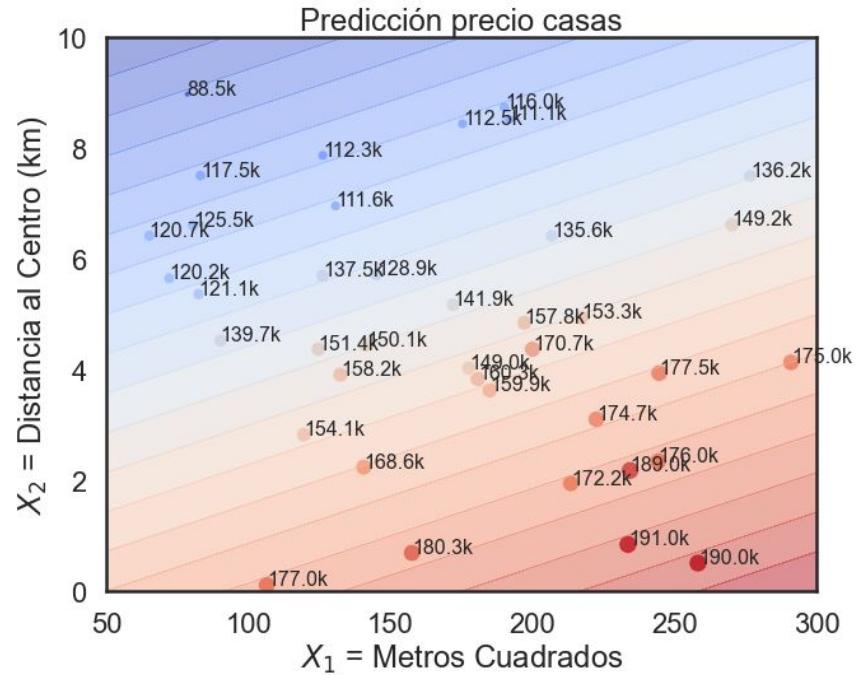
$$\hat{h}_{w_0, \dots, w_p}(x^{(i)}) = w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \dots + w_p x_p^{(i)}$$



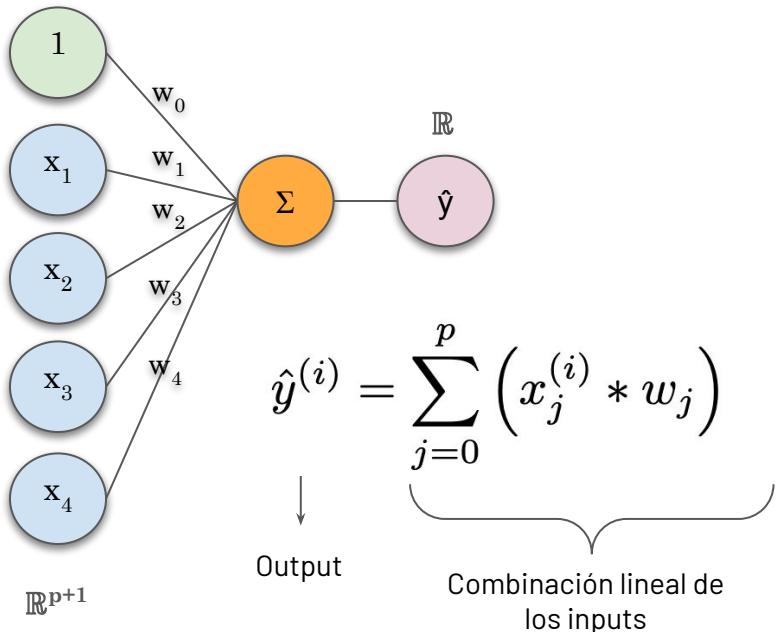
# Convirtiendo a clasificación binaria



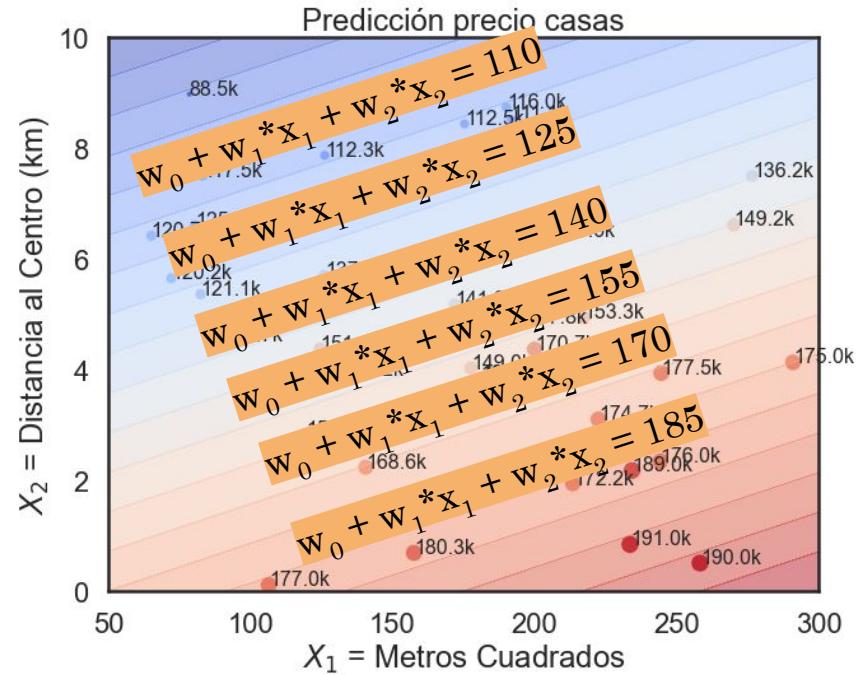
¿Cómo podremos convertir este modelo en un modelo para clasificación? Ej, "CARA" (clase positiva) vs "BARATA"



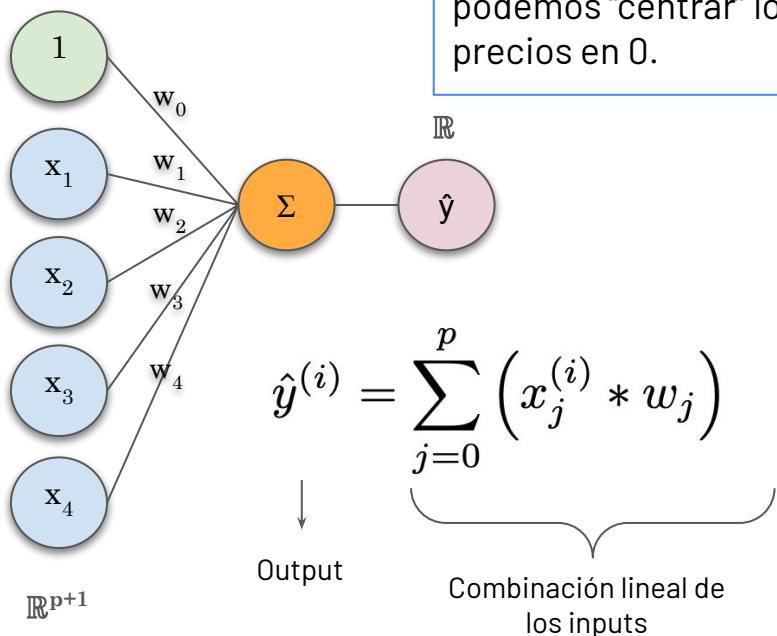
# Convirtiendo a clasificación binaria



¿Cómo podremos convertir este modelo en un modelo para clasificación? Ej, "CARA" (clase positiva) vs "BARATA"

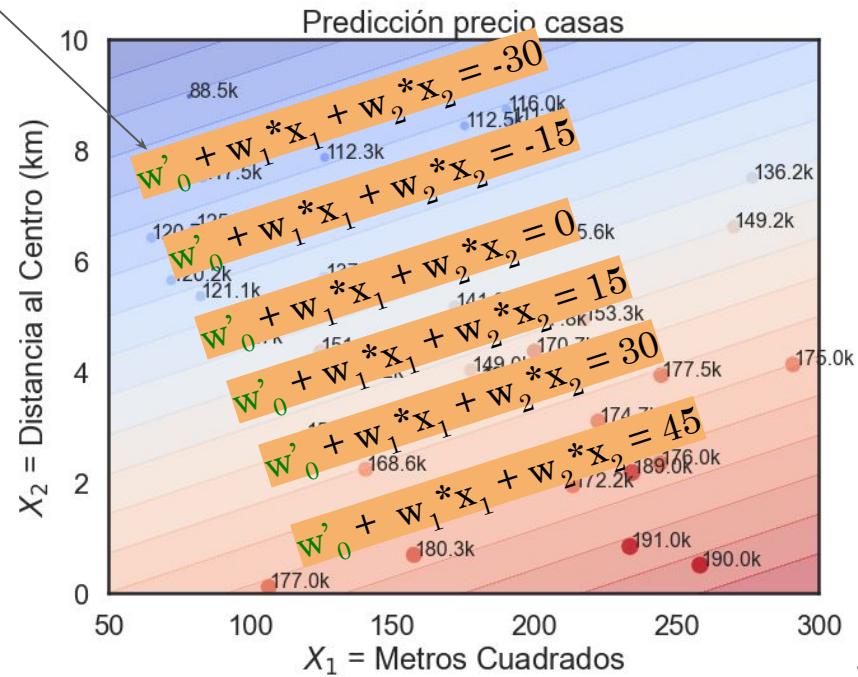


# Convirtiendo a clasificación binaria

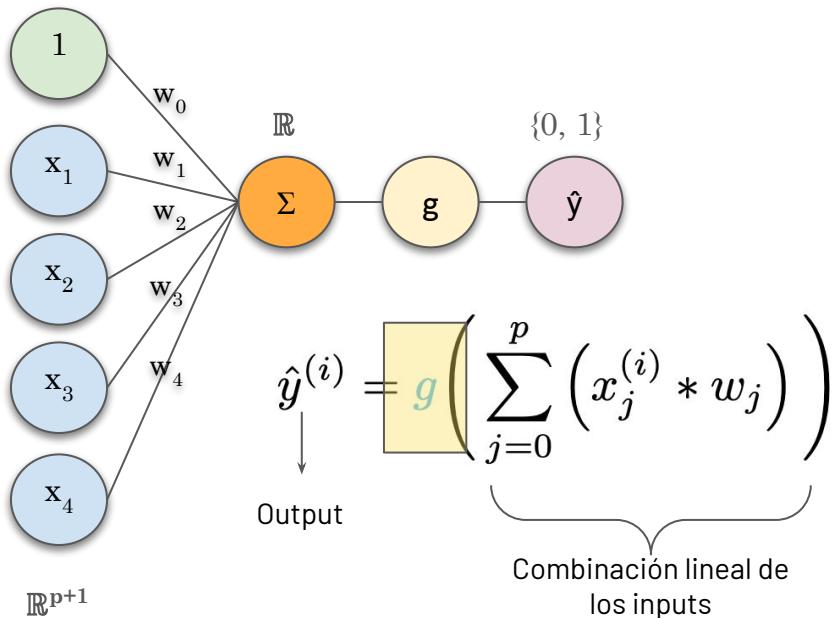


si cambiamos el  $w_0$   
podemos "centrar" los  
precios en 0.

¿Cómo podremos convertir este modelo en un modelo para  
clasificación? Ej, "CARA" (clase positiva) vs "BARATA"

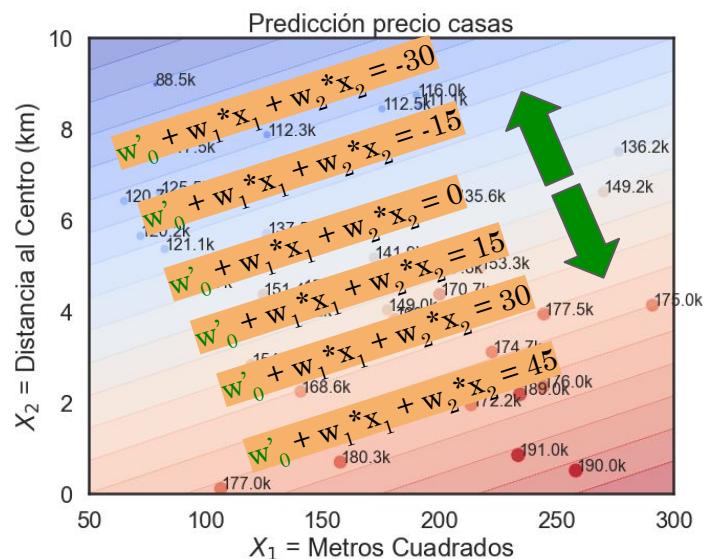
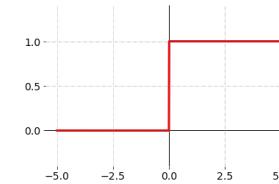


# Convirtiendo a clasificación binaria

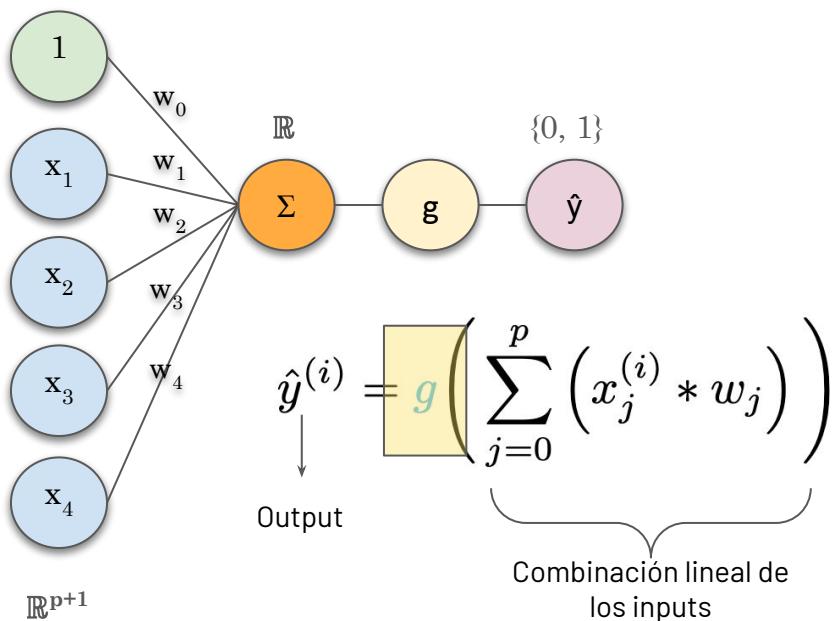


Una idea natural es agregar una función  $g$  de la siguiente manera:

$$g(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

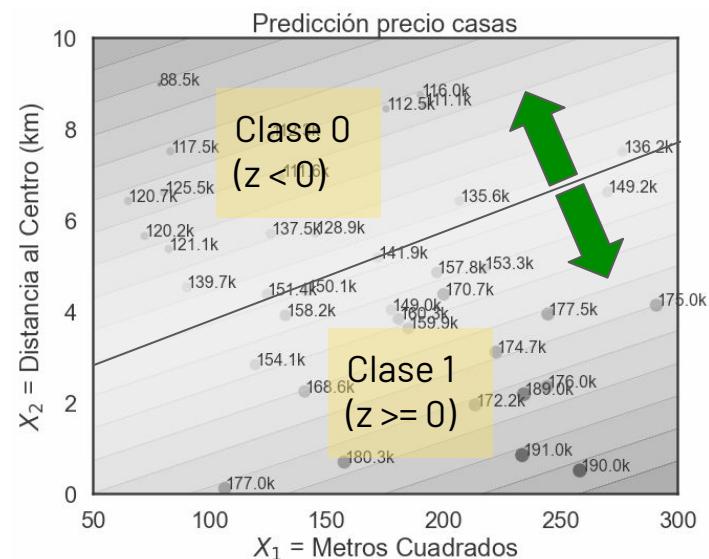
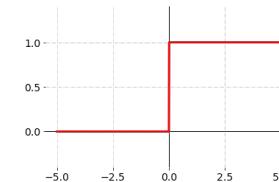


# Convirtiendo el problema en un problema de clasificación

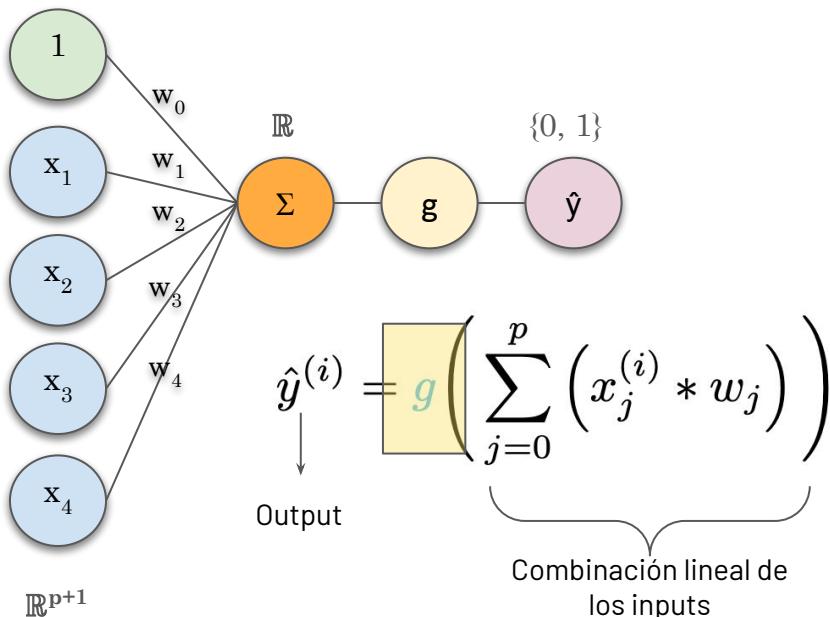


Una idea natural es agregar una función  $g$  de la siguiente manera:

$$g(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

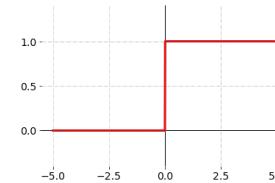


# Convirtiendo el problema en un problema de clasificación



Una idea natural es agregar una función  $g$  de la siguiente manera:

$$g(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$



¿Por qué esta función **no es la más** apropiada?

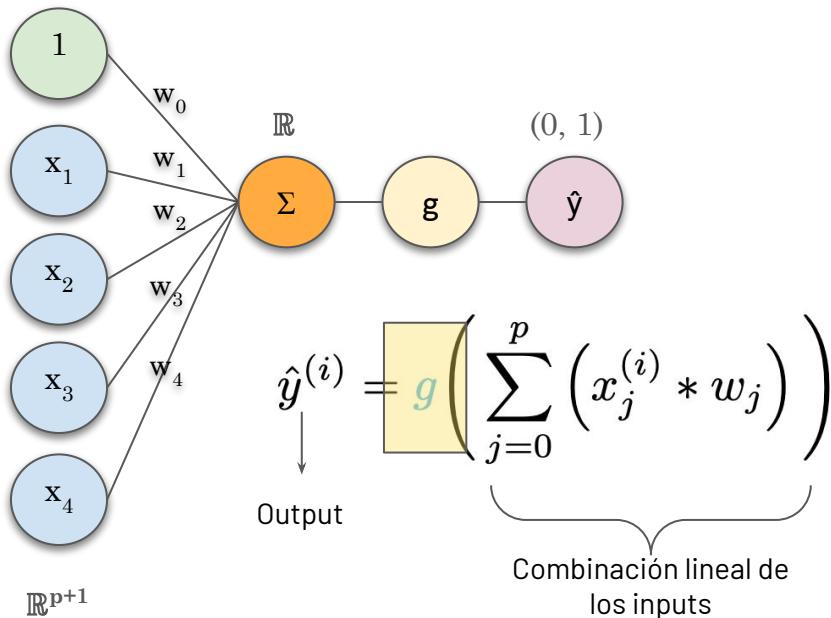
Recordar estuvimos midiendo cómo pequeños cambios aplicados a los pesos  $w$  afectan a la predicción final  $\hat{y}$ . Es decir, midiendo como:

$w + \Delta w$  se relaciona con  $\hat{y} + \Delta \hat{y}$  y cómo esos cambios nos acercan al output esperado, a través de  $\nabla_w \text{MSE}(w)$  lo cual debería permitir saber para qué dirección moverse.

El problema ahora es que cambios pequeños en  $w$  en general no afectarán a la nueva  $\hat{y}$ .

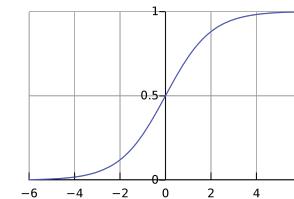
# “Regresión” logística

(un modelo para clasificación)



Probamos con otra  $g$  de la siguiente pinta:

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



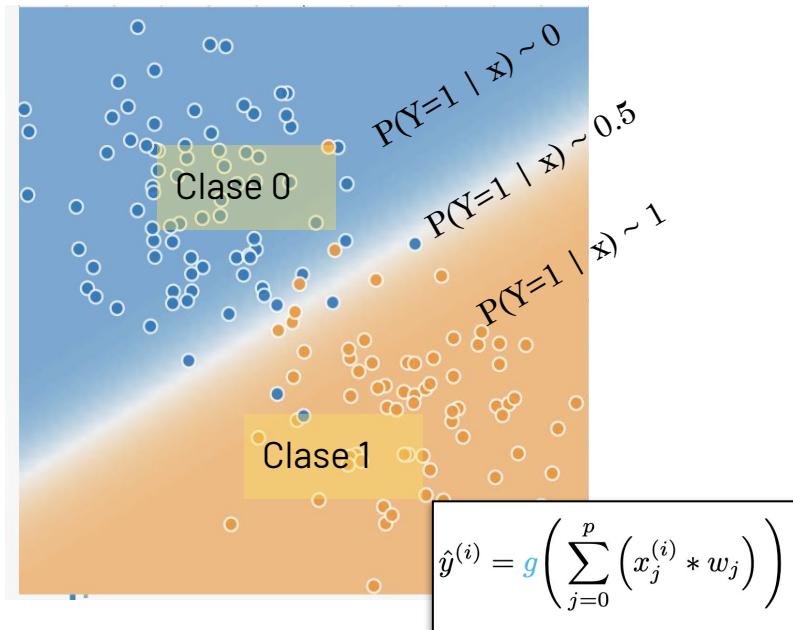
Esta función es conocida como **función sigmoidea devuelve**:

- Cerca de 1 cuando  $z$  es muy grande  
En nuestro caso cuando  $X^t w \gg 0$
- 0.5 cuando  $z = 0$ 
  - En nuestro caso cuando  $X^t w = 0$
- Cerca de 0 cuando  $z$  es muy chico  
○ En nuestro caso cuando  $X^t w \ll 0$

Podemos considerar:  $\hat{y}^{(i)} = P(Y=1 | X=x^{(i)})$   
(tomamos la salida como la **probabilidad** para la clase positiva)

# "Regresión" logística

(un modelo para clasificación)

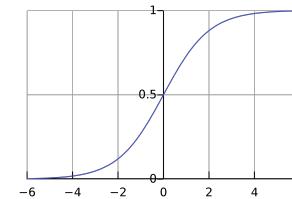


El problema ahora es, ¿cómo encontramos los W?

Nota: Al ser un problema de clasificación, los targets serán valores  $[0, 1]$ . Ya no tiene sentido utilizar  $MSE_{X,y}$

Probamos con otra  $g$  de la siguiente pinta:

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



Esta función es conocida como **función sigmoidea devuelve**:

- Cerca de 1 cuando  $z$  es muy grande  
En nuestro caso cuando  $X^t w \gg 0$
- 0.5 cuando  $z = 0$ 
  - En nuestro caso cuando  $X^t w = 0$
- Cerca de 0 cuando  $z$  es muy chico  
○ En nuestro caso cuando  $X^t w \ll 0$

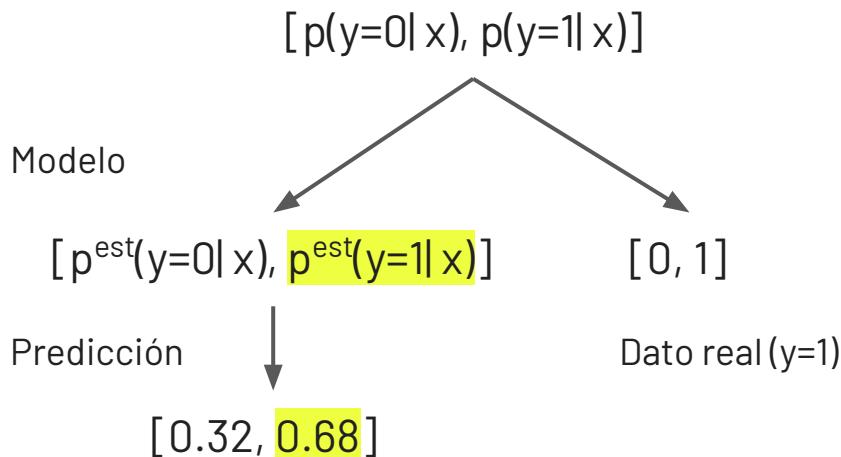
Podemos considerar:  $\hat{y}^{(i)} = P(Y=1 | X=x^{(i)})$

(tomamos la salida como la **probabilidad** para la clase positiva)

Abrimos paréntesis 1:  
Entropía Cruzada Binaria

# Entropía cruzada

Entonces nosotros tenemos ahora un modelo que devuelve probabilidades, ¿cómo medimos el error de este modelo? ¿Sirve mirar por ejemplo el error absoluto entre la probabilidad predicha y la etiqueta (0-1)? ¿Cómo generaliza eso a muchas etiquetas?



iEstamos comparando distribuciones de probabilidad discretas!

Una buena manera de hacerlo es con la **entropía cruzada entre distribuciones con C clases**

$$H(p, q) = - \sum_{i=1}^C p(y_i) \log(q(y_i))$$

$$H(p, q) = - \sum_{i=1}^C p(y=c_i|x) \log q(y=c_i|x)$$

Esta métrica está intrínsecamente relacionada (es equivalente) a la Divergencia de Kullback-Leibler (Ver [The Key Equation Behind Probability](#))

# Entropía cruzada

(para caso binario)

La idea de esta métrica es poder calcular el error en un dataset para el cual tenemos la **probabilidad estimada** de que cada instancia pertenezca a la **clase positiva** en el caso de clasificación binaria.

Veamos qué pinta tiene esta métrica de error en caso binario, es decir cuando:

$$P(Y=1 | X=x^{(i)}) = 1 - P(Y=0 | X=x^{(i)})$$

$$\text{Costo\_BinCE}_{X,y} = \frac{1}{n} \sum_{i=1}^n \text{Binary-CE}^{(i)}$$

$$\text{Binary-CE}^{(i)} = -[y^{(i)} \log(\hat{h}^{\text{bin}}(x^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{h}^{\text{bin}}(x^{(i)}))]$$

equivalentemente

$$\text{Binary-CE}^{(i)} = \begin{cases} -\log(\hat{P}(Y = 1 | X = x^{(i)})) & y^{(i)} = 1 \\ -\log(\hat{P}(Y = 0 | X = x^{(i)})) & y^{(i)} = 0 \end{cases}$$

logaritmo natural (base e)

$$P(Y=1 | X=x^{(i)})$$

$$y^{(i)}$$

$$\text{Binary-CE}^{(i)}$$

$$\text{Costo_BinCE}_{X,y} = <\text{completar}>$$

$$0.775$$

$$1$$

$$<\text{completar}>$$

$$0.116$$

$$0$$

$$<\text{completar}>$$

$$0.884$$

$$1$$

$$<\text{completar}>$$

$$0.744$$

$$0$$

$$<\text{completar}>$$

$$0.320$$

$$0$$

$$<\text{completar}>$$

¿Cuál es el mínimo valor que podemos obtener aquí? ¿Qué debe ocurrir para llegar a ese valor?

Cerramos paréntesis

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

# "Regresión" logística

(un modelo para clasificación)

$$\hat{h}_w^{bin}(x^{(i)}) = \hat{P}(Y = 1 | X = x^{(i)}) = g\left(\sum_{j=0}^p (x_j^{(i)} * w_j)\right)$$

Como vimos anteriormente, es común utilizar estas métricas como **funciones de costo** a minimizar:

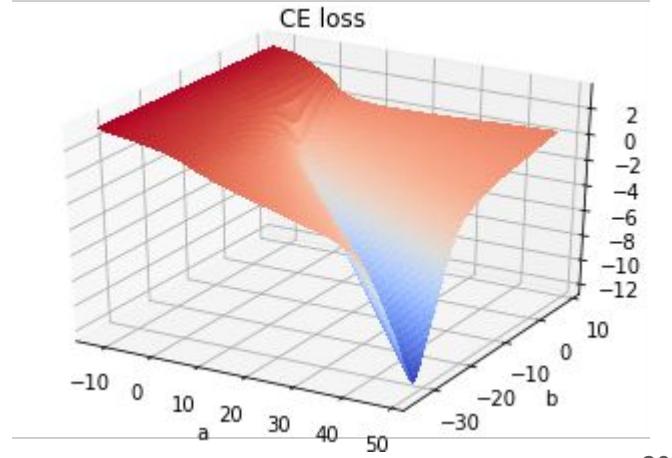
$$J_{X,y}(w) = \frac{1}{n} \sum_{i=1}^n \text{Binary\_CE}^{(i)}(w) = \frac{1}{n} \sum_{i=1}^n -[y^{(i)} \log(\hat{h}_w^{bin}(x^{(i)})) + (1-y^{(i)}) \log(1-\hat{h}_w^{bin}(x^{(i)}))]$$

Y conociendo la forma de  $h$ , ya que seguimos en el caso regresión logística (binaria).

Podemos calcular analíticamente su gradiente (y por lo tanto utilizar descenso de gradiente):

$$\nabla_w J_{X,y}(w) = \frac{1}{n} \sum_{i=1}^n (\hat{h}_w^{bin}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

(ojo, esta forma es porque  $\hat{h}_w^{bin}$  es una reg. logística)



¡Otra función convexa!

# "Regresión" logística (un modelo para clasificación)

Ejercicio: Lean y discutan este código en grupos, mientras tomo mate. Discutan cómo se transformaron las fórmulas de la slide anterior.

```
def fit(self, X_train, y_train):
    # Suponemos y_train es un arreglo de 0 y 1, ej: np.array([0,0,1,0,1,...])
    N = len(y_train)
    ones_column = np.ones(X_train.shape[0])
    X_train_ext = np.hstack((ones_column, X_train))

    def cost_binary_ce(w):
        probas = _pred(w) # Para pensar, qué son estas probas, ¿probabilidad de qué?
        return -(1 / N) * np.sum(y_train * log(probas) + (1 - y_train) * log(1 - probas))

    def grad_cost_binary_ce(w):
        probas = _pred(w)
        return 1 / N * (X_train_ext.T @ (probas - y_train))

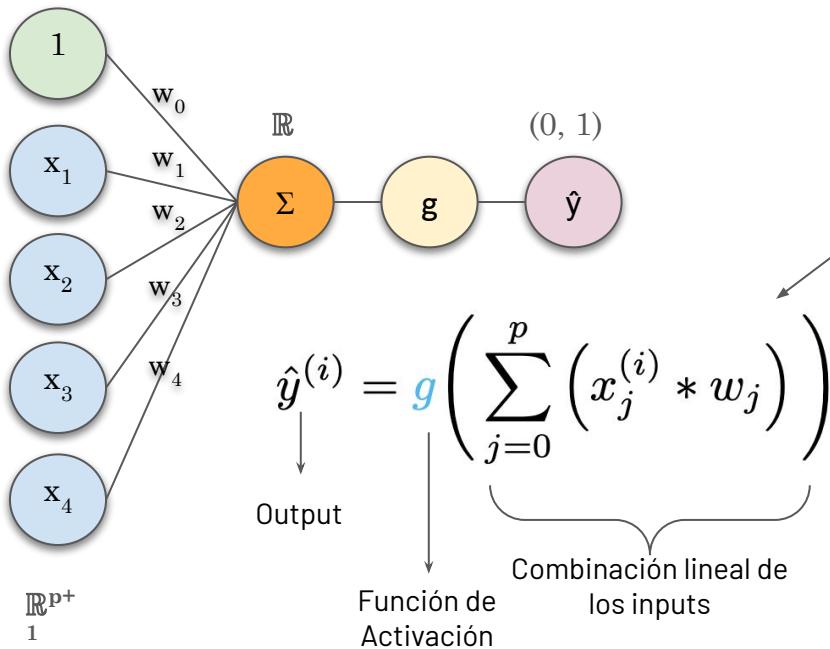
    def _pred(w):
        return _sigmoid(X_train_ext @ w)

    def _sigmoid(z):
        return 1 / (1 + np.exp(-z))

    zeros = np.zeros(X_train_ext.shape[1])
    self.w = descenso_gradiente(cost_binary_ce, grad_cost_binary_ce, z_init=zeros, alpha=0.01)
```

# Redes neuronales

# Redes neuronales simples

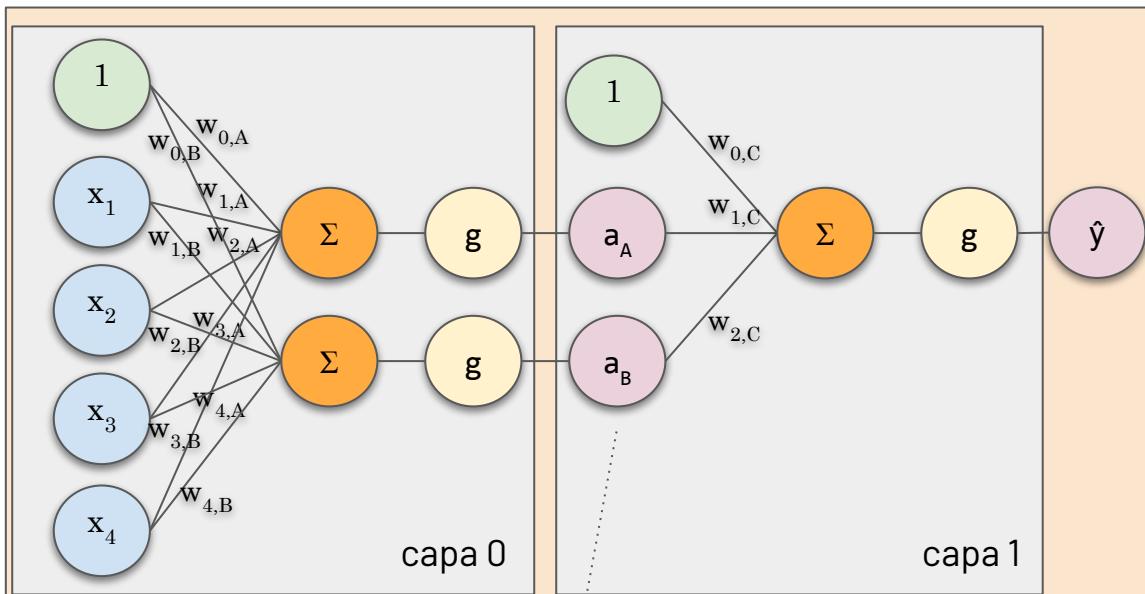


La red neuronal más sencilla es la conocida como "Perceptrón Simple" ([Rosenblatt, 1959](#)).

El Perceptrón Simple tiene la pinta que estuvimos analizando.

Tanto la regresión logística como la regresión lineal son instancias del perceptrón simple en donde **g** es la función **sigmoidea** (caso logística) o la función **identidad** (caso reg. lineal).

# Redes neuronales multicapa

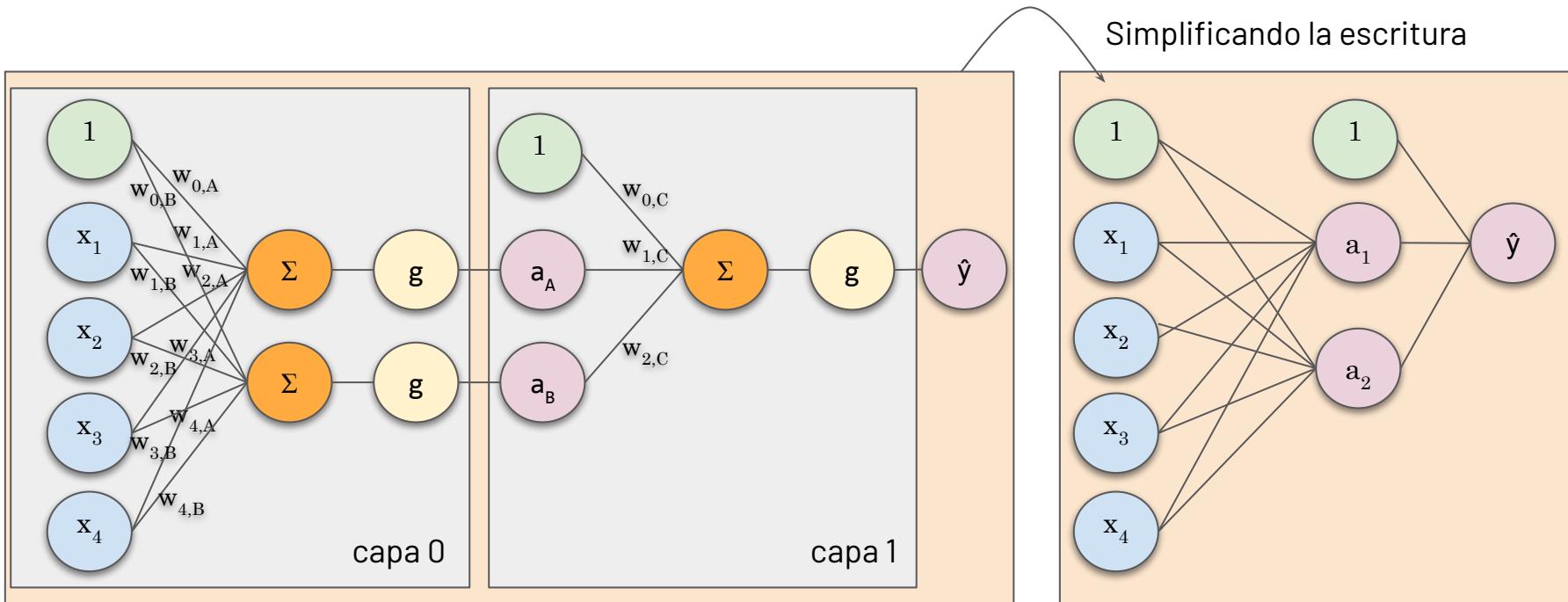


Una **red multicapa** genera **representaciones intermedias** que serán inputs de las capas sucesivas. En este ejemplo, la primera capa tiene dimensión 4 y la segunda capa, dimensión 2.

**Ejercicio,** ¿Cuánto valdrá  $\hat{y}$  si todo vale 0 salvo lo siguiente?:

- $x_4 = 1$
- $g(z) = z$  (*func act. identidad*)
- $w_{0,A} = -3$
- $w_{4,B} = 0.5$
- $w_{0,C} = 1$
- $w_{1,C} = 2$
- $w_{2,C} = 4$

# Redes neuronales multicapa



Una **red multicapa** genera **representaciones intermedias** que serán inputs de las capas sucesivas. En este ejemplo, la primera capa tiene dimensión 4 y la segunda capa, dimensión 2.

# Redes neuronales multicapa

Capa "oculta" (no tenemos output esperado contra el cual comparar)

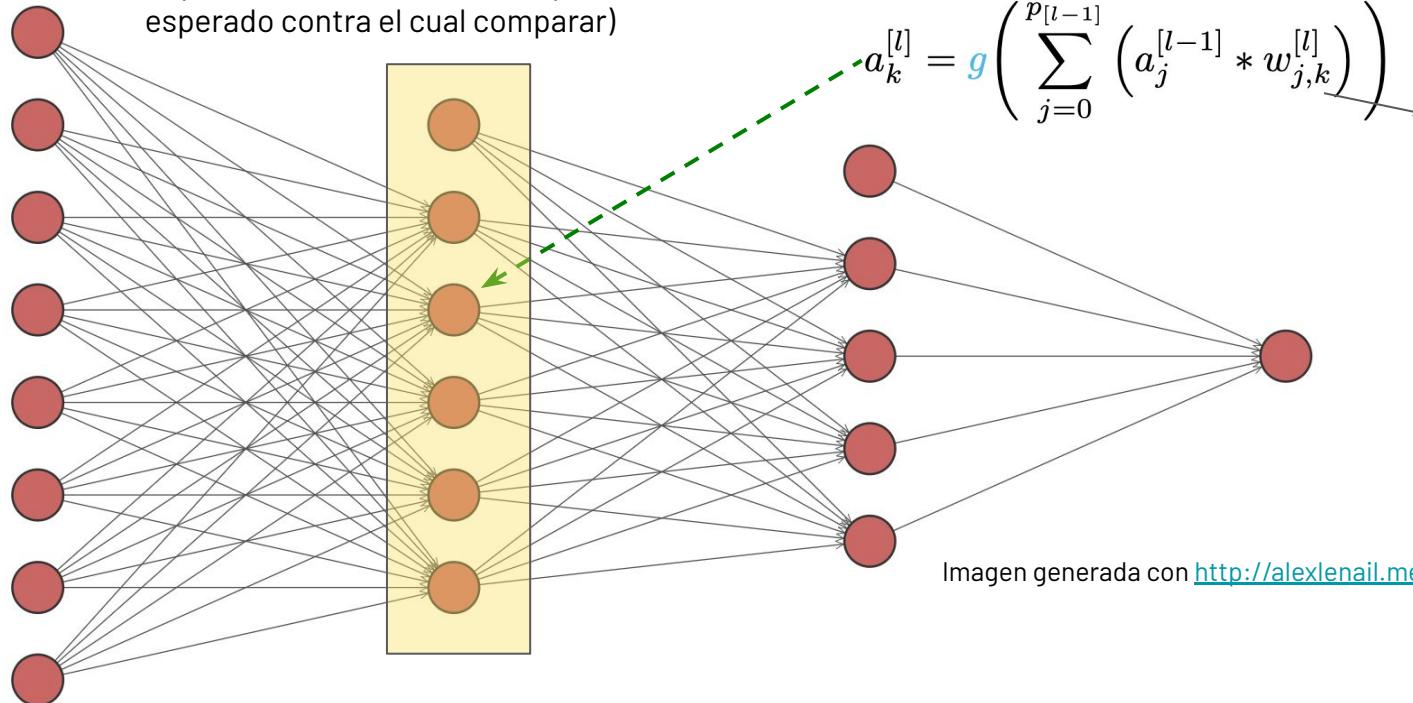


Imagen generada con <http://alexlenail.me/NN-SVG/index.html>

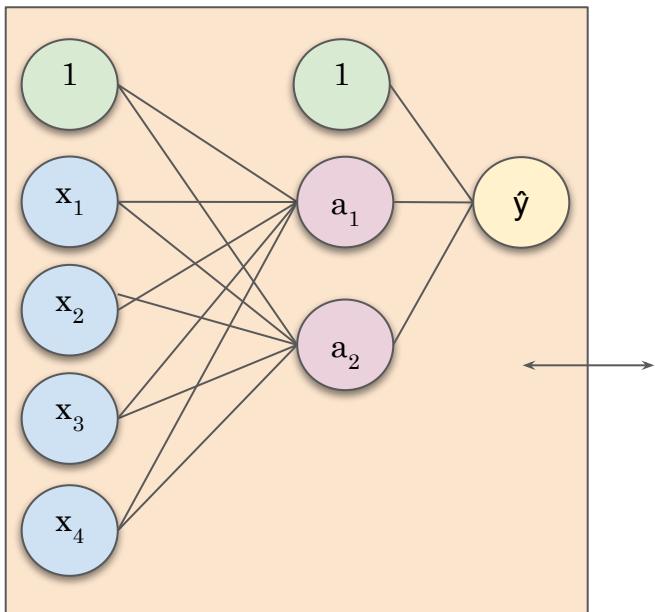
Capa de entrada  
 $\in \mathbb{R}^8$  (input en  $\mathbb{R}^7$ )

Capa oculta  
 $\in \mathbb{R}^6$

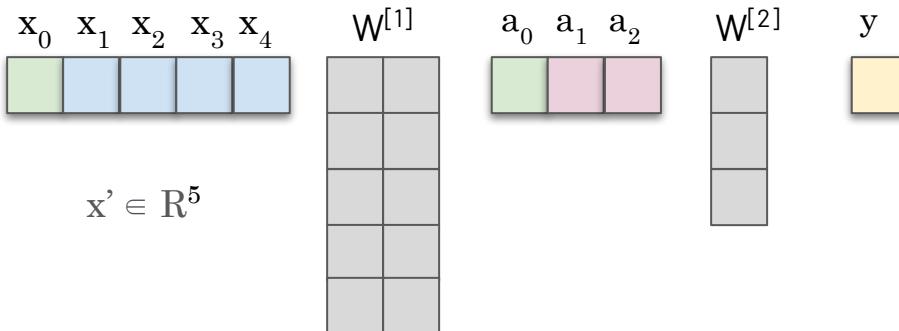
Capa oculta  
 $\in \mathbb{R}^5$

Capa de salida  
 $\in \mathbb{R}^1$

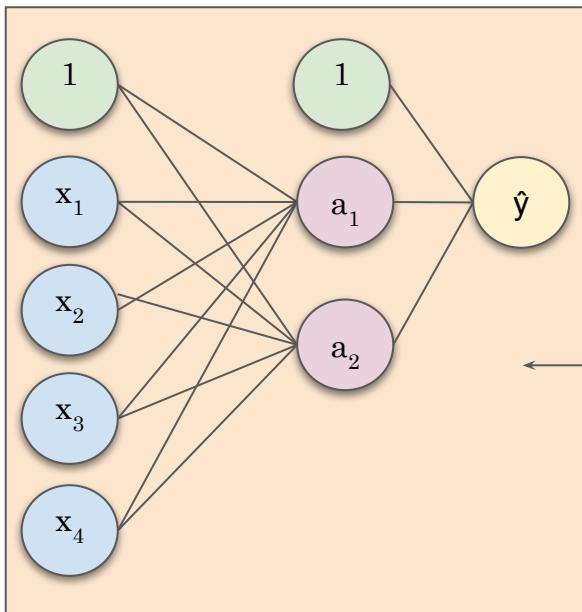
# Redes neuronales multicapa



Para cada capa, ahora tenemos una Matriz de pesos  $W^{[l]}$ .



# Redes neuronales multicapa



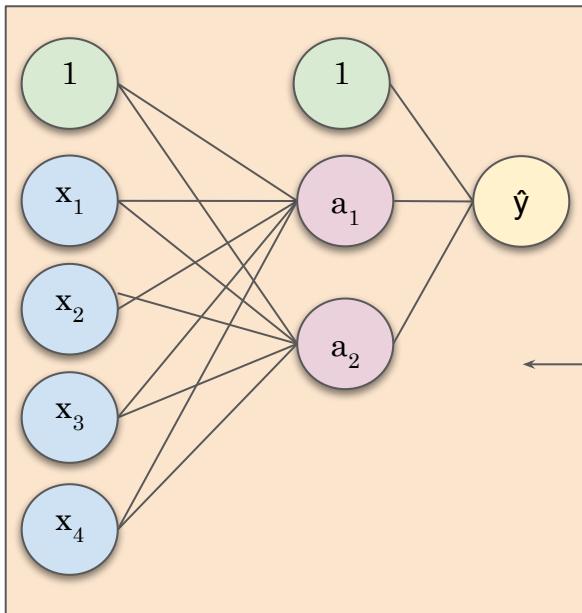
Para cada capa, ahora tenemos una Matriz de pesos  $W^{[l]}$ . Esta forma matricial también muestra que **agregar instancias** no cambia la dimensión de los pesos

$$\begin{matrix} & x_0 & x_1 & x_2 & x_3 & x_4 \\ x^{(0)} & \text{green} & \text{blue} & \text{blue} & \text{blue} & \text{blue} \\ x^{(1)} & \text{green} & \text{blue} & \text{blue} & \text{blue} & \text{blue} \\ x^{(2)} & \text{green} & \text{blue} & \text{blue} & \text{blue} & \text{blue} \end{matrix}$$
$$X' \in \mathbb{R}^{3 \times 5}$$

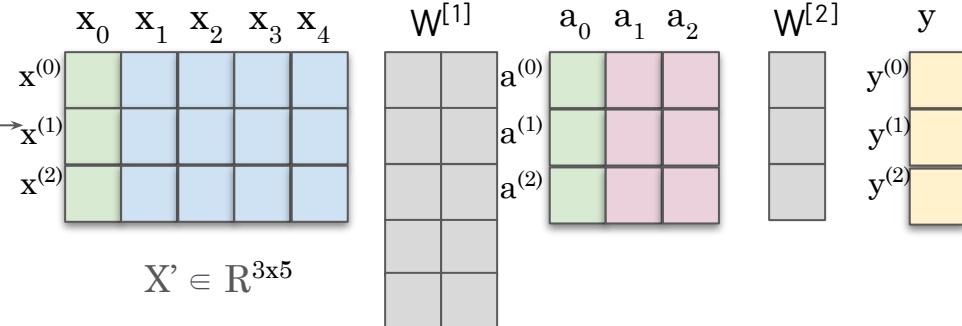
$$\begin{matrix} & a_0 & a_1 & a_2 \\ a^{(0)} & \text{green} & \text{pink} & \text{pink} \\ a^{(1)} & \text{green} & \text{pink} & \text{pink} \\ a^{(2)} & \text{green} & \text{pink} & \text{pink} \end{matrix}$$

$$\begin{matrix} & y \\ y^{(0)} & \text{yellow} \\ y^{(1)} & \text{yellow} \\ y^{(2)} & \text{yellow} \end{matrix}$$

# Redes neuronales multicapa



Para cada capa, ahora tenemos una Matriz de pesos  $W^{[l]}$ . Esta forma matricial también muestra que **agregar instancias** no cambia la dimensión de los pesos

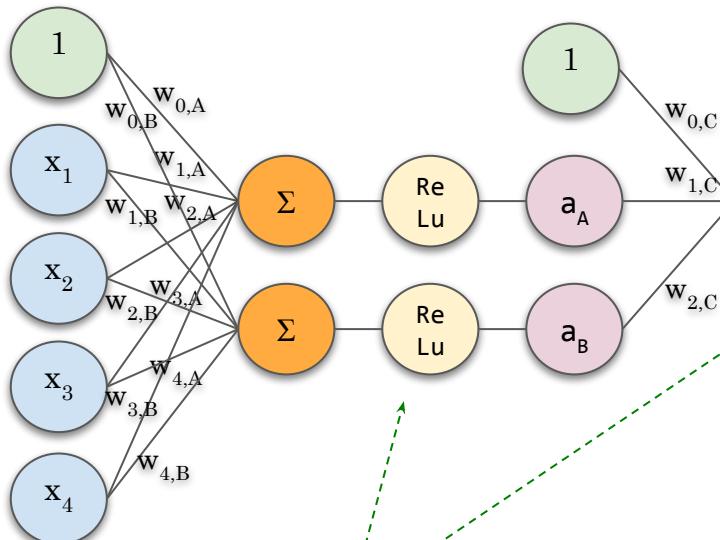


**Ejercicio:** rehacer estos dos diagramas y completar las dimensiones faltantes para el caso de un dataset X que tiene 4 instancias con 2 atributos ( $\text{mts}^2$  y distancia obelisco) para predecir el precio.

$X' \in \mathbb{R}^{? \times ?}$ ,  $W^{[1]} \in \mathbb{R}^{? \times 2}$ ,  $A^{[1]} \in \mathbb{R}^{? \times ?}$ ,  $W^{[2]} \in \mathbb{R}^{? \times 1}$ ,  $A^{[2]} \in \mathbb{R}^{? \times ?}$ ,  $W^{[3]} \in \mathbb{R}^{? \times 1}$ ,  $Y \in \mathbb{R}^{? \times 1}$

# Redes neuronales multicapa

## Función de activación

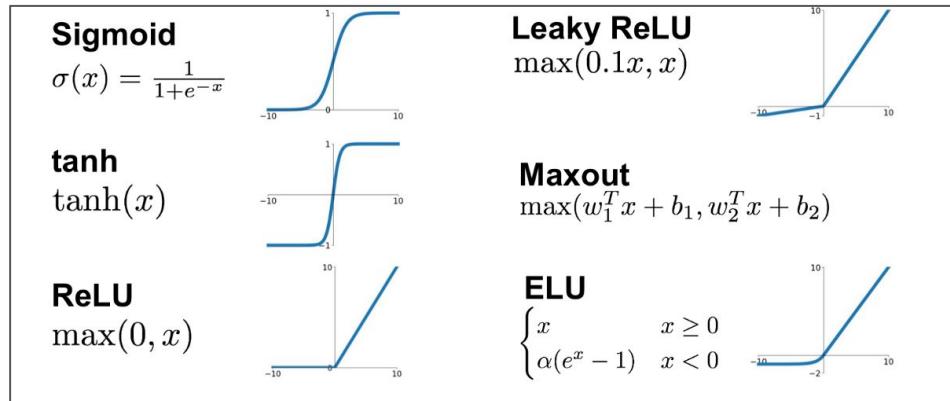


En general, se utiliza la misma función de activación en todas las capas intermedias (en este caso sólo una capa intermedia). Pero la de la última capa depende del tipo de problema.

Además de tener una función no lineal como última operación en el caso de clasificación, es común (necesario) **introducir no linealidades** entre las distintas capas del modelo (tanto para clasificación como para regresión).

Ver  
<https://dashee87.github.io/deep%20learning/visualizing-activation-functions-in-neural-networks/>

## Funciones de activación



Fuente <https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092>

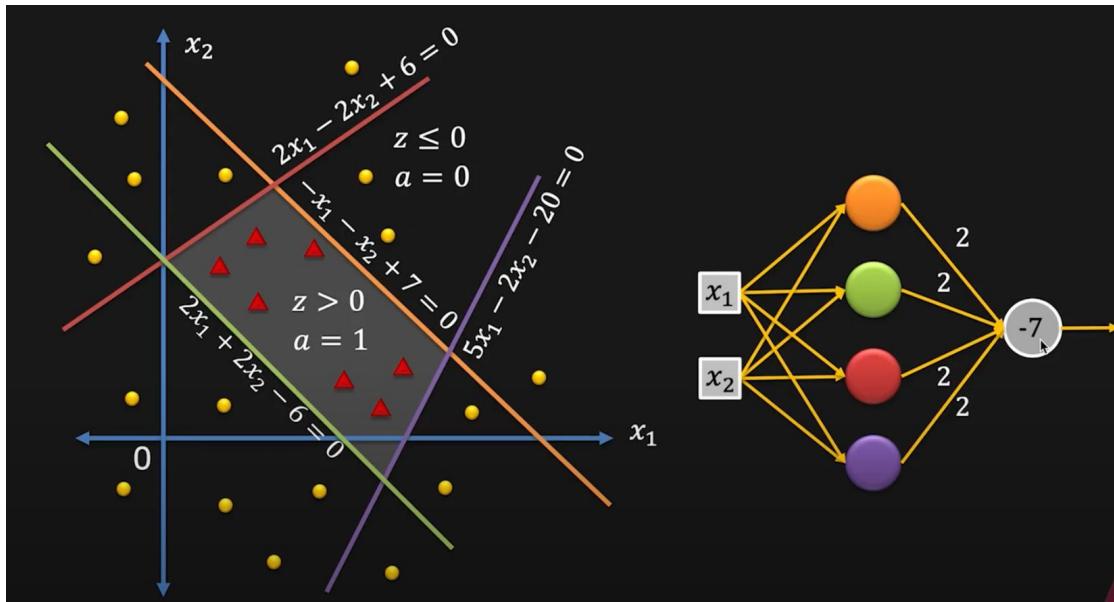
# Redes neuronales multicapa

[PDF] Kolmogorov's mapping neural network existence theorem

R Hecht-Nielsen - Proceedings of the international conference on ..., 1987 - cs.uwaterloo.ca

An improved version of Kolmogorov's powerful 1957 theorem concerning the representation of arbitrary continuous functions from the  $n$ -dimensional cube to the real numbers in terms of one dimensional continuous functions is reinterpreted to yield an existence theorem for mapping neural networks.

☆ Guardar ⌂ Citar Citado por 2088 Artículos relacionados Las 4 versiones ☰



Veamos <https://www.youtube.com/watch?v=torNuKNLwBE>  
y <https://playground.tensorflow.org/>

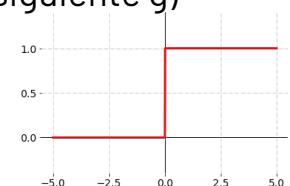
## ¿Para qué multicapa?

Una red neuronal multicapa con funciones de activación no lineales permiten capturar patrones complejos en el espacio de atributos:

## Teorema de aproximación universal

(este ejemplo usa la siguiente g)

$$g(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$



# Redes neuronales profundas

cuando una red tiene muchas capas ( $> 2$  en general) la llamamos "red profunda"

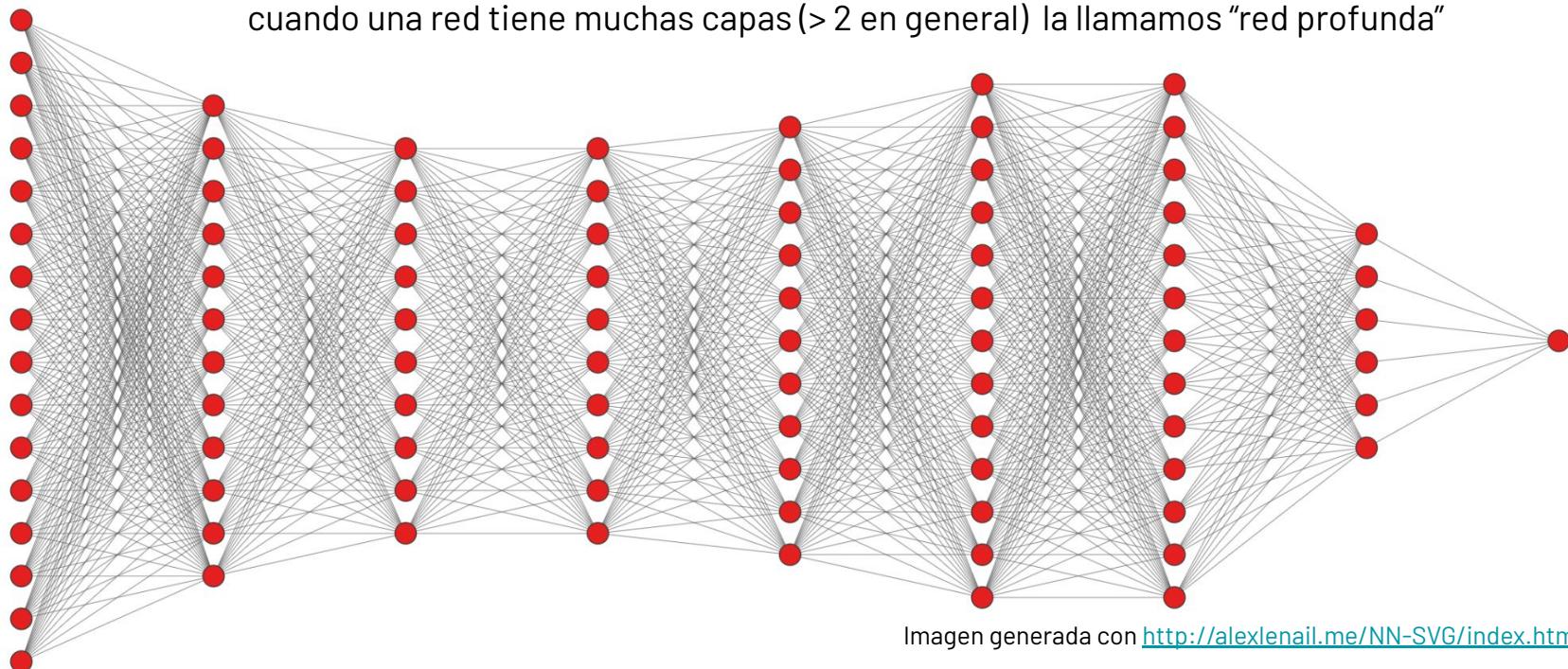


Imagen generada con <http://alexlenail.me/NN-SVG/index.html>



# Entrenamiento

(Ahora vienen las cuentas)



# Descenso por el Gradiente en NNs

Una vez definida la arquitectura de la red y la función de costo/pérdida (mse / cross-entropy) la idea es **aprender** los pesos por descenso por el gradiente como hicimos en regresión.

## Método de descenso por el gradiente

1. Inicializar pesos  $W$ , elegir *learning rate*  $\eta$
2. Durante  $N$  epochs o algún criterio de convergencia:
  - a. Calcular la salida  $h_w$  para todas las muestras
  - b. Calcular la función de pérdida  $L(h_w(x), y)$
  - c. Para los pesos aprendibles  $W$  computar el gradiente de la función de costo  $\nabla_w J_{X,y}(W)$
  - d. Actualizar los pesos:

$$W^{(i+1)} := W^{(i)} - \eta \nabla_W J_{X,y}(W)$$

**Forward pass**

**Backward pass**

# Descenso por el Gradiente en NNs

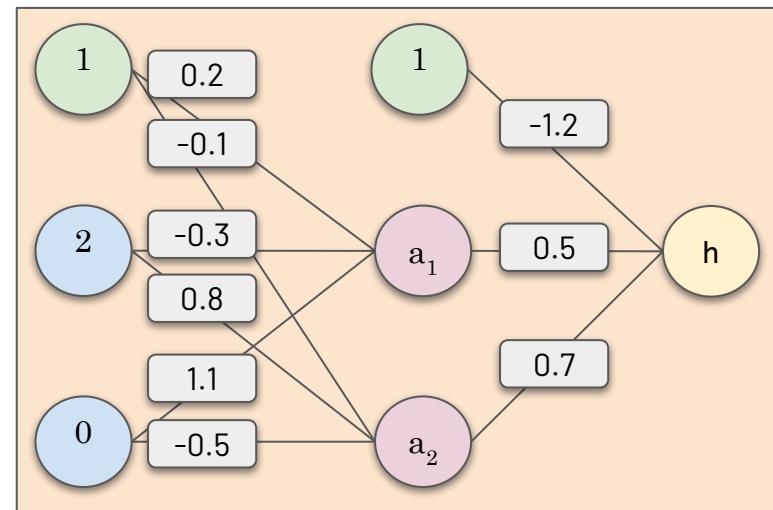
## Método de descenso por el gradiente

1. Inicializar pesos  $W$ , elegir *learning rate*  $\eta$
2. Durante  $N$  epochs o algún criterio de convergencia:
  - a. Calcular la salida  $h_w$  para todas las muestras
  - b. Calcular la función de pérdida  $L(h_w(x), y)$
  - c. Para cada peso aprendible  $W$  computar el gradiente de la función de costo  $\nabla_w J_{X,y}(W)$
  - d. Actualizar los pesos:

$$W^{(i+1)} := W^{(i)} - \eta \nabla_W J_{X,y}(W)$$

## Forward pass

Las predicciones fluyen hacia adelante a través de la red. Calculamos el error.



# Descenso por el Gradiente en NNs

## Método de descenso por el gradiente

1. Inicializar pesos  $W$ , elegir *learning rate*  $\eta$
2. Durante  $N$  epochs o algún criterio de convergencia:
  - a. Calcular la salida  $h_w$  para todas las muestras
  - b. Calcular la función de pérdida  $L(h_w(x), y)$
  - c. Para cada peso aprendible  $W$  computar el gradiente de la función de costo  $\nabla_w J_{X,y}(W)$
  - d. Actualizar los pesos:

$$W^{(i+1)} := W^{(i)} - \eta \nabla_W J_{X,y}(W)$$

### Backward pass

Tenemos que computar los gradientes propagando los errores hacia atrás, desde el final al principio.

Para eso tenemos que poder computar:

$$\nabla_W J_{X,y}(W)$$



$$\nabla_W J_{X,y}(W)$$

Una red neuronal profunda define una función/modelo de la pinta:

$$h_W(x^{(i)}) = h^{[L]} \left( \dots h^{[2]} \left( h^{[1]}(x^{(i)}) \right) \right)$$

La función de costo es la función de pérdida vista como función de los parámetros:

$$J_{X,y}(W) = \frac{1}{n} \sum_{i=1}^n L(h_W(x^{(i)}), y^{(i)})$$

Necesitamos  $\nabla_W J_{X,y}(W)$  para poder aplicar **descenso por gradiente**.

**¿Cómo podemos hacerlo?**

$$\nabla_W J_{X,y}(W)$$

## ¿Cómo podemos hacerlo?

- **Derivada analítica** (derivar a mano, escribir el código) – una pesadilla.
- **Diferenciación numérica** (diferencias finitas, metnum) – puede introducir errores de redondeo en el proceso de discretización y cancelación. Además, lento.
- **Diferenciación simbólica** (obtener la expresión matemática que describe a la derivada automáticamente) – explota rápidamente por el tamaño de la expresión generada (“expression swell”). Además, lento.

Diferenciación automática.

Construcción de un grafo computacional + definir reglas para primitivas + regla de la cadena con programación dinámica.

### Diferenciación simbólica

$$\begin{array}{ll} \frac{64x(1-x)(1-2x)^2}{(1-8x+8x^2)^2} & \frac{128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2}{(1-8x+8x^2)^3} \\ f(x) & f'(x) \end{array}$$

# Backpropagation y la regla de la cadena

**Backpropagation** es el procedimiento que calcula de **forma eficiente** los **gradientes de la función de costos** a partir del concepto de “la regla de la cadena” en redes neuronales.

la derivada de una función compuesta ( $f \circ g)(p)$  se puede descomponer el “encadenamiento” de multiplicaciones de la pinta:

$$\frac{\partial f(g(p))}{\partial p} = \frac{\partial f(u)}{\partial u} * \frac{\partial g(p)}{\partial p} \quad \text{en donde } g(p) = u$$

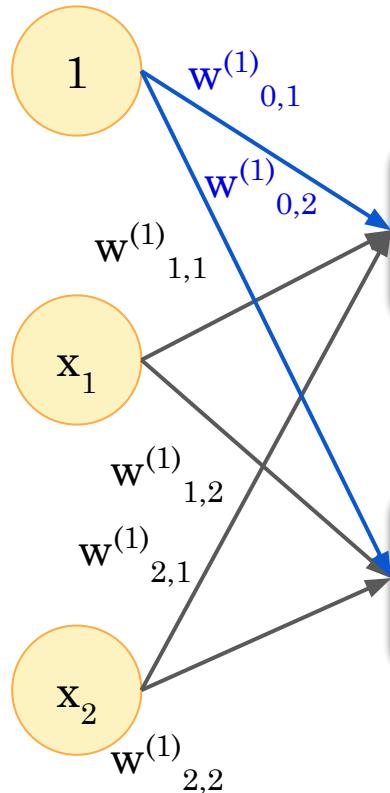
Ejemplo, cuál es la derivada de  $e^{(\sin(p))}$  respecto a  $p$ ?

$$\frac{\partial e^{\sin(p)}}{\partial p} = \frac{\partial e^u}{\partial u} * \frac{\partial \sin(p)}{\partial p} = (e^u) * \cos(p) = (e^{\sin(p)}) * \cos(p)$$

$$u = \sin(p)$$

Veamos un ejemplo más detallado pero con una NN paso por paso...

# Backpropagation

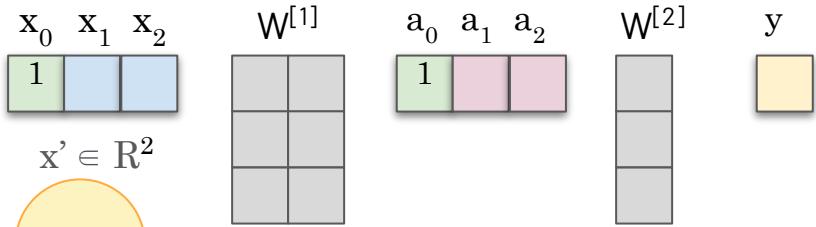


$$g(z^{(1)}_1) = a^{(1)}_1$$

$$g(1 \cdot w^{(1)}_{0,1} + x_1 \cdot w^{(1)}_{1,1} + x_2 \cdot w^{(1)}_{2,1})$$

$$g(z^{(1)}_2) = a^{(1)}_2$$

$$g(1 \cdot w^{(1)}_{0,2} + x_1 \cdot w^{(1)}_{1,2} + x_2 \cdot w^{(1)}_{2,2})$$



$$x' \in \mathbb{R}^2$$

1

$$w^{(2)}_0$$

$$w^{(2)}_1$$

$$w^{(2)}_2$$

$$h_w(z^{(2)}_1)$$

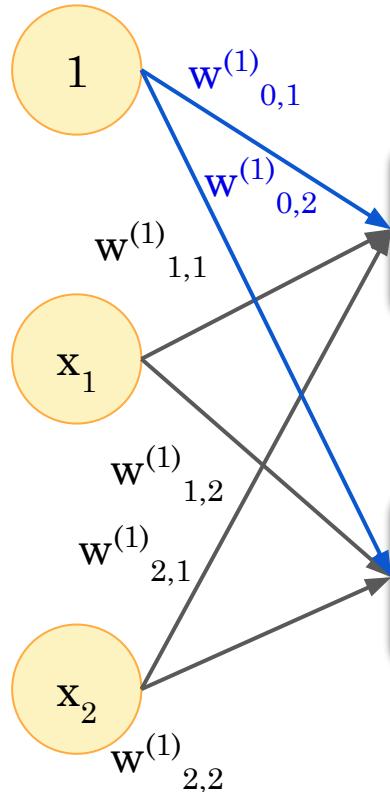
Loss  
( $h_w(x), y$ )

$$L(h_w(\mathbf{x}), y) = \frac{1}{2} \cdot (h_w(\mathbf{x}) - y)^2$$

**Nota<sub>1</sub>** Vamos a utilizar intercambiablemente  $L$  en vez de  $J$  porque vamos a pensarla a nivel de una instancia.

**Nota<sub>2</sub>** Vamos a usar como pérdida el MSE

# Forward pass

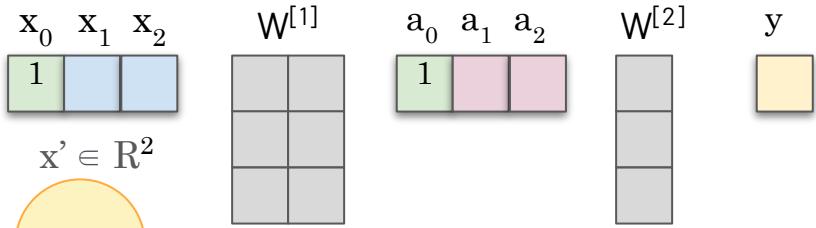


$$g(z^{(1)}_1) = a^{(1)}_1$$

$$g(1 \cdot w^{(1)}_{0,1} + x_1 \cdot w^{(1)}_{1,1} + x_2 \cdot w^{(1)}_{2,1})$$

$$g(z^{(1)}_2) = a^{(1)}_2$$

$$g(1 \cdot w^{(1)}_{0,2} + x_1 \cdot w^{(1)}_{1,2} + x_2 \cdot w^{(1)}_{2,2})$$



**z:** pre activaciones  
**a:** activación de las neuronas  
**g:** función de activación  
**h:**  $\sigma(z^{(2)})$

$$w^{(2)}_0$$

$$w^{(2)}_1$$

$$w^{(2)}_2$$

$$h_w(z^{(2)}_1)$$

$$\text{Loss}(h_w(x), y)$$

$$L(h_w(x), y) = \frac{1}{2} \cdot (h_w(x) - y)^2$$

**Nota<sub>1</sub>** Vamos a utilizar intercambiablemente  $L$  en vez de  $J$  porque vamos a pensarla a nivel de una instancia.

**Nota<sub>2</sub>** Vamos a usar como pérdida el MSE

# Forward pass

$$z_1^{(1)} = w_{0,1}^{(1)} \cdot 1 + w_{1,1}^{(1)} \cdot x_1 + w_{2,1}^{(1)} \cdot x_2$$

$$z_2^{(1)} = w_{0,2}^{(1)} \cdot 1 + w_{1,2}^{(1)} \cdot x_1 + w_{2,2}^{(1)} \cdot x_2$$

$$a_1^{(1)} = g(z_1^{(1)})$$

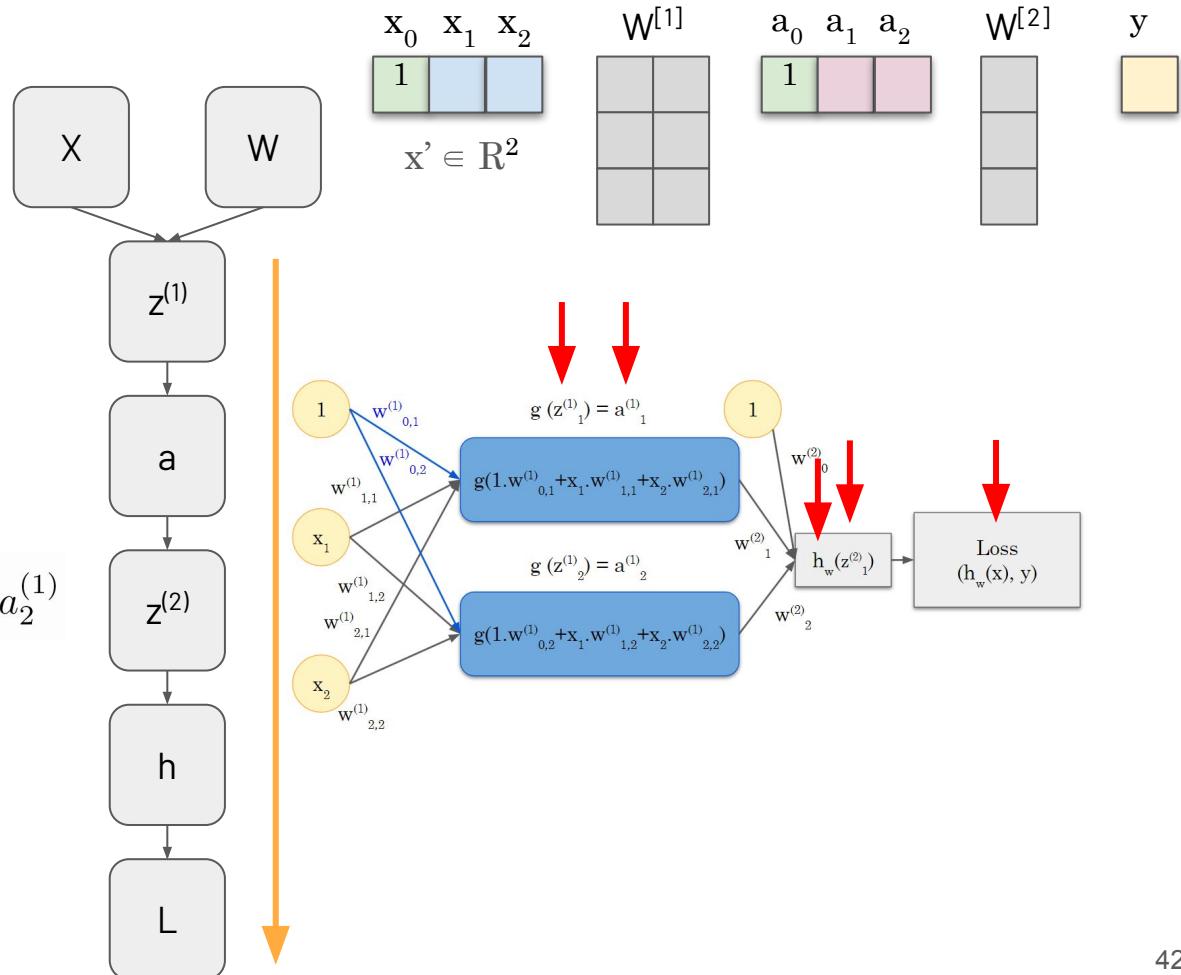
$$a_2^{(1)} = g(z_2^{(1)})$$

$$z_1^{(2)} = w_{0,2}^{(1)} \cdot 1 + w_{1,2}^{(1)} \cdot a_1^{(1)} + w_{2,2}^{(1)} \cdot a_2^{(1)}$$

$$z^{(2)} = \mathbf{a}^{(1)} W^{(2)}$$

$$h_w(\mathbf{x}) = g(z^{(2)})$$

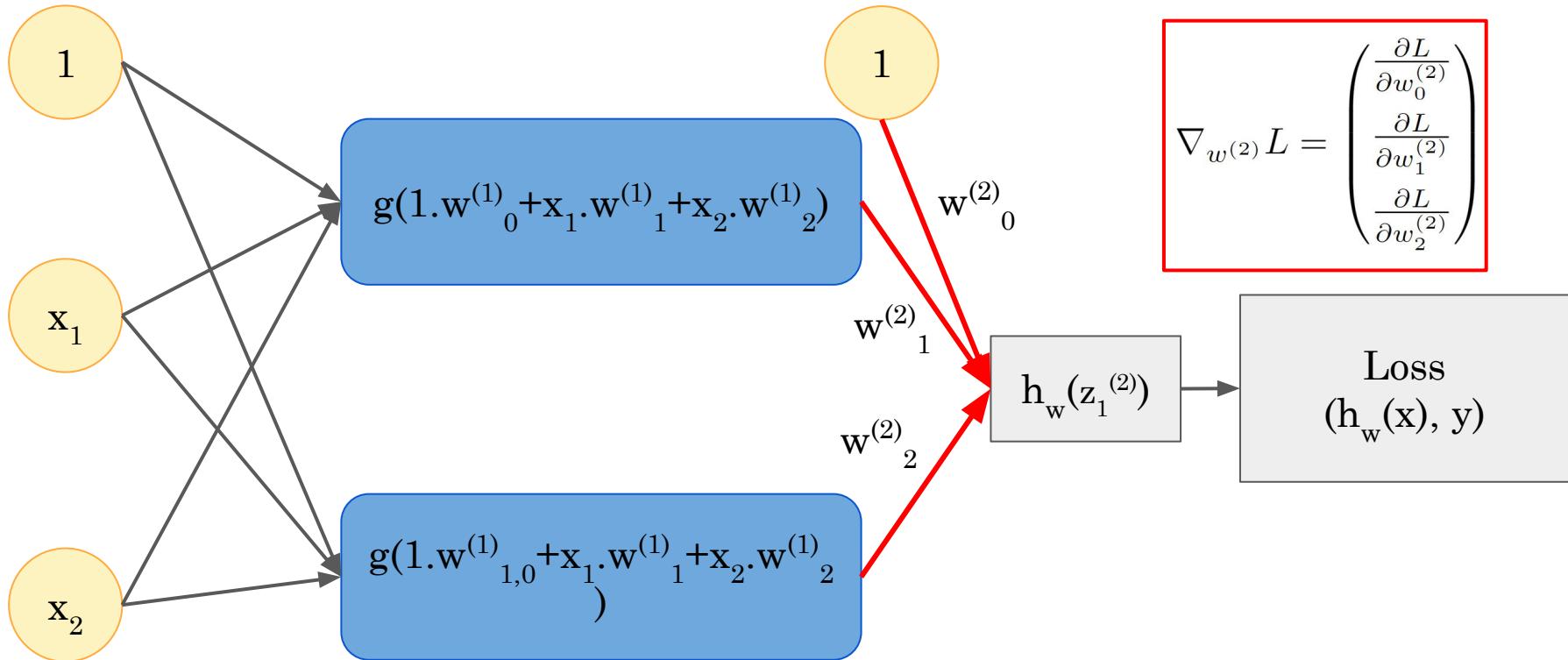
$$L(h_w(\mathbf{x}), y) = \frac{1}{2} \cdot (h_w(\mathbf{x}) - y)^2$$



# Backward propagation

Última capa

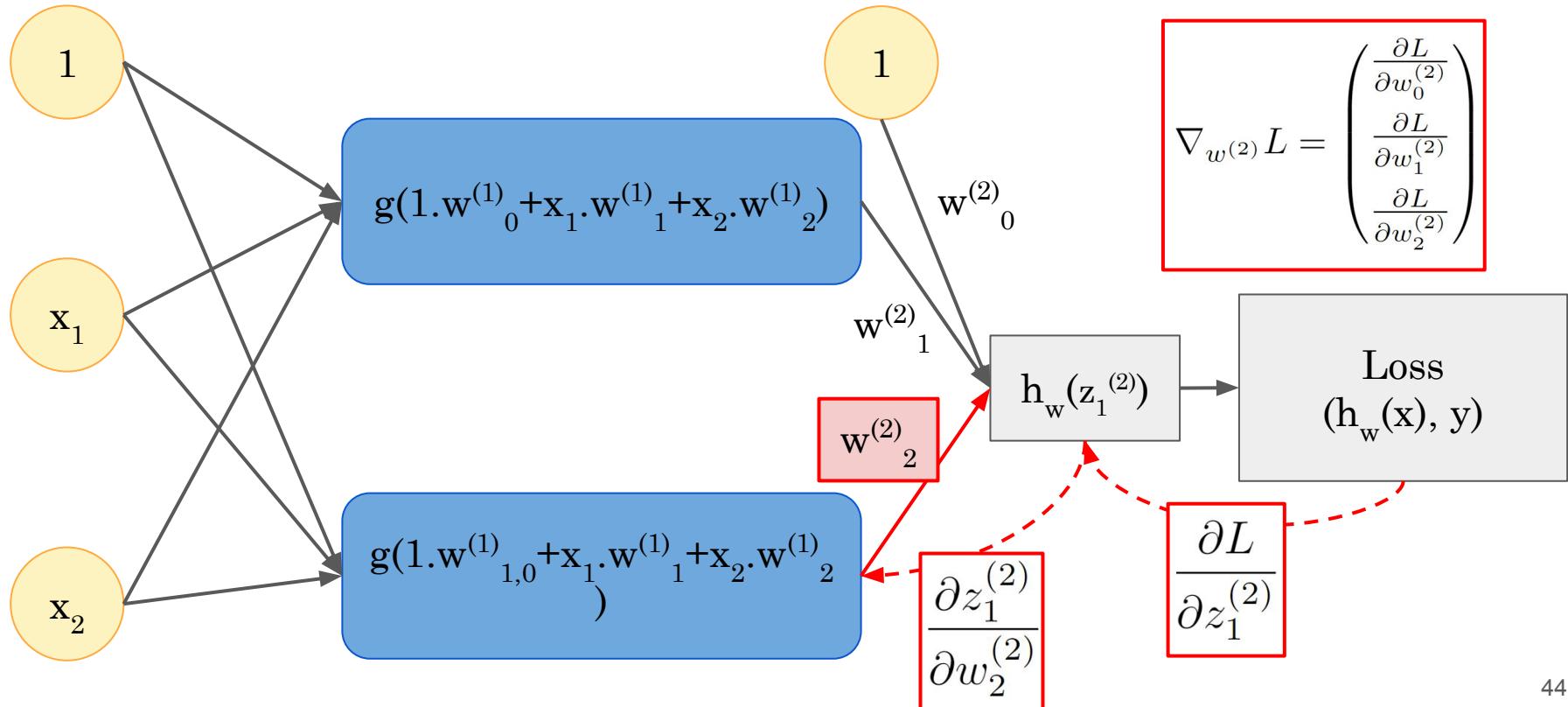
Vamos a calcular los gradientes de atrás para adelante, primero para la última capa  $W^{(2)}$



# Backward propagation

Última capa

Vamos a calcular los gradientes de atrás para adelante, primero para la última capa  $\mathbf{W}^{(2)}$

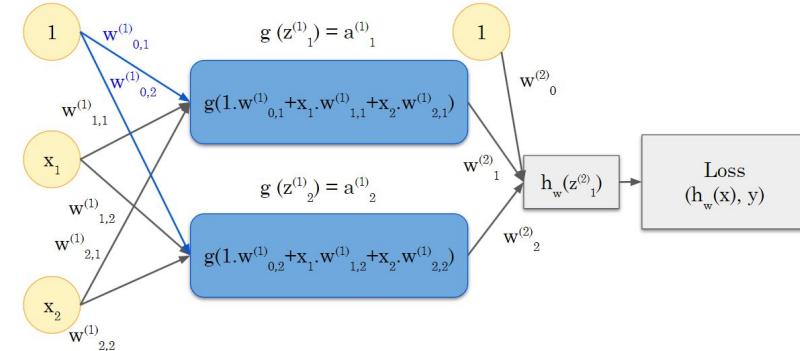


# Backward propagation

## Última capa

Entonces para actualizar los pesos de la última capa quedará como:

$$\frac{\partial L}{\partial w_2^{(2)}} = \frac{\partial L}{\partial z_1^{(2)}} \cdot \frac{\partial z_1^{(2)}}{\partial w_2^{(2)}}$$



Si llamamos **delta** a la tasa/error sobre el costo producida por cambios en la preactivación

$$\delta_j^{(2)} := \frac{\partial L}{\partial z_j^{(2)}}$$

Reemplazando y derivando la preactivación con respecto al peso w



$$\frac{\partial L}{\partial w_2^{(2)}} = \delta_1^{(2)} a_2^{(1)}$$

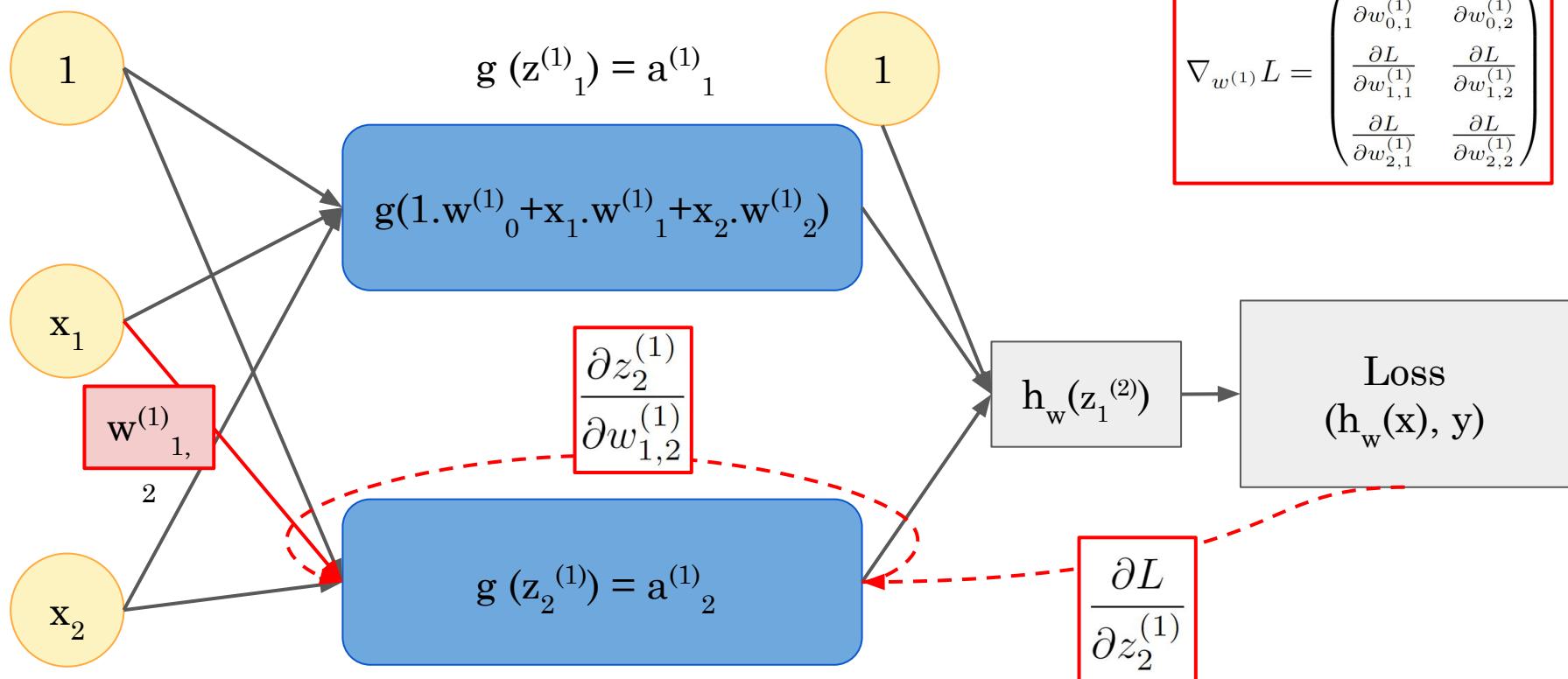
Respectivamente para cada peso en la última capa.

Pequeño ejercicio para el hogar, calcular la expresión de delta en la última capa.

# Backward propagation

Capa oculta

Ahora seguimos con los pesos asociados a la capa oculta  $W^{(1)}$



# Backward propagation

Capa oculta

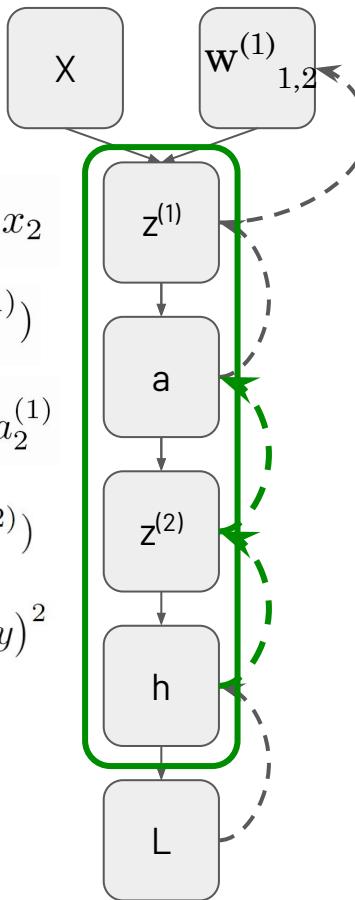
$$z_2^{(1)} = w_{0,2}^{(1)} \cdot 1 + w_{1,2}^{(1)} \cdot x_1 + w_{2,2}^{(1)} \cdot x_2$$

$$a_2^{(1)} = g(z_2^{(1)})$$

$$z_1^{(2)} = w_{0,2}^{(1)} \cdot 1 + w_{1,2}^{(1)} \cdot a_1^{(1)} + w_{2,2}^{(1)} \cdot a_2^{(1)}$$

$$h_w(\mathbf{x}) = g(z^{(2)})$$

$$L(h_w(\mathbf{x}), y) = \frac{1}{2} \cdot (h_w(\mathbf{x}) - y)^2$$



$$\frac{\partial L}{\partial w_{1,2}^{(1)}} = \frac{\partial L}{\partial h_w} \cdot \frac{\partial h_w}{\partial z_1^{(2)}} \cdot \frac{\partial z_1^{(2)}}{\partial a_2^{(1)}} \cdot \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \cdot \frac{\partial z_2^{(1)}}{\partial w_{1,2}^{(1)}}$$

# Backward propagation

Capa oculta

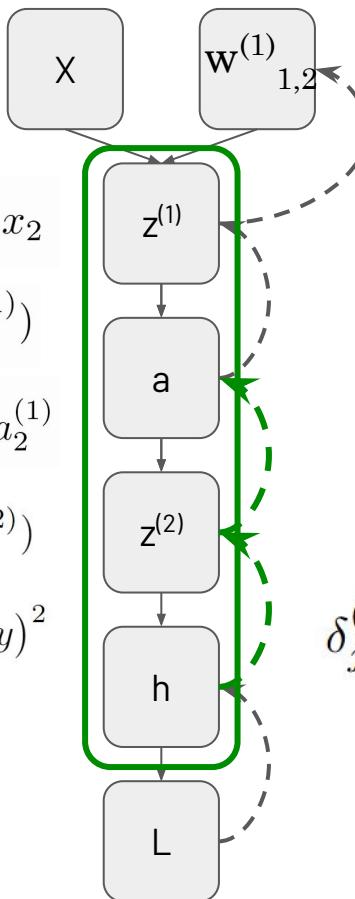
$$z_2^{(1)} = w_{0,2}^{(1)} \cdot 1 + w_{1,2}^{(1)} \cdot x_1 + w_{2,2}^{(1)} \cdot x_2$$

$$a_2^{(1)} = g(z_2^{(1)})$$

$$z_1^{(2)} = w_{0,2}^{(1)} \cdot 1 + w_{1,2}^{(1)} \cdot a_1^{(1)} + w_{2,2}^{(1)} \cdot a_2^{(1)}$$

$$h_w(\mathbf{x}) = g(z^{(2)})$$

$$L(h_w(\mathbf{x}), y) = \frac{1}{2} \cdot (h_w(\mathbf{x}) - y)^2$$



Ya lo vamos a tener  
calculado del paso anterior

$$\frac{\partial L}{\partial z_1^{(2)}} = \delta_1^{(2)}$$

$$\frac{\partial L}{\partial w_{1,2}^{(1)}} = \underbrace{\frac{\partial L}{\partial h_w} \cdot \frac{\partial h_w}{\partial z_1^{(2)}}}_{\text{green bracket}} \cdot \underbrace{\frac{\partial z_1^{(2)}}{\partial a_2^{(1)}} \cdot \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \cdot \frac{\partial z_2^{(1)}}{\partial w_{1,2}^{(1)}}}_{\text{green bracket}}$$

$$\delta_j^{(2)} := \frac{\partial L}{\partial z_j^{(2)}} \quad \delta_j^{(1)} := \frac{\partial L}{\partial z_j^{(1)}}$$

# Backward propagation

## Capa oculta

Entonces para actualizar los pesos de la capa oculta quedará como:

$$\frac{\partial L}{\partial w_{1,2}^{(1)}} = \boxed{\frac{\partial L}{\partial h_w} \cdot \frac{\partial h_w}{\partial z_1^{(2)}} \cdot \frac{\partial z_1^{(2)}}{\partial a_2^{(1)}} \cdot \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \cdot \frac{\partial z_2^{(1)}}{\partial w_{1,2}^{(1)}}}$$

$$\frac{\partial L}{\partial w_{1,2}^{(1)}} = \frac{\partial L}{\partial z_1^{(2)}} \cdot \frac{\partial z_1^{(2)}}{\partial a_2^{(1)}} \cdot \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \cdot \frac{\partial z_2^{(1)}}{\partial w_{1,2}^{(1)}}$$

$$\delta_j^{(1)} := \frac{\partial L}{\partial z_j^{(1)}}$$

$$\frac{\partial L}{\partial w_{1,2}^{(1)}} = \delta_1^{(2)} \cdot \frac{\partial z_1^{(2)}}{\partial a_2^{(1)}} \cdot \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \cdot \frac{\partial z_2^{(1)}}{\partial w_{1,2}^{(1)}}$$

$$\frac{\partial L}{\partial w_{1,2}^{(1)}} = \delta_1^{(2)} \cdot w_1^{(2)} \cdot g'(z_2^{(1)}) \cdot x_1 = \delta_2^{(1)} \cdot x_1$$

# Backward propagation

## Capa oculta

Llegamos a esta expresión

$$\frac{\partial L}{\partial w_{1,2}^{(1)}} = \delta_1^{(2)} \cdot w_1^{(2)} \cdot g'(z_2^{(1)}) \cdot x_1 = \delta_2^{(1)} \cdot x_1$$

Acomodandolo un poco para escribirlo de manera general para todos los pasos vectorialmente:

$$\frac{\partial L}{\partial w^{(1)}} = (w^{(2)})^T \cdot \delta_2^{(1)} \odot g'(z^{(1)}) \cdot x$$

**Tarea para el hogar**  
Verificar esta expresión



Producto posición a posición (Producto de Hadamard)

- **Ojo** con el *bias*, se suele escribir por separado en muchos lados (acá pensamos en vectores extendidos con 1 cuando sea necesario)

# Backpropagation pasado en limpio en nuestro ejemplo

## Cálculo de gradientes

$$\frac{\partial L}{\partial w^{(1)}} = \delta^{(1)} \cdot x$$

$$\frac{\partial L}{\partial w^{(2)}} = \delta^{(2)} \cdot a^{(1)}$$

## Paso de actualización de pesos

$$w^{(1)} := w^{(1)} - \eta \nabla_{w^{(1)}} L$$

$$w^{(2)} := w^{(2)} - \eta \nabla_{w^{(2)}} L$$

# Backpropagation generalización

Esto se puede generalizar con algunas sofisticaciones de notación pero solo con un poco de álgebra a una capa cualquiera  $L$  de un MLP.

Referencias recomendadas:

- [How the backpropagation algorithm works](#)
- [Understanding Backpropagation. A visual derivation of the equations... | by Brent Scarff | Towards Data Science](#)
- [Backpropagation calculus | DL4](#)
- Sección 5 del [capítulo 6](#) de Deep Learning de I. Goodfellow

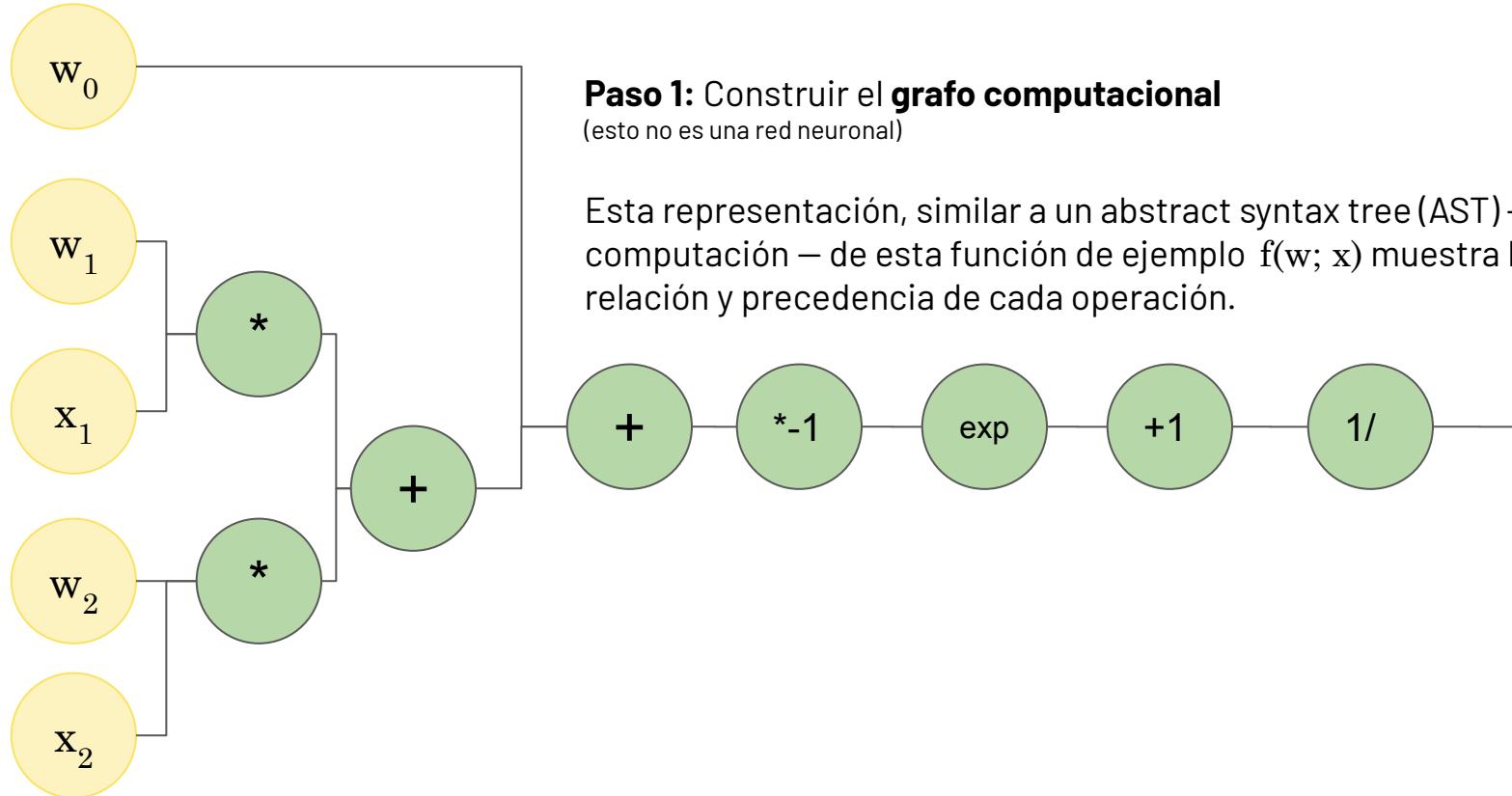
*¿Cómo se computa esto de manera eficiente algorítmicamente?*



**Diferenciación Automática  
Reversa**

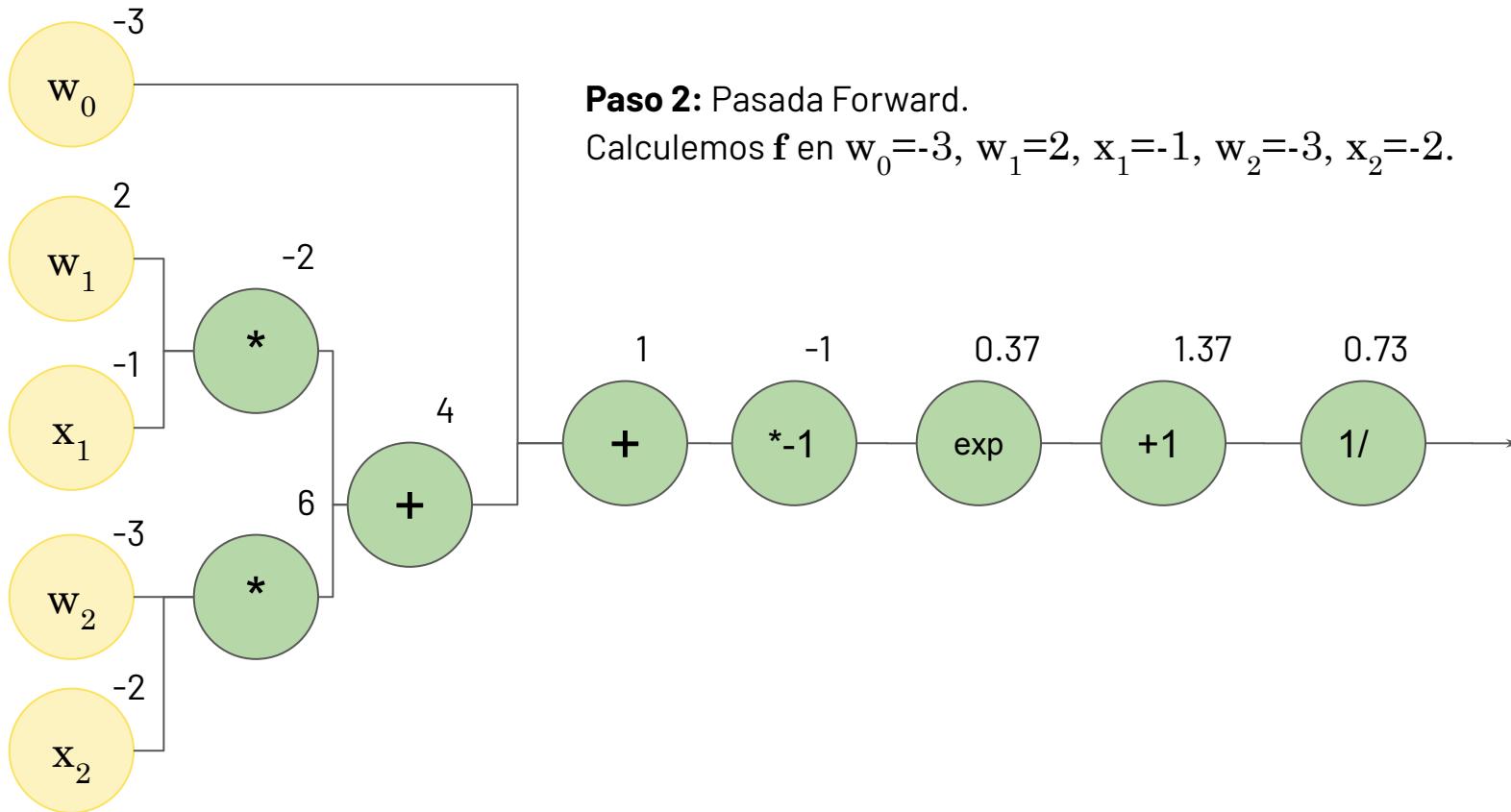
$$f(w_0, w_1, x_1, w_2, x_2) = \frac{1}{1 + e^{-(w_0 + w_1 * x_1 + w_2 * x_2)}}$$

# Dif. Automática Reversa



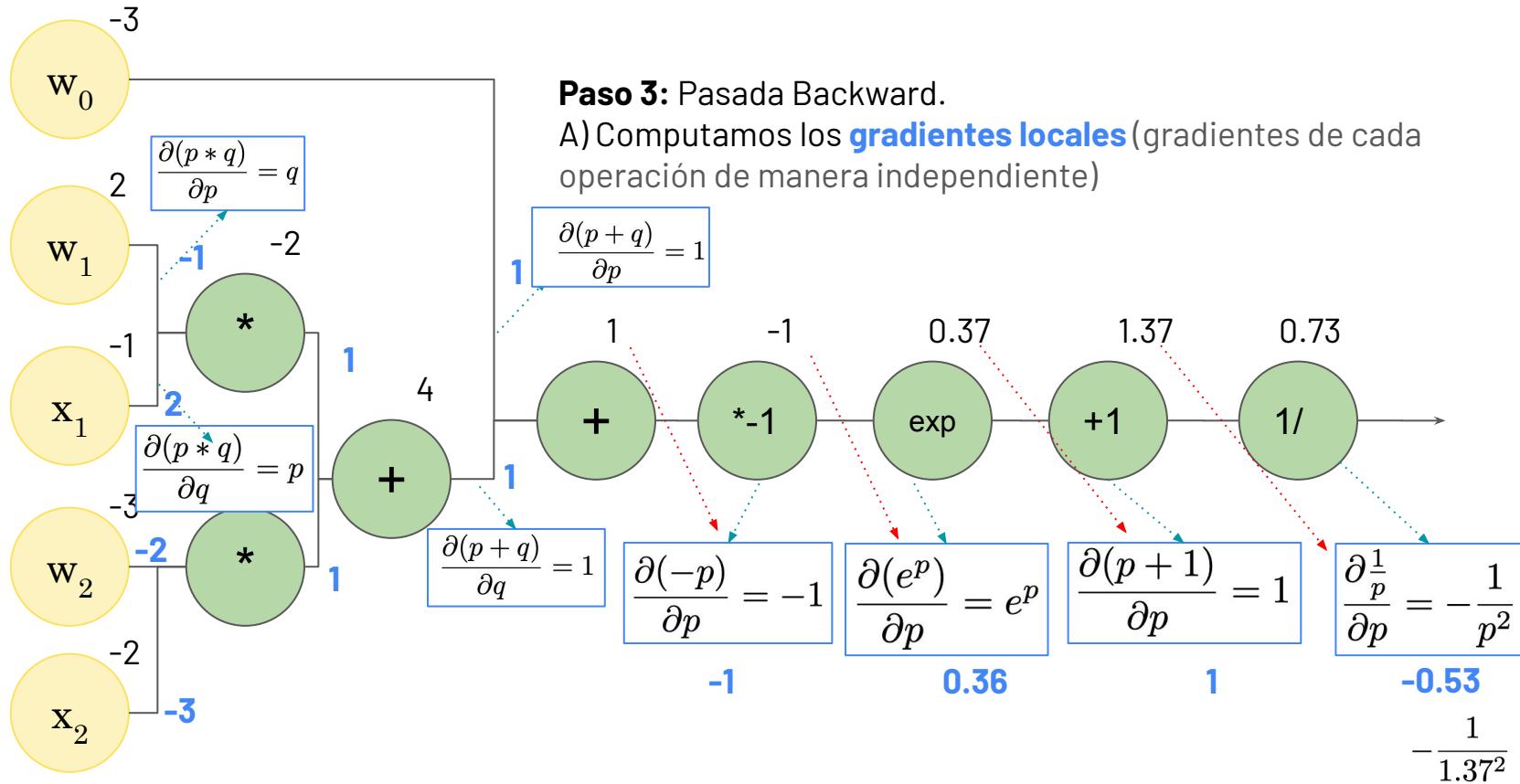
# Dif. Automática Reversa

$$f(w_0, w_1, x_1, w_2, x_2) = \frac{1}{1 + e^{-(w_0 + w_1 * x_1 + w_2 * x_2)}}$$



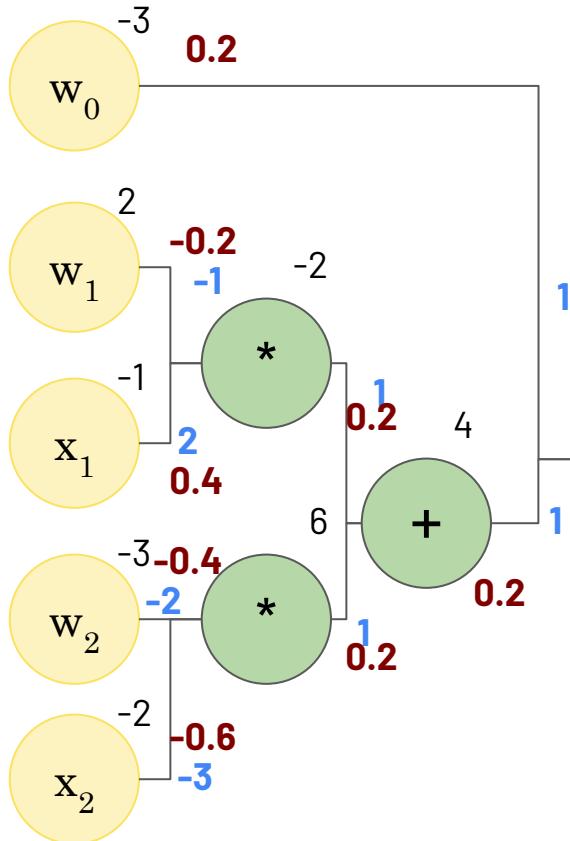
# Dif. Automática Reversa

$$f(w_0, w_1, x_1, w_2, x_2) = \frac{1}{1 + e^{-(w_0 + w_1 * x_1 + w_2 * x_2)}}$$



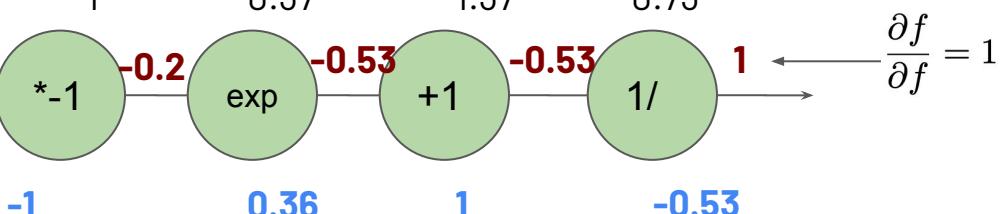
# Dif. Automática Reversa

$$f(w_0, w_1, x_1, w_2, x_2) = \frac{1}{1 + e^{-(w_0 + w_1 * x_1 + w_2 * x_2)}}$$



**Paso 3:** Pasada Backward.

B) Cálculo del **gradiente** usando la regla de la cadena (empezamos desde 1 y vamos multiplicando por los **gradientes locales**)



Resulta entonces  
 $\nabla_w f = (0.2, -0.2, -0.4)$

De paso también calculamos  
 $\nabla_x f = (0.4, -0.6)$

# Por suerte.. los frameworks hacen este trabajo

Pytorch

```
import torch

x = torch.tensor([1., -1., -2.]) # Tensor = vector en este ejemplo
w = torch.tensor([-3., 2., -3.], requires_grad=True) # pesos
z = x @ w
a = 1 / (1 + torch.exp(-z))
# Ultimas dos lineas equivalentes a
# a = torch.sigmoid(x @ w)

a.backward()
print("Gradiente con respecto a w:", w.grad)
print("Gradiente con respecto a x:", x.grad) # "No me lo da, no le
pedí que lo calcule"
```

```
Gradiente con respecto a w: tensor([ 0.1966, -0.1966, -0.3932])
Gradiente con respecto a x: None
```

(da lo mismo, errores de redondeo en el ejemplo)

¿Cómo definir funciones  
"base" propias?

```
class Exp(torch.autograd.Function):
    @staticmethod
    def forward(ctx, i):
        result = i.exp()
        ctx.save_for_backward(result)
        return result

    @staticmethod
    def backward(ctx, grad_output):
        result, = ctx.saved_tensors
        return grad_output * result

# Call the function
Exp.apply(torch.tensor(0.5, requires_grad=True))
# Outputs: tensor(1.6487, grad_fn=<ExpBackward>)
```

<https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/>

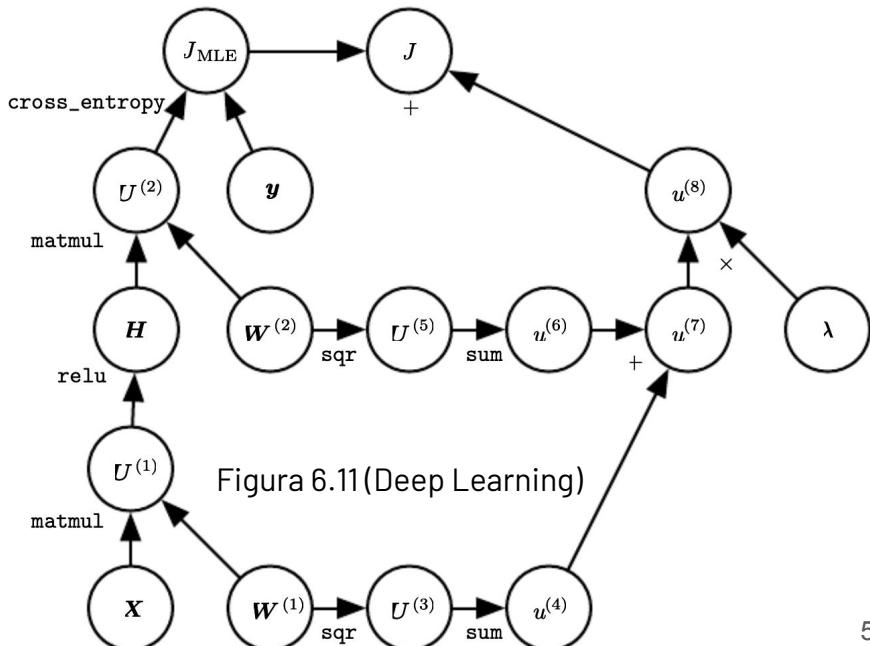
# Gradiente de la función de costo final

Una vez definida la arquitectura de la red, incluyendo la función de costo (mse / cross-entropy) e incluyendo regularizaciones y demás operaciones:

Ejemplo (CE + regularización L2)

$$J_{MLE} = \frac{1}{n} \sum_{i=1}^n \text{Multiclass\_CE}^{(i)} = \frac{1}{n} \sum_{i=1}^n \left[ -\sum_{k=1}^K y_k^{(i)} \log(\hat{h}_w^{(k)}(\mathbf{x}^{(i)})) \right]$$

$$J = J_{MLE} + \lambda \left( \sum_{i,j} \left( W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left( W_{i,j}^{(2)} \right)^2 \right)$$



- Se genera un **grafo computacional** (similar a un AST) que representa exactamente la función a minimizar.
- Cada nodo representa un cómputo elemental.
- Hacer una pasada “**forward**” en este grafo permite evaluar la red en un punto dado.
- Hacer una pasada “**backward**” en el grafo subyacente (que contiene las derivadas parciales correspondientes a cada cómputo) permite aplicar la **regla de la cadena** para encontrar la derivada en un punto.
- Utilizando estas estrategias, podemos entrenar la red haciendo **descenso por el gradiente**.

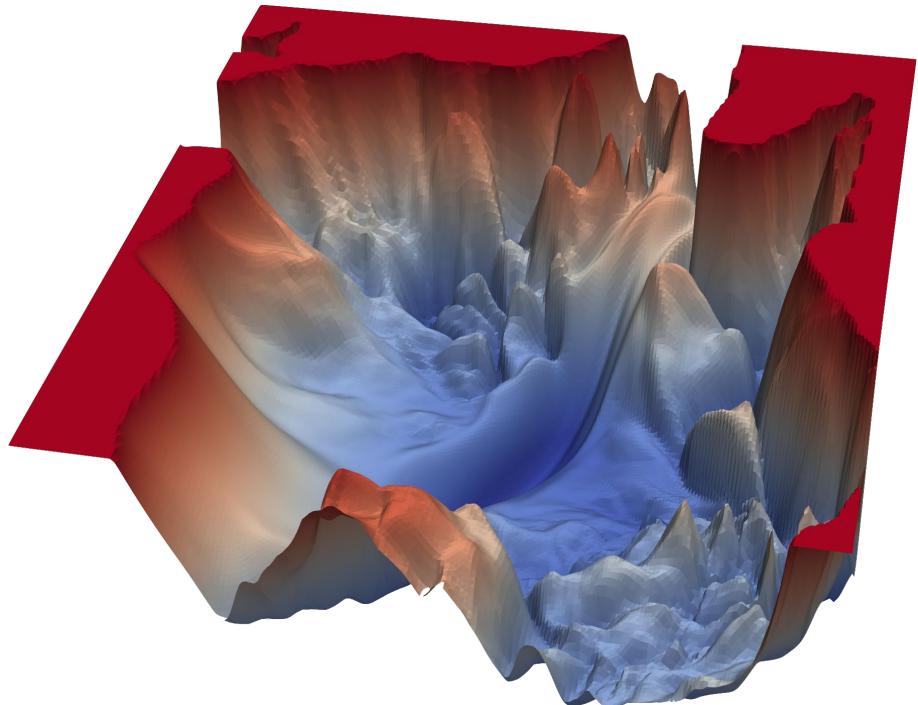
# Descenso de Gradiente en NN

Descenso de gradiente **no garantiza encontrar el mínimo global** en **funciones no convexas**.

Sin embargo, **para redes neuronales “grandes”**, la mayoría de los **mínimos locales son similares** y presentan performance similar en los datasets de tests.

La probabilidad de **encontrar mínimos locales malos decrece con el tamaño de la red**.

Focalizarse demasiado en la búsqueda del mínimo global en el dataset de entrenamiento no es muy útil en la práctica ya que solemos caer en **sobreajuste**.



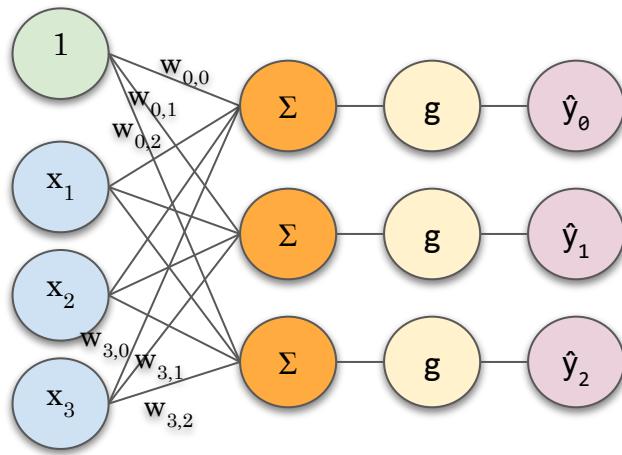
Fuente: <https://www.cs.umd.edu/~tomq/projects/landscapes/?ref=jeremyjordan.me>  
Ver también <https://losslandscape.com/> (arte, tremendo)

# Redes Neuronales Multi Output

# Redes Neuronales Multi Output

Imaginen que ahora queremos clasificar una instancia entre 3 clases distintas (ej: "perro", "gato", "pato").

Podríamos usar la siguiente arquitectura:

 $\mathbb{R}^{p+1}$  $\mathbb{R}^C$  $\mathbb{R}^C$ 

Es decir, podemos utilizarlo para obtener:

$$\hat{y}_0 = P(Y = \text{clase 0} \mid X = x^{(i)})$$

$$\hat{y}_1 = P(Y = \text{clase 1} \mid X = x^{(i)})$$

$$\hat{y}_2 = P(Y = \text{clase 2} \mid X = x^{(i)})$$

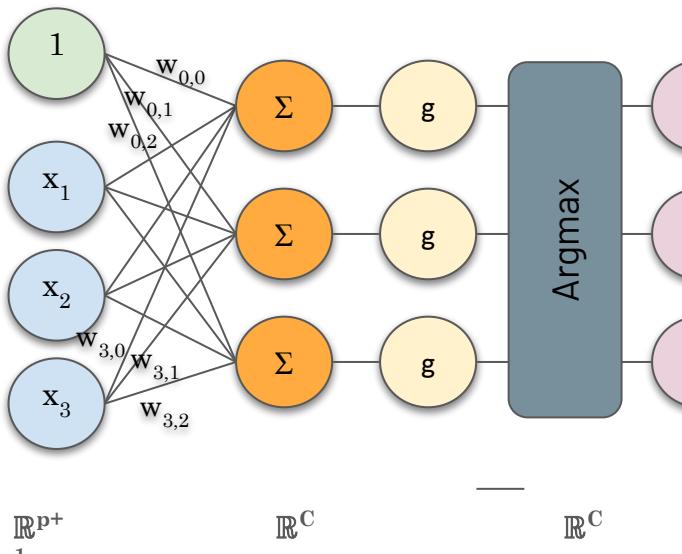
Vimos que si  $\mathbf{g(z)} = \text{sigmoidea}(z)$  obtenemos scores entre 0 y 1. ¿Estaría bien usarla en este caso?

¿Cómo elegimos la clase final?

# Redes Neuronales Multi Output

Imaginen que ahora queremos clasificar una instancia entre 3 clases distintas (ej: "perro", "gato", "pato").

Podríamos usar la siguiente arquitectura:



Es decir, podemos utilizarlo para obtener:

$$\begin{aligned}\hat{y}_0 &= P(Y = \text{clase 0} \mid X = x^{(i)}) \\ \hat{y}_1 &= P(Y = \text{clase 1} \mid X = x^{(i)}) \\ \hat{y}_2 &= P(Y = \text{clase 2} \mid X = x^{(i)})\end{aligned}$$

Una opción sería usar

- $g(z) = x$
- o quizás  $g(z) = \text{sigmoidea}(z)$

y luego usar  $\text{arg\_max}(\hat{y})$ .

Problemas:

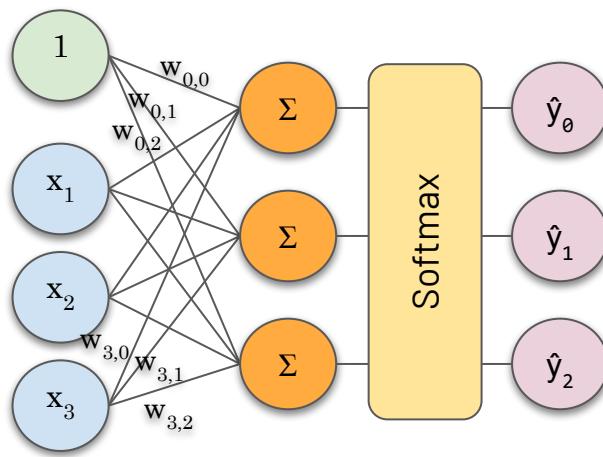
- 1) Perdemos las probabilidades para las distintas clases (no suman 1).
- 2)  $\text{arg\_max}(\hat{y})$  no es diferenciable, cuando querramos calcular la loss, no se propaga el gradiente!

# Redes Neuronales Multi-Output

## Softmax

Imaginen que ahora queremos clasificar una instancia entre 3 clases distintas (ej: "perro", "gato", "pato").

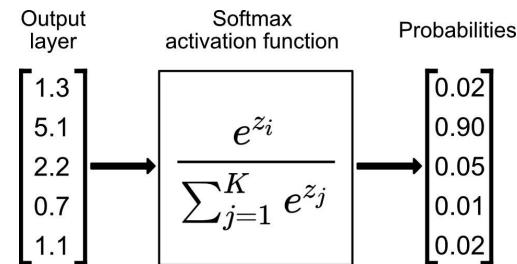
Podríamos usar la siguiente arquitectura:



Es decir, podemos utilizarlo para obtener:

$$\begin{aligned}\hat{y}_0 &= P(Y = \text{clase 0} \mid X = x^{(i)}) \\ \hat{y}_1 &= P(Y = \text{clase 1} \mid X = x^{(i)}) \\ \hat{y}_2 &= P(Y = \text{clase 2} \mid X = x^{(i)})\end{aligned}$$

Usamos **Softmax** una función de activación (que toma en cuenta los valores del resto de las neuronas). Su salida **suma 1 y es diferenciable**.



# Redes Neuronales Multi-Output

## Softmax

<https://playground.tensorflow.org/>



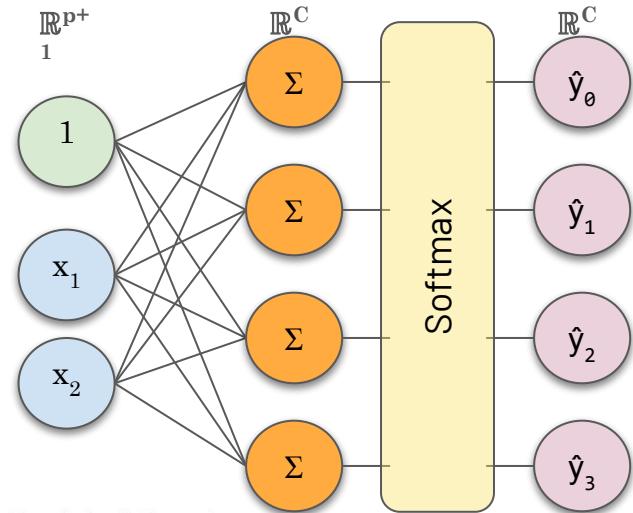
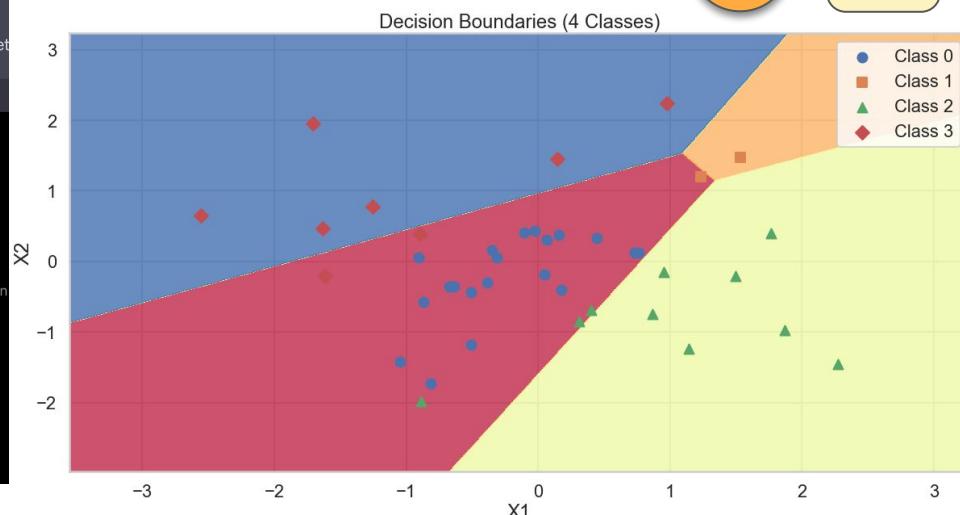
create a code in python that trains a neural network using pytorch, the network is only a linear layer with softmax activation. And there is a plot showing the decision boundaries for the example.

Certainly! Here's a Python code that trains a simple neural network with a linear layer and softmax activation using PyTorch and then plots the decision boundaries for a given dataset

```
python Copy code
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# Define a simple neural network with a linear layer and softmax activation
class SimpleNN(nn.Module):
    def __init__(self, input_size, output_size):
        super(SimpleNN, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        return self.linear(x)
```



# Entropía Cruzada (multiclasificación)

# Entropía cruzada (multiclas)

Una métrica muy utilizada en problemas de clasificación que permite calcular el error en un dataset para el cual tenemos la probabilidad estimada **para cada clase**.

$$J_{X,y} = \frac{1}{n} \sum_{i=1}^n \text{CE}^{(i)}$$

$$\hat{h}(x^{(i)}) = \text{softmax}(f^{(L)}(f^{(L-1)} \dots (x^{(i)})))$$

$$\begin{bmatrix} \hat{P}(Y = 0|X = x^{(i)}) \\ \hat{P}(Y = 1|X = x^{(i)}) \\ \vdots \\ \hat{P}(Y = C|X = x^{(i)}) \end{bmatrix}$$

$$\text{CE}^{(i)} = \sum_{c=1}^C -[y_c^{(i)} \log(\hat{h}_c(x^{(i)}))] \quad \text{en donde } \hat{h}_c(x^{(i)}) \text{ representa la probabilidad asignada a la clase c.}$$

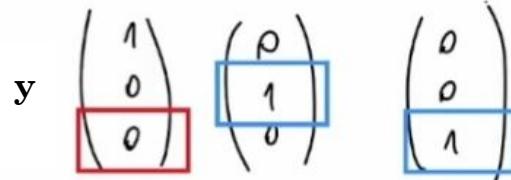
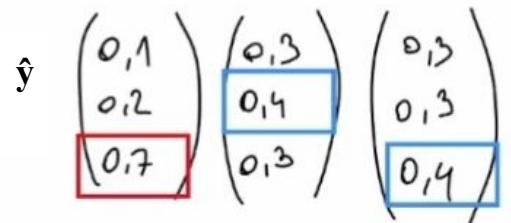
$y^{(i)}$  representa un vector de ceros salvo en la posición de la clase c, en donde vale 1.

equivalentemente

$$\text{CE}^{(i)} = \begin{cases} -\log(\hat{P}(Y = 0|X = x^{(i)})) & y_1^{(i)} = 1 \\ -\log(\hat{P}(Y = 1|X = x^{(i)})) & y_2^{(i)} = 1 \\ \dots \\ -\log(\hat{P}(Y = C|X = x^{(i)})) & y_C^{(i)} = 1 \end{cases}$$

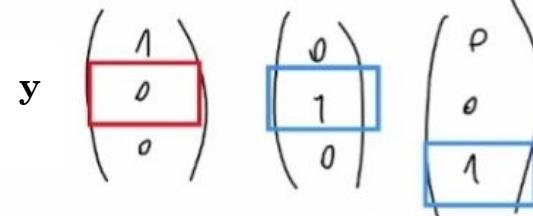
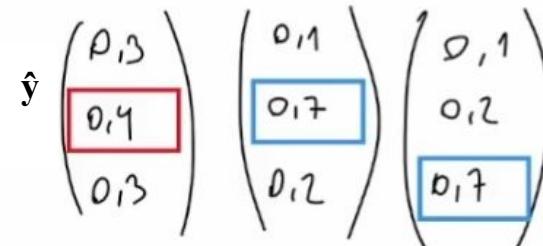
Notar que, como en caso binario, sólo un término sobrevive para cada instancia (todos los términos de la sumatoria son cero salvo uno)

# Entropía cruzada vs Accuracy



Accuracy =  $\frac{2}{3}$

Entropía cruzada = 4.14



Accuracy =  $\frac{2}{3}$

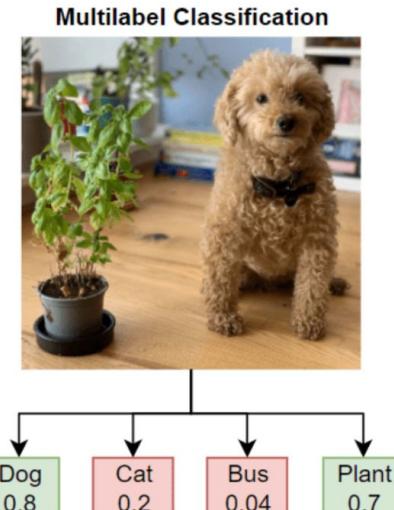
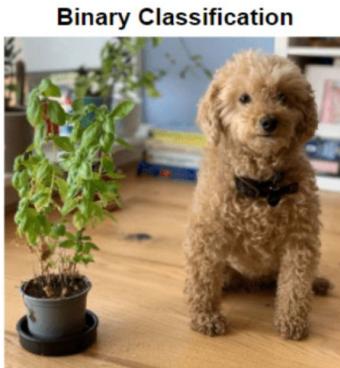
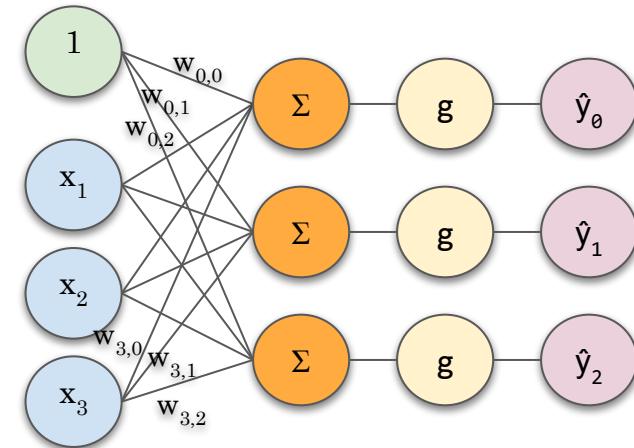
Entropía cruzada = 1.92

# Redes Neuronales Multi Label

# Redes Neuronales Multi-label

Hay problemas en los cuales tenemos más de una etiqueta por instancia.

- ¿Qué  $g$  se les ocurre utilizar en este caso?
- ¿Cómo definirían la función de pérdida?
- ¿Qué métrica se podría usar para medir el error bajo este problema? Buscar por ejemplo: Precision at k (P@k).



# Tarea

## Completar el notebook y formulario

### Lecturas obligatorias:

- ISLR, capítulo 10: hasta la sección **10.2 (inclusive)**
- ISLR, capítulo 10: sección **10.7**
- Deep Learning: **Intro del Capítulo 6 + Sección 6.1 + Sección 6.5 hasta 6.5.2 inclusive.**

### Opcional:

- Deep learning: El resto de la **Sección 6.5**.
- ISLR, capítulo 4: sección **4.3** Logistic Regression
- ISLR, capítulo 10: sección **10.8** Interpolation and Double Descent.
- Deep Learning: Resto del **Capítulo 6**.