



# Clase 4 - Redes Neuronales

## Aprendizaje Automático Profundo



# Aprendizaje Automático - Los Elementos

## Ejemplos (etiquetados)

Estu dio	Edad	Pro medi o	N1	N2
2	24	4	7	7.2
5	22	3	4.5	5.2
7	25	4	6.3	6

## Algoritmo de Aprendizaje

- Descenso de Gradiente
- otros

## Error a optimizar

- Error cuadrático
- Entropía
- Otros

## Modelo

Parámetros  
(aleatorios)

## Ejemplos nuevos

Estudi o	Edad	Prom edio
10	19	4
11	20	3

## Modelo

Parámetros  
(entrenados)

## Predicciones

P1	P2
7	7.2
4.5	5.2

# Regresión Lineal con Múltiples Variables de Salida

Previo a Redes Neuronales

# Regresión Lineal Múltiple - Problema

Predecir una variable

Estudio	Edad	Promedio	Nota
2	24	4	7
5	22	3	4.5
7	25	4	6.3
9	20	7	5.4
10	19	4	8.2
11	20	3	7.2
13,4	21	5	5.5
14	20	3	3



Predecir múltiples variables

Estudio	Edad	Promedio	N1	N2
2	24	4	7	7.2
5	22	3	4.5	5.2
7	25	4	6.3	6
9	20	7	5.4	4.4
10	19	4	8.2	9.5
11	20	3	7.2	8.1
13,4	21	5	5.5	10
14	20	3	3	4.3

# Regresión Lineal Múltiple - Motivación

## Motivación

- Predecir M valores, a partir de N valores
- Transformar un vector en otro
  - Vector de flotantes = codificación universal de cualquier información
- Puede usarse para
  - Predecir la posición de un objeto ( vector de 2 dimensiones) a partir de una imagen (vector de dimensiones: ancho\*alto\*colores)
  - Generar una imagen en base a un vector de características.
  - Generar un sonido (vector de t dimensiones) en base a un texto (vector de P dimensiones)
  - Etc...

Estudio	Edad	Promedio	N1	N2
2	24	4	7	7.2
5	22	3	4.5	5.2
7	25	4	6.3	6
9	20	7	5.4	4.4
10	19	4	8.2	9.5
11	20	3	7.2	8.1
13,4	21	5	5,5	10
14	20	3	3	4.3

# Regresión Lineal Múltiple - Versión Simple

¿Con lo que sabemos? Dos salidas → Dos modelos

Estudio	Edad	Promedio	N1
2	24	4	7
5	22	3	4.5
7	25	4	6.3
9	20	7	5.4
10	19	4	8.2
11	20	3	7.2
13,4	21	5	5,5
14	20	3	3

Modelo para N1

Estudio	Edad	Promedio	Nota N2
2	24	4	7.2
5	22	3	5.2
7	25	4	6
9	20	7	4.4
10	19	4	9.5
11	20	3	8.1
13,4	21	5	10
14	20	3	4.3

Modelo para N2

# Regresión Lineal Múltiple - Versión Simple

## Modelo para N1

### Entrada

X: 3 números

### Parámetros

W1: 3 números

B1: 1 número

### Salida

N1: 1 número

## Modelo para N2

### Entrada

X: 3 números

### Parámetros

W2: 3 números

B2: 1 número

### Salida

N2: 1 número

## Fórmula del modelo para N1

$$\begin{aligned}N1(X) &= (X[1] * W1[1] + X[2] * W1[2] + X[3] * W1[3]) + B1 \\&= X \cdot W1 + B1\end{aligned}$$

### En Python

```
N1 = np.dot(W1,X) + B1
```

## Fórmula del modelo para N2

$$\begin{aligned}N2(X) &= (X[1] * W2[1] + X[2] * W2[2] + X[3] * W2[3]) + B2 \\&= X \cdot W2 + B2\end{aligned}$$

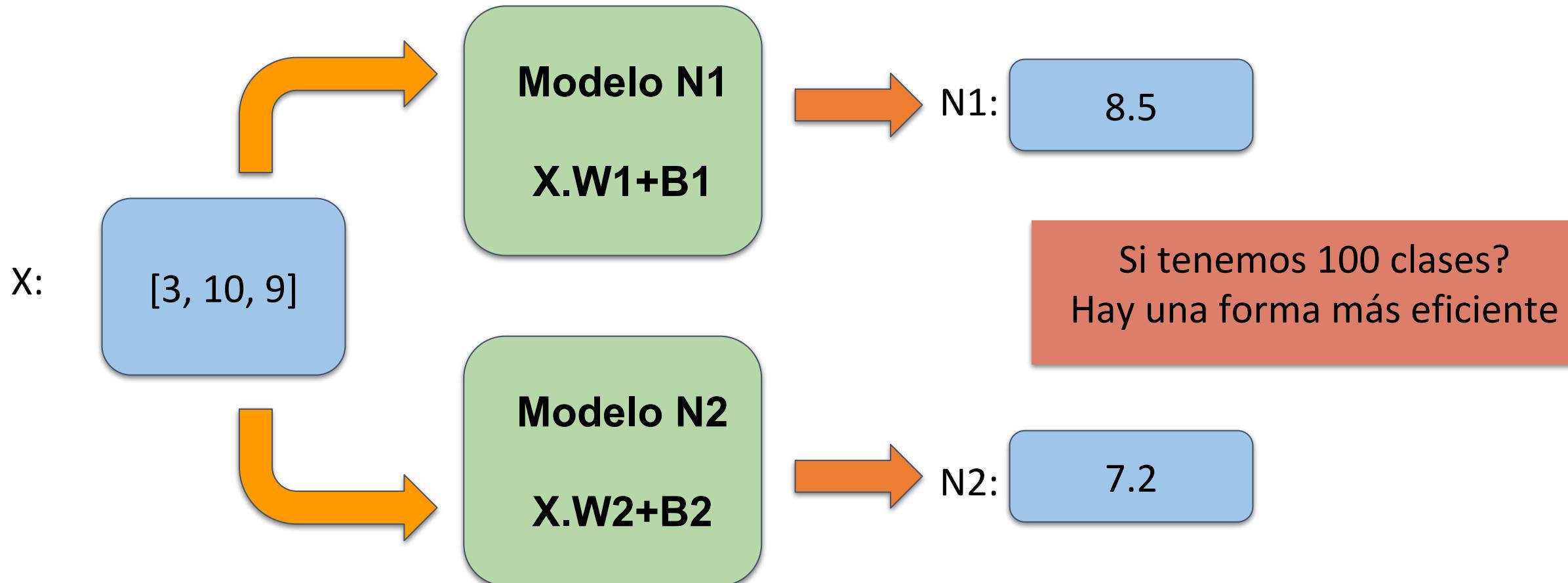
### En Python

```
N2 = np.dot(W2,X) + B2
```

X . W es el producto punto o escalar.

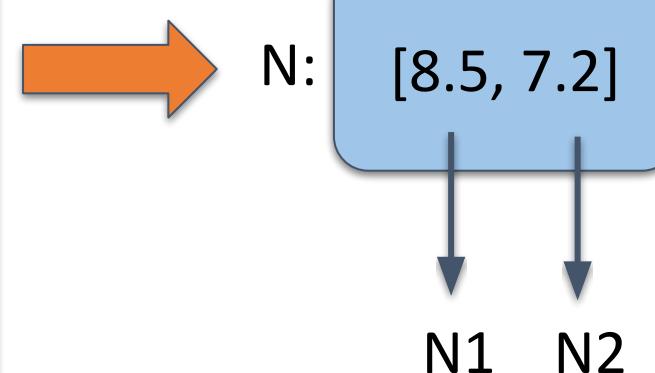
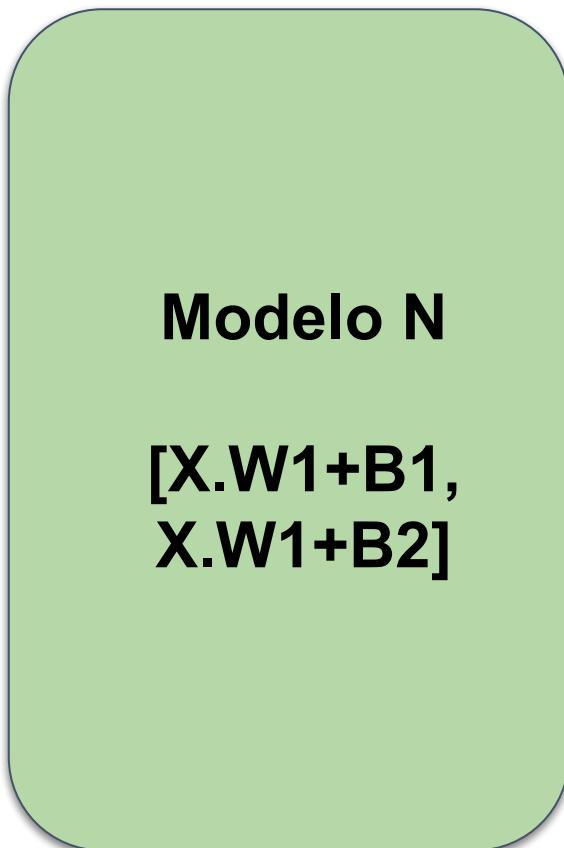
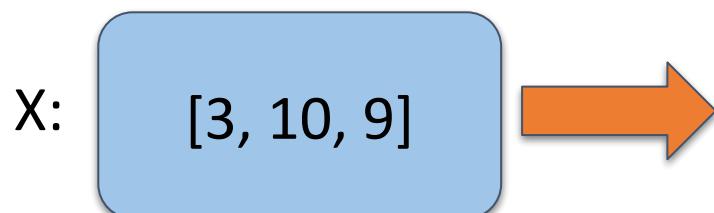
```
def dot(X,W):  
    p=0  
    for i in range(3):  
        p=p+(X[i]*W[i])  
    return p
```

# Regresión Lineal Múltiple - Versión Simple



# Regresión Lineal Múltiple - Modelo

**Combinamos N1 y N2  
en un solo modelo N**



Regresión Múltiple  
Una “sola” salida  
Vector de 2 números

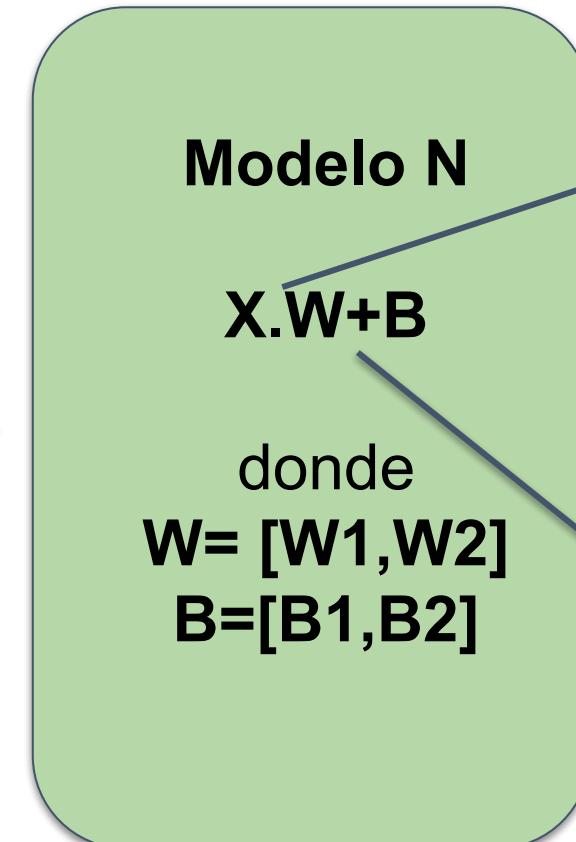
# Regresión Lineal Múltiple - Modelo

Combinamos los parámetros en:

- Matriz W de 3x2
- Vector B de 2x1

X:

[3, 10, 9]



Producto punto con matrices

N:

[8.5, 7.2]

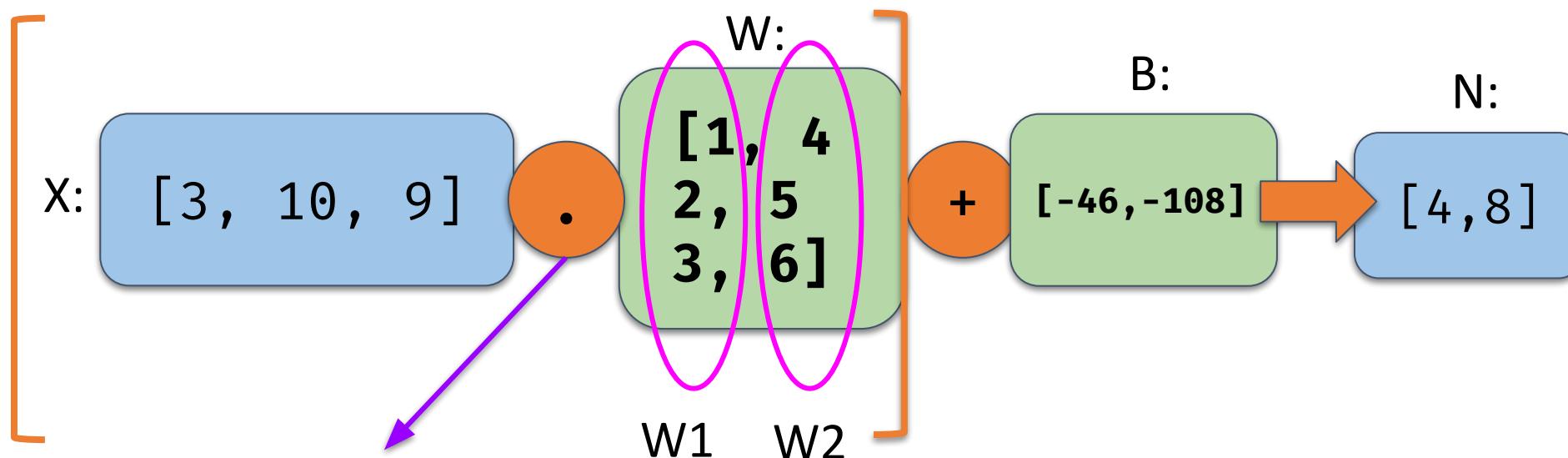


Suma vectorial

# Regresión Lineal Múltiple - Cálculo

- $W_1 [1, 2, 3]'$
- $B_1 -46$

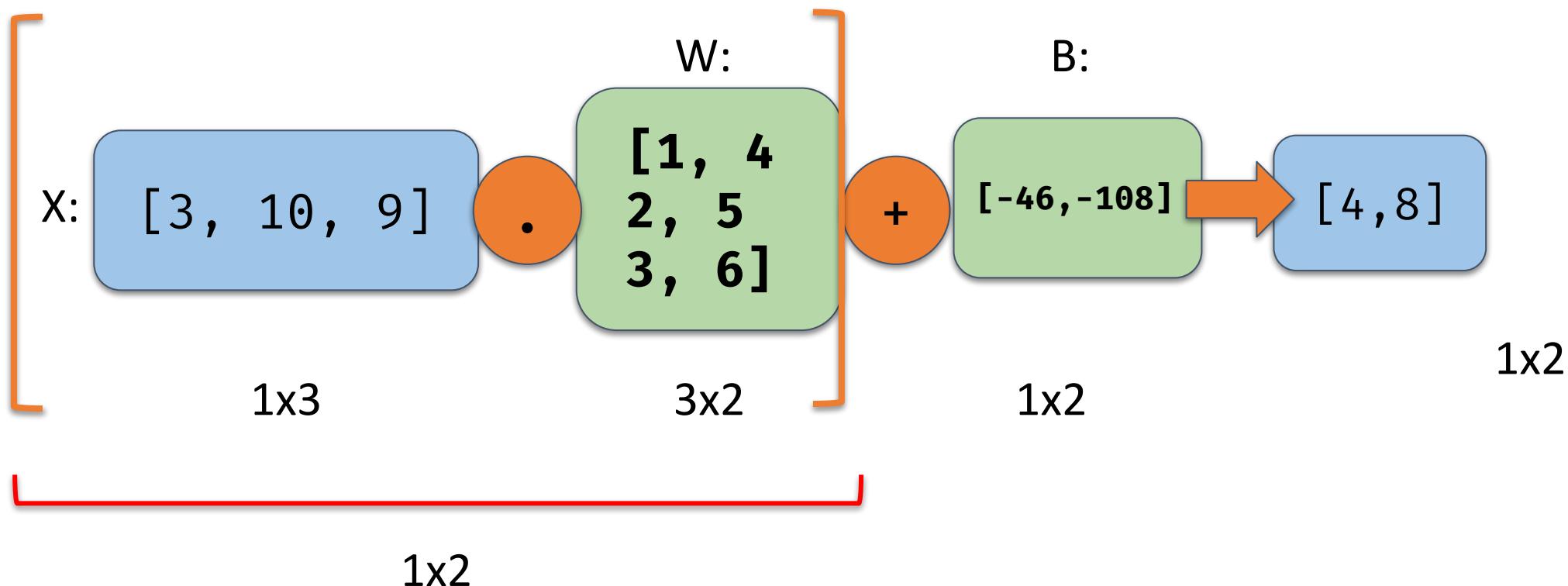
- $W_2 [4, 5, 6]'$
- $B_2 -108$



Multiplicación de matrices

# Regresión Lineal Múltiple - Tamaños

## Tamaños de vectores y matrices

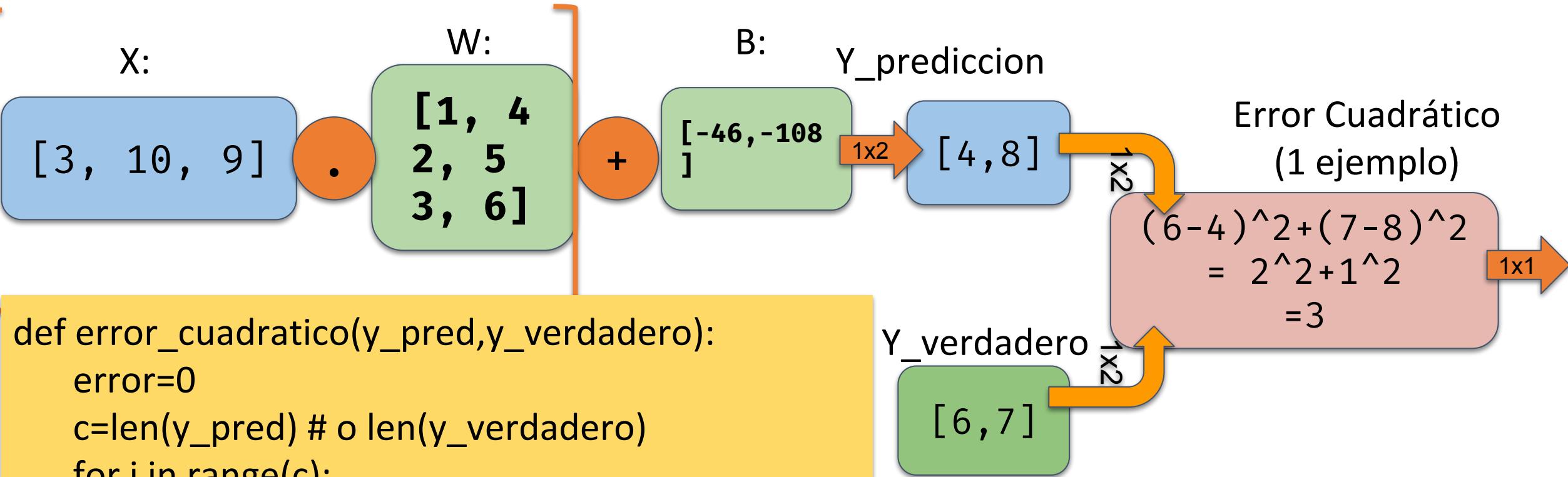


# Recursos para repasar operaciones vectoriales

- Producto punto o escalar entre dos vectores:  
<https://www.youtube.com/watch?v=N5f7pYTNcFM>
- Multiplicación de un vector por un escalar:  
<https://es.khanacademy.org/math/algebra-home/alg-vectors/alg-scalar-multiplication/v/understanding-multiplying-vectors-by-scalars>
- Multiplicación de una matriz por un escalar:  
<https://www.youtube.com/watch?v=-ArUqjhQIBM>
- Producto de una matriz y un vector  
<https://www.youtube.com/watch?v=2Gdy1xRnqjk>
- Producto de una matriz y otra matriz  
<https://www.youtube.com/watch?v=Tjrm3HsqBXE>  
<https://es.khanacademy.org/math/linear-algebra/matrix-transformations/composition-of-transformations/v/linear-algebra-matrix-product-examples>

# Regresión Lineal Múltiple - Función de Error

## Error Cuadrático para un solo ejemplo



# Regresión Lineal Múltiple - Función de Error

X = (Estudio, Edad, Promedio)

Y\_verdadero = (N1,N2)

Y\_prediccion = (P1,P2)

Estudio	Edad	Promedio	N1	N2	P1	P2	E
2	24	4	7	7.2	16	44	1435.24
5	22	3	4.5	5.2	12	40	1267.29
7	25	4	6.3	6	23	69	4247.89
9	20	7	5.4	4.4	24	70	4649.32
10	19	4	8.2	9.5	14	51	1755.89
11	20	3	7.2	8.1	14	54	2153.05
13.4	21	5	5.5	10	24.4	80.6	5341.57
14	20	3	3	4.3	17	66	4002.89

Promedio(E) = 3106.64

```
def error_cuadratico_medio(x,y,w,b):  
    n=x.shape[0] #==y.shape[0]  
    error=0  
    for i in range(n):  
        P = x[i,:]*w+b #prediccion  
        N = y[i,:]      #verdadero  
        error+=error_cuadratico(P,N)  
    return error/n
```

```
def error_cuadratico(y_pred,y_verdadero):  
    error=0  
    c=len(y_pred) # o len(y_verdadero)  
    for i in range(c):  
        error+= (y_pred[i]-y_verdadero[i])**2  
    return error
```

# Regresión Lineal Múltiple - En Keras

```
x,y=cargar_dataset()
nx,d_in = x.shape # x tiene tamaño n x d_in
ny,d_out = y.shape # y tiene tamaño n x d_out
assert(nx==ny) # misma cantidad de ejemplos en ambos vectores

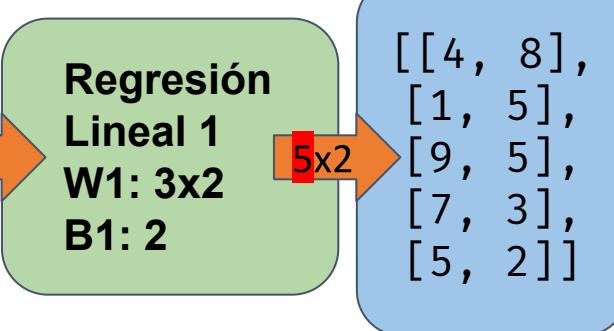
import keras
#Creamos el modelo
model=keras.Sequential()
#Capa lineal que convierte vector de d_in a d_out dims
model.add(keras.Dense(d_out,input_shape=[d_in]))
model.compile(loss='mse', # error cuadratico medio
              optimizer='sgd') # descenso de gradiente
model.fit(x,y,epochs=100,batch_size=32)
y_predicted=model.predict(x)
```

# Regresión Lineal - Batch Size

- No predecir/entrenar con **todos** los ejemplos ni con **uno**.
- Subconjuntos llamados **batchs**
- **batch\_size** = cantidad de ejemplos
- Ejemplo con **batch\_size=5**:

x:

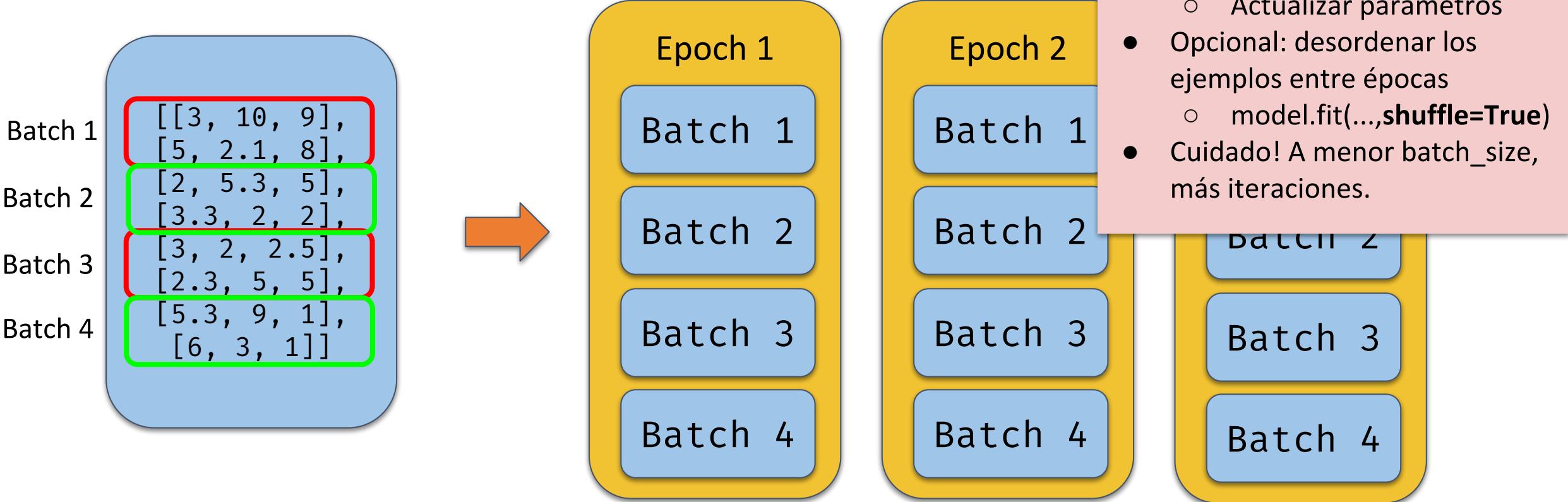
```
[[3, 10, 9],  
 [5, 2.1, 8],  
 [2, 5.3, 5],  
 [3.3, 2, 2],  
 [6, 3, 1]]
```



- Más eficiente
  - Multiplicaciones de matrices
  - pre-alocación de memoria
- Generalmente **batch\_size** = potencia de 2
  - 8, 16, 32, 64, 128, 256, 512
  - Más de 512 es redundante para entrenar
- Con GPUs:
  - Maximiza el uso de la memoria
  - Minimiza overhead copias
- No se puede usar todo el dataset al mismo tiempo!
  - $N = O(\text{millones})$
  - No entra en memoria

# Regresión Lineal - Epochs

- Permite indicar la cantidad de iteraciones de entrenamiento
  - En relación al tamaño del dataset
- epochs 5 => 5 pasadas por el dataset
- Ejemplo con **N=8, batch\_size=2**, y **epochs = 3**



# Regresión Lineal Múltiple - Resumen

## Resumen

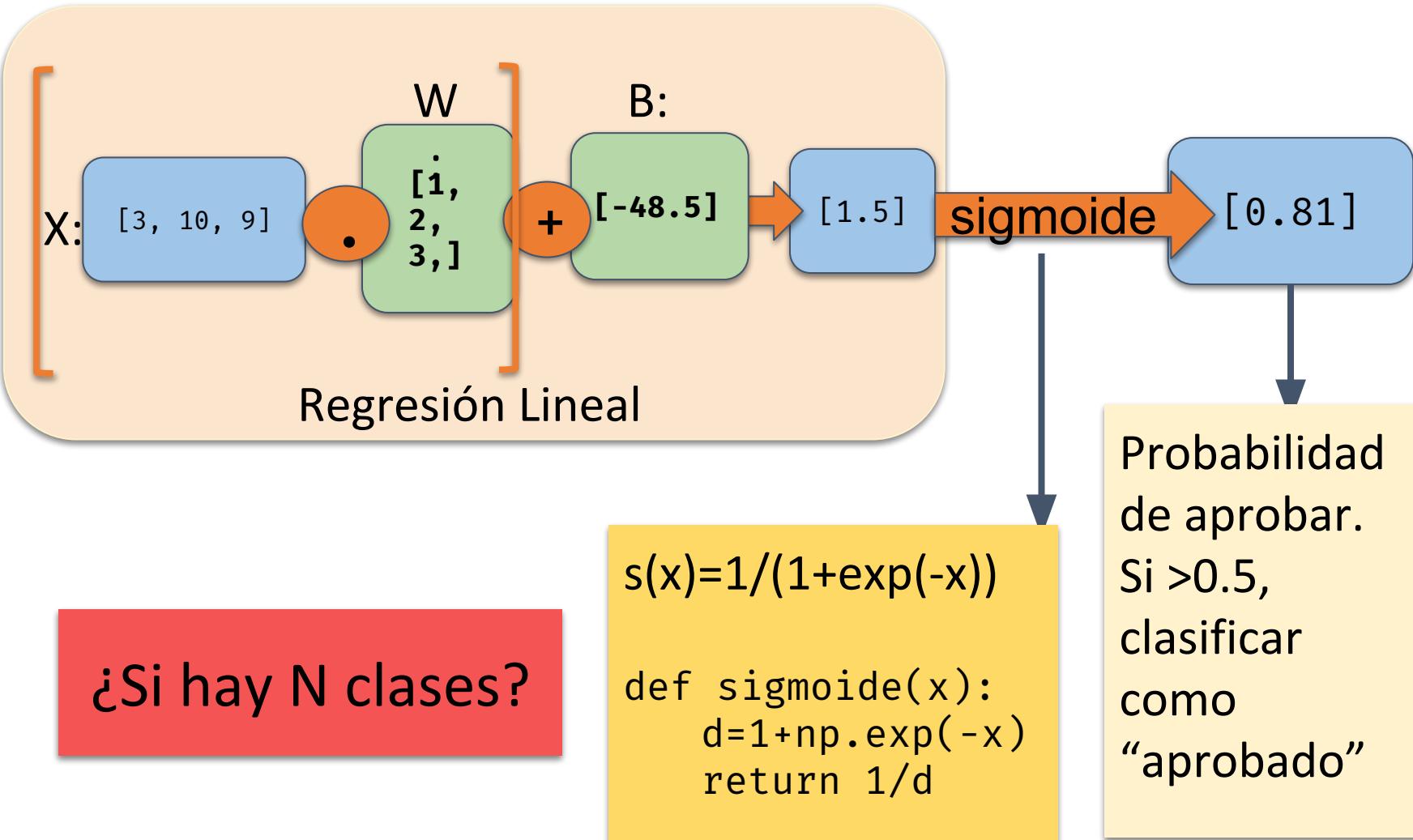
- Predecir N valores
- Transformar un vector en otro
  - Vector de entrada de dimension N
  - Vector de salida de dimension M
  - Matriz de pesos W, de dimension NxM
  - Vector de bias B, de dimensión M
- ¿Cómo se entrena?
  - Se suman los errores cuadráticos de cada salida para el descenso de gradiente
  - Sigue siendo convexo el error (solución única).
- ¿Es más potente que Regresión Lineal Simple?
  - No, sigue siendo un modelo lineal.

Estudio	Edad	Promedio	N1	N2
2	24	4	7	7.2
5	22	3	4.5	5.2
7	25	4	6.3	6
9	20	7	5.4	4.4
10	19	4	8.2	9.5
11	20	3	7.2	8.1
13.4	21	5	5.5	10
14	20	3	3	4.3

# **Regresión Logísitca con Múltiples Variables de Salida (previa a Redes Neuronales)**

# Regresión Logística (dos clases)

Estudio	Edad	Promedio	Aprobó (y)
2	24	4	0
5	22	3	1
7	25	4	0
9	20	7	1
10	19	4	0
11	20	3	0
13,4	21	5	1
14	20	3	0



# Regresión Logística Múltiple

Múltiples opciones  
=> predicciones independientes



Variables (x)			Carrera (y)		
Estudio	Edad	Promedio	P1	P2	P3
2	21	4	0	1	1
5	22	3	1	1	1
7	25	1	0	1	0
9	20	7	1	0	1
10	19	4	0	1	0
11	20	3	1	1	1
13,4	21	5	1	0	1
14	20	3	0	1	0

Clases mutuamente exclusivas  
=> Problema de **clasificación**

Variables (x)			Carrera (y)		
Estudio	Edad	Promedio	LS	LI	IC
2	24	4	1	0	0
5	22	3	0	1	0
7	25	1	1	0	0
9	20	7	0	0	1
10	19	4	0	0	0
11	20	3	0	0	1
13,4	21	5	0	1	0
14	20	3	1	0	0

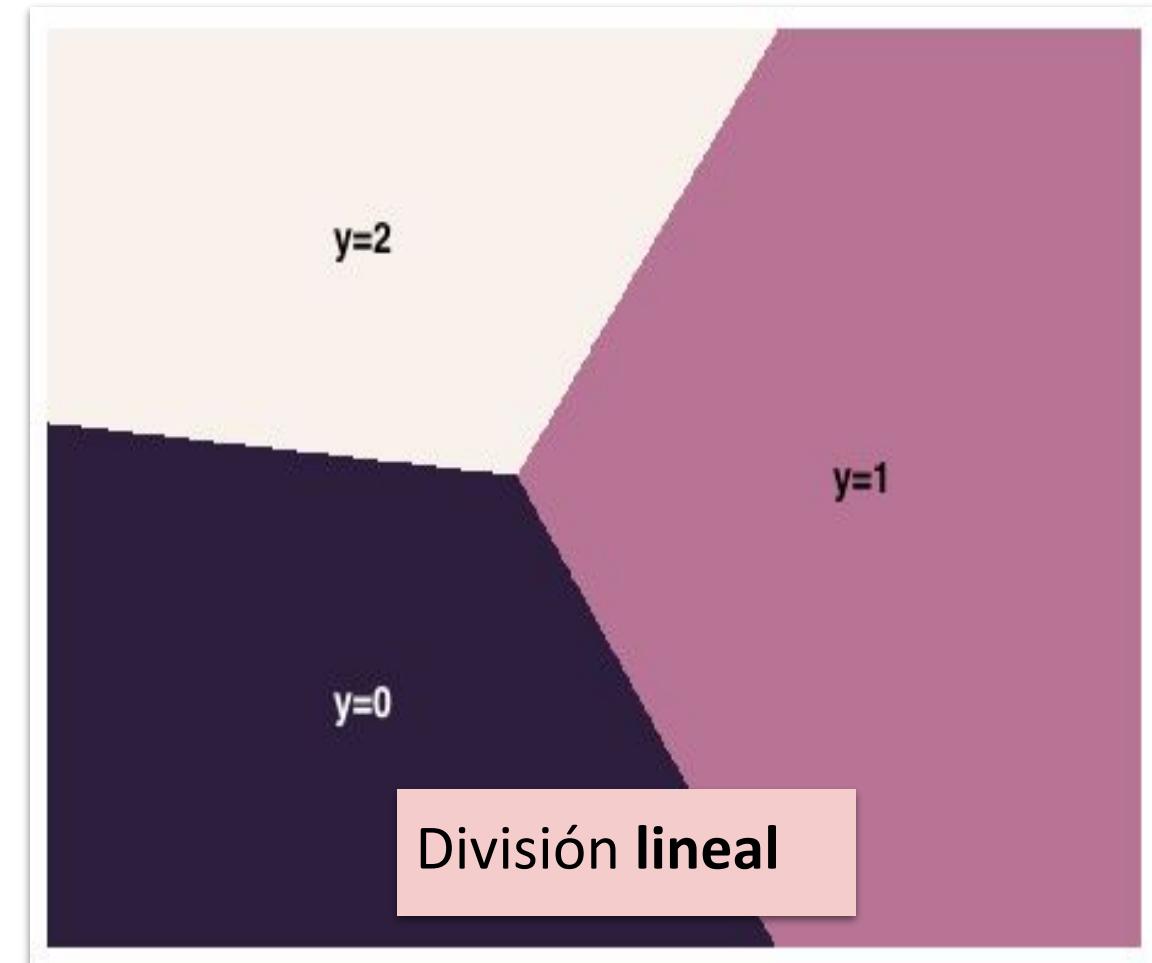
En esta materia:  
**problemas de  
clasificación**

# Regresión Logística Múltiple - Intuición

Un hiperplano por cada clase



Dividen el espacio en regiones por clase



# Regresión Logística Múltiple - Codificación

Codificación “One Hot”. Cada clase con su columna. Tamaño de y: N x C

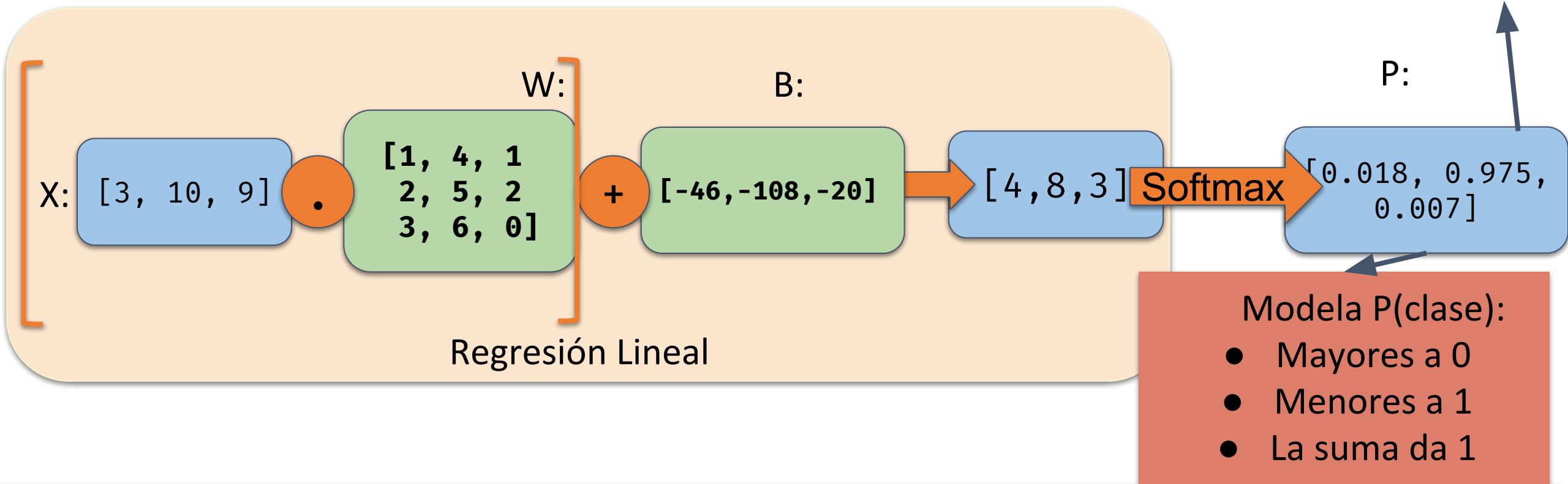
Variables (x)			Carrera (y)		
Estudio	Edad	Promedio	LS	LI	IC
2	24	4	1	0	0
5	22	3	0	1	0
7	25	4	0	1	0
9	20	7	0	0	1
10	19	4	1	0	0
11	20	3	0	0	1
13,4	21	5	0	1	0
14	20	3	1	0	0

Codificación con etiquetas. Cada clase con su ID. Tamaño de y: N x 1

Variables (x)			Carrera (y)
Estudio	Edad	Promedio	LS = 0, LI = 1, IC = 2
2	24	4	0
5	22	3	1
7	25	4	1
9	20	7	2
10	19	4	0
11	20	3	2
13,4	21	5	1
14	20	3	0

# Regresión Logística Múltiple - Softmax

- Similar a **Regresión Lineal** pero para predecir probabilidades
- Función **softmax** aplicada a la salida
  - Genera **distribución de probabilidad  $P(y)$**
- **Clasificación de N clases**



# Regresión Logística Múltiple - Softmax

- Función softmax
  - Convierte salida de Regresión Lineal en distribución de probabilidades de c/ clase

¿Por qué  $e^x$ ?

- Resultado  $> 0$
- Exponencial, acentúa los valores altos
- Podría ser con otra base

Y:

[4, 8, 3]

## Softmax

### 1) Cálculo del vector E

$$E = [e^4, e^8, e^3] = [54.59, 2980.95, 20.08]$$

### 2) Cálculo del valor N

$$N = e^4 + e^8 + e^3 = 54.59 + 2980.95 + 20.08 = 3055.63$$

### 3) Cálculo del vector P de probabilidades

$$\begin{aligned}P &= E / N = [e^4/N, e^8/N, e^3/N] \\&= [54.59/3055.63, 2980.95/3055.63, 20.08/3055.63]\end{aligned}$$

P:

[0.018, 0.975, 0.007]

# Regresión Logística Múltiple - Softmax

Y:

```
[4,8,3]
```

```
def softmax(Y):  
    n=len(Y)  
    E=np.zeros_like(Y)  
    for i in range(n):  
        E[i] = np.exp(Y[i])  
    N=0  
    for i in range(n):  
        N=N+E[i]  
    P=np.zeros_like(Y)  
    for i in range(n):  
        P[i]=E[i]/N  
    return P
```

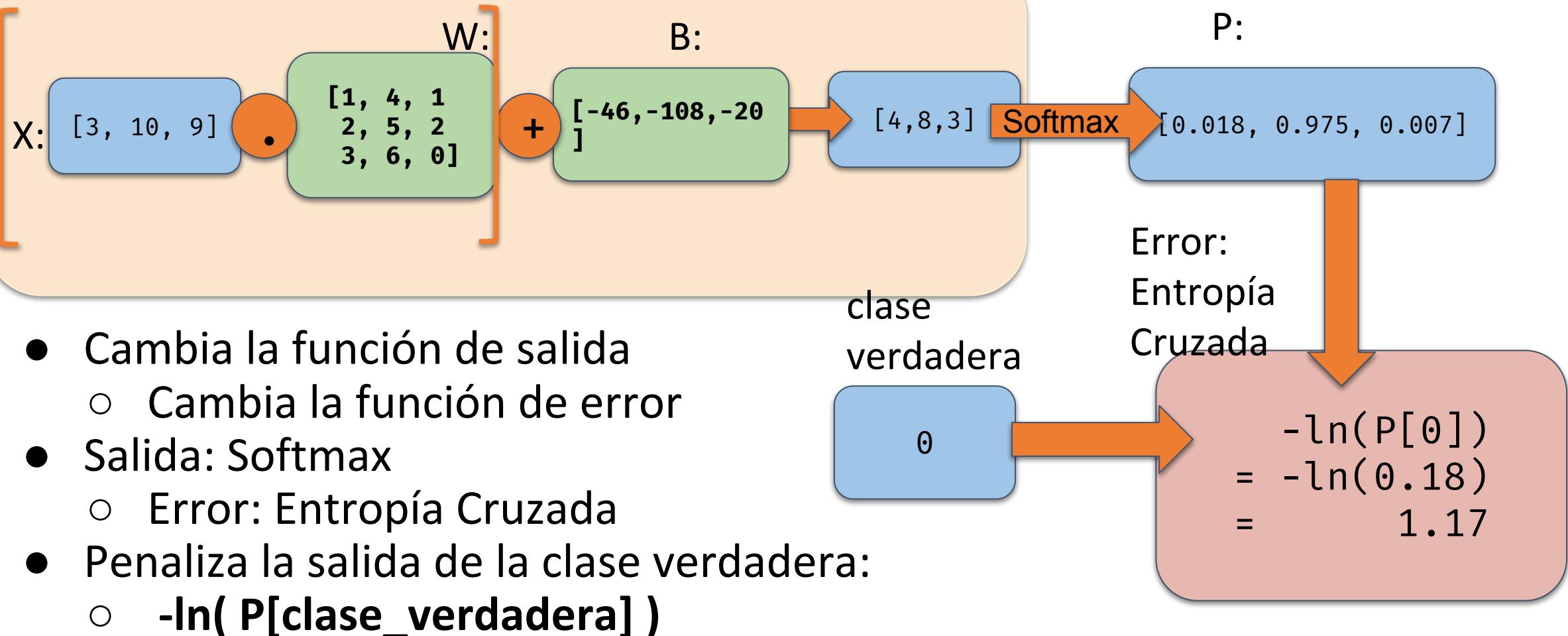
P:

```
[0.018, 0.975,  
 0.007]
```

Versión vectorial:

```
def softmax(Y):  
    E=np.exp(Y)  
    N=np.sum(E)  
    return E/N
```

# Regresión Logística Múltiple - Entropía Cruzada



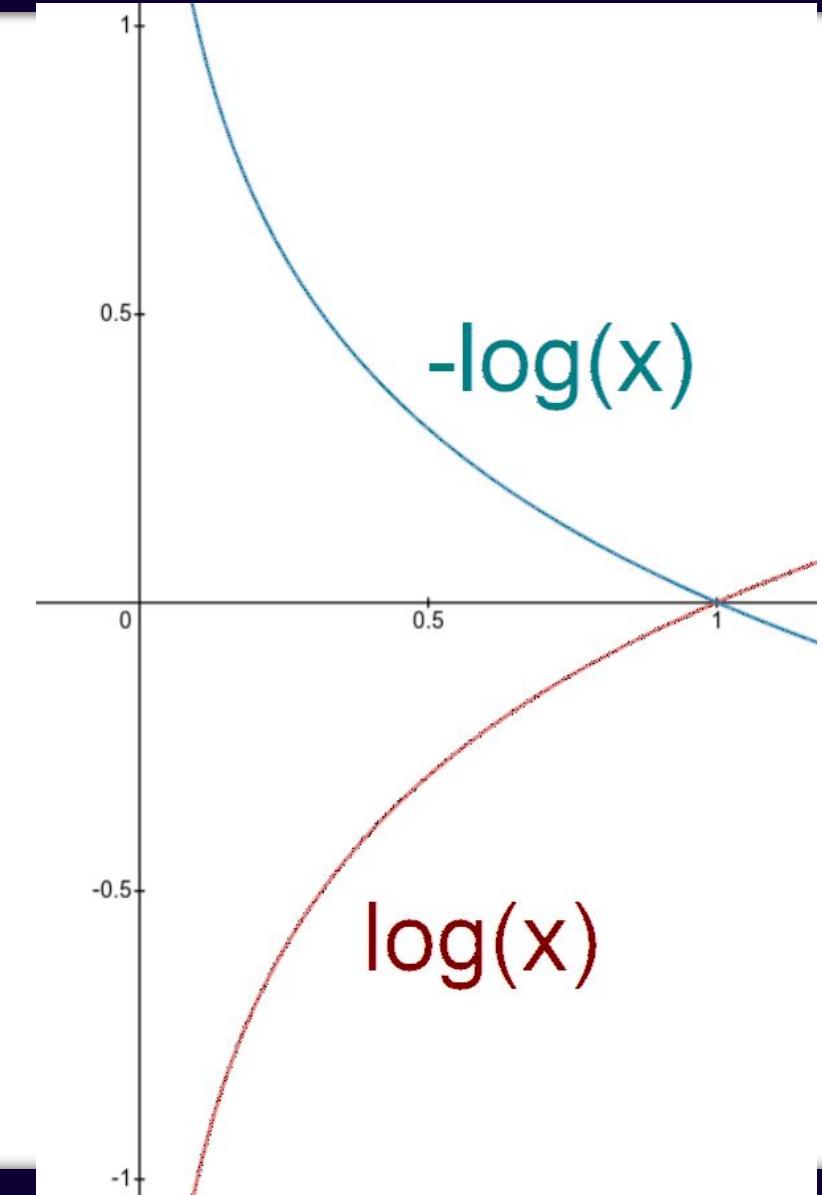
# Regresión Logística Múltiple - Entropía Cruzada

## Repaso

- Funciones  $\ln(x)$  y  $-\ln(x)$ 
  - Cuando  $0 <= x <= 1$
- Si  $x$  tiende a 1
  - $\ln(x)$  tiende a 0
  - **$-\ln(x)$  tiende a 0**
- Si  $x$  tiende a 0
  - $\ln(x)$  tiende a -infinito
  - **$-\ln(x)$  tiende a +infinito**

## Entonces

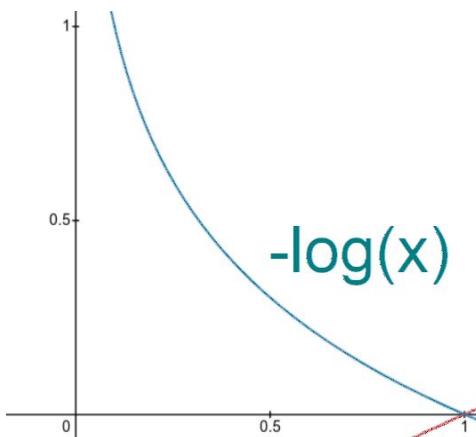
- $-\ln(P[\text{clase\_verdadera}])$ 
  - Penaliza si  $P[\text{clase\_verdadera}] = 0$
  - NO penaliza si  $P[\text{clase\_verdadera}] = 1$



# Regresión Logística Múltiple - Entropía Cruzada

Función de error = Entropía Cruzada =  $\ln(P[\text{clase\_verdadera}])$

Variables			Carrera	xW+b			P(Carrera)			E	
Estudio	Edad	Promedio	LS = 0, LI = 1, IC = 2	LS	LI	IC	LS	LI	IC		
2	24	4	0	46	50	50.2	0.01	0.45	0.55	-log(0.01)	4.61
14	20	3	0	69	66	66.1	0.91	0.05	0.05	-log(0.91)	0.09



- $-\ln(P[\text{clase\_verdadera}])$ 
  - Penaliza **mucho** si  $P[\text{clase\_verdadera}] \approx 0$
  - Penaliza **poco** si  $P[\text{clase\_verdadera}] \approx 1$

# Regresión Logística Múltiple - Entropía Cruzada

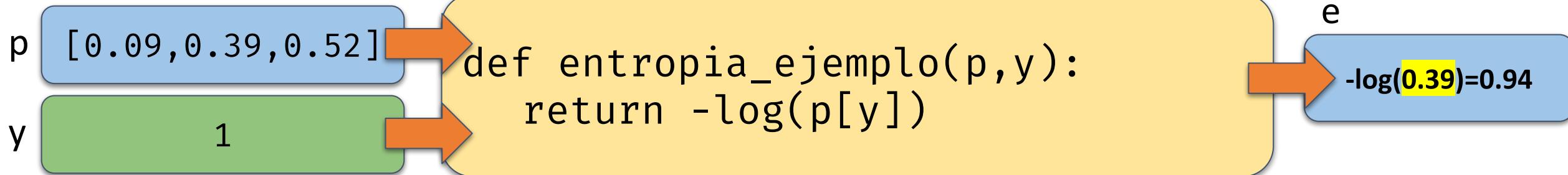
Función de error = Entropía Cruzada **Promedio** = E.mean()

Variables			Carrera	xW+b			P(Carrera)			E	
Estudio	Edad	Promedio	LS = 0, LI = 1, IC = 2	LS	LI	IC	LS	LI	IC		
2	24	4	0	46	50	50.2	0.01	0.45	0.55	-log(0.01)	4.61
5	22	3	1	49.5	51	51.3	0.09	0.39	0.52	-log(0.39)	0.94
7	25	4	1	60	61.5	61.8	0.09	0.39	0.52	-log(0.39)	1.20
9	20	7	2	60.5	64	62.9	0.02	0.73	0.24	-log(0.24)	1.43
10	19	4	0	58.5	58.5	58.2	0.36	0.36	0.27	-log(0.36)	1.35
11	20	3	2	61.5	60	60.1	0.68	0.15	0.17	-log(0.17)	0.39
13,4	21	5	1	71	70.3	69.9	0.55	0.27	0.18	-log(0.55)	1.31
14	20	3	0	69	66	66.1	0.91	0.05	0.05	-log(0.91)	0.09

Error Promedio = 1.41

# Regresión Logística Múltiple - Función de Error

Entropía Cruzada: Penaliza con  $-\ln(P[y])$  (y es la clase verdadera)



Y	P		
LS = 0, LI = 1, IC = 2	LS	LI	IC
0	0.01	0.45	0.55
1	0.09	0.39	0.52
1	0.09	0.39	0.52
2	0.02	0.73	0.24
0	0.36	0.36	0.27
2	0.68	0.15	0.17
1	0.55	0.27	0.18
0	0.91	0.05	0.05

The diagram illustrates the calculation of average cross-entropy loss for a batch of examples. On the left, a table shows the values for Y and P. An orange arrow points from this table to a yellow box containing the function definition:

```
def entropia_promedio(P,Y):  
    n=len(Y)  
    es=np.zeros(n)  
    for i in range(n):  
        p=P[i,:]  
        y=Y[i]  
        es[i]=entropia_ejemplo(p,y)  
    return errores.mean()
```

An orange arrow points from the function call to the result: Promedio = 1.41.

# Regresión Logística Múltiple - En Keras

```
x,y=cargar_dataset(one_hot=False)
nx,d_in = x.shape # x tiene tamaño n x d_in
ny = y.shape # y tiene tamaño n x 1 (codificacion etiquetas)
classes=y.max()+1 # etiqueta de clase máxima+1
assert(nx==ny) # misma cantidad de ejemplos en ambos vectores
import keras
#Creamos el modelo
model=keras.Sequential()
#Capa lineal que convierte vector de d_in a d_out dims
model.add(keras.Dense(classes,input_shape=[d_in],
                      activation="softmax"))
model.compile(loss='sparse_categorical_crossentropy',# ent. cruz.
              optimizer='sgd') # descenso de gradiente
history = model.fit(x,y,epochs=100,batch_size=32)
y_predicted=model.predict(x)
```

# Regresión Logística Múltiple - En Keras

```
x,y=cargar_dataset(one_hot=True)
nx,d_in = x.shape # x tiene tamaño n x d_in
ny,classes = y.shape # y tiene tamaño n x clases (one hot)
assert(nx==ny) # misma cantidad de ejemplos en ambos vectores

import keras
#Creamos el modelo
model=keras.Sequential()
#Capa lineal que convierte vector de d_in a d_out dims
model.add(keras.Dense(classes,input_shape=[d_in],
                      activation="softmax"))
model.compile(loss='categorical_crossentropy',# ent. cruz.
              optimizer='sgd') # descenso de gradiente
history = model.fit(x,y,epochs=100,batch_size=32)
y_predicted=model.predict(x)
```

# Resumen Regresión Logística Múltiple

- Permite realizar clasificación de **más de 2 clases**
- Función **softmax** se aplica luego de la regresión lineal
  - Genera una distribución de probabilidades
- **Entropía cruzada:** error adecuado para la **softmax**
- Sigue teniendo una solución única (optimización convexa)

Variables			Carrera
Estudio	Edad	Promedio	LS = 0, LI = 1, IC = 2
2	24	4	0
5	22	3	1
7	25	4	1
9	20	7	2
10	19	4	0
11	20	3	2
13,4	21	5	1
14	20	3	0

# Regresión Logística y Lineal Múltiples

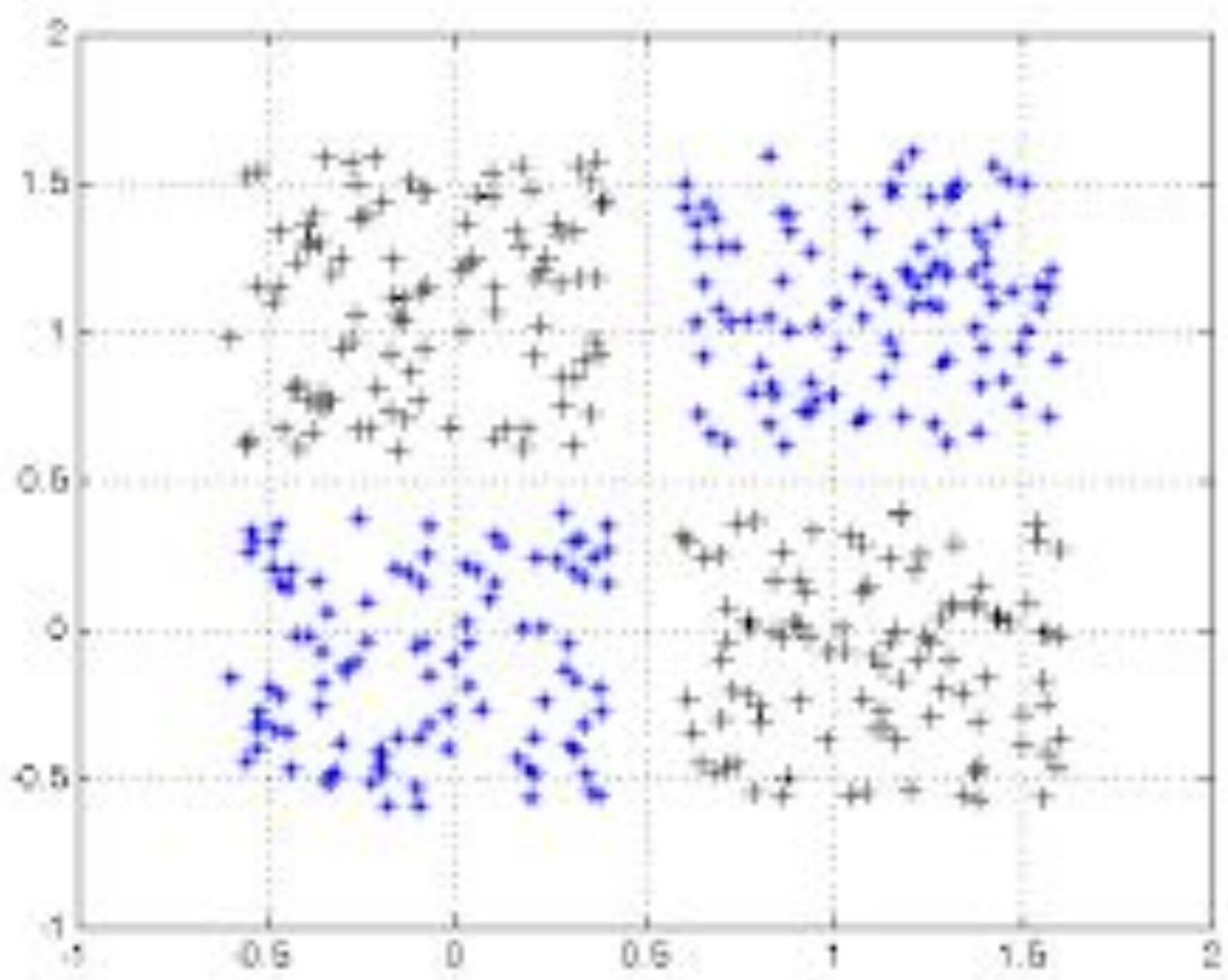
- Herramientas simples pero poderosas.
- Permiten
  - Transformaciones arbitrarias entre vectores (lineal).
  - Dar estructura de distribución de probabilidad a la salida (logística).
- Lo primero a intentar en cualquier problema
- Ventajas
  - Caja blanca, interpretable
  - Error convexo, solución única, fácil de entrenar.
- Desventajas
  - No pueden resolver problemas no-lineales.

# Límites de modelos lineales

# Límites de Regresión Logística

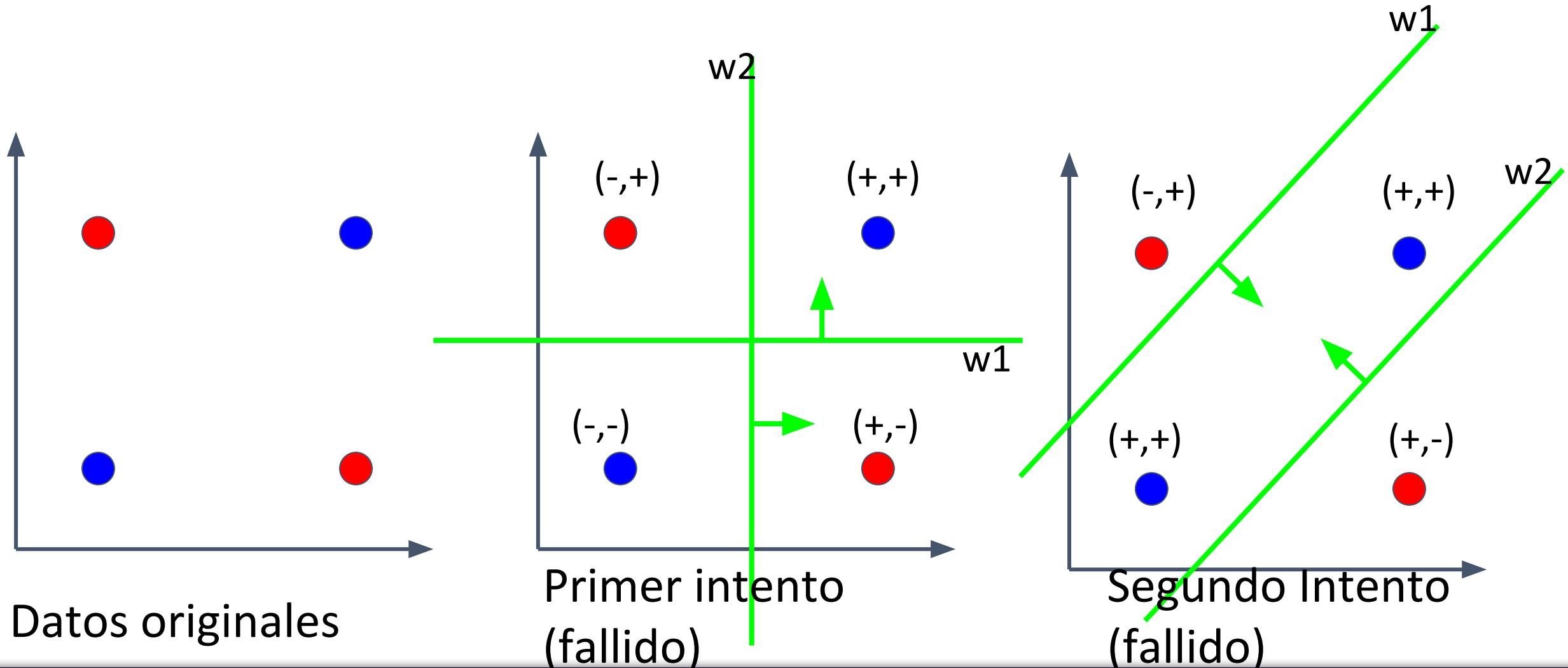
El problema del XOR

Los ejemplos NO son separables con hiperplanos

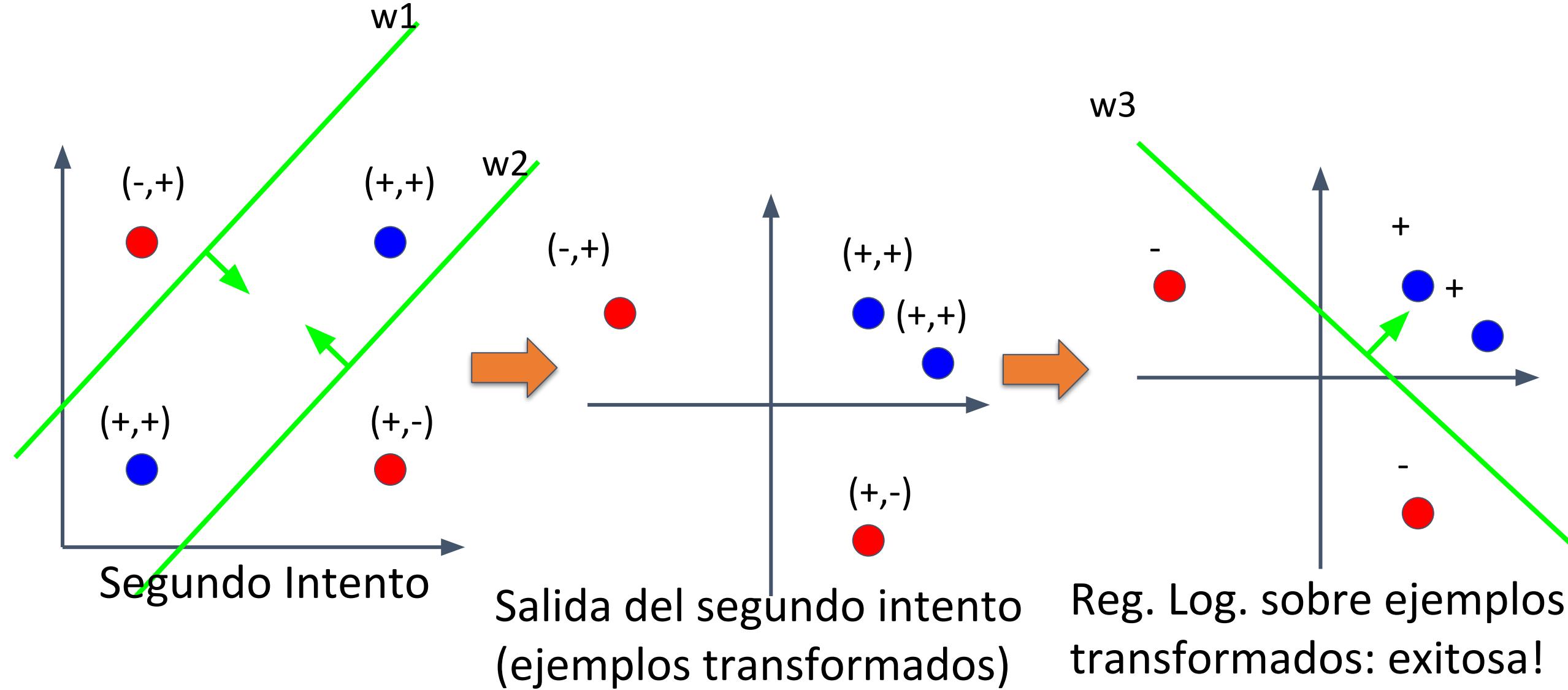


# Límites de Regresión Logística

El problema del XOR. 2 clases, rojo y azul. ¿Que hiperplanos los separan?



# Límites de Regresión Logística

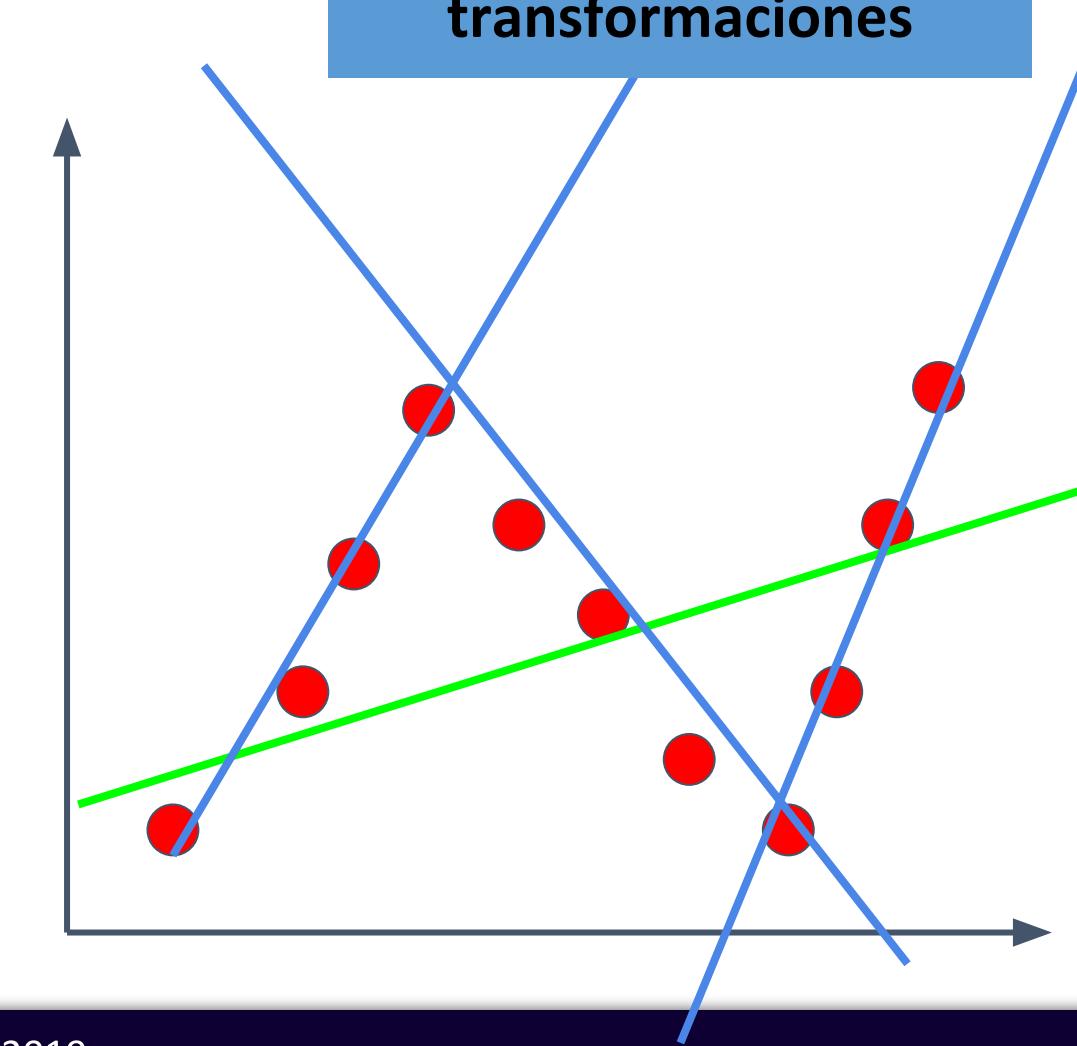


# Límites de Regresión Lineal

Mucho error con una RL



Mejora combinando transformaciones



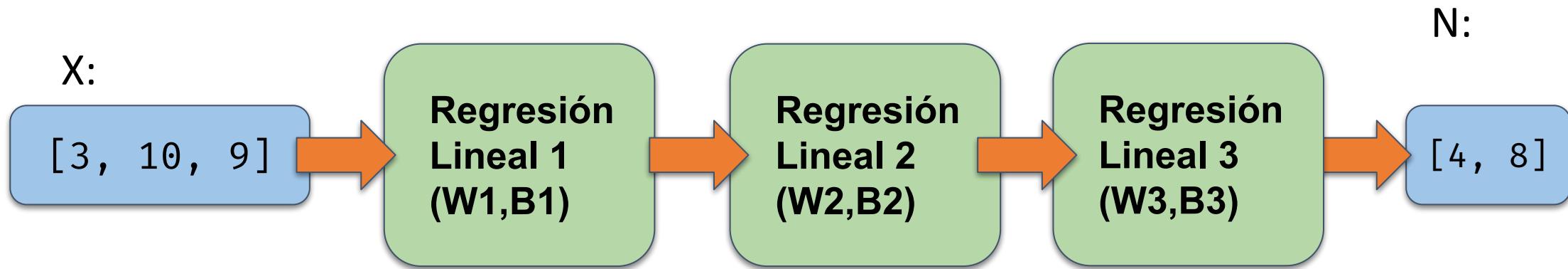
# Límites de Regresión Lineal y Logística

- Hay problemas que **Regresión Lineal** o **Regresión Logística** no pueden resolver fácilmente.
- Transformar los ejemplos con varias regresiones aumenta el poder de los modelos
- **Redes Neuronales** = combinación de varias regresiones (y otras yerbas)

# Redes Neuronales

# Redes Neuronales - Regresión

Red neuronal para predecir notas:



Idea básica: Apilar transformaciones (regresiones lineales).  
Llamamos **capa** a cada transformación.

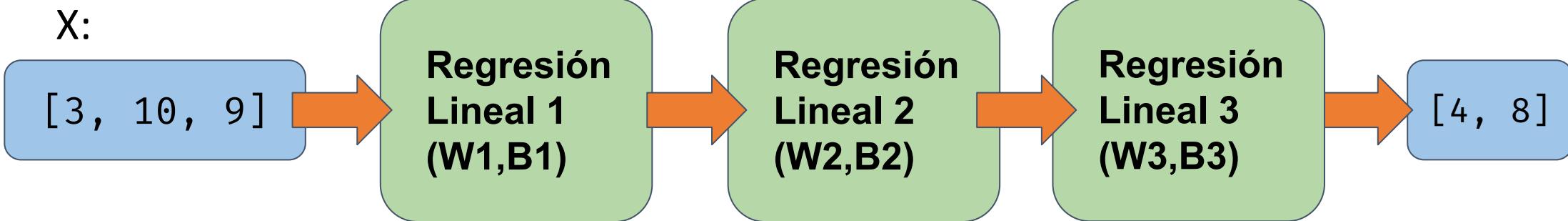
En este caso usamos 3 regresiones.  
Podemos usar N (depende del problema)

# Redes Neuronales - Regresión

Red neuronal para predecir notas:

En este caso usamos 3 regresiones, podemos usar N (depende del problema)

N:



RL 1 tiene que recibir algo de tamaño 3  
=> W tiene tamaño  $3 \times P$

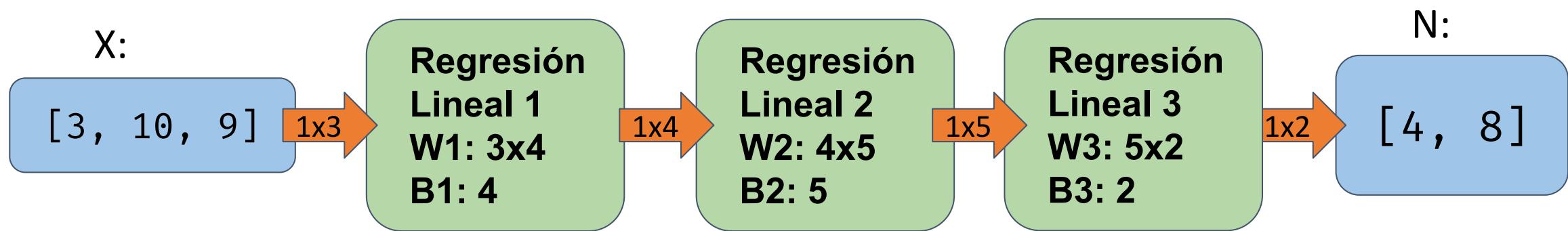
RL 2 es libre de decidir el tamaño de entrada/salida

RL 3 tiene que generar algo de tamaño 2  
=> W tiene tamaño  $P \times 2$   
B tiene tamaño 2

Idea básica: Apilar transformaciones (regresiones lineales).  
Llamamos **capa** a cada transformación.

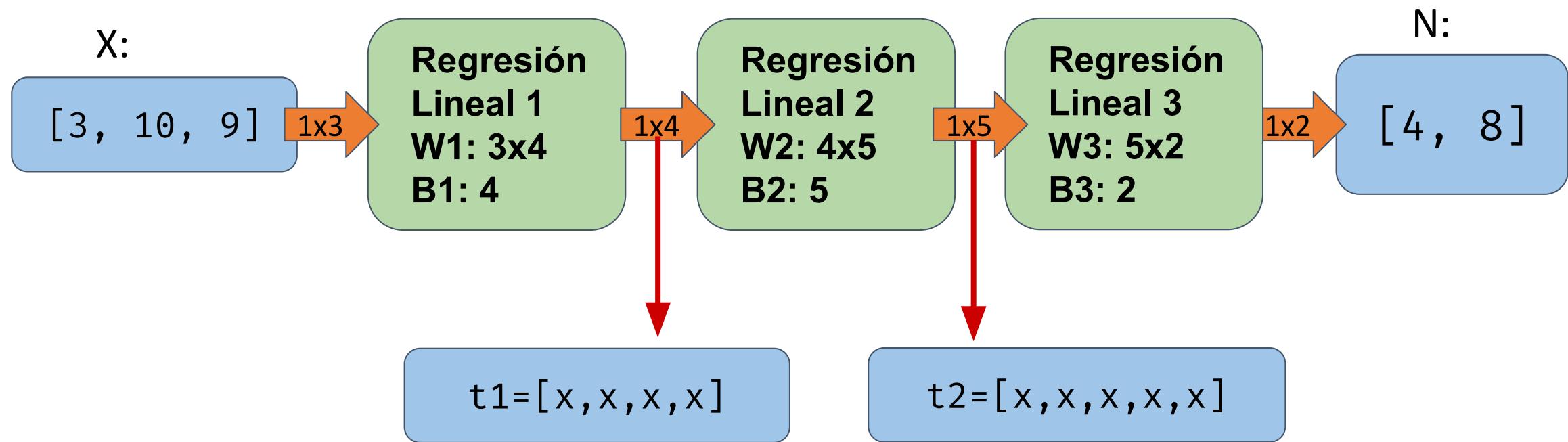
# Redes Neuronales - Tamaños

Ejemplo de tamaños de parámetros y salidas

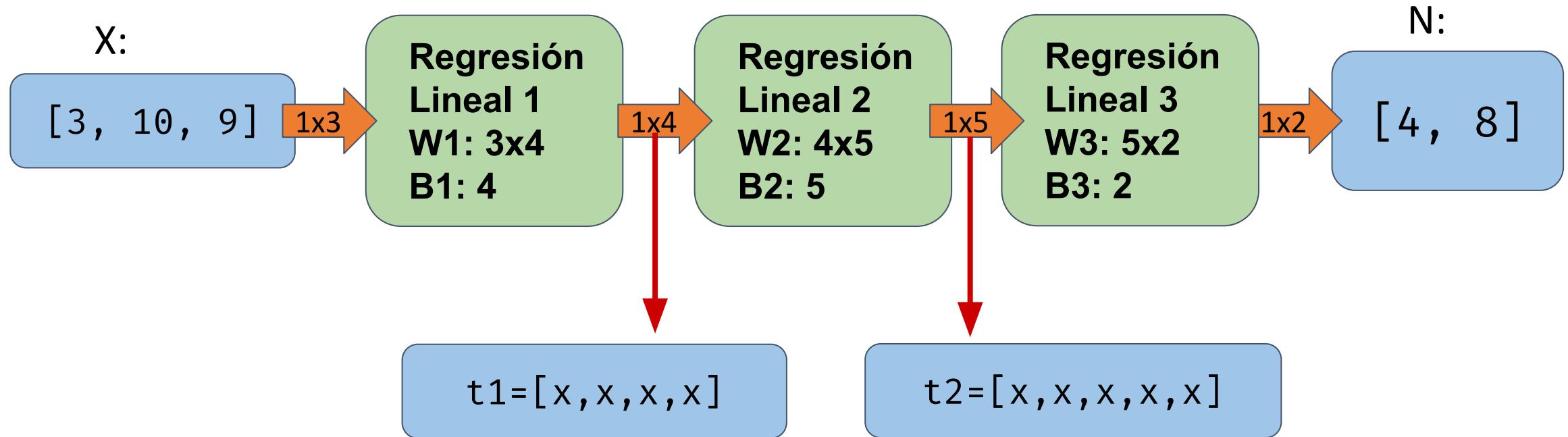


# Redes Neuronales - Ejecución

Función o pasada “forward”



# Redes Neuronales - Ejecución



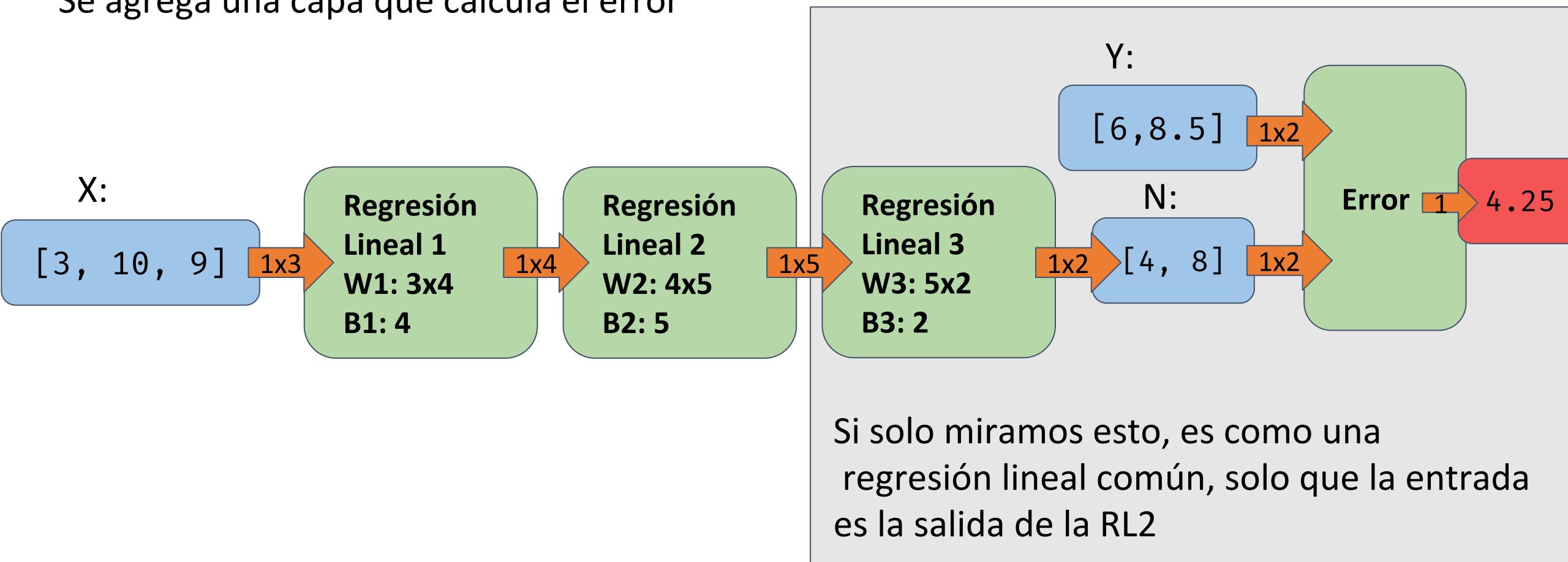
```
N = RL3( RL2( RL1(X) ) )  
t1 = RL1(X)  
t2 = RL2(t1)  
N = RL3(t2)
```

```
N = ((X W1+B1) W2+B2) W3+B3  
t1 = X W1+B1  
t2 = t1 W2 + B2  
N = t2 W3 + B3
```

```
def forward(x, RL1, RL2, RL3)  
    t1=RL1.forward(x)  
    t2=RL2.forward(t1)  
    N = RL3.forward(t2)  
    return N
```

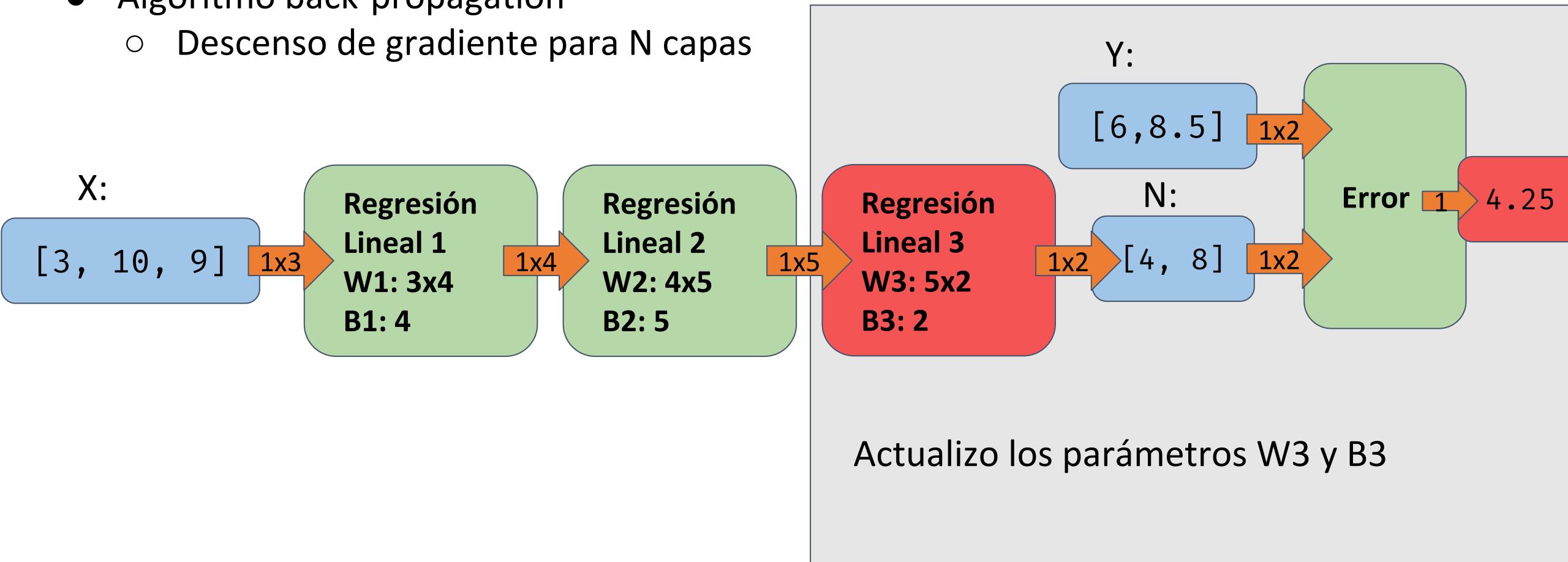
# Redes Neuronales - Aprendizaje

Se agrega una capa que calcula el error



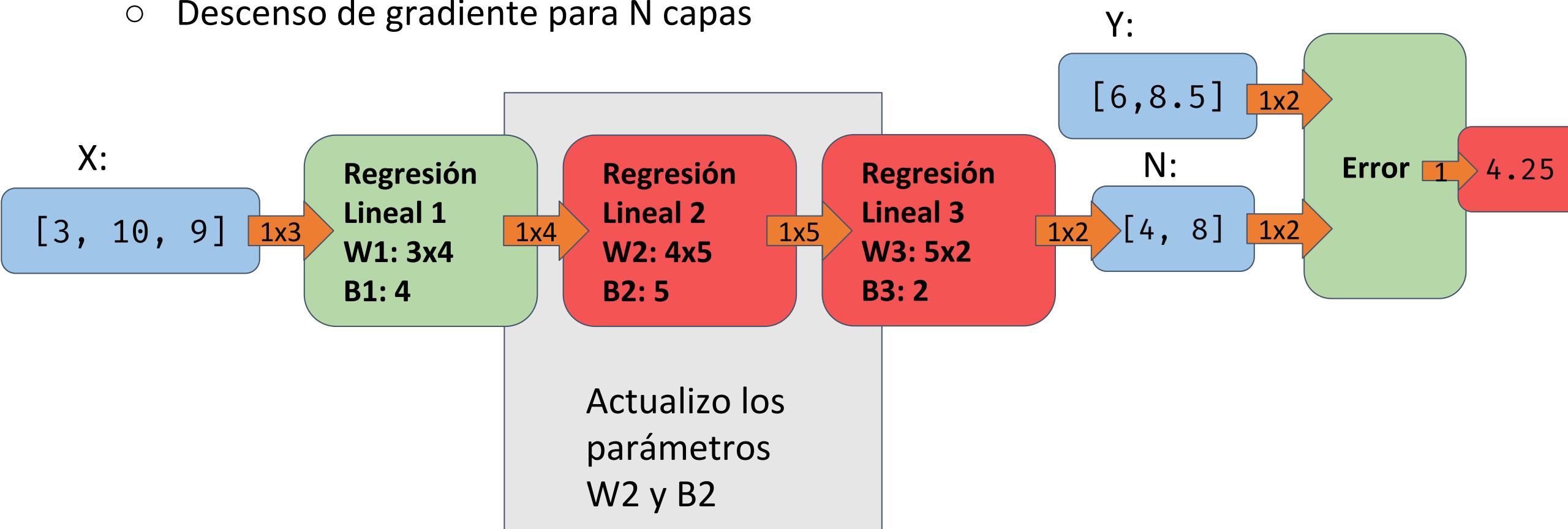
# Redes Neuronales - Aprendizaje

- Función o pasada **backward()**
- Algoritmo back-propagation
  - Descenso de gradiente para N capas



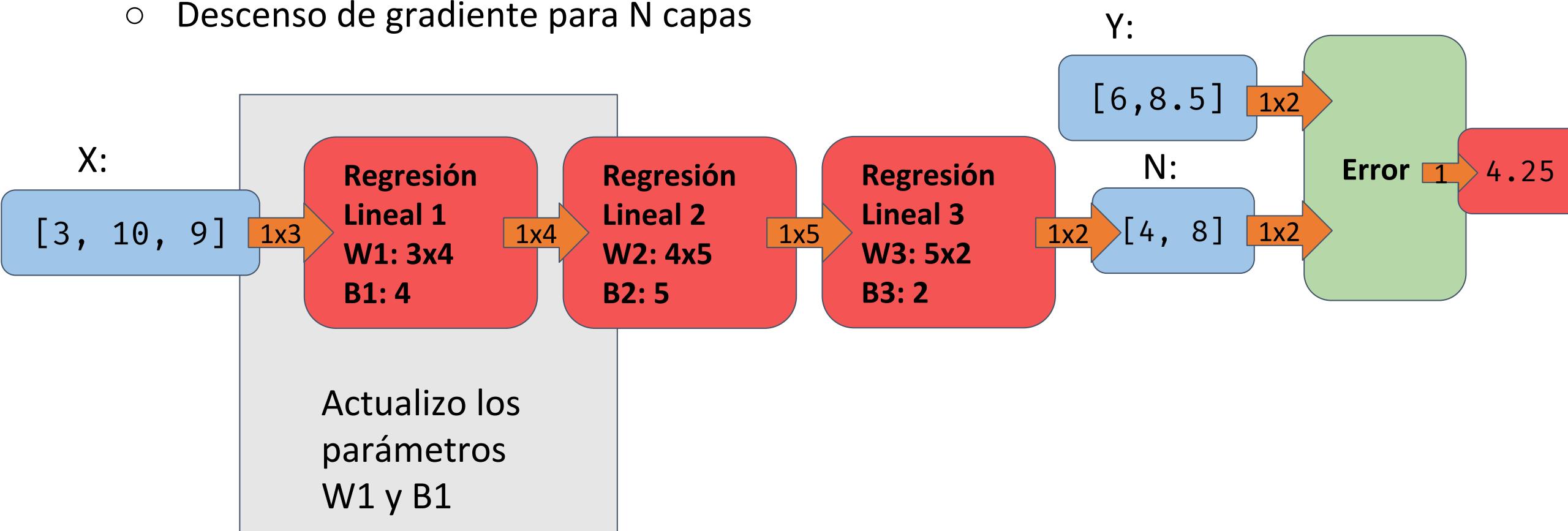
# Redes Neuronales - Aprendizaje

- Función o pasada **backward()**
- Algoritmo back-propagation
  - Descenso de gradiente para N capas



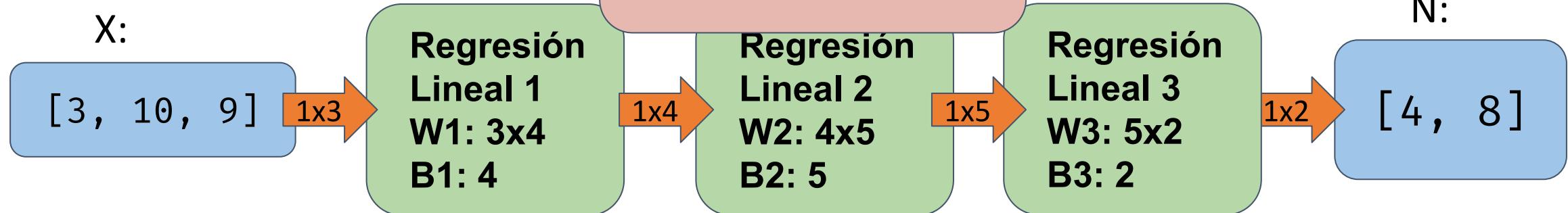
# Redes Neuronales - Aprendizaje

- Función o pasada **backward()**
- Algoritmo back-propagation
  - Descenso de gradiente para N capas



# Redes Neuronales - Funciones de activación

Pero esto no va a funcionar..



3 regresiones = 1  
regresión!

W y V son matrices constantes  
=> W.V es otra matriz.  
W es una matriz y B es un vector  
=> W.B es otro vector

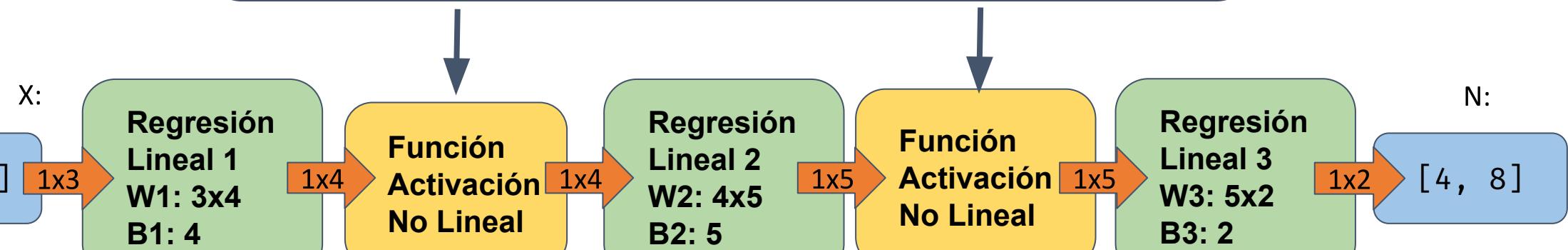
$$\begin{aligned} N &= ((X W1 + B1) W2 + B2) W3 + B3 \\ N &= ((X \textcolor{red}{W1} \textcolor{blue}{W2} + \textcolor{blue}{B1} \textcolor{red}{W2}) + \textcolor{blue}{B2}) W3 + \\ &\quad B3 \\ N &= (X \textcolor{red}{W4} + \textcolor{blue}{B4}) \textcolor{green}{W3} + B3 \\ N &= (X \textcolor{red}{W4} \textcolor{blue}{W3} + \textcolor{blue}{B4} \textcolor{red}{W3}) + \textcolor{blue}{B3} \\ N &= X \textcolor{red}{W5} + \textcolor{blue}{B5} \end{aligned}$$

# Redes Neuronales - Funciones de activación

Cualquier función, siempre que sea **derivable**:

$\sin(x)$ ,  $\cos(x)$ ,  $\tan(x)$

$f(x)=x^2$ ,  $g(x)=x^{25} \cdot 5 - 584$ , etc

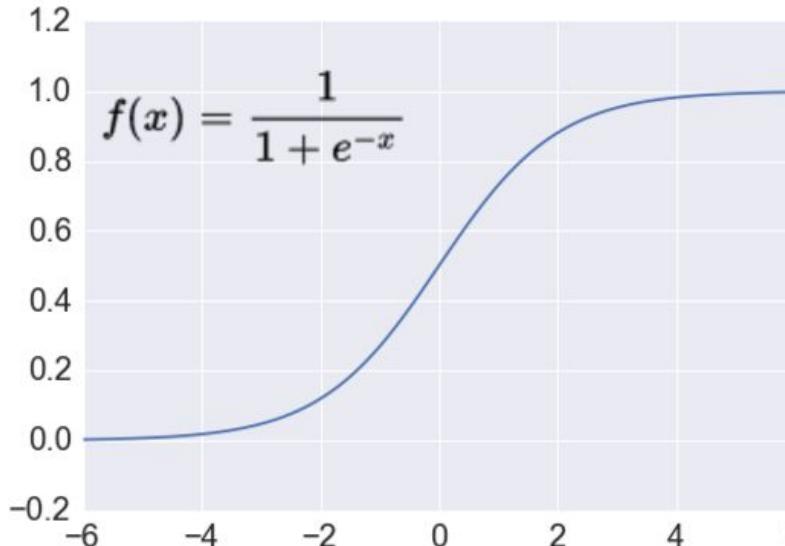


Funciones populares: **ReLU(x)**, **Tanh(x)** o **Sig(x)**

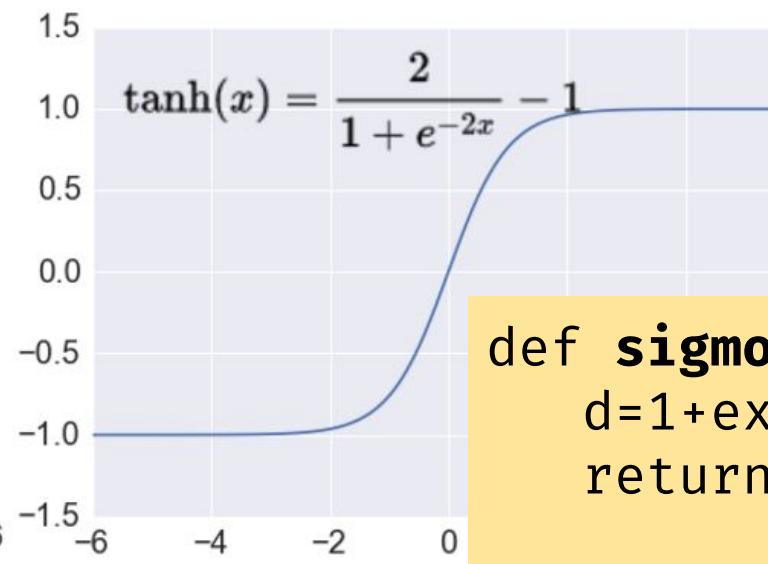
- Capas SIN parámetros
  - No se entrenan!

# Redes Neuronales - Funciones de Activación

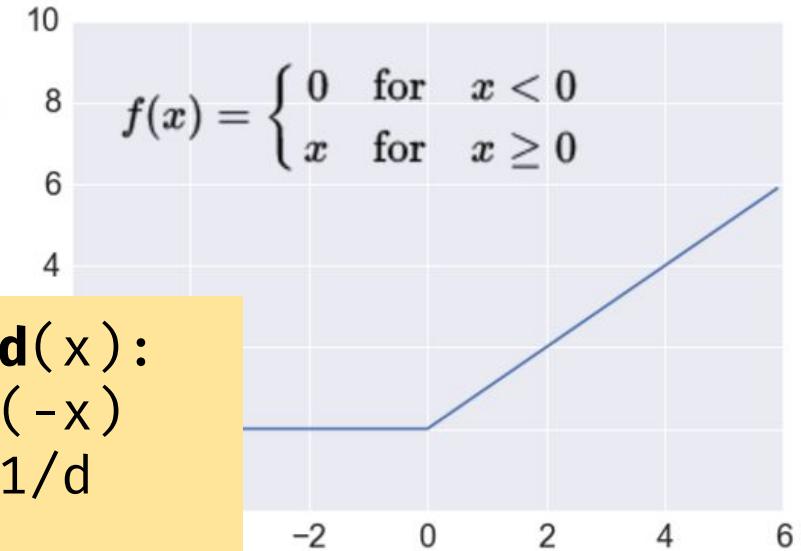
Sigmoid



TanH



ReLU



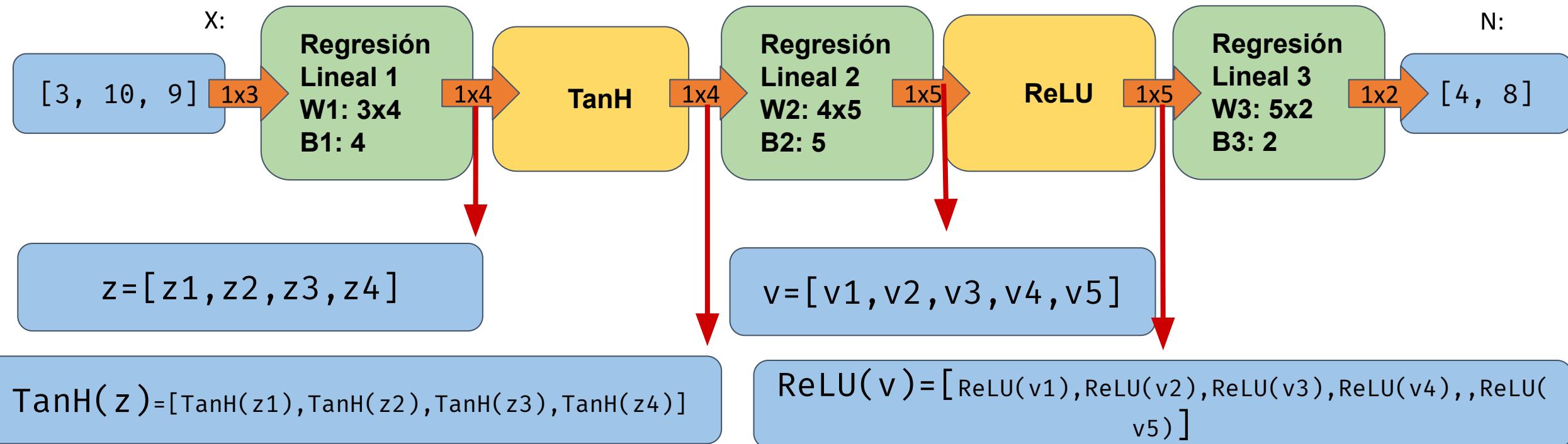
- Sig = la misma que regresión logística
- TanH = (Sigmoid\*2)-1
- ReLU = 0 en los negativos, lineal en los positivos
- Generalmente se usa ReLU o variantes, más fácil de entrenar

```
def sigmoid(x):  
    d=1+exp(-x)  
    return 1/d
```

```
def TanH(x)  
    s=sigmoid(x)  
    return (x*2)-1
```

```
def ReLU(x):  
    if x<=0:  
        return 0  
    else:  
        return x
```

# Redes Neuronales - Funciones de activación



# Regresión Con Redes + Keras

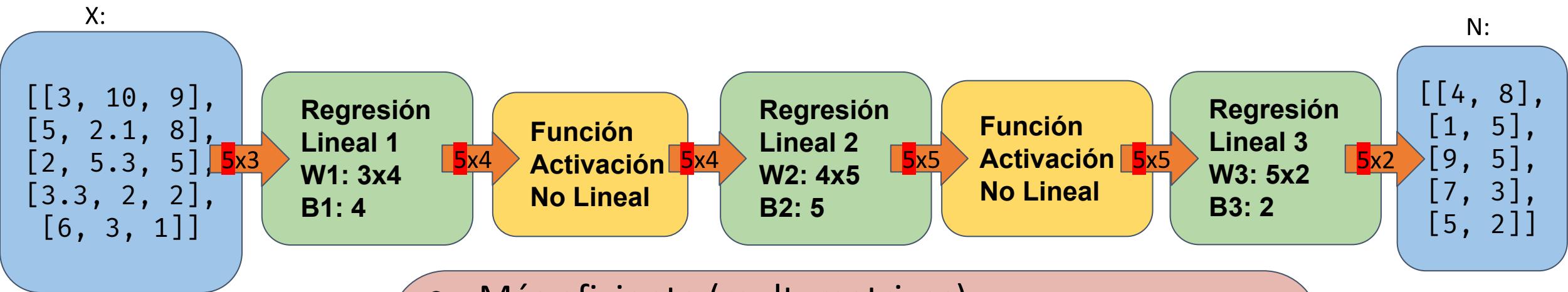
```
x,y=cargar_dataset()
nx,d_in = x.shape # x tiene tamaño n x d_in
ny,d_out = y.shape # y tiene tamaño n x d_out
import keras

model=keras.Sequential()
model.add(keras.Dense(4,input_shape=[d_in],activation='tanh'))
model.add(keras.Dense(5,activation='relu'))
model.add(keras.Dense(d_out,activation='none'))
model.compile(loss='mse', # error cuadratico medio
              optimizer='sgd') # descenso de gradiente
history = model.fit(x,y,epochs=100,batch_size=32)
y_predicted=model.predict(x)
```

- Los valores 4 y 5 son arbitrarios
- La cantidad de capas también
- Son **hiperparámetros**

# Redes Neuronales - Batch Size

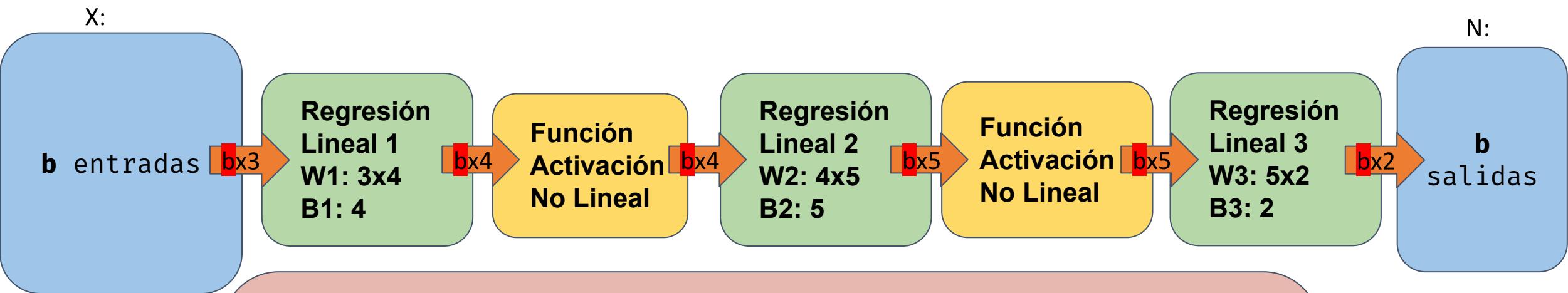
No predecir/entrenar con **todos** los ejemplos o con **uno**. Ejemplo con **batch\_size=5**:



- Más eficiente (mult. matrices)
- Generalmente  $\text{batch\_size} = \text{potencia de } 2$ 
  - 8, 16, 32, 64, 128, 256, 512 (no más)
- Con GPUs:
  - Maximiza el uso de la memoria
  - Minimiza overhead copias
- No se puede usar todo el dataset! ( $N= \text{millones}$ )

# Redes Neuronales - Batch Size

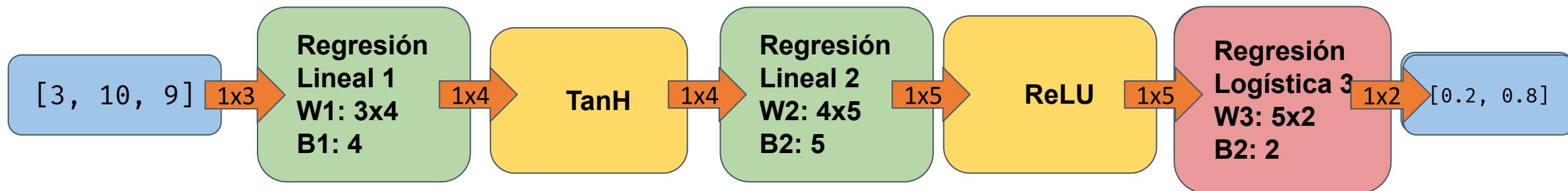
Técnicamente, expresar los tamaños de entrada/salida como  $b \times n$  ( $b$  es el batch\_size)



- **$b = \text{batch\_size}$**  puede modificarse de forma según sea necesario
  - **Entrenar** con una GPU potente con **batch\_size grande**
  - **Predecir** con GPU menos potente con **batch\_size chico**
- Esto aplica a todos los ejemplos vistos en esta teoría (y siguientes) aunque no se diga explícitamente

# Redes Neuronales - Clasificación

Modificar la red anterior para **clasificar**:



- Capas intercambiables
  - Pero verificar tamaños
- Cambio última capa:
  - Cambia el tipo de dato de salida
- Última capa se llama **cabeza** o **head**.

# Clasificación Con Redes + Keras

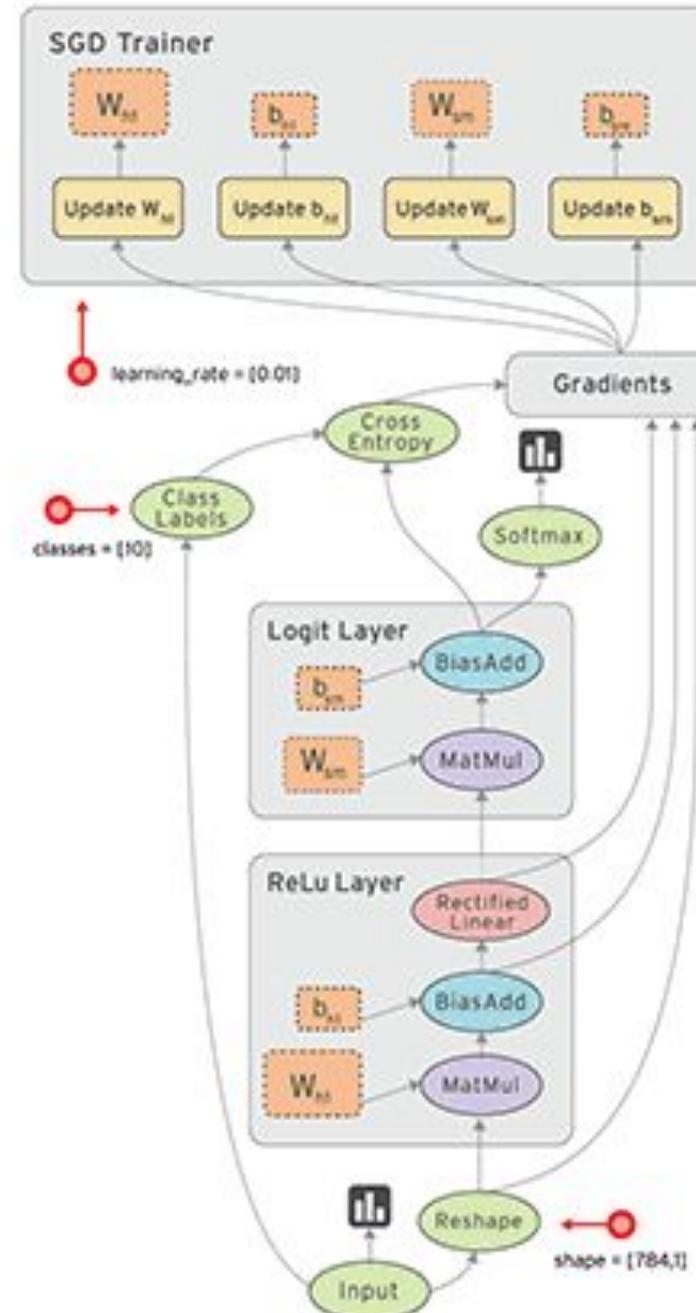
```
x,y=cargar_dataset(one_hot=True)
nx,d_in = x.shape # x tiene tamaño n x d_in
ny,d_out = y.shape # y tiene tamaño n x d_out
import keras

model=keras.Sequential()
model.add(keras.Dense(4,input_shape=[d_in],activation='tanh'))
model.add(keras.Dense(5,activation='relu'))
model.add(keras.Dense(d_out,activation='softmax'))
model.compile(loss='categorical_crossentropy', # ent cruz
              optimizer='sgd', # descenso de gradiente
              metrics='accuracy')
history = model.fit(x,y,epochs=100,batch_size=32)
y_predicted=model.predict(x)
```

# Redes - Entrenamiento (fit)

Por cada **batch**:

- Se calcula la función de salida
  - Softmax
- Se calcula el error
  - Cross Entropy
  - Con las etiquetas de clase
- Se calculan los gradientes
- Se actualizan los parámetros
  - Distintos algoritmos para optimizar
  - Ej: descenso de gradiente estocástico (SGD)



# Clasificación - Accuracy

Accuracy (% aciertos) = Indica % de ejemplos que clasificó correctamente

Carrera	P(Carrera)			Predicción	Acierto
	LS	LI	IC		
LS = 0, LI = 1, IC = 2	0.01	0.45	<b>0.55</b>	2	0
0	0.09	<b>0.39</b>	<b>0.52</b>	2	0
1	0.09	<b>0.39</b>	<b>0.52</b>	2	0
2	0.02	<b>0.73</b>	0.24	1	0
0	<b>0.36</b>	<b>0.36</b>	0.27	0	1
2	<b>0.68</b>	0.15	<b>0.17</b>	0	0
1	<b>0.55</b>	0.27	0.18	0	0
0	<b>0.91</b>	0.05	0.05	0	1

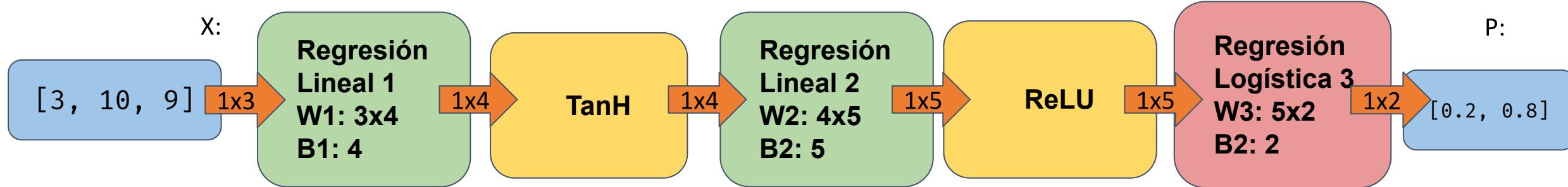
- $P(\text{Carrera}).\text{argmax}()$
- El índice del mayor elemento

- Acierto = carrera == Predicción
- aciertos = Acierto.sum()

## Cálculo del Accuracy

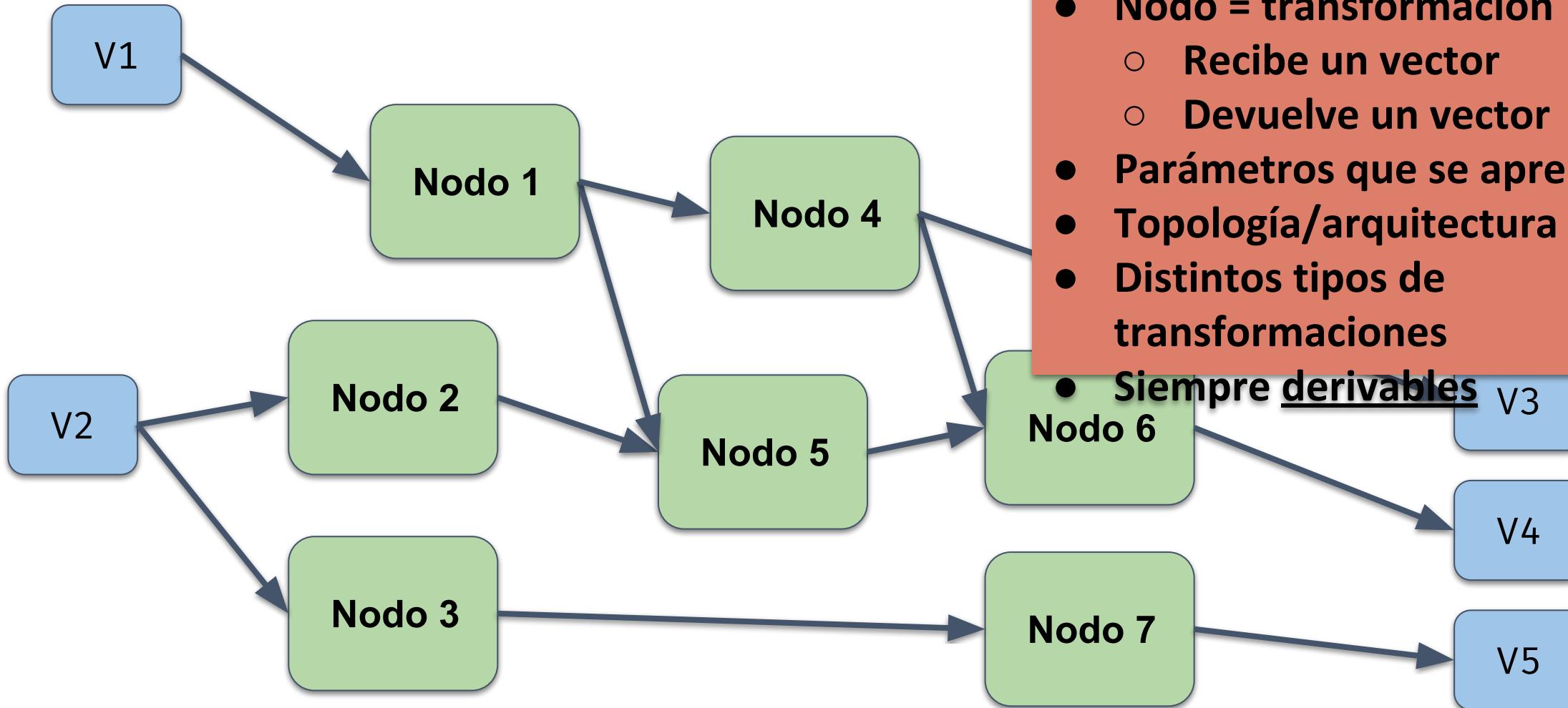
- N = 8
- aciertos=2
- Accuracy = aciertos/N = 2/8 = 0.25
- Accuracy(%) = 25%

# Redes Neuronales - Resumen



- Transforman vectores de entrada en vectores de salida
- Regresión Lineal/Logística + Funciones de activación
  - Transformaciones no lineales
  - Mayor poder de clasificación/regresión
- Capas modulares
  - Combinar de cualquier forma => *arquitecturas o topologías*
- Algoritmo de entrenamiento para **cualquier** red
  - Si todas sus capas son **derivables**

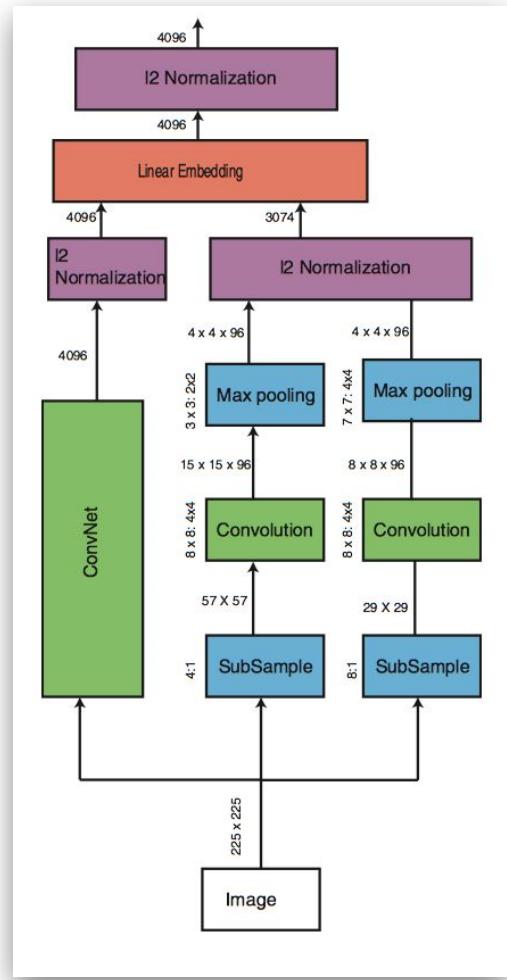
# Redes Neuronales Generales



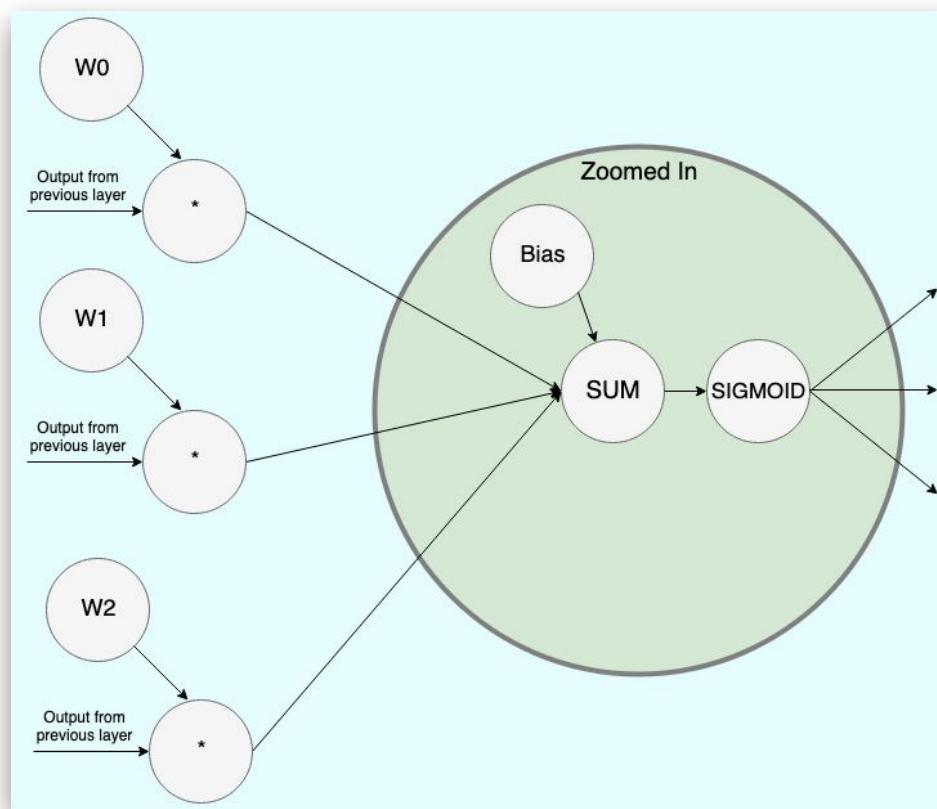
- Grafos computacionales
- Nodo = transformación
  - Recibe un vector
  - Devuelve un vector
- Parámetros que se aprenden
- Topología/arquitectura arbitraria
- Distintos tipos de transformaciones
- Siempre derivables

# Redes Neuronales Generales - Escalas

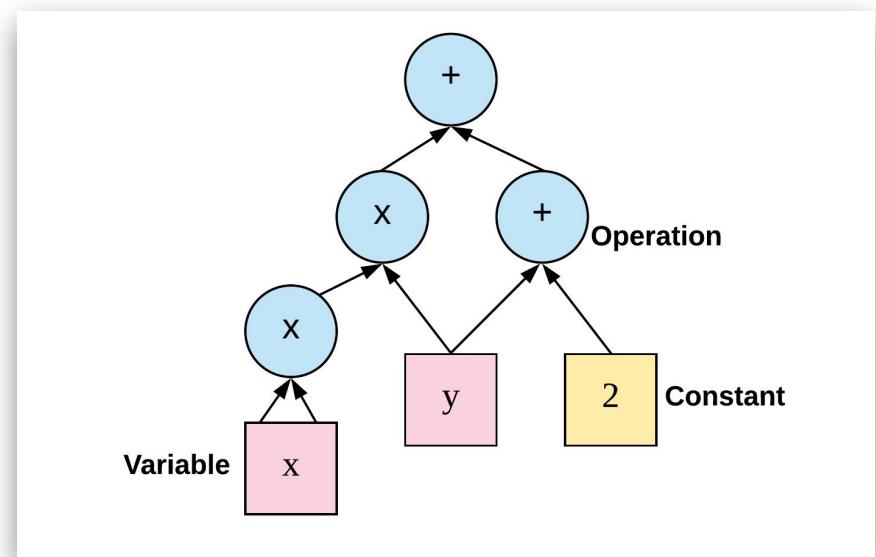
Capas (nivel alto)



Vectores (nivel medio)

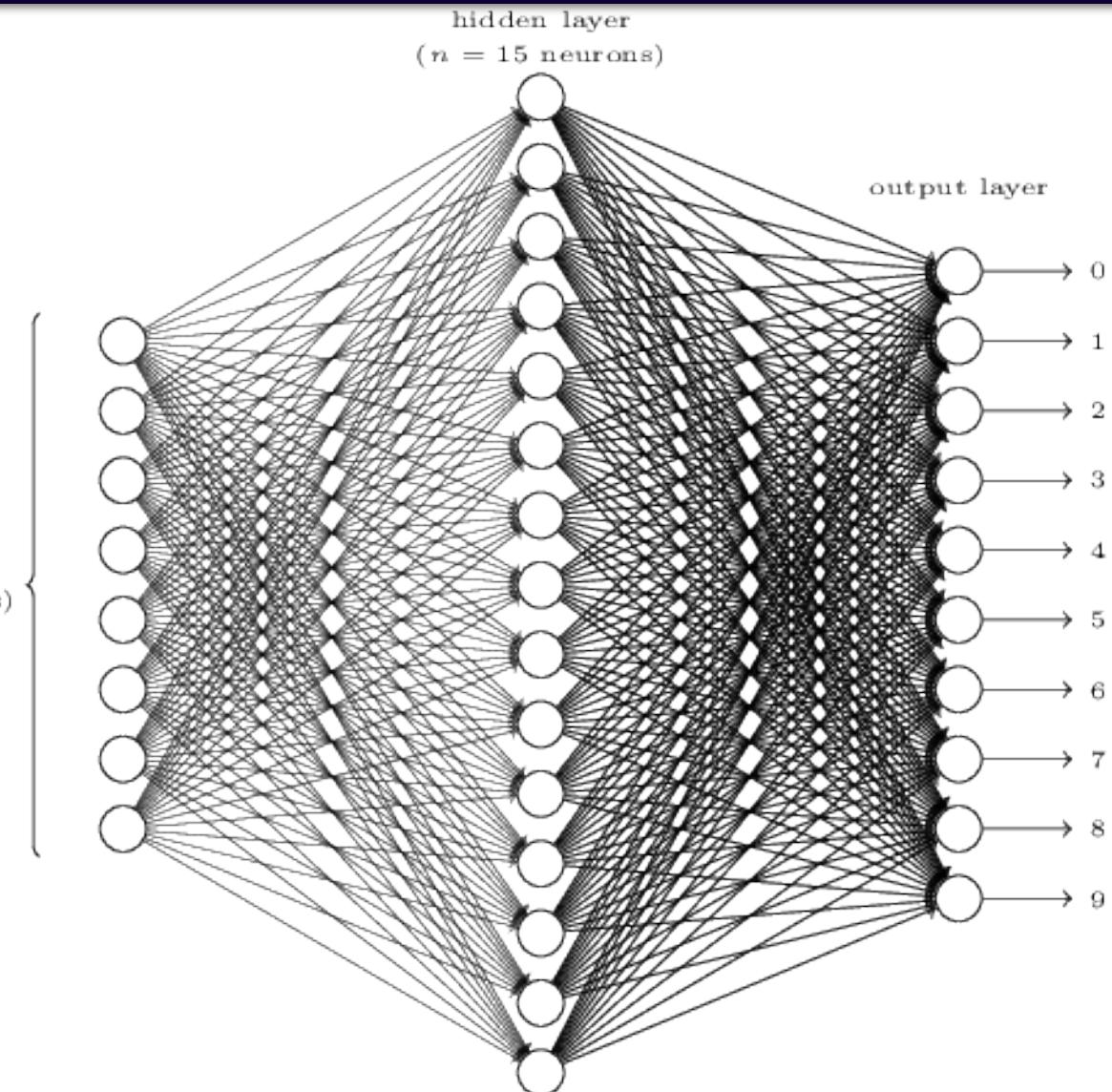


Números (nivel bajo)



# Redes Neuronales Generales - Nomenclatura

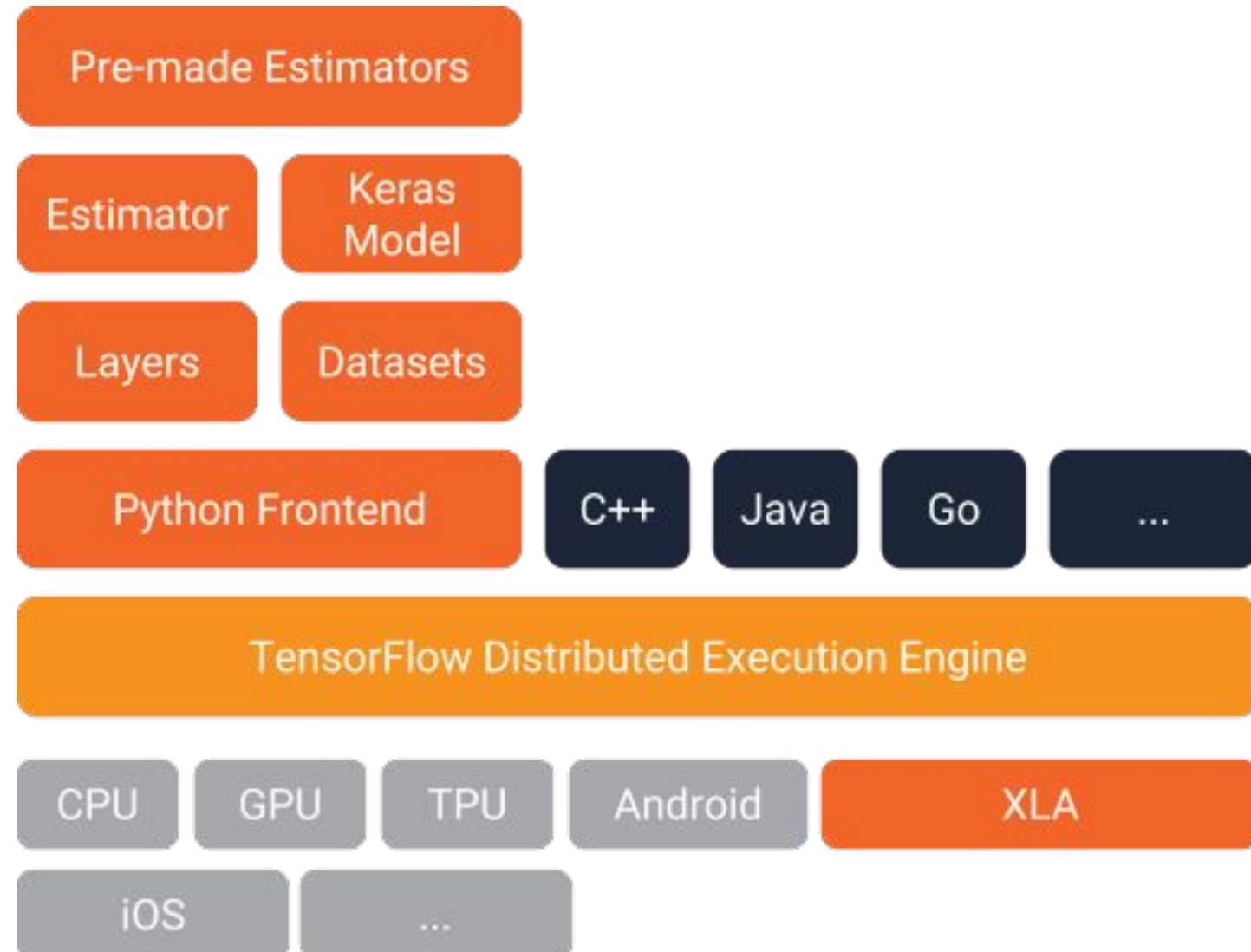
- Nomenclatura inversa
  - Nodos = valores
  - Aristas = funciones/capas
- Interpretación biológica
  - Nodos = Neuronas, con valor de *activación*
  - Aristas = Sinapsis *transformadoras*
- Se usan ambas nomenclaturas
  - Aprender ambas y a distinguir



# Tensorflow y Keras

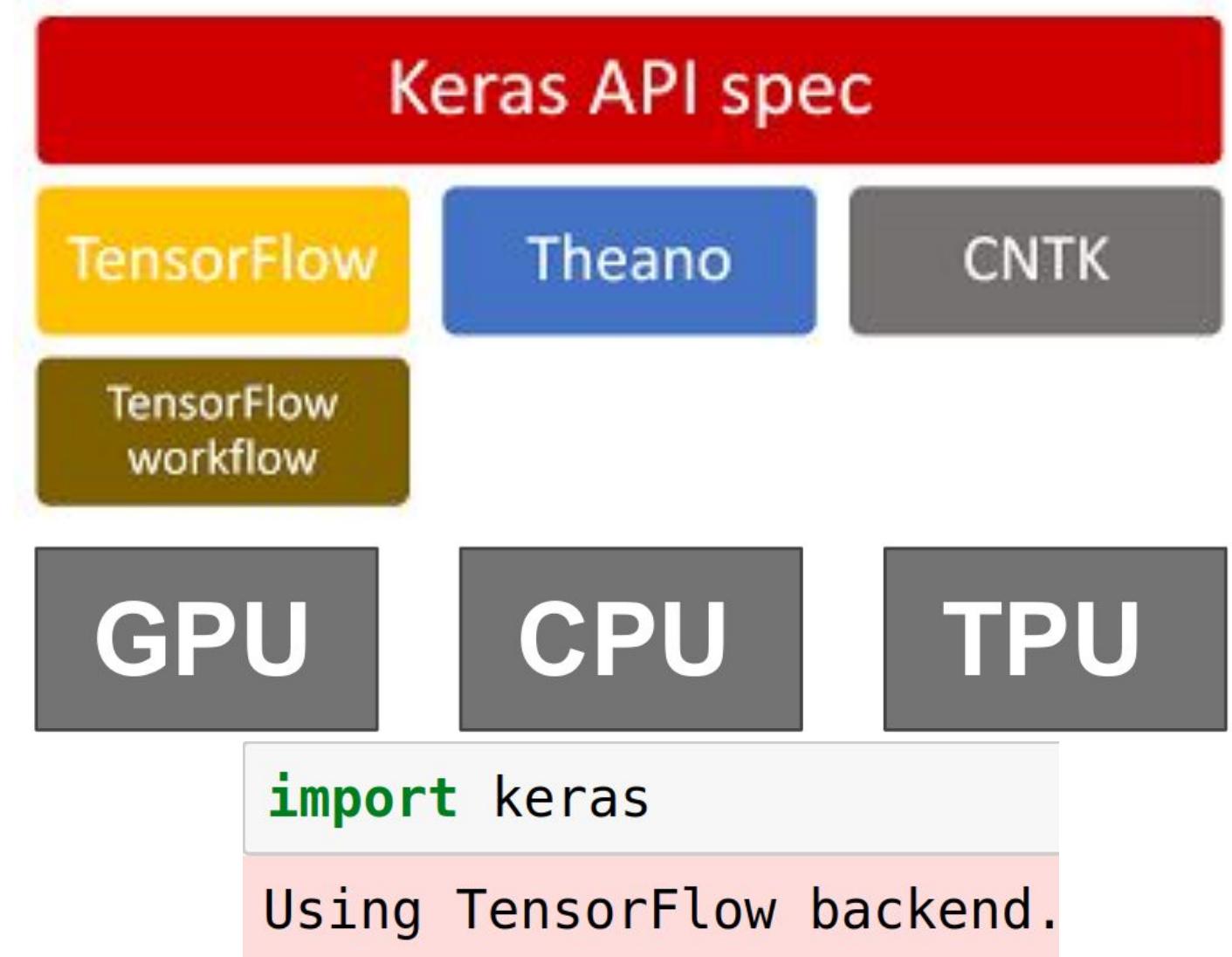
# Tensorflow - Arquitectura

- Core en C++
- API en varios lenguajes
- Elementos
  - API de operadores
    - Grafo de computación
    - **Derivadas automáticas**
  - Implementación optimizada para cada dispositivo
  - Comunicación entre dispositivos



# Keras - Arquitectura

- Keras: interfaz de alto nivel
  - Otras libs hacen el trabajo pesado (*backends*)
- Para definir Redes Neuronales
  - Y otros modelos diferenciables
- Provee API unificada para:
  - Cargar datos
  - Entrenar (fit)
  - Predecir (predict)
  - Evaluar (evaluate)



# Tensorflow vs Keras

## Keras

```
# MODELO
model=keras.Sequential(
    keras.Dense(4,activation='softmax')]

# ENTRENAMIENTO
model.compile(loss='mse',
optimizer='sgd')
model.fit(x,y,epochs=100)

# PREDICCION
y_predicted=model.predict(x)
```

## Tensorflow

```
# MODELO

# Variables de entrada
var_X = tf.placeholder("float")
var_Y = tf.placeholder("float")

# Parámetros
W = tf.Variable(np.random.randn())
b = tf.Variable(np.random.randn())

# Salida
y_abs = tf.add(tf.multiply(x, w), b)
y_pred=tf.softmax(y_abs)
```

# Tensorflow vs Keras

## Keras

```
# MODELO
model=keras.Sequential(
    keras.Dense(4,activation
='softmax'])

# ENTRENAMIENTO
model.compile(loss='mse',
optimizer='sgd')
model.fit(x,y,epochs=100)

# PREDICCION
y_predicted=model.predict(
x)
```

## Tensorflow

```
# ENTRENAMIENTO
errors=tf.nn.sigmoid_cross_entropy_with_logits(logits=y_pred, labels=varY)
error=tf.reduce_mean(errors)

opt = tf.train.GradientDescentOptimizer(
    learning_rate=0.001).minimize(error)

## varias líneas de sarasa ##

# epocas y batches “a mano”
epochs=1000
for i in range(epochs):
    for (batch_x, batch_y) in zip(x, y):
        sess.run(opt,
            feed_dict={varX:batch_x,varY:batch_y})
```

# Tensorflow vs Keras

## Keras

```
# MODELO
model=keras.Sequential(
    keras.Dense(4,activation
='softmax')]

# ENTRENAMIENTO
model.compile(loss='mse',
optimizer='sgd')
model.fit(x,y,epochs=100)

# PREDICCIÓN
y_predicted=model.predict(
x)
```

## Tensorflow

```
# PREDICCIÓN
y_pred_value = sess.run(y_pred,
feed_dict={varX:x})
```

# Tensorflow vs Keras

## Keras

- Alto nivel (capas)
- Usa TF como backend
  - Backend: provee la implementación real de las operaciones
- Fácil uso
- No tan customizable
- Input y output con arreglos de numpy
- Puede usar operadores/capas custom implementadas en TF

## Tensorflow (TF)

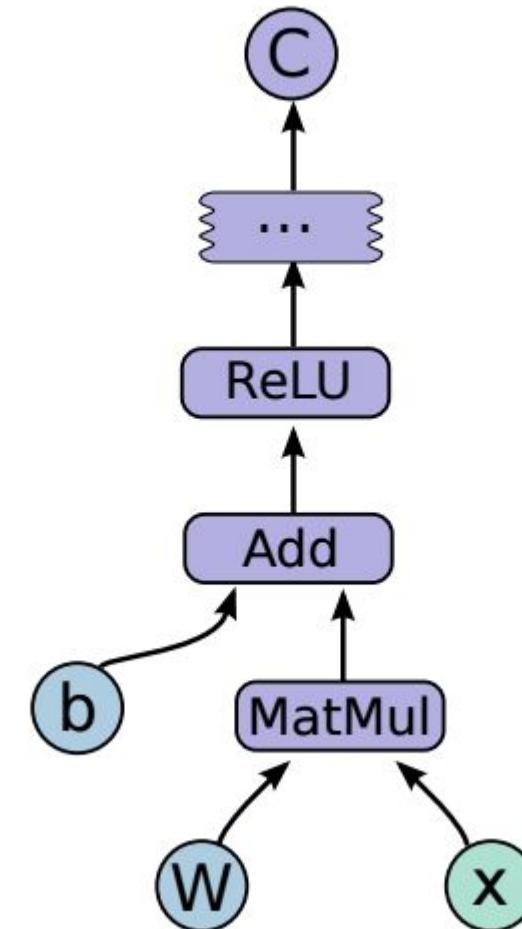
- Nivel intermedio (tensores)
  - tensores = matrices n-dim
- Muy customizable
- Útil para definir nuevos operadores
- tf.tensor: clase propia de tensor
  - convertir desde/hacia numpy
- Repetitivo para redes
  - Keras está incluido desde TF 1.3 para facilitar su uso
- Entrenamiento y predicción “manuales”

# Tensorflow - Grafos Declarativos

- Operaciones de tensorflow **declarativas**
  - NO se ejecutan inmediatamente
  - Definen grafos
  - Ejecución mediante sess.run(op)
- Desacoplan definición de la ejecución
  - Como una función!
  - Permite optimizar el grafo
- Ejemplo:

```
x = tf.placeholder("float")
W = tf.Variable(np.random.randn())
b = tf.Variable(np.random.randn())
```

```
# Salida
C = tf.nn.relu(tf.add(tf.multiply(x, W), b))
C_value = sess.run(C, feed_dict={x: INPUT }) # vectores numpy
```



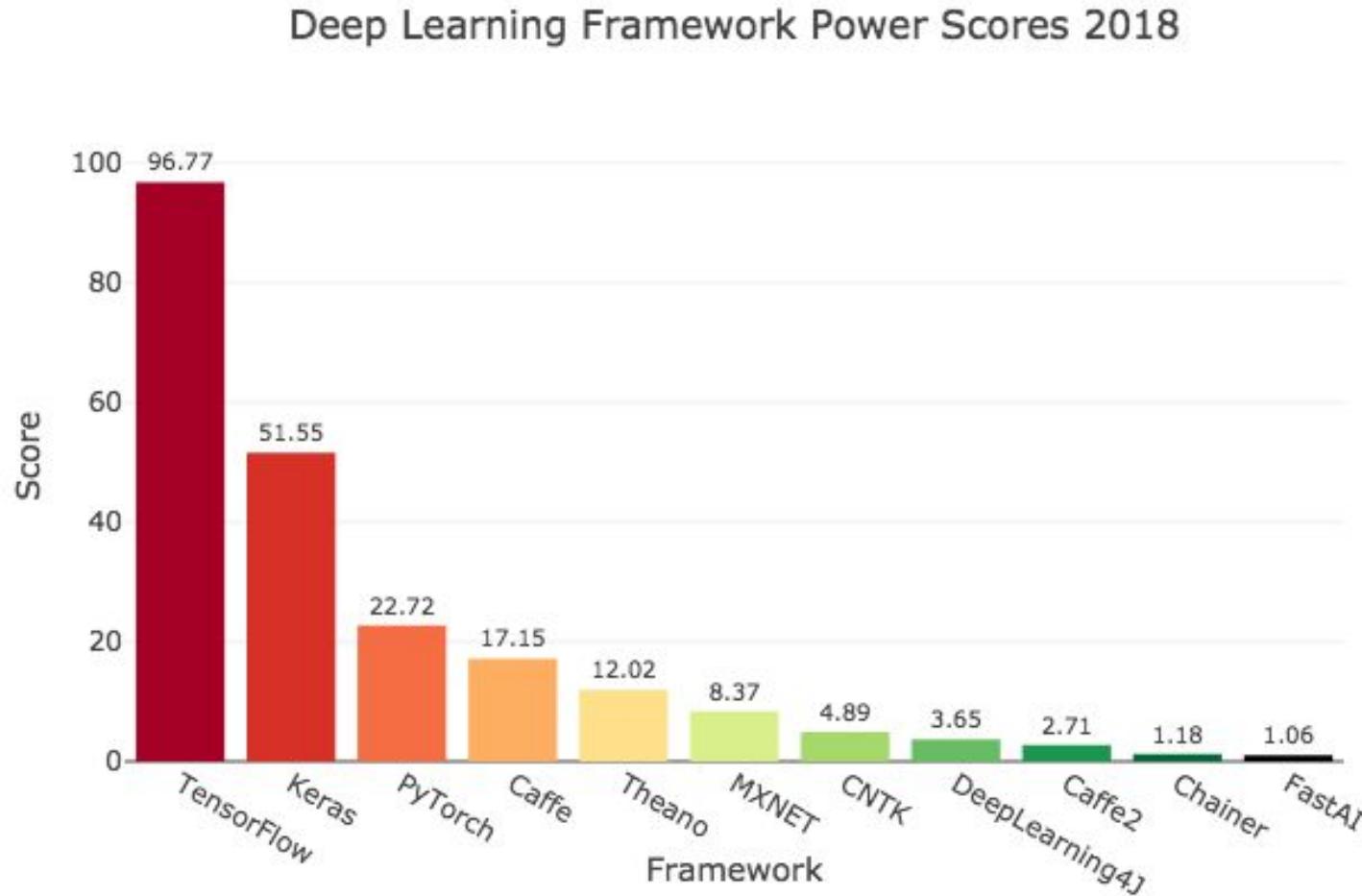
# TensorFlow - Nueva API Eager

- Nueva API permite ejecución **Eager**
  - Imperativa
  - Se ejecutan inmediatamente
  - Keras NO las utiliza
- Ejemplo:
- # habilitar API eager  
tf.enable\_eager\_execution()

```
x = INPUT # un vector de numpy
W = tf.Variable(np.random.randn())
b = tf.Variable(np.random.randn())

# Salida
C = tf.nn.relu(tf.add(tf.multiply(x, W), b))
# C ya tiene el output (vector de numpy)
```

# TensorFlow vs Keras vs Otros (2018)



- En este curso sólo usamos Keras
  - Simple
  - Fácil de aprender
- El interop entre tf y keras es muy bueno
  - Podés combinar ambos en un mismo proyecto
  - Usar tf solo donde es necesario