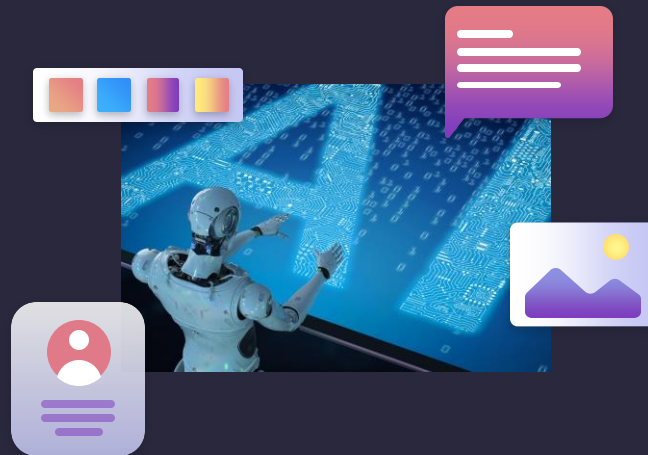


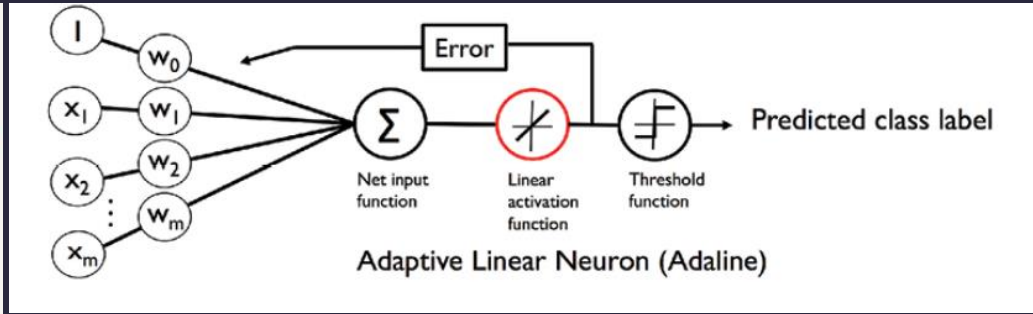
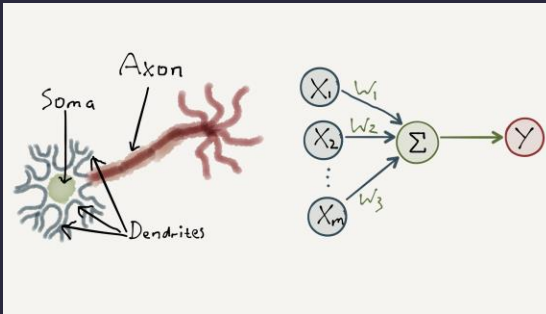
# /TEMA 4

## Algoritmos para A.A.



<https://www.youtube.com/watch?v=8Dotiqbtvoo>





# evolución: Desde el perceptron a ADALINE

entrenar un algoritmo para la detección de tipos de flores



# /CONTENIDOS



**/01** /UN DATAFRAME POPULAR.

**/02** /ANALIZAR LOS DATOS

**/03** /VISUALIZAR LOS DATOS

**/04** /UNA NEURONA FÁCIL DE PROGRAMAR: EL PERCEPTRÓN

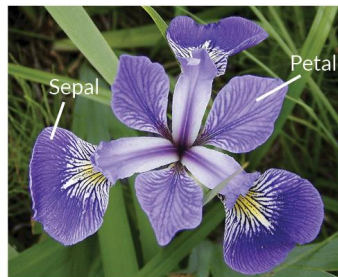
**/05** /LA EVOLUCIÓN: ADaptative Linear NEuron



# /01

# /un DataFrame Popular

Breve historia, sin  
repetirnos



**Iris Versicolor**



**Iris Setosa**



**Iris Virginica**

# /muestras, características y etiquetas de clase

## Muestras

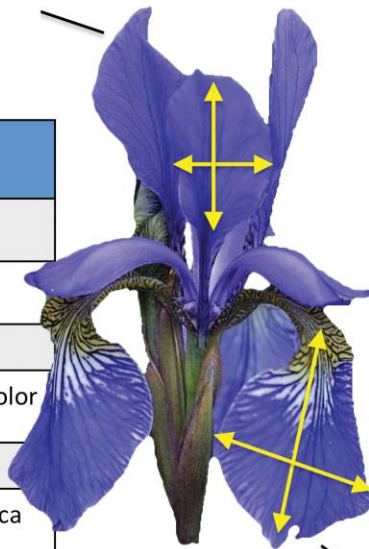
(instancias, observaciones)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

## Características

(atributos, medidas, tamaños)

## Pétalo



## Sépalo

## Etiquetas de clase

(destinos)

# /02 /Analizar Datos

Carga con pandas el DataFrame de: <https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data>

In [5]: `import pandas as pd`

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)  
df.tail()
```

Out[5]:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

1. Crea un array sólo con las variedades 'Iris-setosa' y 'Iris-versicolor' (las primeras 100 líneas)
2. Con numpy, crea un array con un las 100 muestras con valores -1 para setosa y 1 para versicolor

```
In [10]: #con numpy, mapea en un array las muestras con valores -1 para setosa y 1 para versicolor
import numpy as np
y = np.where(y == 'Iris-setosa', -1, 1)
y
```

```
Out[10]: array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

<https://numpy.org/doc/stable/reference/generated/numpy.where.html>

### 3. Extrae en una variable X la longitud del sépalo y del pétalo (columnas 0 y 1) de las 100 primeras muestras (setosa y versicolor)

```
In [11]: X= df.iloc[0:100, [0,2]].values  
X
```

```
Out[11]: array([[5.1, 1.4],  
                [4.9, 1.4],  
                [4.7, 1.3],  
                [4.6, 1.5],  
                [5. , 1.4],  
                [5.4, 1.7],  
                [4.6, 1.4],  
                [5. , 1.5],  
                [4.4, 1.4],  
                [4.9, 1.5],
```

<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html>



## /03 /VISUALIZAR LOS DATOS

*Visualiza en un gráfico de matplotlib las clases setosa y versicolor, usando la longitud de sépalo en el eje de las X y la longitud del pétalo en el eje de las Y*

### RECUERDA

[https://matplotlib.org/stable/api/matplotlib\\_configuration\\_api.html](https://matplotlib.org/stable/api/matplotlib_configuration_api.html)

[https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.scatter.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html)

[https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.xlabel.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.xlabel.html)

[https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.ylabel.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.ylabel.html)

[https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.legend.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.legend.html)

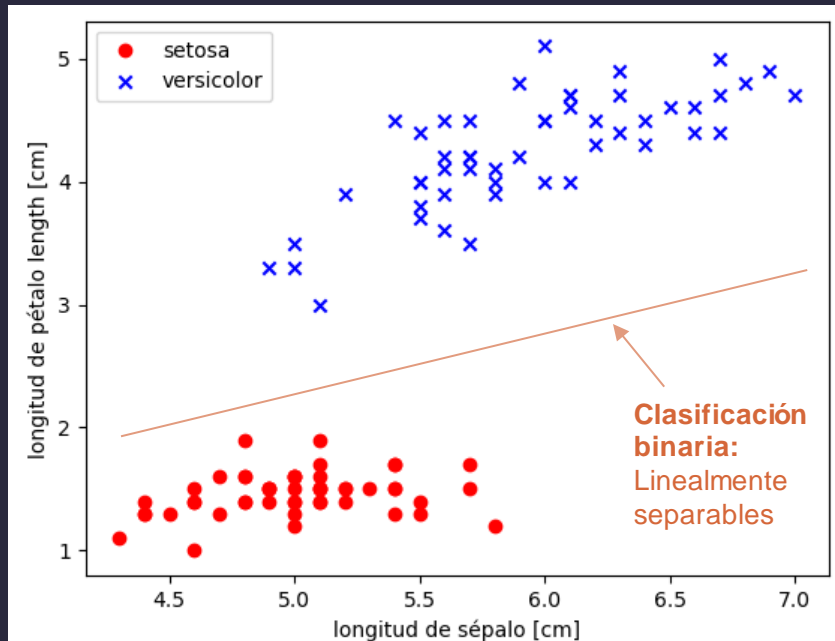
[https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.plot.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html)

## 4. Representa los datos en un gráfico de matplotlib

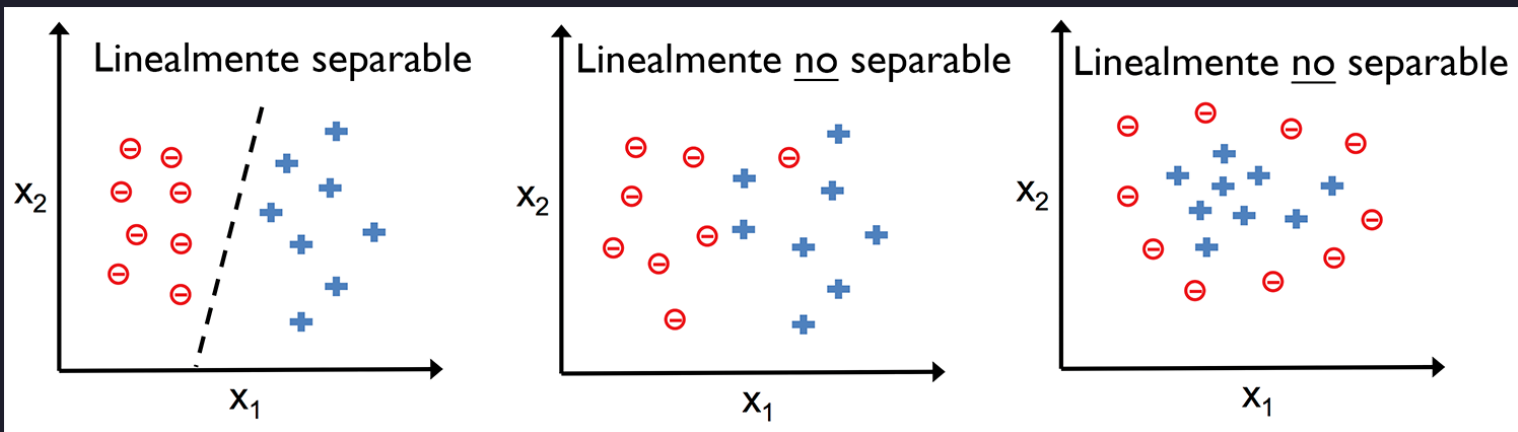
```
import matplotlib.pyplot as plt
# Dibujar los datos
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')

plt.xlabel('longitud de sépalo [cm]')
plt.ylabel('longitud de pétalo length [cm]')
plt.legend(loc='upper left')

plt.savefig('clasificacion.png', dpi=300)
plt.show()
```

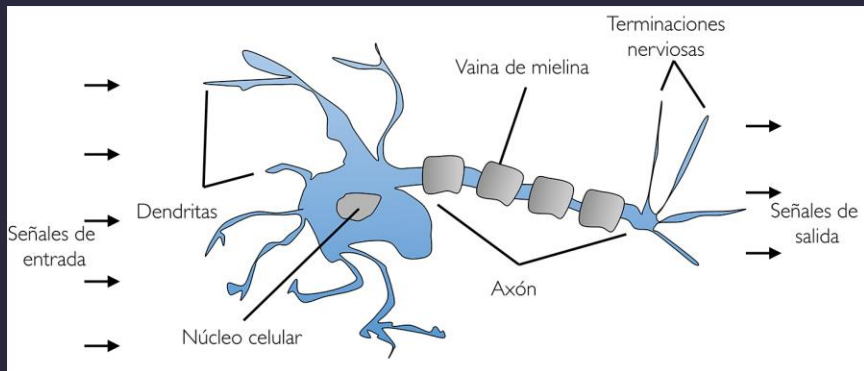


*Si las muestras son linealmente separables y de dos dimensiones, se puede “entrenar” un algoritmo llamado **perceptron** para saber diferenciar las especies.*



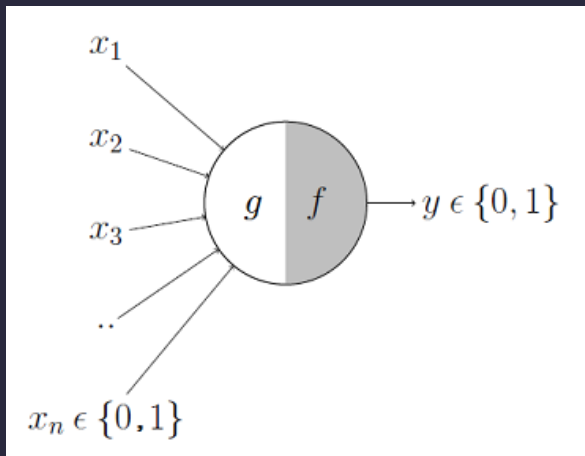
# /04 /UNA NEURONA FÁCIL DE PROGRAMAR: EL PERCEPTRÓN

## ¿Qué es una neurona?



Ramón y Cajal: Padre de la neurociencia

## Una neurona artificial: La neurona MCP (McCullock-Pitts, 1943)



La función  $g$  toma entradas  $X$  y las suma:  
Las  $x$  son entradas que pueden ser **excitadoras** o **inhibitorias**

$$g(x_1, x_2, x_3, \dots, x_n) = g(\mathbf{x}) = \sum_{i=1}^n x_i$$

La función  $f$  toma el valor de  $g(x)$  y produce un 0 o un 1:

$$y = f(g(\mathbf{x})) = \begin{cases} 1 & \text{if } g(\mathbf{x}) \geq \theta \\ 0 & \text{if } g(\mathbf{x}) < \theta \end{cases}$$

Theta representa el parámetro de umbral

## Ejemplo de neurona MCP:

*Neurona que me predice si me apetece ver o no un partido de baloncesto de la NBA*

ENTRADAS a la función g

- ¿Juega Luka Doncic?
  - ¿Juega algún jugador español?
  - ¿Juegan los Lakers?
  - ¿El partido comienza después de la 1am?
- Diagram illustrating the inputs and their weights:
- Inputs 1-3 (Luka Doncic, Spanish player, Lakers) are grouped by a bracket and labeled **excitadoras** (excitatory), leading to **Se suman** (they are added).
  - Input 4 (Start time after 1am) is labeled **inhibitoria** (inhibitory), leading to **Provoca salida 0** (causes output 0).

Mi umbral  $\theta$  es 2 -> veo el partido si ocurren dos de estas:



otra forma de expresarlo:

(suma de entradas) -  $\theta \geq 0$  **veo el partido**  
(suma de entradas) -  $\theta < 0$  **NO veo el partido**



a  $-\theta$  se le llama sesgo



# Limitaciones de la neurona MCP:

¿Qué ocurre con las entradas “no booleanas”?

¿Son todas las entradas iguales?

¿Siempre tengo que fijar el umbral?

¿Qué ocurre con las funciones no linealmente separables?

Frank Rosenblat

1953: ***The perceptron:  
a perceiving and recognizing automaton***



# Perceptrón: Una neurona artificial fácil de implementar

Objetivo: Crear un clasificador. A partir de unas entradas y unos pesos, generar una salida binaria (clasificación positiva o negativa)

$w_0 = -\theta$  es el sesgo (bias),  $x_0 = 1$

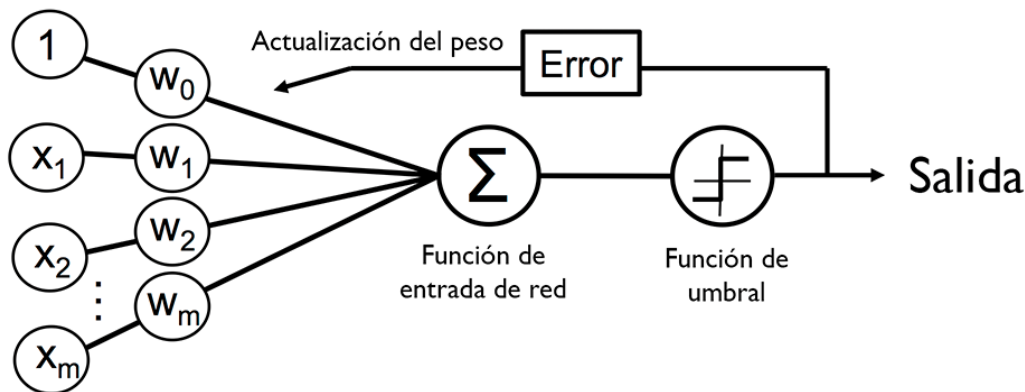
vectores de entrada:

$$w = \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_n \end{bmatrix}, x = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix}$$

$$\left( \sum \right) \text{ entrada} = w_0 x_0 + w_1 x_1 + \dots + w_n x_n = w^t x$$

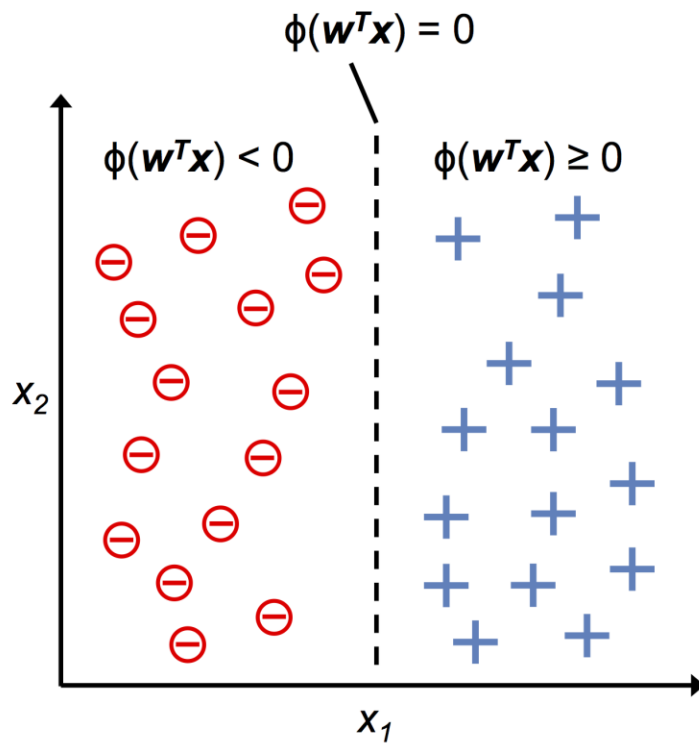
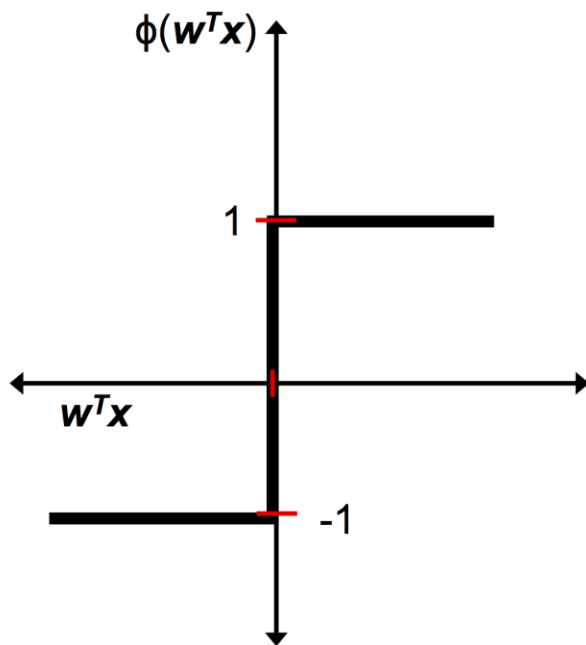
$$\left( \text{Función de umbral} \right) = \begin{cases} 1 & \text{si entrada} \geq 0 \\ -1 & \text{en caso contrario} \end{cases}$$

→ clase positiva  
→ clase negativa





## Función umbral



# La regla de aprendizaje del perceptrón de Rosenblatt

1. Inicializar los pesos a números aleatorios pequeños (distribución normal)
2. Para cada muestra de entrenamiento  $x^i$ 
  - a. Calcular el valor de salida y (etiqueta predicha)
  - b. Actualizar los pesos

$$w_j = w_j + \Delta w_j \rightarrow \Delta w_j = n(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

delta

Etiqueta  
clase  
verdadera

Etiqueta  
clase  
predicha

Rango de  
aprendizaje

El perceptrón predice correctamente una etiqueta de clase -1

$$\Delta w_j = n(-1 - (-1))x_j^{(i)} = 0$$

El perceptrón predice correctamente una etiqueta de clase 1

$$\Delta w_j = n(1 - 1)x_j^{(i)} = 0$$

El perceptrón no corrige nada

El perceptrón hace una predicción errónea de la clase -1

$$\Delta w_j = n(1 - (-1))x_j^{(i)} = n(2)x_j^{(i)}$$

El perceptrón hace una predicción errónea de la clase 1

$$\Delta w_j = n(-1 - 1)x_j^{(i)} = n(-2)x_j^{(i)}$$

El perceptrón “Empuja” hacia la predicción correcta en la siguiente iteración

## PRACTICA 4.1

# Desarrollo de un perceptrón



# 1. Crea una clase “Perceptron” en Python con tres atributos

Parámetros

-----

ratio : float

ratio de aprendizaje (entre 0.0 y 1.0)

n\_iter : int

Pasadas sobre el dataset de entrenamiento.

(llamaremos a este parámetro *épocas*)

semilla : int

Semilla para el generador de números aleatorios

para inicializar el vector de pesos de forma aleatoria

## 2. Añade a la clase dos atributos más:

- El vector de pesos
- Una lista de números enteros para contar cuántas flores se clasifican incorrectamente en cada iteración:

Atributos

-----

w\_ : 1d-array

Pesos después del entrenamiento.

errors\_ : list

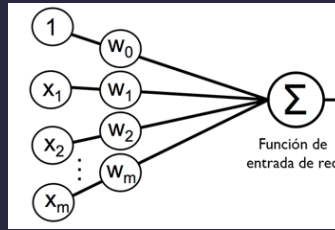
Número de clasificaciones erróneas (updates) en cada época

### 3. Programa el método `__init__` para inicializar los parámetros y los atributos.

El vector de pesos lo puedes inicializar con el método normal de `RandomState`:

```
rgen = np.random.RandomState(semilla)
self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
```

## 4. Programa el método `net_input(self,X)` para calcular la entrada a la red:



$$entrada = w_0x_0 + w_1x_1 + \dots + w_nx_n = w^t x$$

Recuerda:

que  $w_0$  es el sesgo y  $x_0$  es 1

que  $w_1 .. w_n$  son los pesos para cada característica

que  $x_1 .. x_n$  son los valores de las características para una muestra



**5. Programa el método `predict(self,X)` que invoque a `net_input` y devuelva:**

**-1 si es setosa (entrada  $<0$ )**

**1 si es versicolor (entrada  $\geq 0$ )**

## 6. Programa el método $\text{fit}(\text{self}, X, y)$ para entrenar el conjunto de datos y actualizar el vector de pesos $w$ un número de iteraciones $n\_iter$ :

```
#pseudocódigo  
recibe el conjunto de muestras X  
recibe el conjunto de etiquetas reales para cada muestra
```

```
repetir  $n\_iter$  veces (épocas)
```

```
    errors=0
```

```
    para cada Muestra  $X_i$ 
```

```
         $\Delta w_j = \text{ratio} * (\text{EtiquetaReal} - \text{predict}(X_i))$ 
```

```
        Actualizar pesos  $w$  con  $\Delta w_j$ 
```

```
        si  $\Delta w_j \neq 0$  (Error en predicción)
```

```
            errors=errors+1
```

```
    fin-para
```

```
    añadir errors a la lista de errores en cada época
```

```
fin-repetir
```

$$w_j = w_j + \Delta w_j$$

$$\Delta w_j = n(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

↑  
Etiqueta  
clase  
verdadera

↑  
Etiqueta  
clase  
predicha

↑  
Rango de  
aprendizaje

## 7. Crea un objeto perceptrón y entrénalo con los siguientes parámetros:

- 0.1 de ratio de aprendizaje
- 10 épocas
- 1 como valor de semilla para la inicialización de pesos

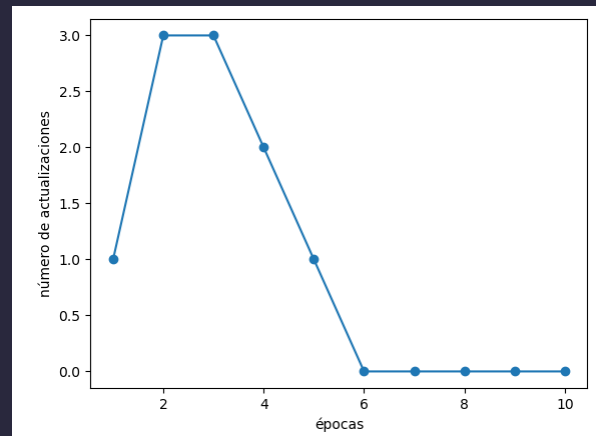
```
#entrenar el perceptrón  
p=Perceptron(0.1,10,1)  
p.fit(X,y)
```

## 8. Muestra convergencia del modelo:

```
p=Perceptron(0.1,10,1)
p.fit(X,y)

plt.plot(range(1, len(p.errors_)+1 ), p.errors_, marker='o')
plt.xlabel('épocas')
plt.ylabel('número de actualizaciones')

# plt.savefig('convergencia.png', dpi=300)
plt.show()
```



¿Cuántas épocas de actualización han sido necesarias para que el perceptrón no falle en sus predicciones?

## 9. Usando el perceptrón entrenado, predice un par de ejemplos de cada flor:

```
sepalo=3  
petalo=5  
print("la flor con longitud de sépalo",sepalo,"y longitud de pétalo", petalo,  
      "es",np.where(p.predict([sepalo, petalo])==1,'versicolor','setosa'))
```

la flor con longitud de sépalo 3 y longitud de pétalo 5 es versicolor

```
sepalo=4.5  
petalo=1.7  
print("la flor con longitud de sépalo",sepalo,"y longitud de pétalo", petalo,  
      "es",np.where(p.predict([sepalo, petalo])==1,'versicolor','setosa'))
```

la flor con longitud de sépalo 4.5 y longitud de pétalo 1.7 es setosa

## 10. Siguientes pasos:

**¿Si deseo crear una app que verifique qué tipo de flor es una muestra en concreto? ¿Qué debo incluir en mi app?**

### AHORA TÚ

DESCARGA EL LIBRO DE JUPYTER  
Y CONTESTA A LAS PREGUNTAS  
DEL DOCUMENTO DEL FP RIBERA

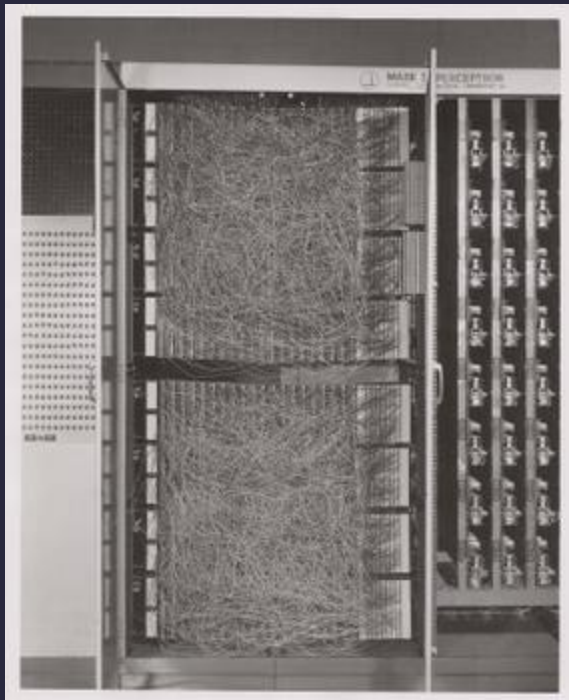
### práctica 4.2 - Desarrollo de un perceptrón



 practica4.2\_perceptron.ipynb



preguntas.docx

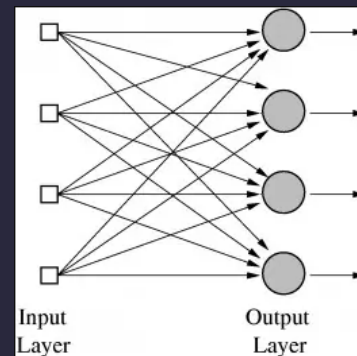


Mark I perceptron (Cornell aeronautical lab)

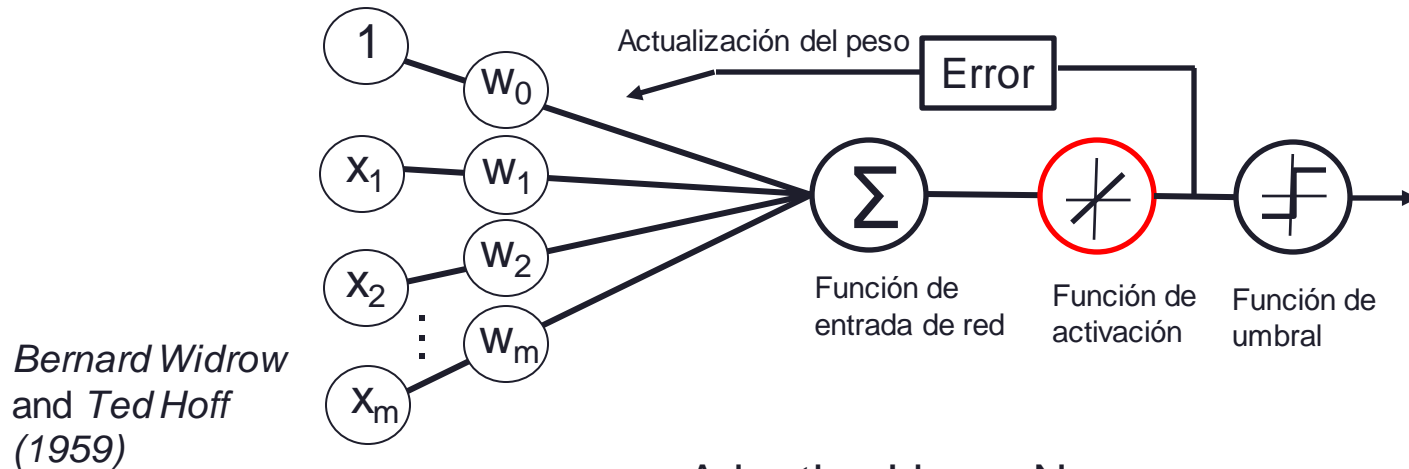
## RESUMEN

La máquina fue usada para clasificar formas de entradas de 20x20 pixels

Las RNAs o redes de neuronas artificiales son capas de perceptrones:



## /05 /LA EVOLUCIÓN: ADaptative LInear NEuron

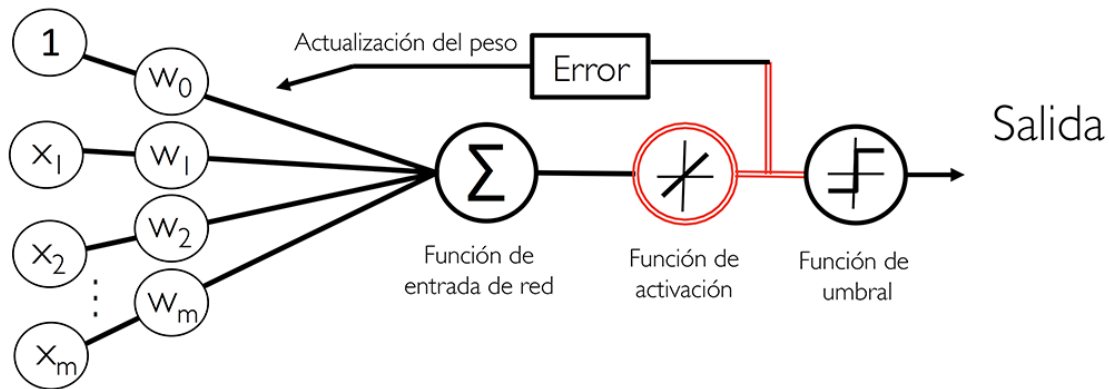
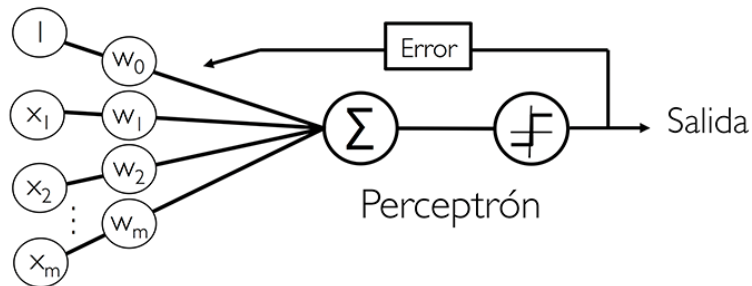


*Bernard Widrow  
and Ted Hoff  
(1959)*

Adaptive Linear Neuron  
(Adaline)



# DIFERENCIA ENTRE LAS NEURONAS PERCEPTRÓN Y ADALINE



Adaptive Linear Neuron (Adaline)

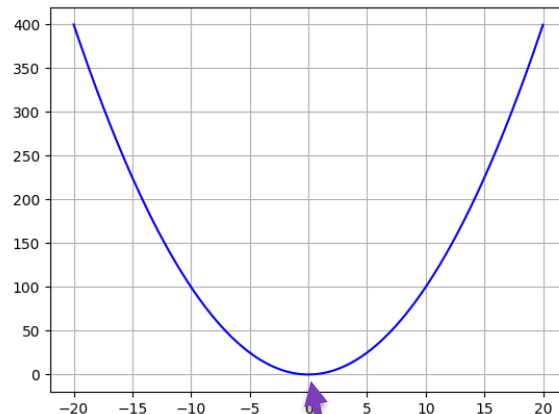
## Motivación:

- ❑ ADaLINE utiliza el método de ajuste por mínimos cuadrados de las predicciones de **una función de activación lineal**.  
(coste: media de los errores al cuadrado)
- ❑ El proceso de aprendizaje se realiza basado **en la salida de una función de activación lineal  $\phi(z)$**  en lugar de la de escalón unitario.
- ❑ El perceptrón actualiza los pesos calculando la diferencia entre la clase verdadera  $y$  y la predicción  $\hat{y}$  : ***sólo aprende cuando falla en una predicción***.
- ❑ ADaLINE actualiza los pesos calculando la diferencia entre la clase esperada  $y$  y la salida de la función de activación  $\hat{y}$  , que es un valor real: **Puede aprender incluso cuando no ha fallado la predicción**.
- ❑ Además, matemáticamente, esto permite minimizar una **función continua de coste** asociado al aprendizaje.

## ¿Función continua de coste?

- Una función de coste es una medida de **cómo de bien o de mal** se ha hecho la predicción.
- Las funciones continuas son fácilmente **derivables** facilitando el entrenamiento de las neuronas y abriendo la puerta a algoritmos no lineales como los perceptrones multicapa, vectores de soporte o regresión logística.
- Adaline utiliza como función de coste  $J(w)$  la **media de los errores al cuadrado** y para su optimización el **descenso del gradiente**  $\nabla J(w)$ .

Función  $J(w)$  “media de los errores al cuadrado”



Objetivo: encontrar el coste mínimo  
“global mínima”

$$J(w) = \frac{1}{2} \sum_i \left( (y^{(i)} - \phi(z^{(i)}))^2 \right)$$

¿Cuál es la función de activación  $\phi(z)$  de Adaline?

En Adaline  $\phi(z) = z$   $\left( \sum \right)$  entrada =  $w_0x_0 + w_1x_1 + \dots + w_nx_n = w^t x$   
 $\phi(w^t x) = w^t x$

Por tanto, Adaline es tan sencillo que **la entrada a la función de activación coincide con la salida**, pero es una función continua, por tanto los pesos varían aunque se haya fallado en la predicción

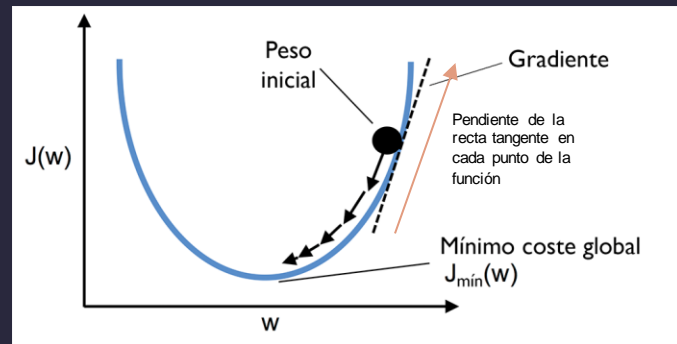
**La magia de las matemáticas....**

Para calcular los pesos usábamos:  $w = w + \Delta w$

Podemos redefinir  $\Delta w$  como el gradiente negativo multiplicado por el rango de aprendizaje (Optimizar la función consiste en dar un paso en la dirección opuesta del gradiente):

$$\Delta w = -n \nabla J(w)$$

El gradiente es una palabra alternativa a “derivada” para funciones con múltiples entradas y una sola salida. Se refiere al ratio de cambio en una función al variar una entrada.



Derivando la función de coste resulta que:  $J(\mathbf{w}) = \frac{1}{2} \sum_i \left( (y^{(i)} - \phi(z^{(i)}))^2 \right)$

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2 =$$

$$= \frac{1}{2} \sum_i 2 (y^{(i)} - \phi(z^{(i)})) \frac{\partial J}{\partial w_j} (y^{(i)} - \phi(z^{(i)})) =$$

$$= \sum_i (y^{(i)} - \phi(z^{(i)})) \frac{\partial J}{\partial w_j} (y^{(i)} - (\sum_j w_j^{(i)} x_j^{(i)})) =$$

$$= \sum_i (y^{(i)} - \phi(z^{(i)})) (-x_j^{(i)})$$

$$= - \sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

$$\Delta w = -n \nabla J(\mathbf{w}) = n \sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Idéntica a la regla de aprendizaje del perceptrón solo que la actualización del peso se calcula en base a todas las muestras del conjunto de entrenamiento.

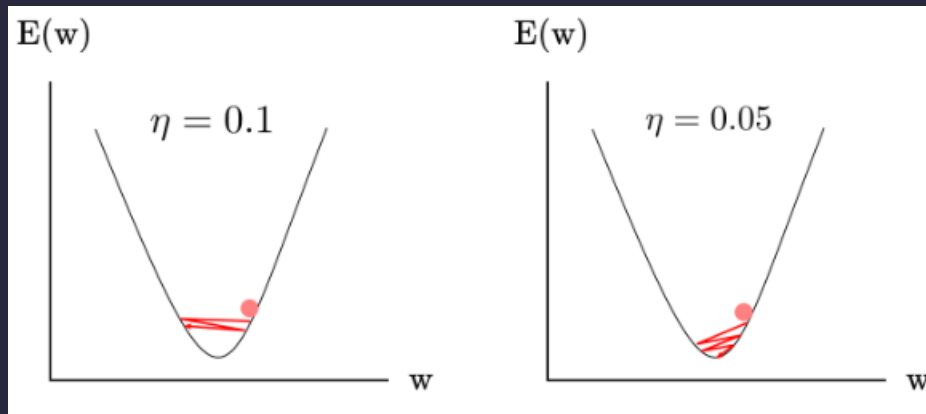
Permite el entrenamiento parcial “Parcial Fit” con diversas muestras de entrenamiento “Descenso del gradiente estocástico”

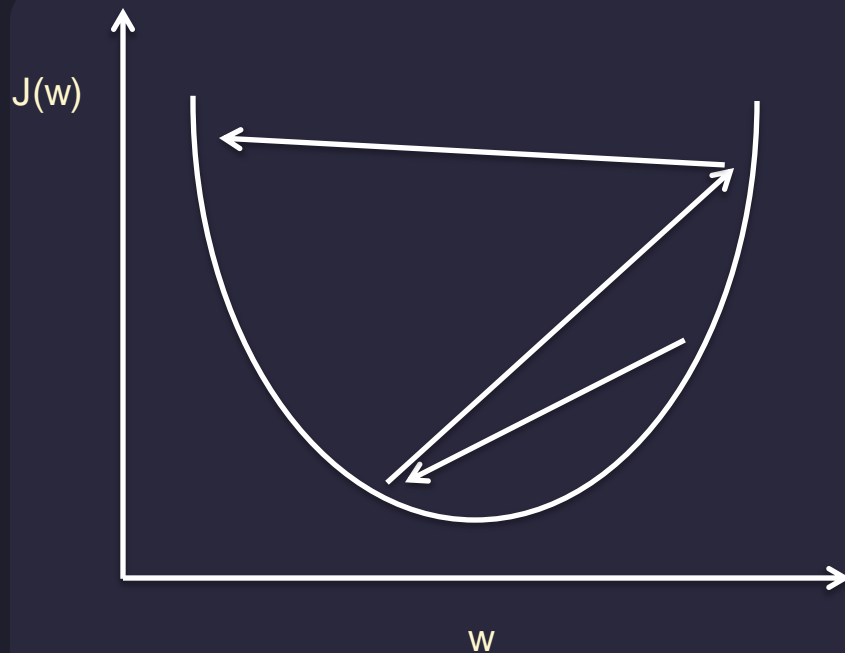
## El ratio de aprendizaje $\eta$ (hiperparámetro):

Si seleccionamos un ratio de aprendizaje muy rápido (0.1) el método, no converge al mínimo global, por tanto ADALINE desaprende lo que ha aprendido según pasan las épocas

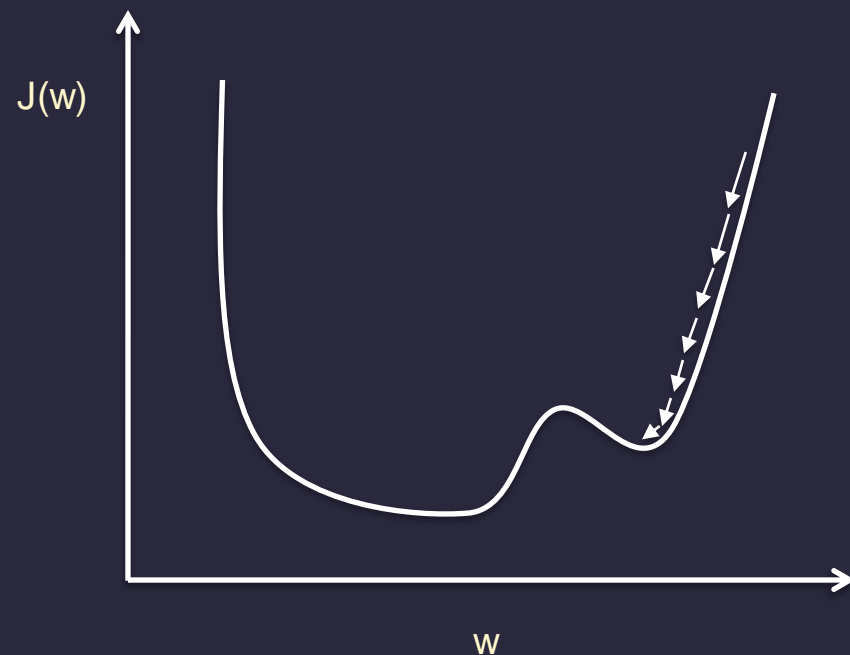
Si seleccionamos un ratio de aprendizaje muy lento, puede ser que el método tarde mucho en encontrar el mínimo global (muchas épocas).

Hay que seleccionar un ratio de aprendizaje apropiado en cada caso






Rango de aprendizaje amplio: *Overshooting*.



Rango de aprendizaje pequeño: Muchas iteraciones hasta la convergencia y capturas en mínimo local.

## PRÁCTICA 4.2:

 adaline.py

```
def fit(self, X, y):  
    rgen = np.random.RandomState(self.random_state)  
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])  
    self.cost_ = []  
  
    for i in range(self.n_iter):  
        net_input = self.net_input(X)  
        output = self.activation(net_input)  
        errors = (y - output)  
        self.w_[1:] += self.eta * X.T.dot(errors) →  
        self.w_[0] += self.eta * errors.sum()  
        cost = (errors ** 2).sum() / 2.0  
        self.cost_.append(cost)  
    return self
```

### Multiplicación matriz por vector

Ejemplo:

```
matriz=np.array([[1,2],[3,4],[5,6]])
```

```
vector=np.array([7,8,9])
```

```
result=matriz.T.dot(vector)
```

```
print(result) → [76 100]
```

1×7+3×8+5×9

2×7+4×8+6×9



## Muestras (pétalo,sépalo)

t

	0	1
0	5.1	1.4
1	4.9	1.4
2	4.7	1.3
3	4.6	1.5
4	5	1.4
5	5.4	1.7
6	4.6	1.4
7	5	1.5
8	4.4	1.4
9	4.9	1.5

100 elementos

Errores (y - output)

dot

`[-0.97764947 -0.97887298 -0.98062467 ... 1.04439683 1.03080128`

100 elementos

## numpy.dot

`numpy.dot(a, b, out=None)`

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using `matmul` or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to `multiply` and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where *M* ≥ 2), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

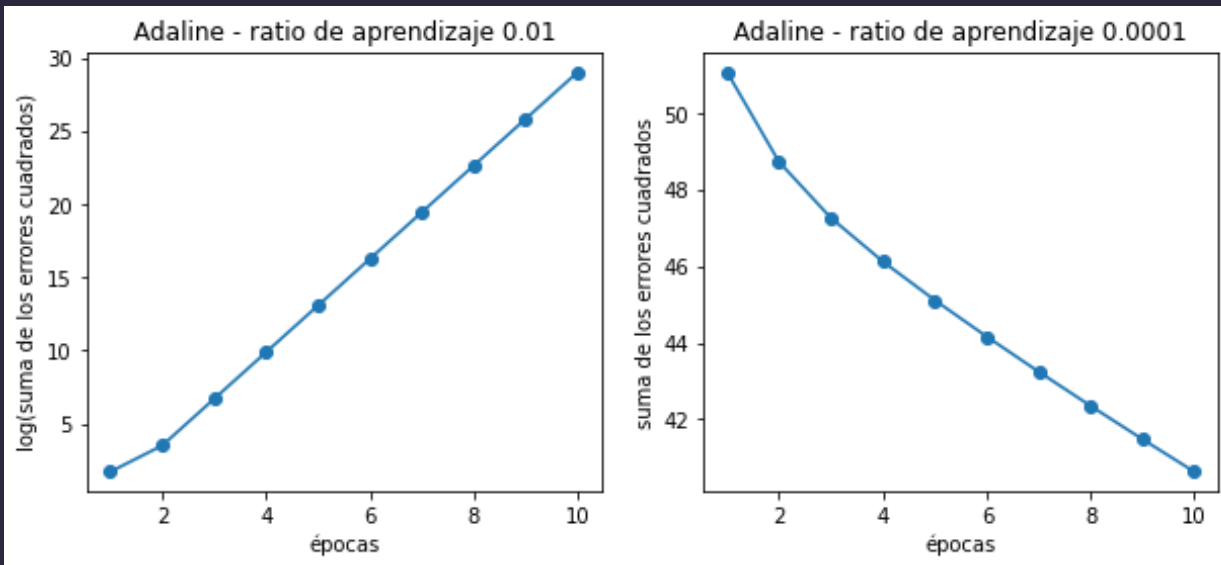
## El ratio de aprendizaje $\eta$ (hiperparámetro):

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
```

```
ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)  
ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
```

```
ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)  
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
```

```
plt.show()
```



## Escalado de características:

Muchos algoritmos requieren un escalado de características para su buen funcionamiento. Por ejemplo, para  $n=0.01$ , hemos visto que el algoritmo no converge, pero si los **normalizamos**, obtenemos **mejores resultados**

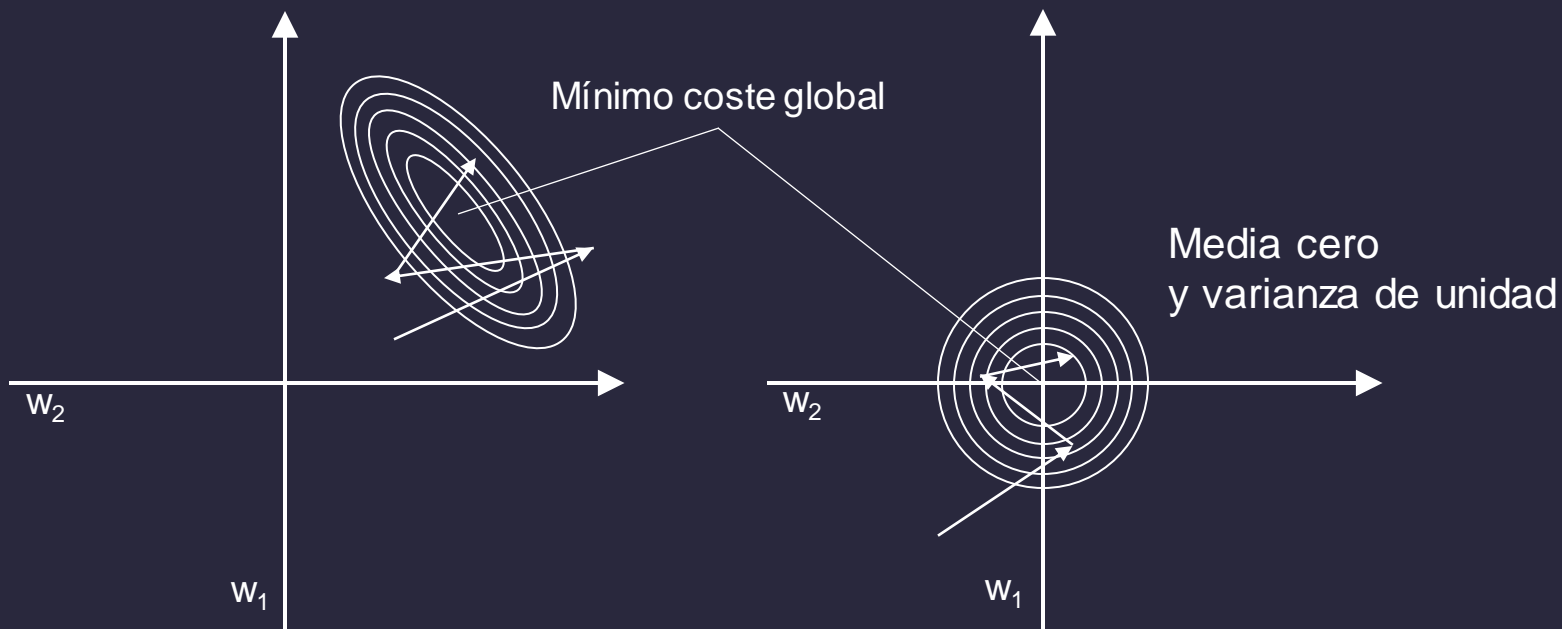
$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Media

Desviación típica

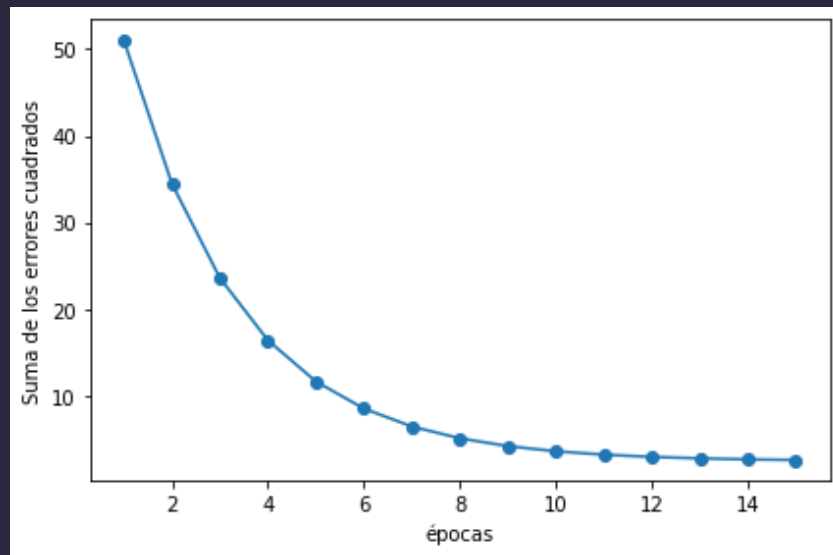
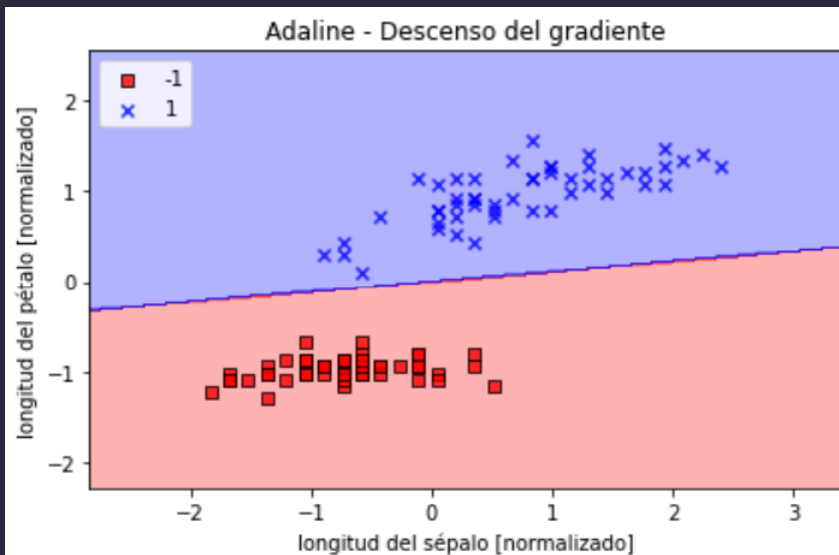
```
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
```

## Reescalando el algoritmo encuentra la solución más fácil









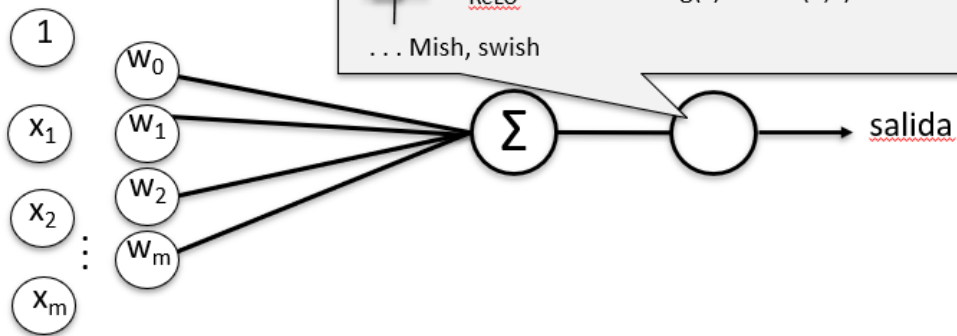
## Reescalando el algoritmo encuentra la solución más fácil: ratio 0.01

```
ada = AdalineGD(n_iter=15, eta=0.01)  
ada.fit(X_std, y)
```



**Selección de funciones de activación muy utilizadas para neuronas artificiales.**

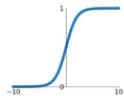
	Escalón unitario	$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{si no.} \end{cases}$
	Escalón unitario (base 0)	$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{si no.} \end{cases}$
	Lineal	$g(z) = z$
	Logística (sigmoide)	$g(z) = 1 / (1 + \exp(-z))$
	Tangente hiperbólica (sigmoide)	$g(z) = \frac{\exp(2z) - 1}{\exp(2z) + 1}$
	ReLU	$g(z) = \max(0, x)$
... Mish, swish		



## Activation Functions

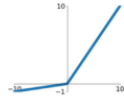
### Sigmoid

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



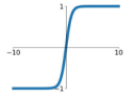
### Leaky ReLU

$$\max(0.1x, x)$$



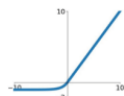
### tanh

$$\tanh(x)$$



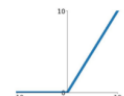
### Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$



### ReLU

$$\max(0, x)$$



### ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

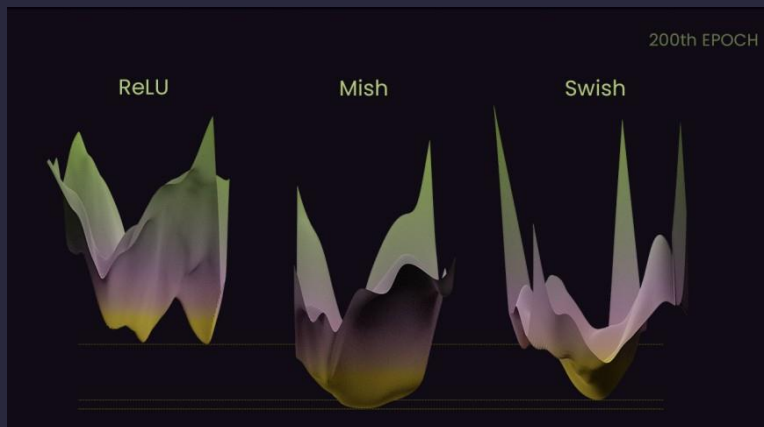
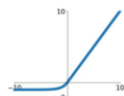
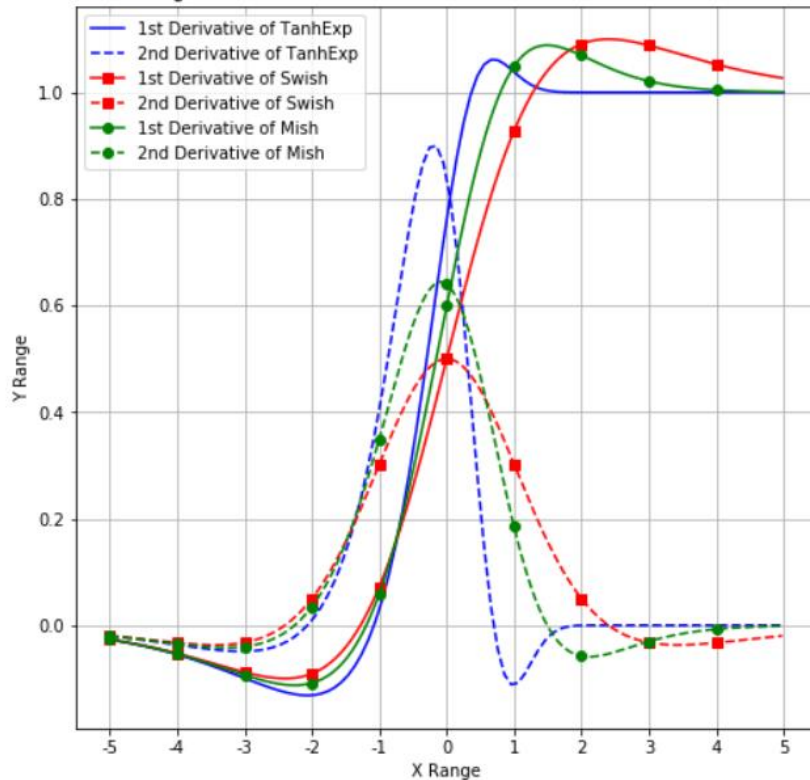
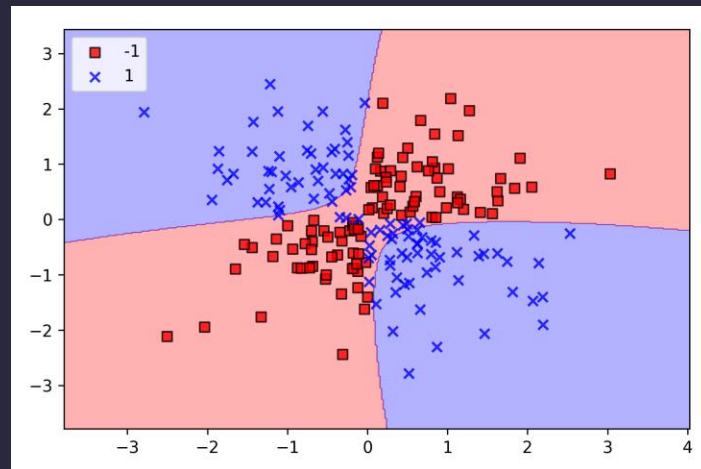
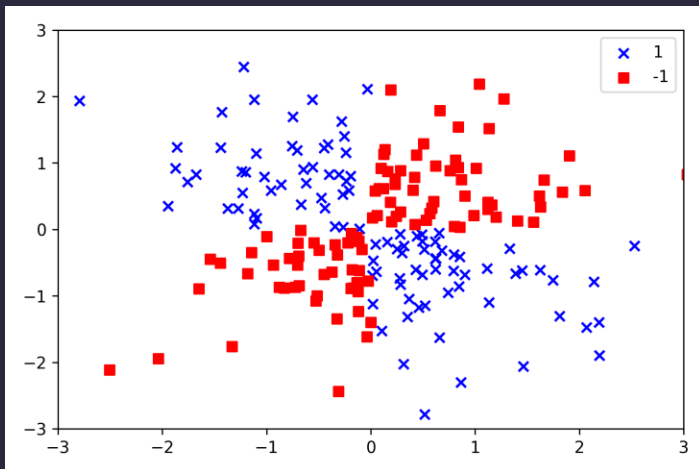


Figure of The Derivatives of Different Activation Functions



Este tipo de neuronas no pueden modelar representaciones no lineales, por ejemplo, una función XOR:

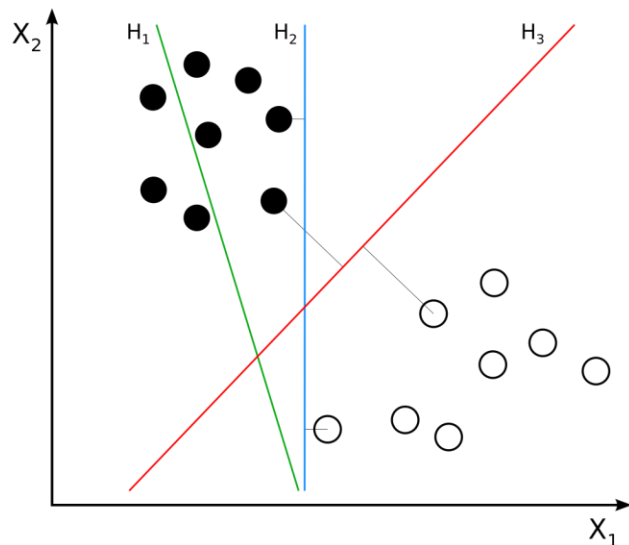


Para resolver este problema se usa una función de coste que maximiza el margen entre las clases: Support Machine Vector (SVM):

Hay que usar una función de mapeo:  $\phi(X_1, X_2) = (Z_1, Z_2, Z_3) = (X_1, X_2, X_1^2 + X_2^2)$



## Maximizar el margen entre las clases: Support Machine Vector (SVM)



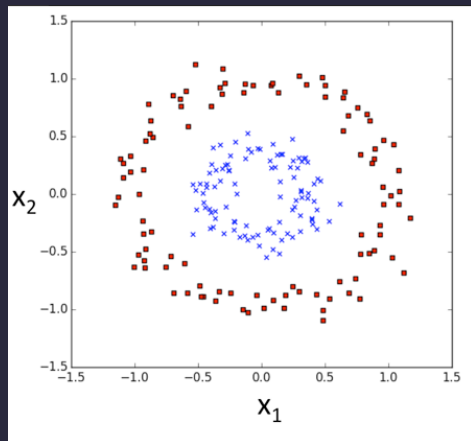
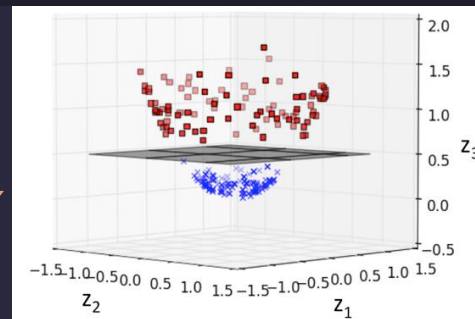
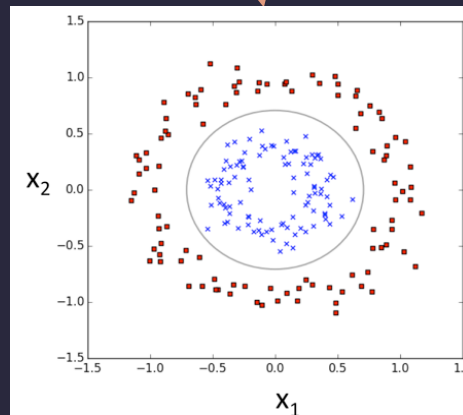
$H_1$  no separa las clases

$H_2$  las separa, pero con muy poco margen

$H_3$  las separa con el margen máximo

Método llamado SVM (support vector machine) kernelizado

$$\phi(X_1, X_2) = (Z_1, Z_2, Z_3) = (X_1, X_2, X_1^2 + X_2^2)$$

 $\phi$  $\phi^{-1}$ 

**Overfitting:** Un modelo funciona bien en el entrenamiento, pero no generaliza bien

**Underfitting:** nuestro modelo no es suficientemente complejo como para capturar bien el patrón en los datos de entrenamiento

