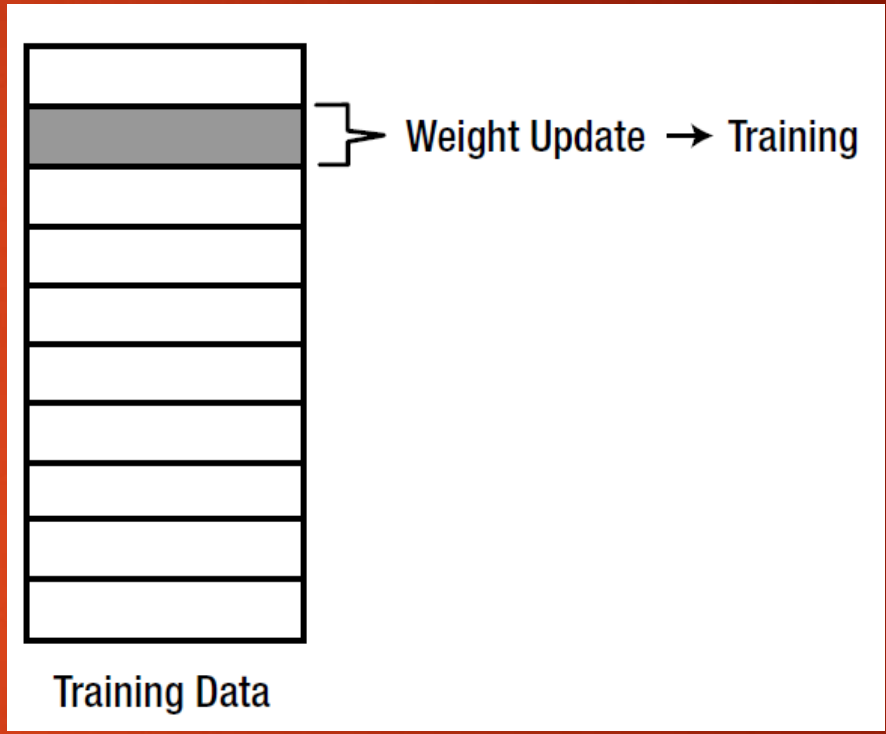
Redes Neuronales y Aprendizaje Profundo

BACKPROPAGATION

Modos de entrenamiento de las Redes Neuronales

Stochastic Gradient Descent (SGD)

- Con este nombre, se le denomina al proceso mediante el cual se calcula el error en datos de entrenamiento y de inmediato se ajustan los pesos de la red neuronal. Si tenemos 100 patrones de entrenamiento, haremos 100 ajustes.



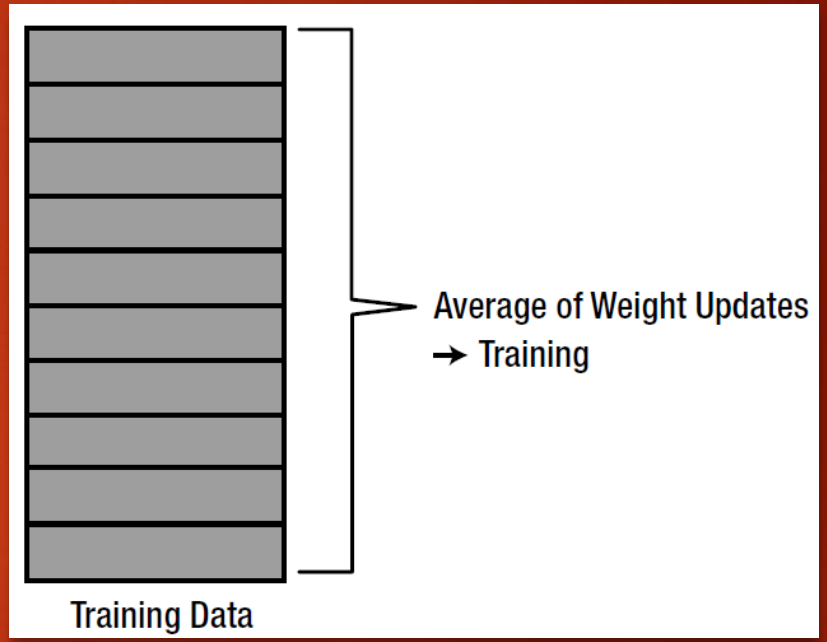
SGD

- Conforme se ajustan los pesos por cada punto de entrenamiento, el desempeño de la red neuronal puede alterarse a lo largo del proceso de entrenamiento.
- El término estocástico implica el comportamiento aleatorio del proceso de entrenamiento. Por cada patrón, los pesos se ajustarían bajo la regla delta generalizada.

$$\Delta w_{ij} = \alpha \delta_i x_j$$

Batch

- En el método batch, la actualización de los pesos se calcula a partir de todos los errores de todos los datos de entrenamiento, el promedio de las actualizaciones de los pesos se utilizará para ajustar los pesos. Este método usará todos los datos de entrenamiento y actualizará los pesos una sola vez.

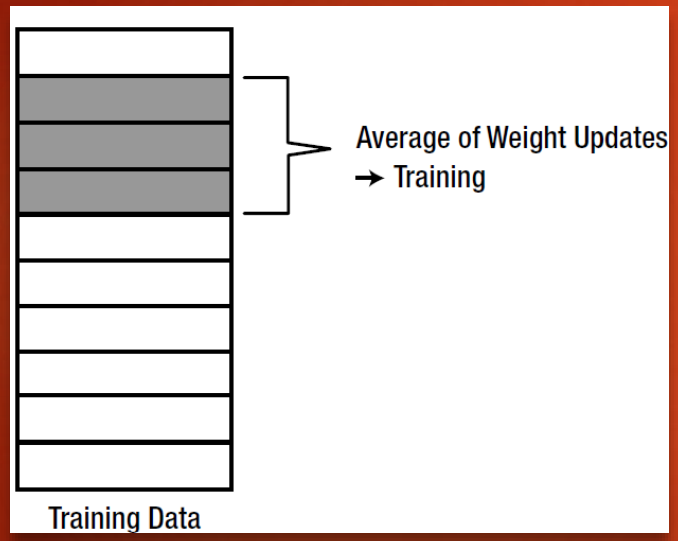


Match

- La actualización de los pesos se calcula como:

$$\Delta w_{ij} = \frac{1}{N} \sum_{k=1}^{N} \Delta w_{ij}(k)$$

Mini Batch



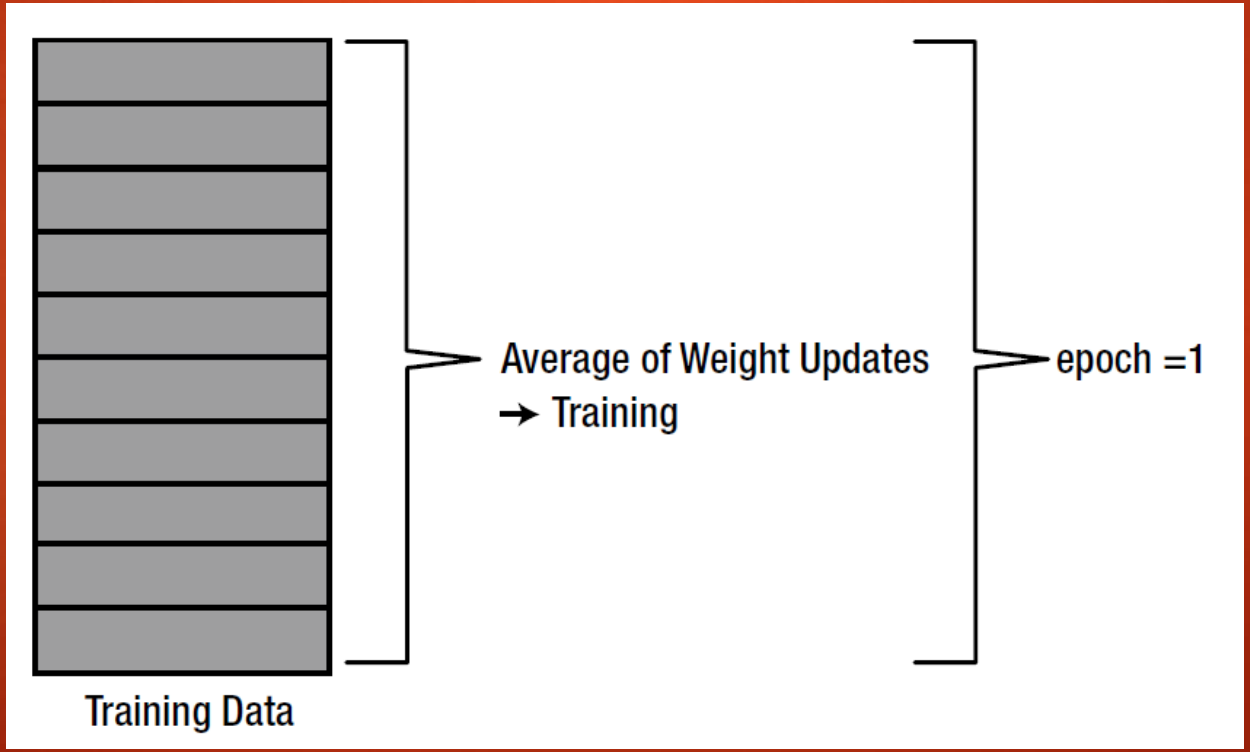
- Este método es una versión intermedia del SGD y el batch.
- Se selecciona una parte del conjunto de entrenamiento y se utiliza con el método batch.
- Calcula las actualizaciones de los pesos de los datos seleccionados y entrena la red neuronal con la actualización de pesos promediada.

Mini Batch

- Si tenemos 100 datos, y aplicamos mini batch de 20; haremos un máximo de 5 ajustes de pesos.
- Este método busca aprovechar la velocidad del SGD y la estabilidad del Batch.
- Por tal razón es empleado en Deep Learning

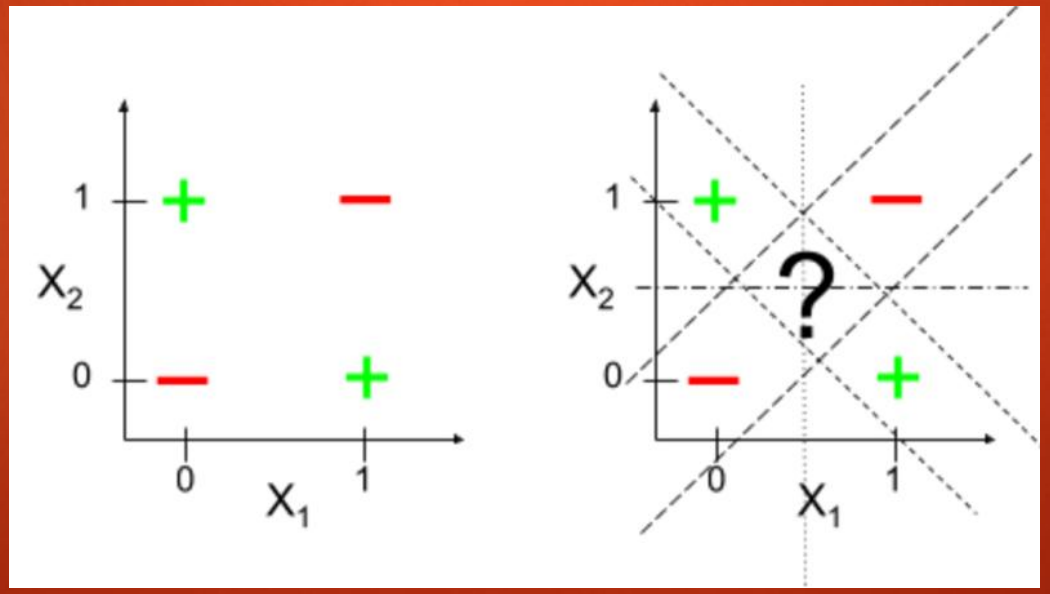
Épocas o *Epochs*

- Una época se define como el número de ciclos de entrenamiento completados para todos los datos de aprendizaje.



Solución a los límites del Perceptrón

- Las limitaciones del Perceptrón que fueron formalizadas en 1969, tardaron 17 años en solucionarse.
- Entre 1984 y 1986 se retoma y fundamente una solución: unir dos o más perceptrones.



Solución a los límites del Perceptrón

- Lo más complicado resultó ser que la regla delta empleada en los perceptrones simples, no funcionaba para dos o más perceptrones encadenados.
- En el perceptrón simple:

$$w'_{ij} = w_{ij} + \Delta w_{ij}$$
$$\Delta w_{ij} = (d_i - y_i)\alpha x_j$$

En otras palabras:

$$w' = w + (error * \alpha * x_j)$$

Regla delta generalizada

- La versión anterior de la regla delta funcionaba adecuadamente para problemas básicos como funciones lógicas.
- Sin embargo, hay una actualización más vigente de esta regla.

$$w_{ij} = w_{ij} + \alpha \delta_i x_j$$

La regla delta generalizada

La nueva variable delta se define de la siguiente forma:

$$\delta_i = f'(v_i)e_i$$

- En donde:
- e_i el error del nodo de salida i.
- v_i la suma ponderada del nodo de salida i.
- f' la derivada de la función de activación f del nodo de salida i.

La regla delta generalizada

- Recordemos que en el caso de emplear una función de activación lineal $f(x) = x$ podemos hacer que $f'(x) = 1$, sustituimos en la expresión anterior.
- $\delta_i = e_i$
- $w_{ij} = w_{ij} + \alpha e_i x_j$

La regla delta generalizada

- Para el caso de la función de activación sigmoide tenemos que:
- $f(x) = \frac{1}{1+e^{-x}}$
- $f'(x) = f(x)\big(1 - f(x)\big)$
- Sustituimos en $w_{ij} = w_{ij} + \alpha\delta_i x_j$
- $\delta_i = f'(v_i)e_i = f(v_i)\big(1 - f(v_i)\big)e_i$
- Para la función sigmoide la regla delta es:
- $w_{ij} = w_{ij} + \alpha f(v_i)\big(1 - f(v_i)\big)e_i x_j$

Funciones de costo

- Muchas ocasiones además del accuracy o de otras medidas de desempeño, es importante conocer el impacto de los errores dentro del proceso de entrenamiento o prueba.
- Para eso se han definido algunas funciones de costo como las siguientes.

Funciones de costo

Esta primera se conoce como suma del error cuadrático.

$$J = \sum_{i=1}^{M} (d_i - y_i)^2$$

También tenemos, algunas variantes:

$$J = \frac{1}{2} \sum_{i=1}^{M} (d_i - y_i)^2$$

$$J = \frac{1}{M} \sum_{i=1}^{M} (d_i - y_i)^2$$

Funciones de costo

- Y tenemos la función de costo cross entropy

$$E = \sum_{i=1}^{M} \{-d_i \ln(y_i) - (1 - d_i) \ln(1 - y_i)\}$$

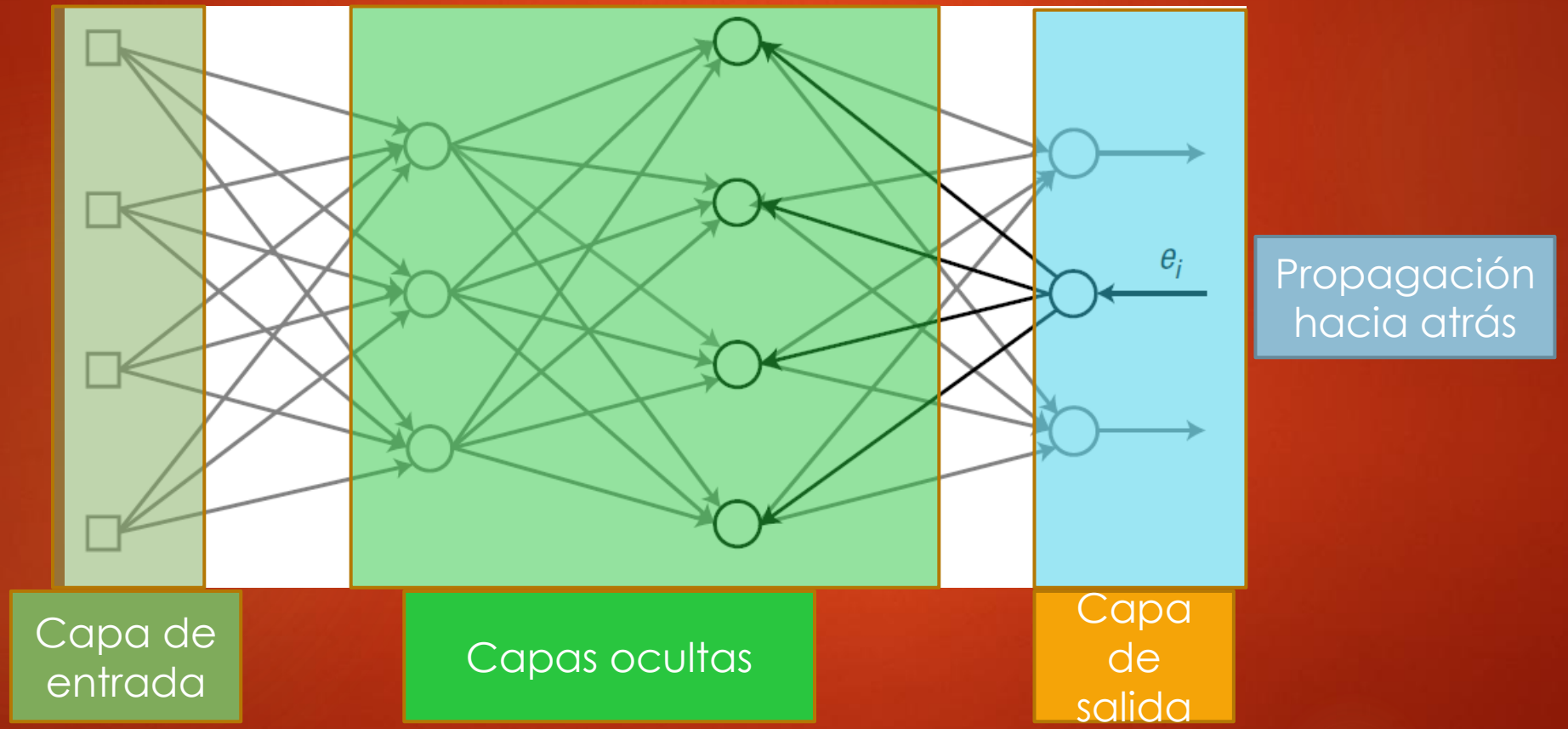
$$E = -d \ln(y) - (1 - d) \ln(1 - y)$$

El algoritmo Backpropagation

Perceptrones Multicapa (MLP)

- A partir de la propuesta inicial de Wervos y después de Rumelhart, Hinton y Williams, surgió un nuevo modelo de Red Neuronal, el cual se ha denominado como Perceptrón Multicapa o MLP.
- De acuerdo con sus investigaciones, una red neuronal puede conformarse por una capa de entrada, una o más capas ocultas de procesamiento y una capa de salida.
- Para que este modelo de red funcione, es importante que los errores de la capa de salida se propaguen hacia atrás, es decir a las capas anteriores.

Arquitectura del MLP



Algoritmo del Backpropagation

1. Inicializar los pesos con los valores adecuados.
2. Por cada patrón del conjunto de entrenamiento E.

a. Calcular el error de salida y el valor delta de los nodos de salida

$$e = d - y$$
$$\delta = f'(v)e$$

b. Propagar hacia atrás el valor δ hacia atrás, y calcular las deltas inmediatas de los nodos anteriores.

$$e^k = W^T \delta$$
$$\delta^k = f'(v^k)e^k$$

c. Repetir el paso b, hasta que se llegue a la primera capa oculta.

Algoritmo del Backpropagation

d. Ajustar los pesos de acuerdo con la siguiente regla de aprendizaje.

$$\Delta w_{ij} = \alpha \delta_i x_j$$

$$w_{ij} = w_{ij} + \Delta wij$$

e. Repetir los pasos a-d para cada patrón de entrenamiento.

3. Repetir el paso 2 hasta que la red neuronal está propiamente entrenada (cuando alcancemos un valor mínimo de error o se alcance un número máximo de épocas o repeticiones establecidas).

Algoritmo Backpropagation (Ejemplo)

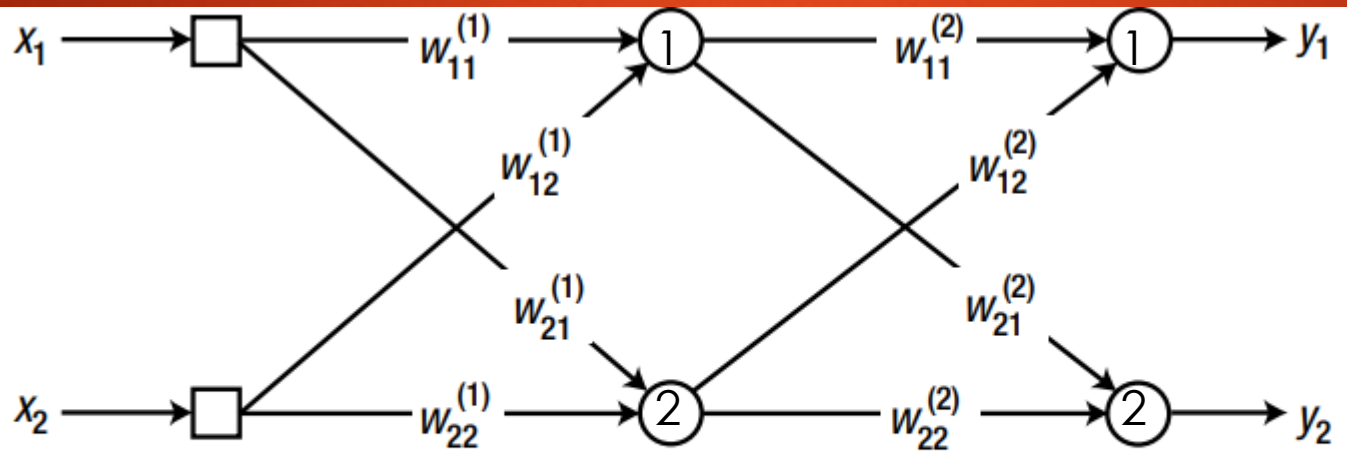
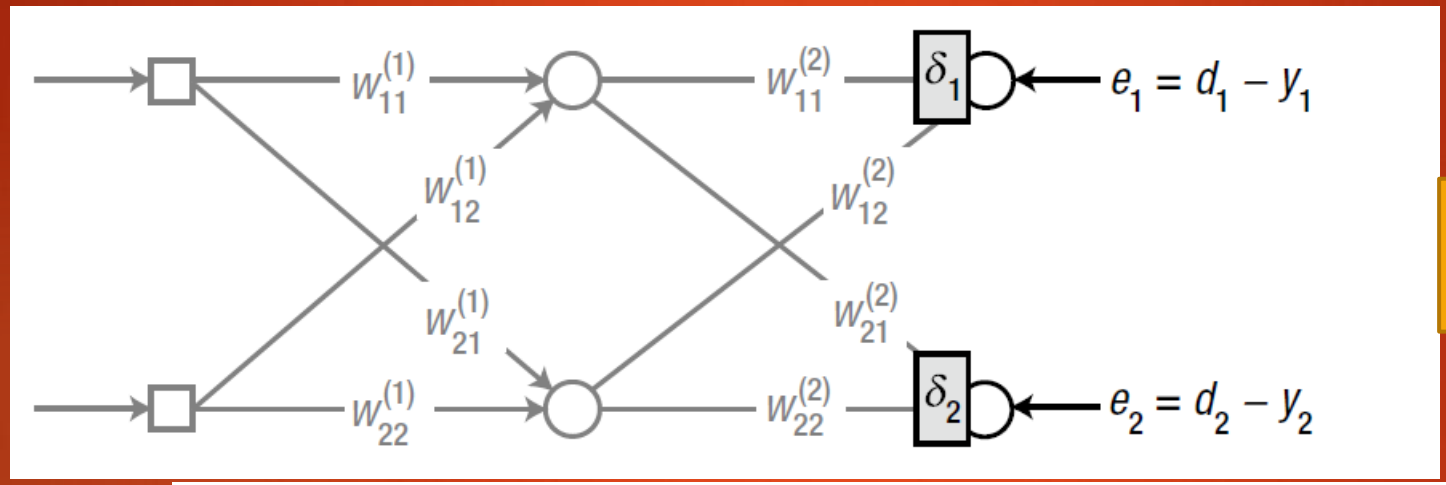


Figure 3-2. *Neural network that consists of two nodes for the input and output and a hidden layer, which has two nodes*

Algoritmo Backpropagation (Ejemplo)



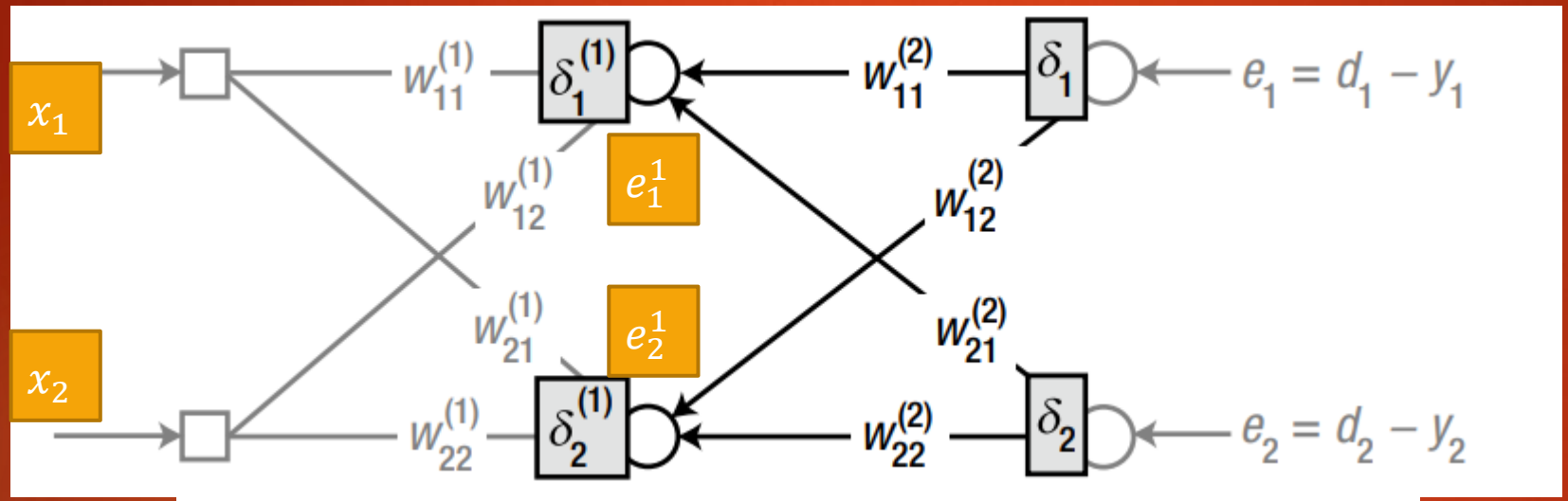
$$e^k = W^T \delta$$
$$\delta^k = f'(v^k) e^k$$

$$e_1 = d_1 - y_1$$
$$\delta_1 = \varphi'(v_1) e_1$$

$$e_2 = d_2 - y_2$$
$$\delta_2 = \varphi'(v_2) e_2$$

(Equation 3.3)

Algoritmo Backpropagation (Ejemplo)



$$e_1^{(1)} = w_{11}^{(2)}\delta_1 + w_{21}^{(2)}\delta_2$$

$$\delta_1^{(1)} = \varphi'\left(v_1^{(1)}\right)e_1^{(1)}$$

$$e_2^{(1)} = w_{12}^{(2)}\delta_1 + w_{22}^{(2)}\delta_2$$

$$\delta_2^{(1)} = \varphi'\left(v_2^{(1)}\right)e_2^{(1)}$$

(Equation 3.4)

Algoritmo Backpropagation (Ejemplo)

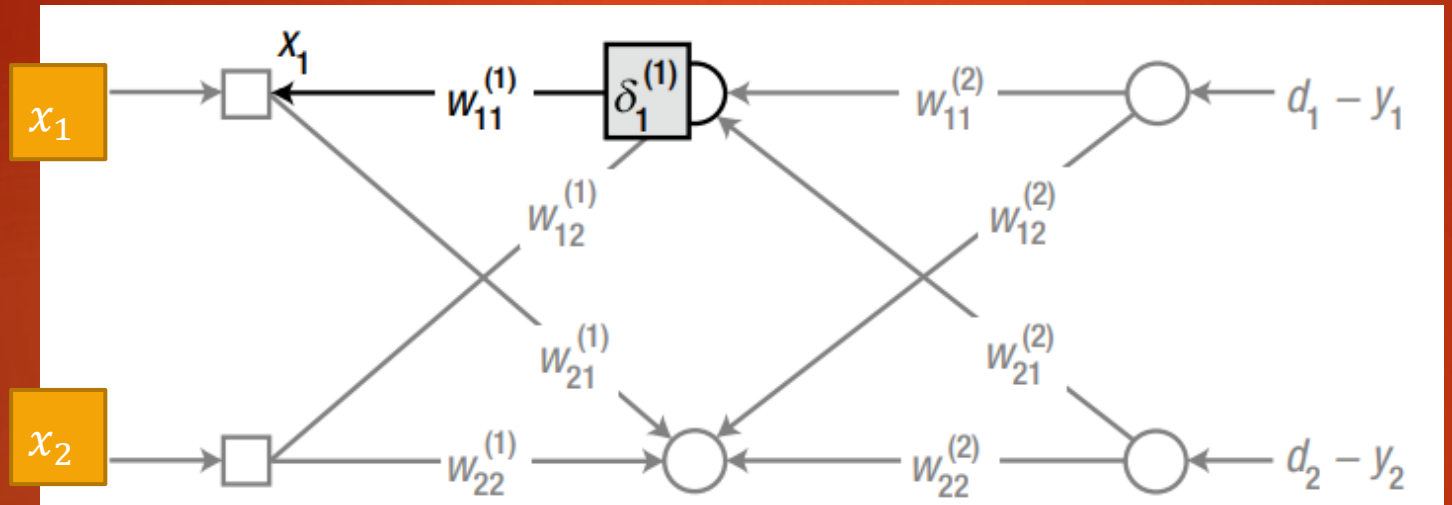


Figure 3-7. *Derive the equation to adjust the weight, again*

$$w_{11}^{(1)} \leftarrow w_{11}^{(1)} + \alpha \delta_1^{(1)} x_1$$

Algoritmo Backpropagation (Ejemplo)

$$\varphi(\upsilon) = y$$

$$\delta = \varphi'(\upsilon) \times e$$

Figure 3-5. *The error calculation is the only difference*

Teorema de la aproximación universal

- Una Red Neuronal con una sola capa oculta es suficiente para representar cualquier función, pero dicha capa puede ser muy grande y eventualmente podría fallar en aprender y generalizar correctamente.

Demo

- https://playground.tensorflow.org/

¿Preguntas?

Implementación

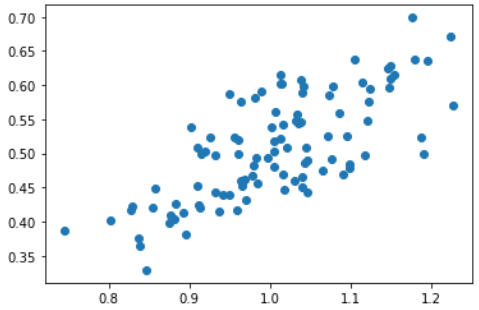
- https://github.com/andreiosgf/neural_nets

Implementación

Full example available here

•[1]:
```
import numpy as np
import tensorflow as tf #Work wit Tensor Flow 2
import matplotlib.pyplot as plt
```

•[2]:
```
np.random.seed(0)#Set random seeds
x_vals = np.random.normal(1, 0.1, 100).astype(np.float32) #Create syntetic data
y_vals = (x_vals * (np.random.normal(1, 0.05, 100) - 0.5)).astype(np.float32)
plt.scatter(x_vals, y_vals)#Plot the data
plt.show()
```



•[3]:
```
#Create a function to perform the weighted sum of values and biases.
def my_output(X, weights, biases):
    return tf.add(tf.multiply(X, weights), biases)
```

•[4]:
```
#Calculate loss
def loss_func(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_pred - y_true))
```

•[5]:
```
#Use a learning algorithm Stochastic Gradient Descent
my_opt = tf.optimizers.legacy.SGD(learning_rate=0.02)
```

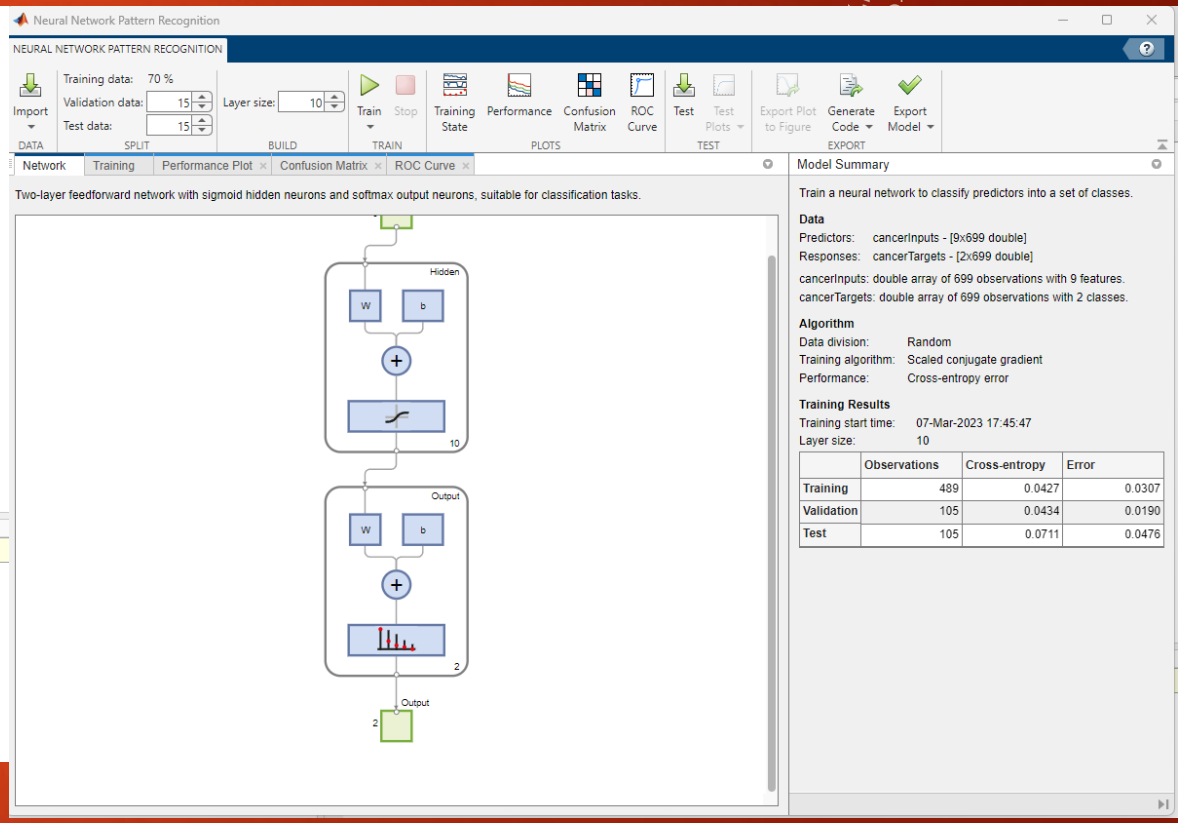
•[6]:
```
#Initialize weights and variables with random values
tf.random.set_seed(1)
np.random.seed(0)
weights = tf.Variable(tf.random.normal(shape=[1]))
biases = tf.Variable(tf.random.normal(shape=[1]))
history = list()
```

```
clear all
X = [ 0 0 1;
 0 1 1;
 1 0 1;
 1 1 1;
 ];
D = [ 0
 1
 1
 0
 ];
W1 = 2*rand(4, 3) - 1;
W2 = 2*rand(1, 4) - 1;
for epoch = 1:10000 % train
 [W1 W2] = BackpropXOR(W1, W2, X, D);
end
N = 4; % inference
for k = 1:N
 x = X(k, :)';
 disp(x)
 v1 = W1*x;
 y1 = Sigmoid(v1);
 v = W2*y1;
 y = Sigmoid(v)
end
```

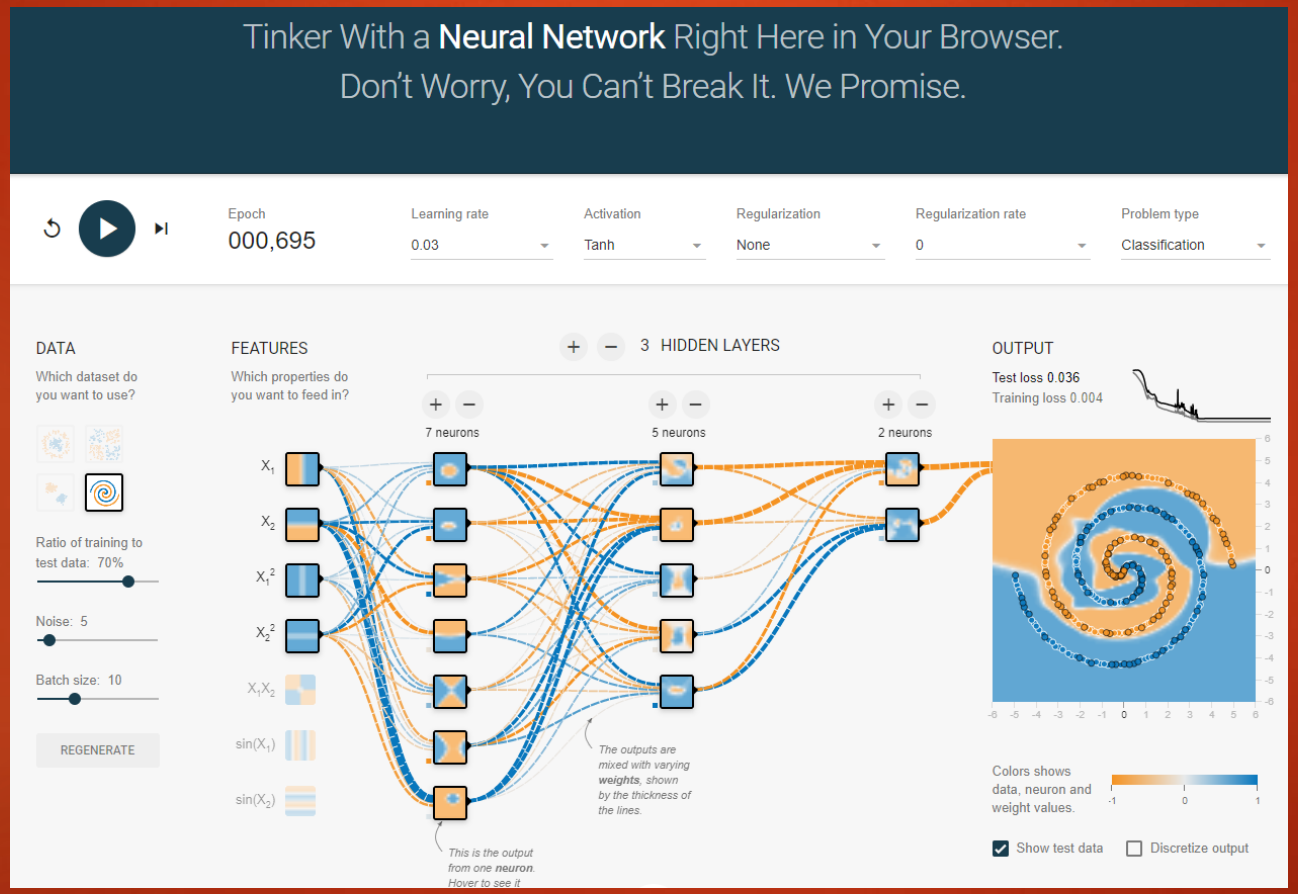
Command Window

New to MATLAB? See resources for Getting Started.

```
    0.9882

     1
     1
     1


y =

    0.0154
```



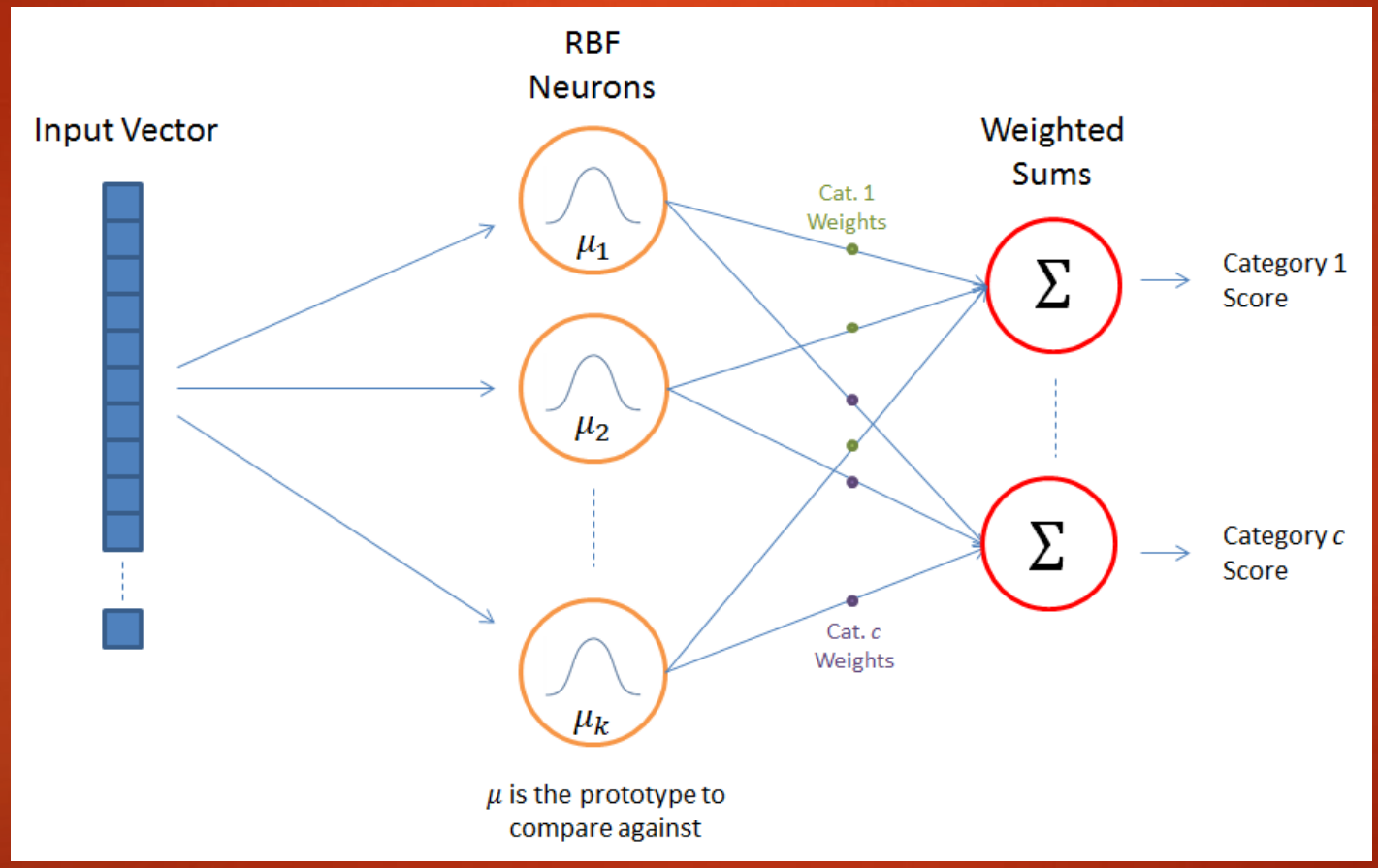
Demo


Tinker With a Neural Network Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.
Epoch
000,695
Learning rate
0.03
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 70%
Noise: 5
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
3 HIDDEN LAYERS
7 neurons
5 neurons
2 neurons
The outputs are mixed with varying weights, shown by the thickness of the lines.
This is the output from one neuron. Hover to see it
OUTPUT
Test loss 0.036
Training loss 0.004
Colors shows data, neuron and weight values.
Show test data
Discretize output

Otros modelos de Redes Neuronales (RBF Network: Redes de Base Radial)

- A diferencia de las MLP estas redes solo tienen una capa de entrada, una capa oculta (formada por neuronas con funciones de base radial) y una de salida.
- En la capa oculta el entrenamiento consiste en:
  - Calcular un centroide que representa a una parte del espacio en el que están los datos (clase).
  - Calcular un parámetro de escala σ^2 a partir de las distancias de los patrones a los centroides.
  - $\sigma_j^2 = \frac{1}{N_j}\sum_{x\in j}^{p}\|x - c_j\|^2$

Otros modelos de Redes Neuronales (RBF Network: Redes de Base Radial)



Otros modelos de Redes Neuronales (RBF Network: Redes de Base Radial)

- En la capa de salida el entrenamiento consiste en:
  - Ajustar los pesos de conexión de las neuronas de la siguiente forma:
  - $w_{ij} = w_{ij} + \frac{\alpha}{2}(d_i - z_i)\phi(r_j)$
  - La neurona de salida tiene como valor

$$z_i = \sum_j w_{ij}\,\phi(r_j) + \theta_i$$

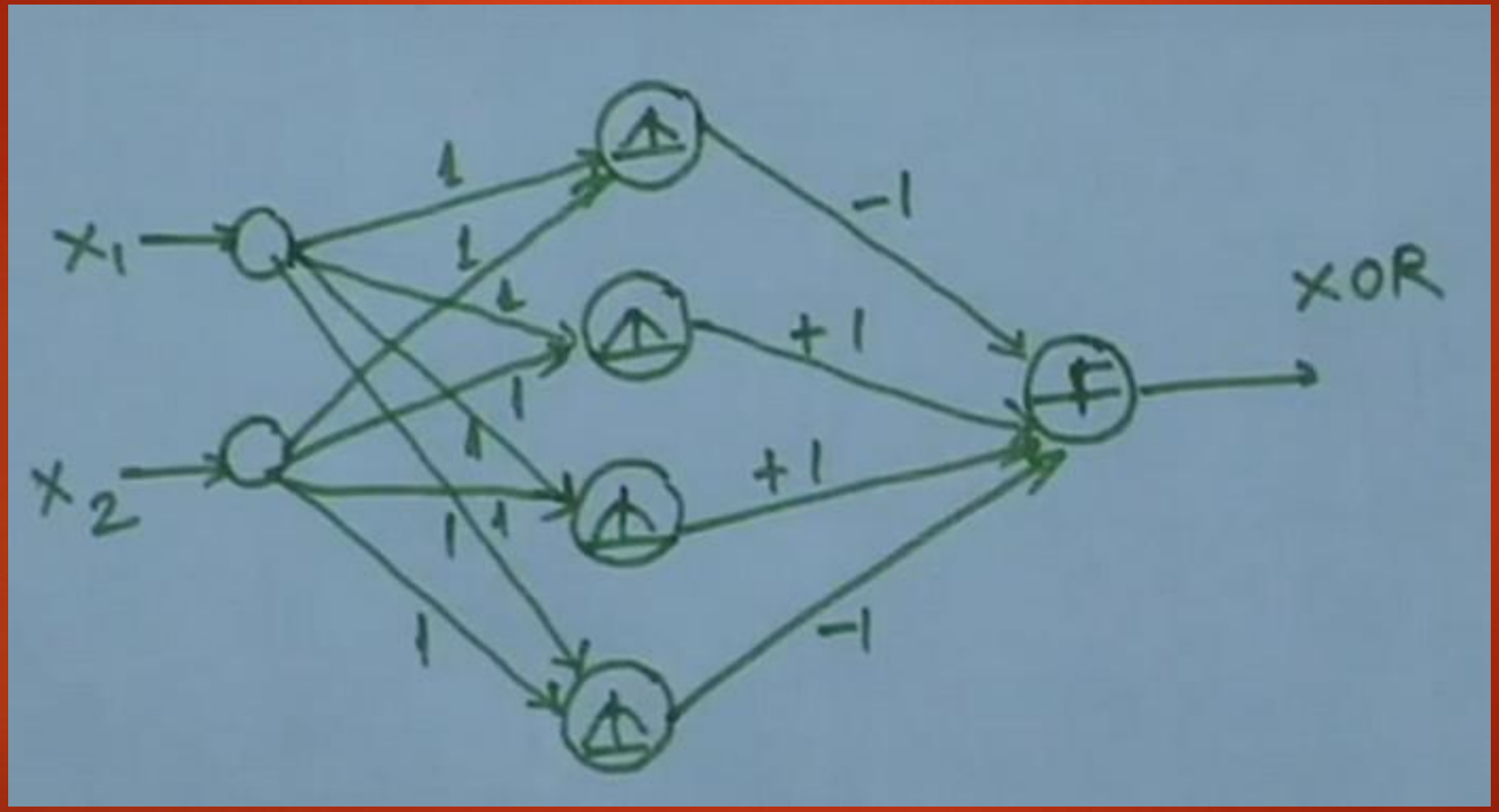
RBF Network: Redes de Base Radial Procesamiento de patrones

- Por cada patrón en $x \in E$
  - Calcular la distancia euclidiana entre el patrón x y el centroide almacenado en la neurona j
  - $r_j^2 = \|x - c_j\|^2$
  - La salida de la neurona se calcula a través de una función de base radial
  - $y = \phi(r) = e^{-\frac{r^2}{2\sigma^2}}$ o $y = \phi(r) = r^2 \ln(r)$
  - La respuesta de la capa de salida es la siguiente
  - $z_i = \sum_j w_{ij}\, y_j + \theta_i$

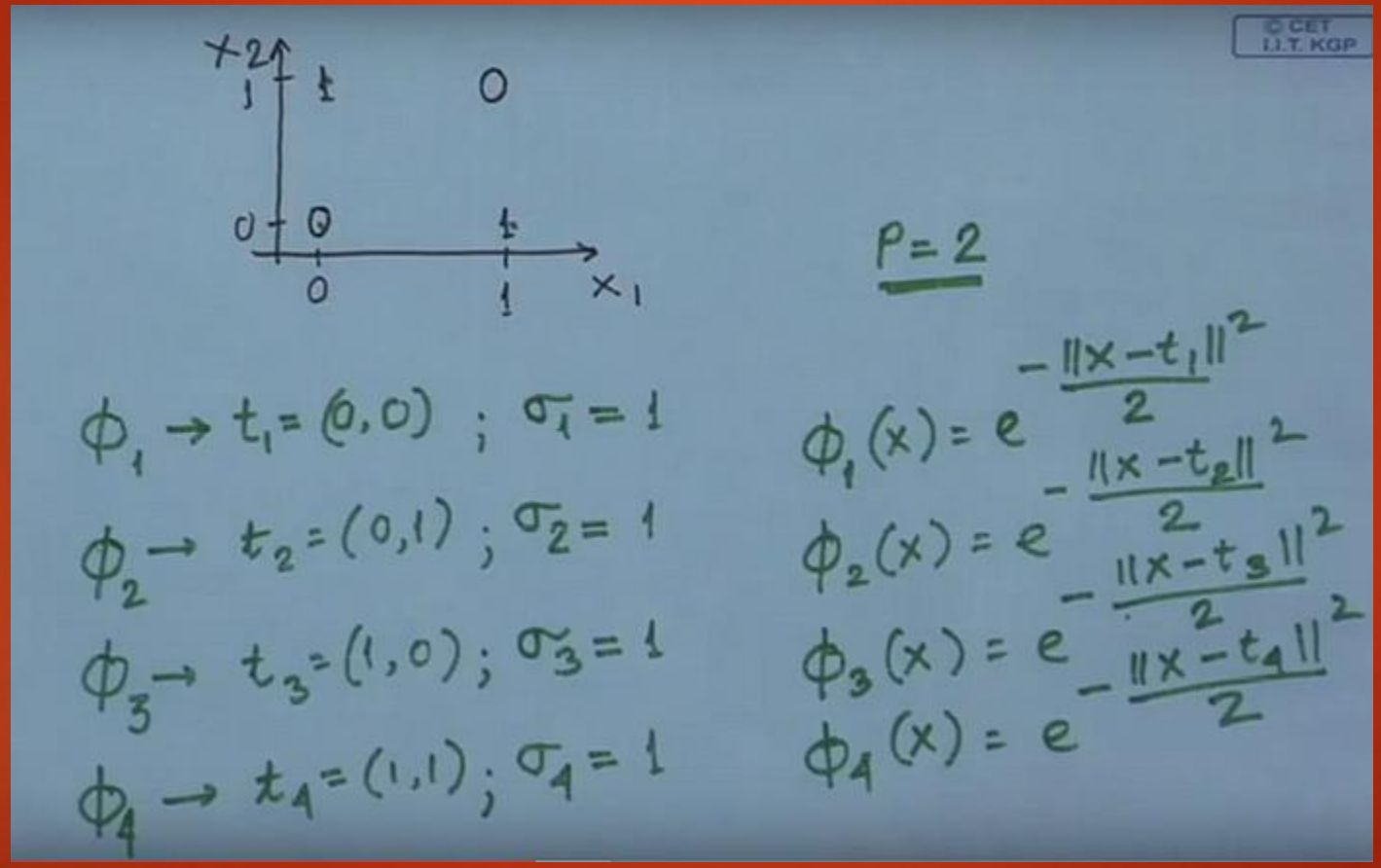
Ejemplo e Información adicional

- https://towardsdatascience.com/radial-basis-functions-neural-networks-all-we-need-to-know-9a88cc053448

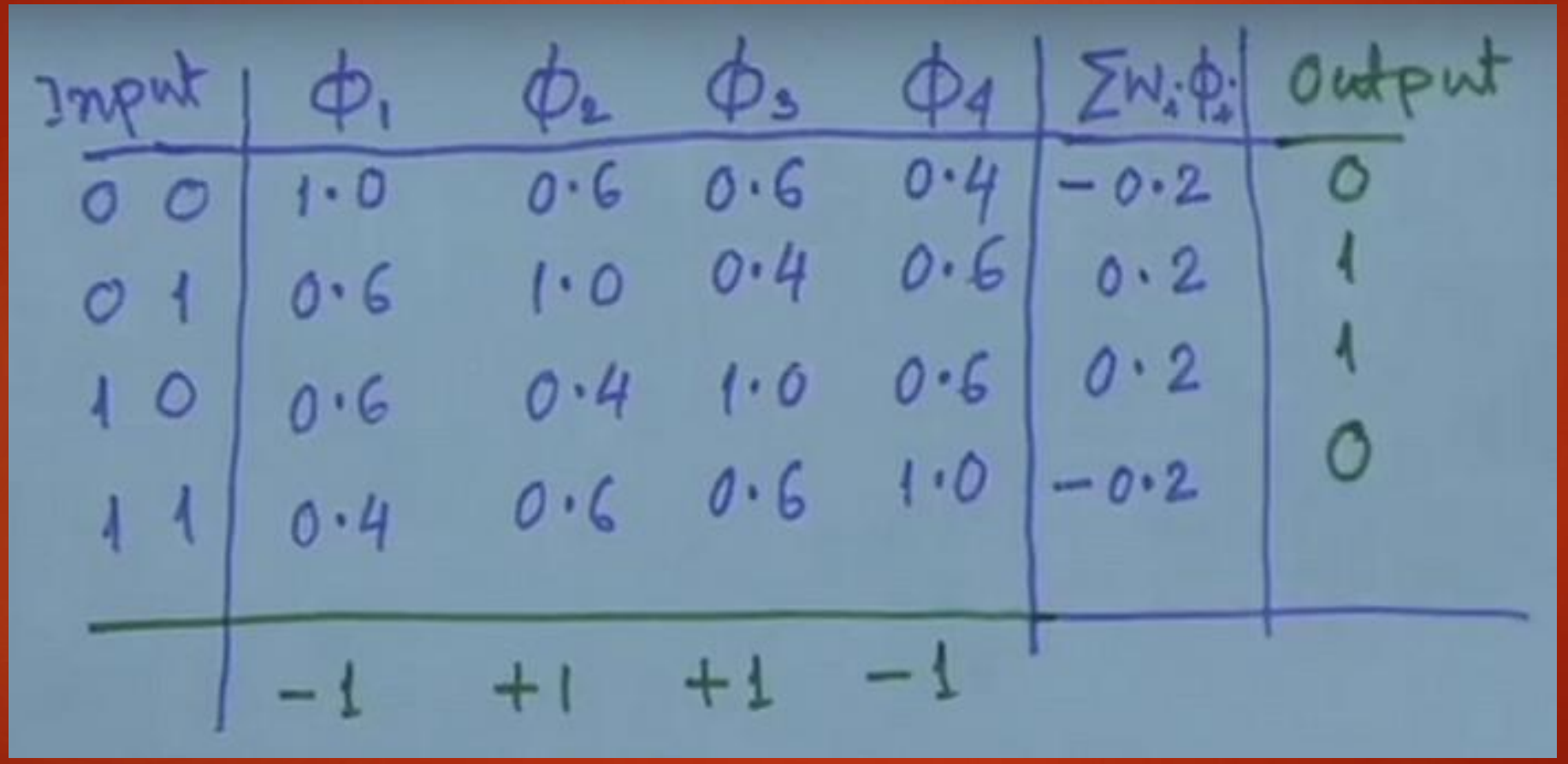
Ejemplo e Información adicional



Ejemplo e Información adicional



Ejemplo e Información adicional



Input	ϕ_1	ϕ_2	ϕ_3	ϕ_4	$\sum W_i \phi_i$	Output
0 0	1·0	0·6	0·6	0·4	−0·2	0
0 1	0·6	1·0	0·4	0·6	0·2	1
1 0	0·6	0·4	1·0	0·6	0·2	1
1 1	0·4	0·6	0·6	1·0	−0·2	0
	−1	+1	+1	−1		

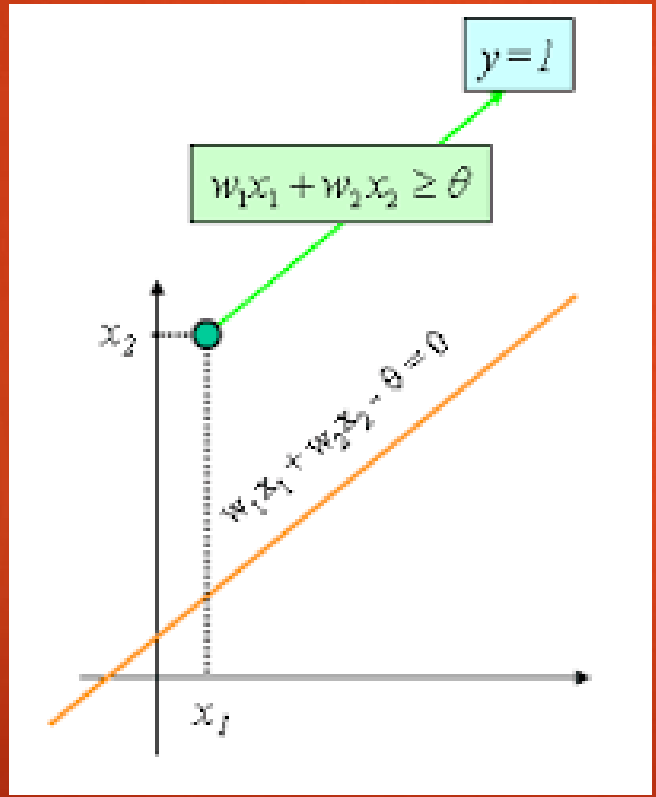
¿Dudas o comentarios?

Introducción

Entrenamiento de un perceptrón simple

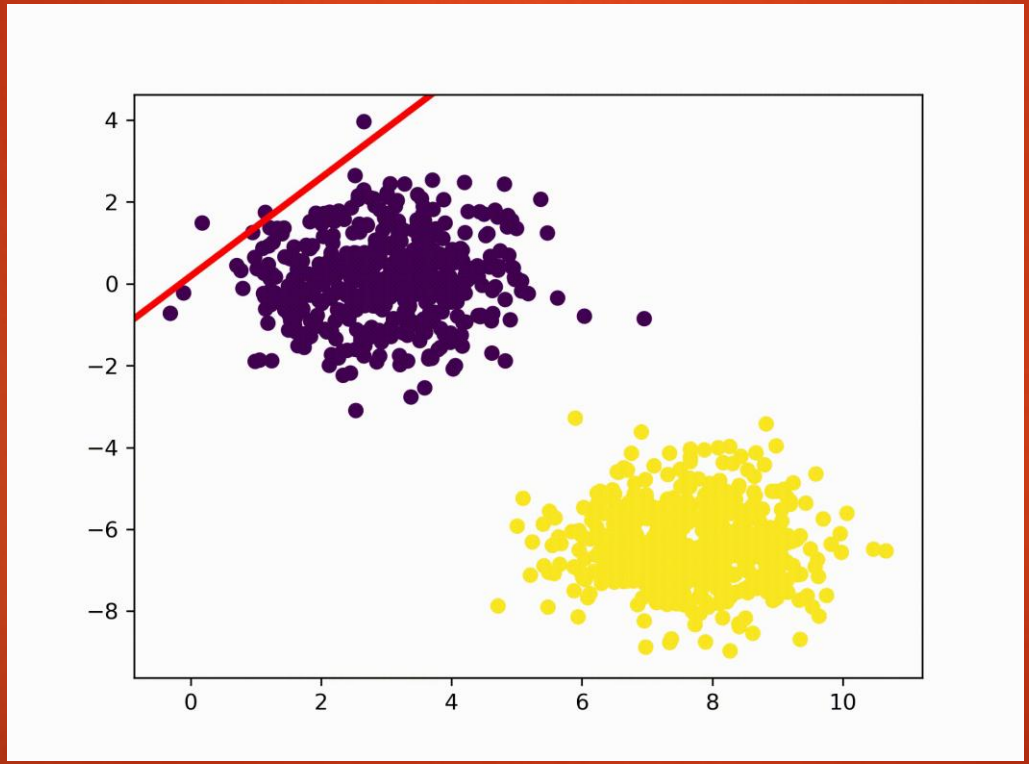
1. Inicializar los pesos.
2. Por cada uno de los patrones de entrenamiento:
    1. Calcular el error $e_i = d_i - y_i$
    2. Calcular la actualización del peso de acuerdo con la regla $\Delta w_{ij} = \alpha e_i x_j$
    3. Actualizar los pesos $w_{ij} = w_{ij} + \Delta w_{ij}$
    4. Contabilizar los errores
3. Ejecutar el paso 2 hasta que el error tenga un nivel de tolerancia adecuado. Cada ejecución del paso 2 será considerado como una época o epoch. El proceso puede también detenerse tras un número específico de épocas.

Interpretación geométrica

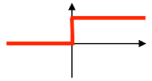
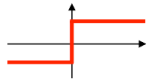
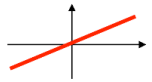
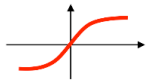
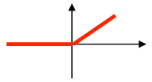


Las entradas y los pesos, junto con el umbral forman una ecuación de la recta de la forma Ax+By+C=0

Interpretación geométrica



Funciones de activación para RNA

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