

**Ciencia de Datos  
Igeniería Industrial  
UTN FRBA  
curso I5521**

# **Clase\_11: Redes Neuronales**



## Borges and AI

Léon Bottou <sup>†</sup> and Bernhard Schölkopf <sup>‡</sup>

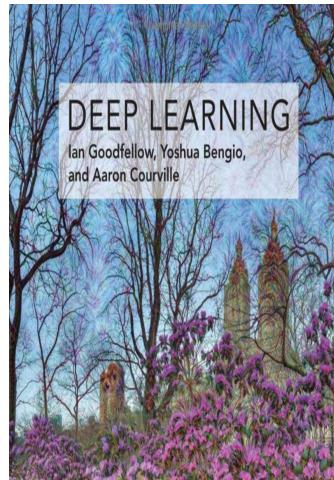
<sup>†</sup> FAIR, Meta, New York, NY, USA

<sup>‡</sup> Max Planck Institute for Intelligent Systems, Tübingen, Germany

### Abstract

Many believe that Large Language Models (LLMs) open the era of Artificial Intelligence (AI). Some see opportunities while others see dangers. Yet both proponents and opponents grasp AI through the imagery popularised by science fiction. Will the machine become sentient and rebel against its creators? Will we experience a paperclip apocalypse? Before answering such questions, we should first ask whether this mental imagery provides a good description of the phenomenon at hand. Understanding weather patterns through the moods of the gods only goes so far. The present paper instead advocates understanding LLMs and their connection to AI through the imagery of Jorge Luis Borges, a master of 20th century literature, forerunner of magical realism, and precursor to postmodern literature. This exercise leads to a new perspective that illuminates the relation between language modelling and artificial intelligence.

# Lectura sugerida



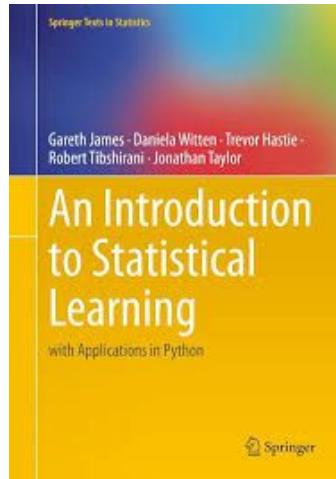
## Deep Learning

<https://www.deeplearningbook.org/>

Uno de los libros más importantes en redes neuronales contemporáneas

- Part II: Modern Practical Deep Networks
  - Capítulo 6: Deep Feedforward Networks
  - Capítulo 8: Optimization for Training Deep Models
  - Capítulo 10: Sequence Modeling: Recurrent and Recursive Nets
  - Capítulo 11: Practical Methodology

# Lectura sugerida



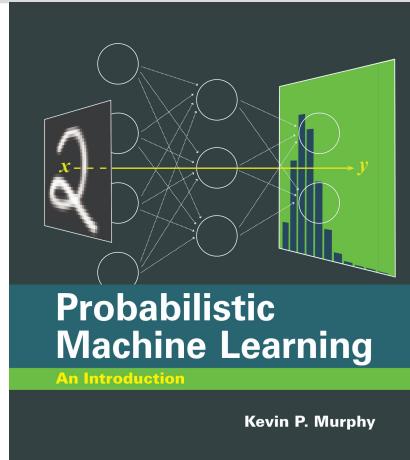
## **Introduction to Statistical Learning (with applications in Python)**

<https://www.statlearning.com/>

[https://hastie.su.domains/ISLP/ISLP\\_website.pdf.download.html](https://hastie.su.domains/ISLP/ISLP_website.pdf.download.html)

- Capítulo 10 -> Deep Learning

# Lectura sugerida



## Probabilistic Machine Learning: An Introduction

<https://probml.github.io/pml-book/book1.html>

- Capítulo 13: Neural Networks for Tabular Data

## Redes Neuronales

- Perceptrón
- Funciones de activación
- Multilayer Perceptrón (MLP)
- Arquitecturas de Redes Neuronales (NN)
- Entrenamiento de una NN
- Autoencoder

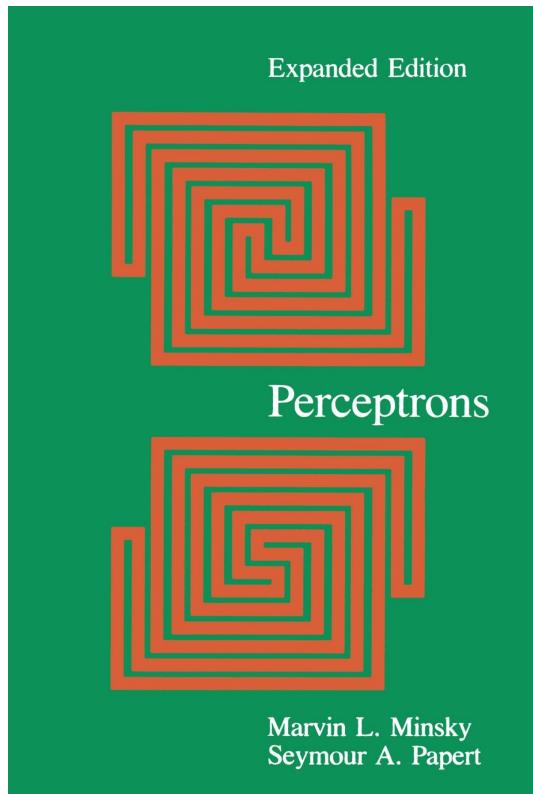
# ¿Que es una red neuronal artificial?

$$y = f_w(x)$$

$$w = \underset{w}{\operatorname{argmin}} L(y, \hat{y}) = \underset{w}{\operatorname{argmin}} L(y, \hat{f}_w(x))$$

Una red neuronal, es una función  $f_{\mathbf{w}(\mathbf{x})}$  lineal o no lineal caracterizada por un conjunto de parámetros “W”. Esta función toma como entrada un vector  $\mathbf{x}$  d-dimensional para generar una respuesta/salida unidimensional ‘y’ o multidimensional ‘z’. Los parámetros  $\mathbf{w}$  se ajustan resolviendo un problema de optimización tal que minimice una función de costo  $\mathbf{L}$  entre la respuesta ‘y’ del dataset y la respuesta de la red  $y_{\text{hat}}$ .

# Perceptrón



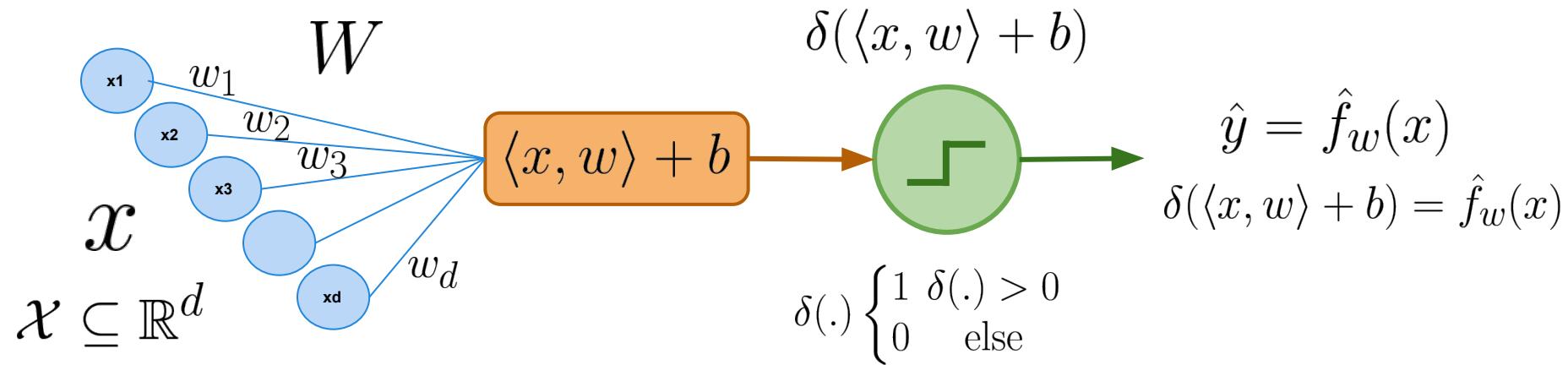
*Psychological Review*  
Vol. 65, No. 6, 1958

## THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN<sup>1</sup>

F. ROSENBLATT

*Cornell Aeronautical Laboratory*

# Perceptrón



Para entender las redes neuronales primero debemos estudiar el modelo **perceptrón**. Es un algoritmo de Clasificación Lineal originado en los años 50s. El modelo tiene **d** features/variable de entradas, donde cada feature de entrada estará combinada linealmente a un “peso **w**”. Es decir que tenemos una combinación lineal ( $\mathbf{x} \cdot \mathbf{w}$ ). Una vez realizada la combinación lineal se determina el valor de salida ( $\mathbf{w}^* \mathbf{x} + \mathbf{b}$ ) =  $y$ . Al resultado de la combinación lineal se lo afecta por una función de activación “step” que en el caso de ser binario resultará como 0 o 1.

El objetivo es aprender los pesos **w,b** que minimizan el error de clasificación **y\_hat**.

# Perceptrón

- Es un modelo que trabaja “online”. Procesa una muestra a la vez. De esta manera los pesos también se actualizan gradualmente.
- Únicamente en los casos donde el modelo realiza una predicción errónea los pesos se actualizan.
- El perceptrón computa una función de costo  $L(y, \hat{y})$  basada en la cantidad de veces que hubo una mal clasificación.
- La función de activación ('neurona') que utiliza es “signo o step” -> es 0 si  $x < 0$  y +1 si  $x > 0$ .

# Neurona

Las NN artificiales, inspiradas en su contraparte biológica, el cerebro humano, son aproximadores de funciones y se clasifican como modelos paramétricos.

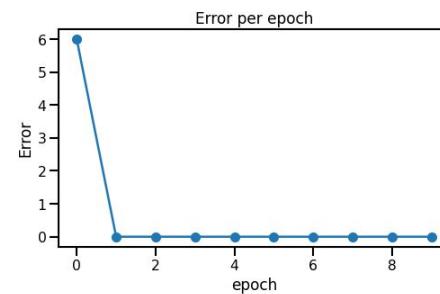
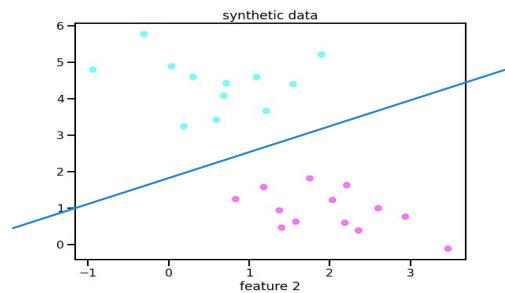
$$z(s) = \sigma \left( \sum_{j=1}^{n_{in}} W_j s_j + b \right) \quad (1)$$

La Eq. 1 proporciona la formulación básica de una sola neurona:

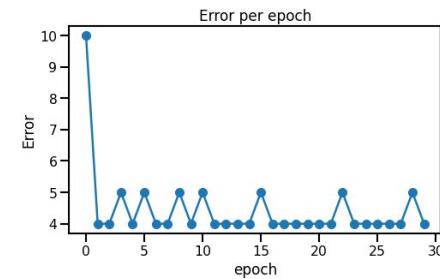
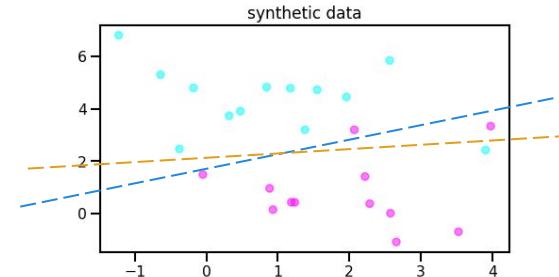
Donde  $z$  y  $s_j$  se refieren a la salida y la entrada de la neurona, y  $W_j$  y  $b$  se refieren a los parámetros que se pueden aprender. La agregación lineal de la entrada  $s_j$  ponderada por los parámetros  $W_j$  y  $b$  se denomina comúnmente *preactivación* de la neurona. Para enriquecer la complejidad de las funciones que la **NN** puede aproximar, la *preactivación* se utiliza como entrada de una función de activación  $\sigma(\cdot)$ . Las funciones de activación añaden capacidad no lineal a la **NN** que simula la estimulación producida por una neurona.

# Limitaciones del perceptrón

Linearly separable  
classes

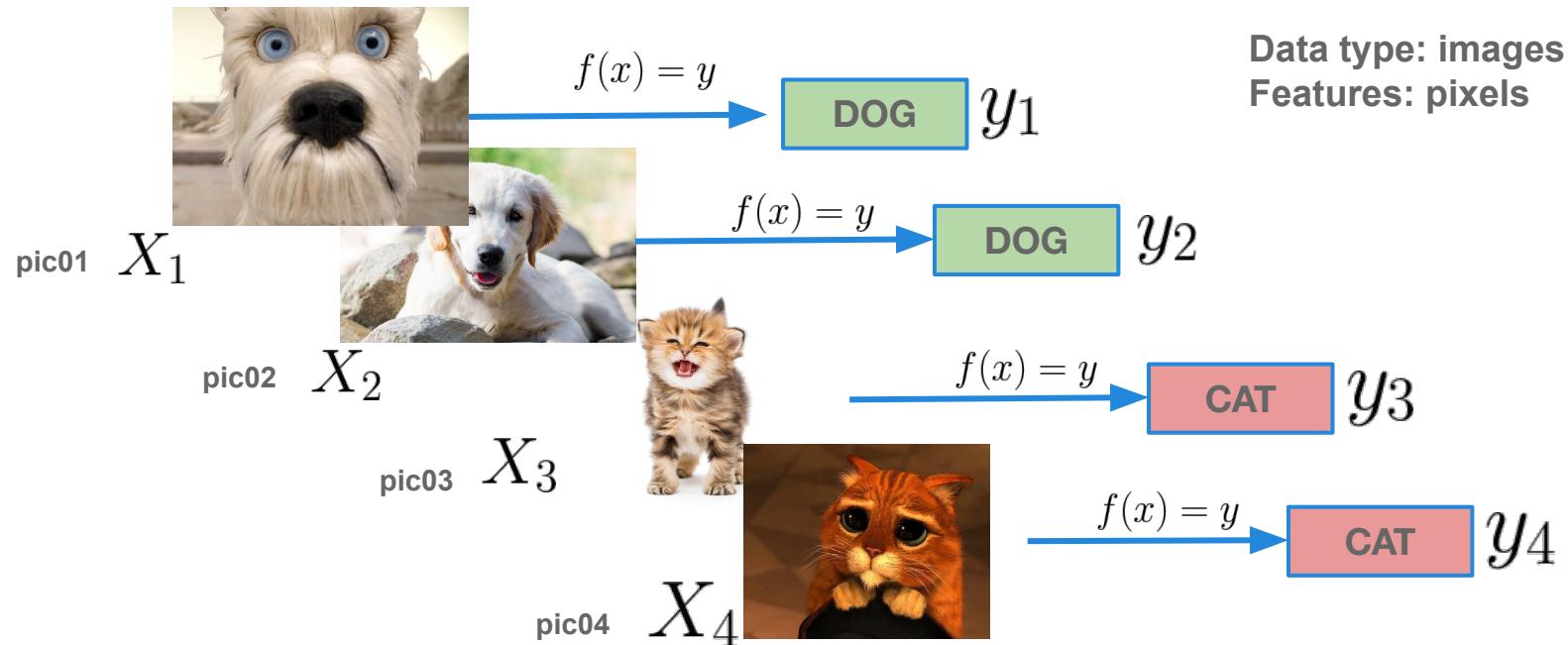


Non Linear separable classes



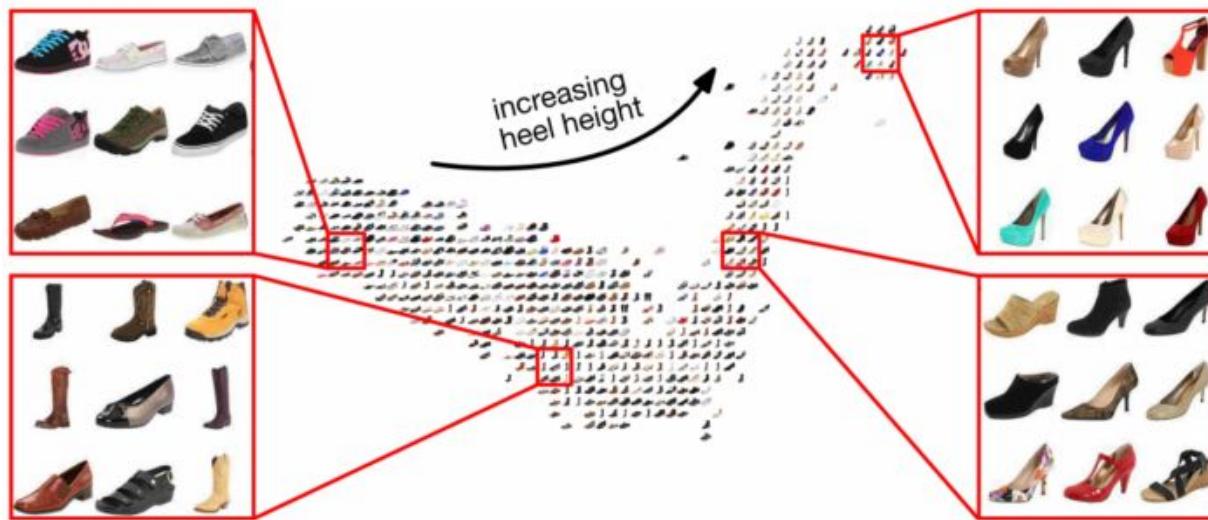
Durante el entrenamiento el dataset es procesado por el perceptrón múltiples veces iterativamente donde cada iteración se la denomina "epoch". El perceptrón únicamente actualiza los parámetros cuando el output  $\hat{y}$  es distinto a la verdadera etiqueta  $y$ . En clases linealmente separables el perceptron convergerá a una solución. En clases no separables el perceptrón iterara la cantidad de veces que el usuario le indique sin poder llegar a una solución definitiva.

# Aprendizaje Supervisado

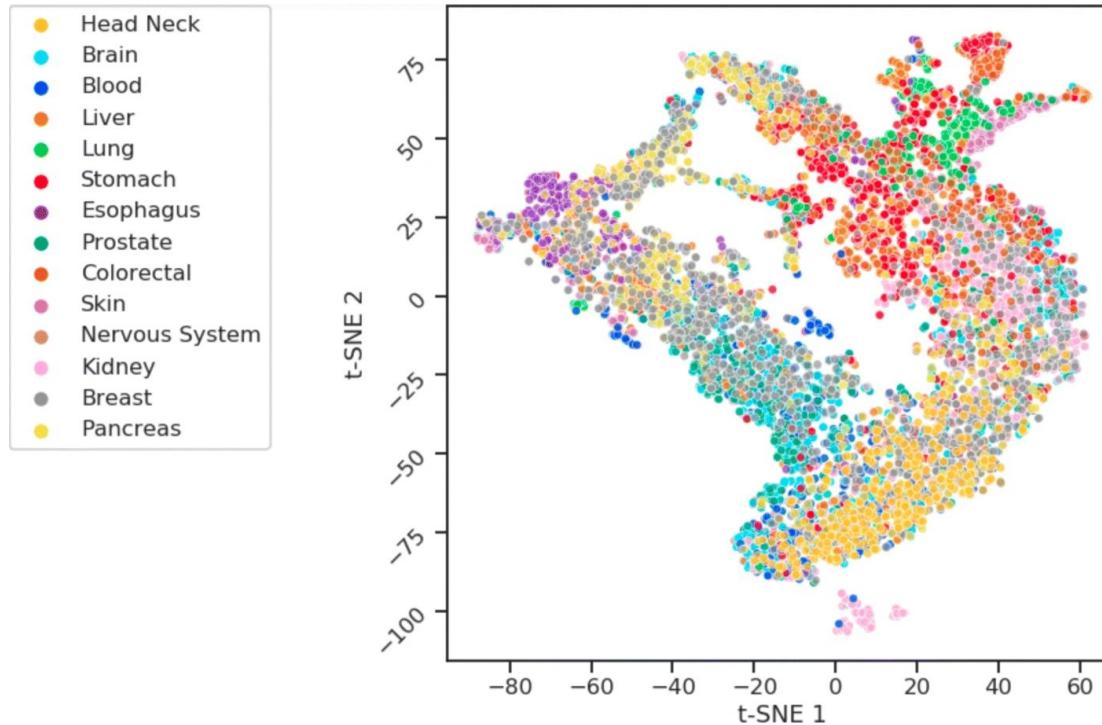


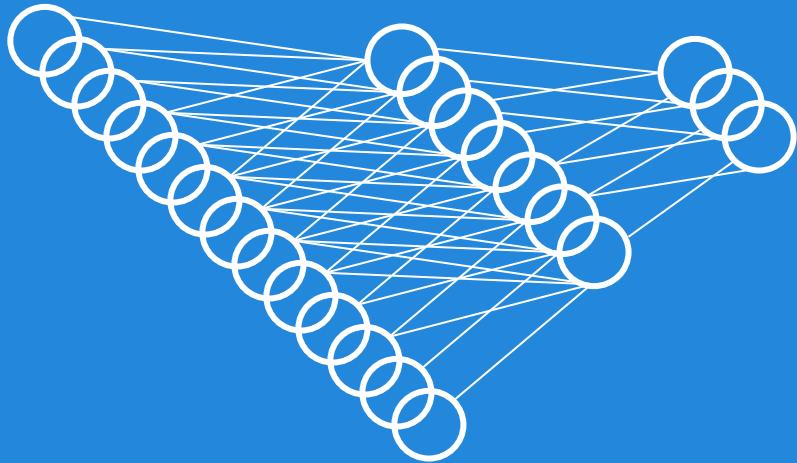
cada instancia esta asociada a una etiqueta determinada por un humano.

# Image embedding



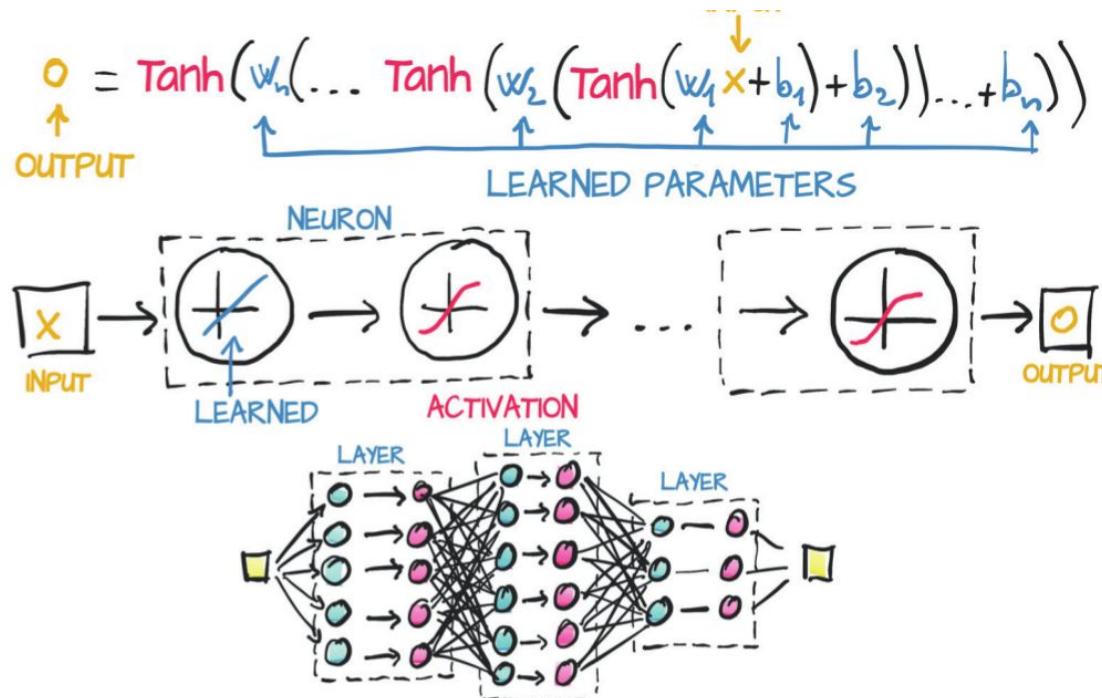
# Tumor embedding (tum2vec)



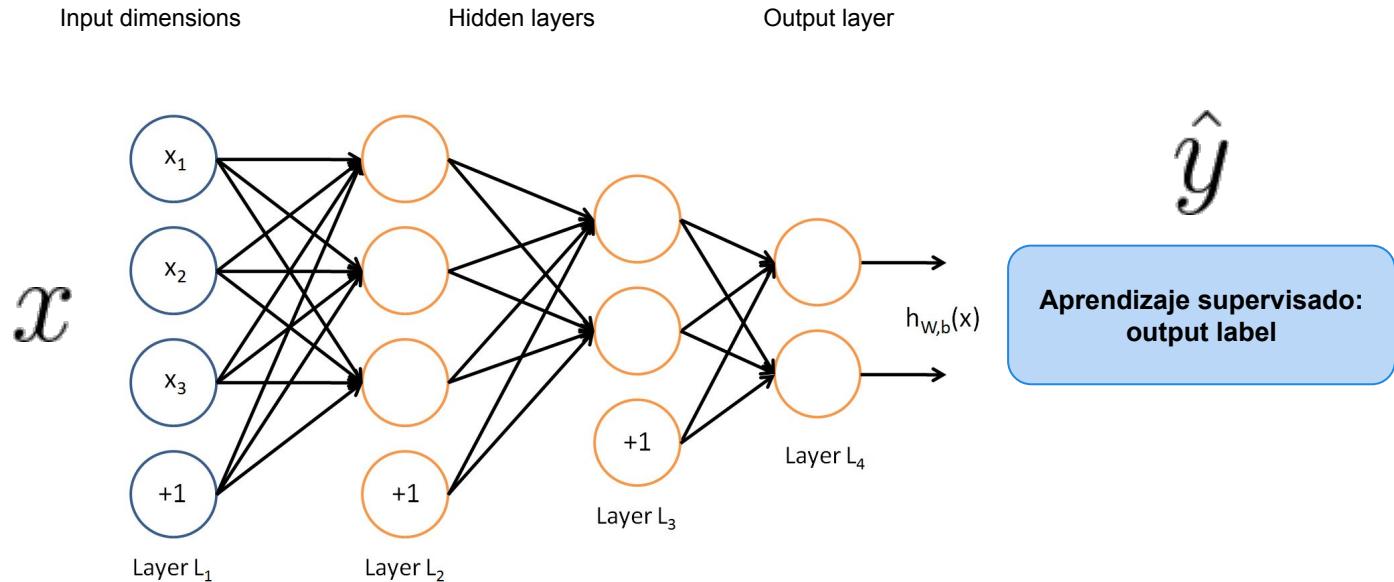


Redes Neuronales Artificiales

# Neural Networks: Arquitectura



# Neural Networks: Arquitectura



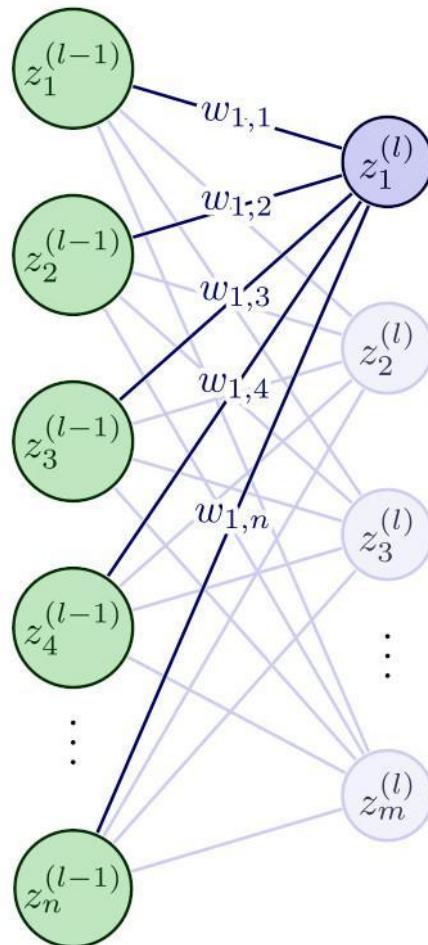
La arquitectura multi-capa o perceptrón multi-capa permite capturar patrones mas complejos en los datos. Cada capa es una representación de los datos donde cada neurona de una capa es una feature/variable aprendida por la combinación de variables de la capa anterior. A medida que mas capas se agregan mas complejas pueden ser las variables/features aprendidas. De todos modos agregar mas capas implica agregar mas complejidad y parámetros al modelo y de beneficio aprender representaciones mas beneficiosas de la distribución de los datos.

# Multilayer Perceptrón / Feedforward Network

Cuando se combinan varias neuronas  $n_{out}$  que reciben el vector dimensional  $n_{in}$   $s_j$  y producen el vector dimensional de activación, se crea una *capa*. Este punto de vista colectivo parametriza una capa mediante un vector de sesgos  $b_i$  y una matriz de pesos  $W_{ij}$ , donde  $i = 1, \dots, n_{out}$  y  $j = 1, \dots, n_{in}$ , junto con una función de activación fija  $\sigma(\ )$ . El apilamiento de muchas capas define una de las arquitecturas básicas de las **NN**, la **MLP**. En la Ec. 2 se puede encontrar la formulación matemática de un **MLP** para la capa  $l$ .

$$z_i^{(l)} = \sigma \left( \sum_{j=1}^{n_{l-1}} W_{ij}^{(l)} z_j^{(l-1)} + b_i^{(l)} \right) \begin{cases} \forall i = 1, \dots, n_l \\ \forall l = 1, \dots, L \end{cases} \quad (2)$$

# Multilayer Perceptrón / Feedforward Network

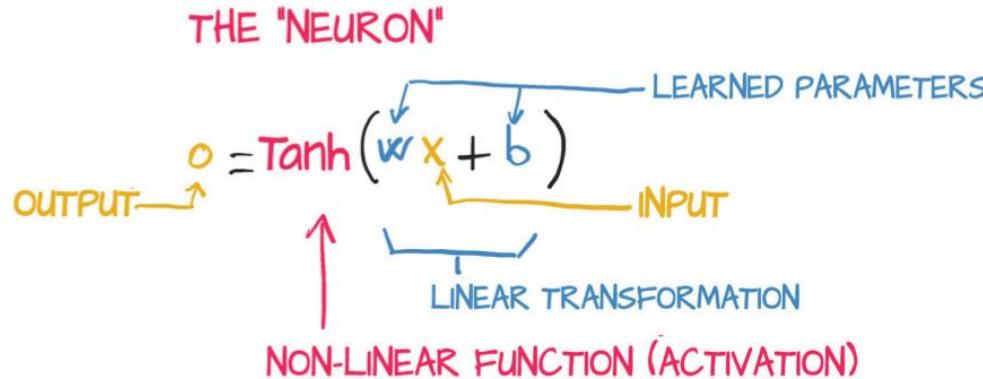


$$\begin{aligned} z_1^{(l)} &= \sigma(w_{1,1}z_1^{(l-1)} + w_{1,2}z_2^{(l-1)} + \dots + w_{1,n}z_n^{(l-1)} + b_1^{(l)}) \\ z_2^{(l)} &= \sigma\left(\sum_{j=1}^n w_{1,j}z_j^{(l-1)} + b_1^{(l)}\right) \end{aligned}$$

$$\begin{pmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_m^{(l)} \end{pmatrix} = \sigma \left[ \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \dots & w_{m,n} \end{pmatrix} \begin{pmatrix} z_1^{(l-1)} \\ z_2^{(l-1)} \\ \vdots \\ z_n^{(l-1)} \end{pmatrix} + \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_m^{(l)} \end{pmatrix} \right]$$

$$\mathbf{z}^{(l)} = \sigma(\mathbf{W}^{(l)} \mathbf{z}^{(l-1)} + \mathbf{b}^{(l)})$$

# Funciones de activación

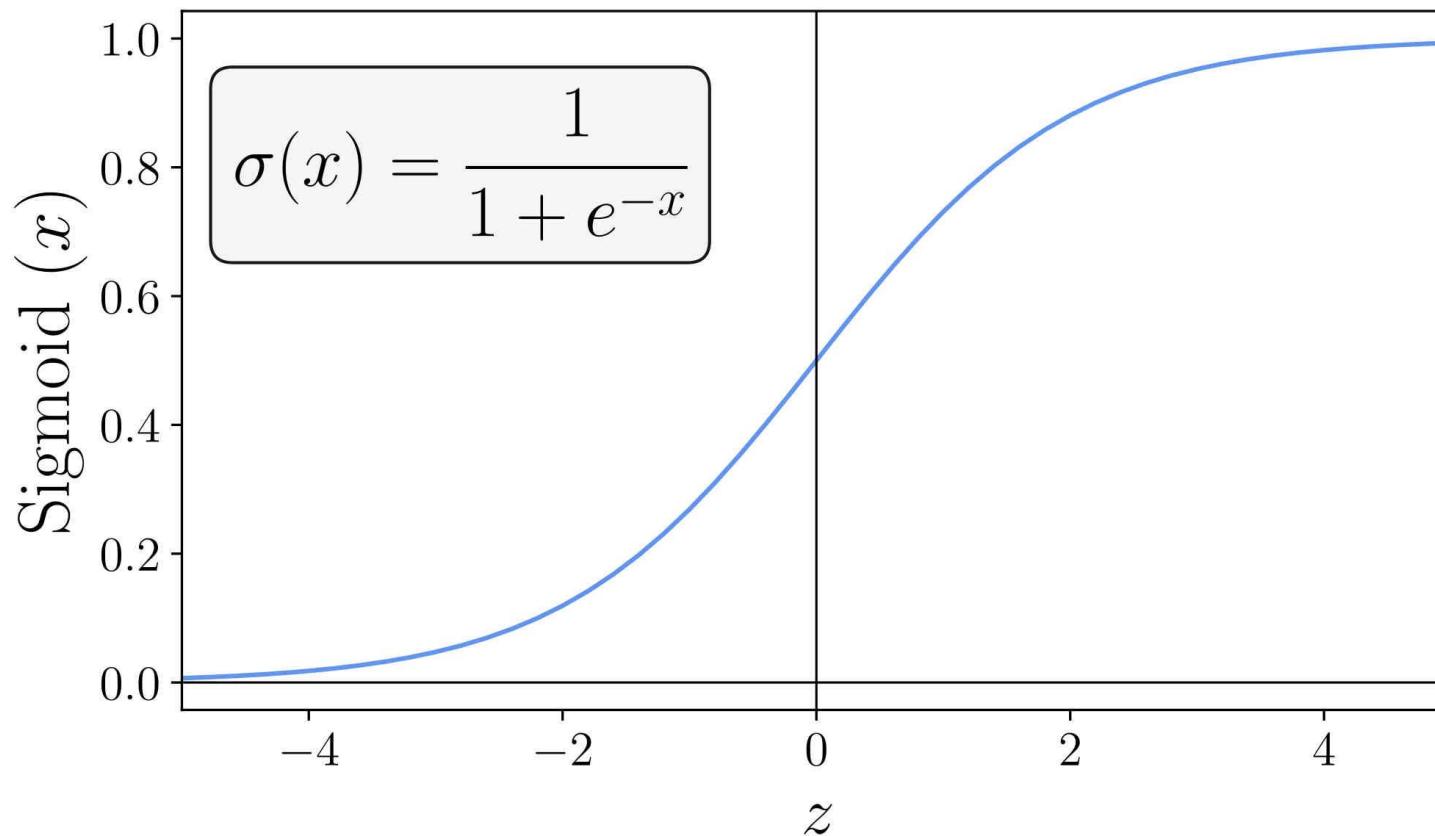


\*Deep Learning with Pytorch, 2019

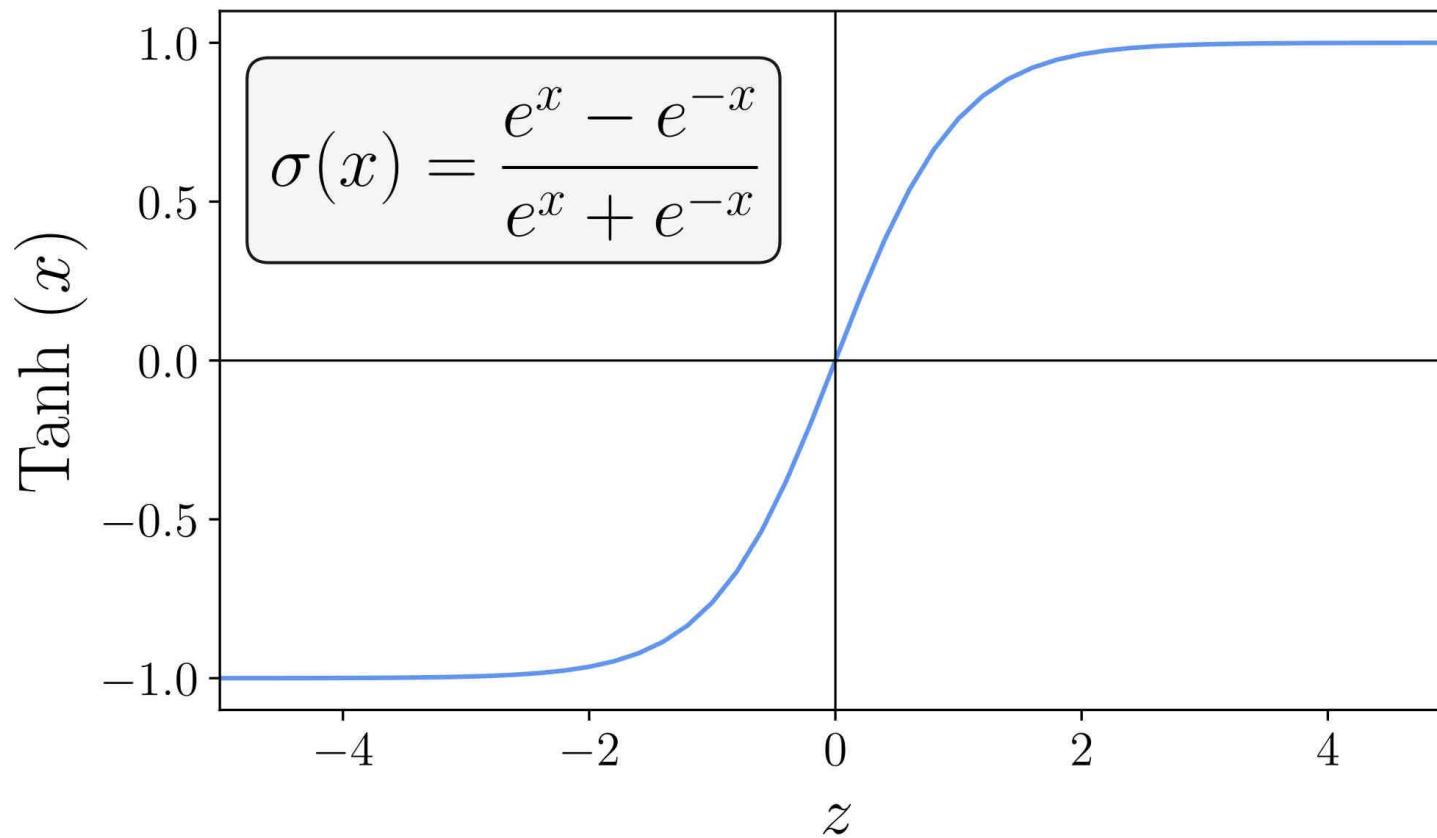
La función de activación es un **nodo o neurona** que tiene múltiples entradas y define una determinada salida. Las funciones de activación pueden estar “encendidas” o “desactivadas”. Algunas populares son:

- Sigmoid
- TanH
- ReLU
- Leaky-ReLu
- ELU
- Etc ...

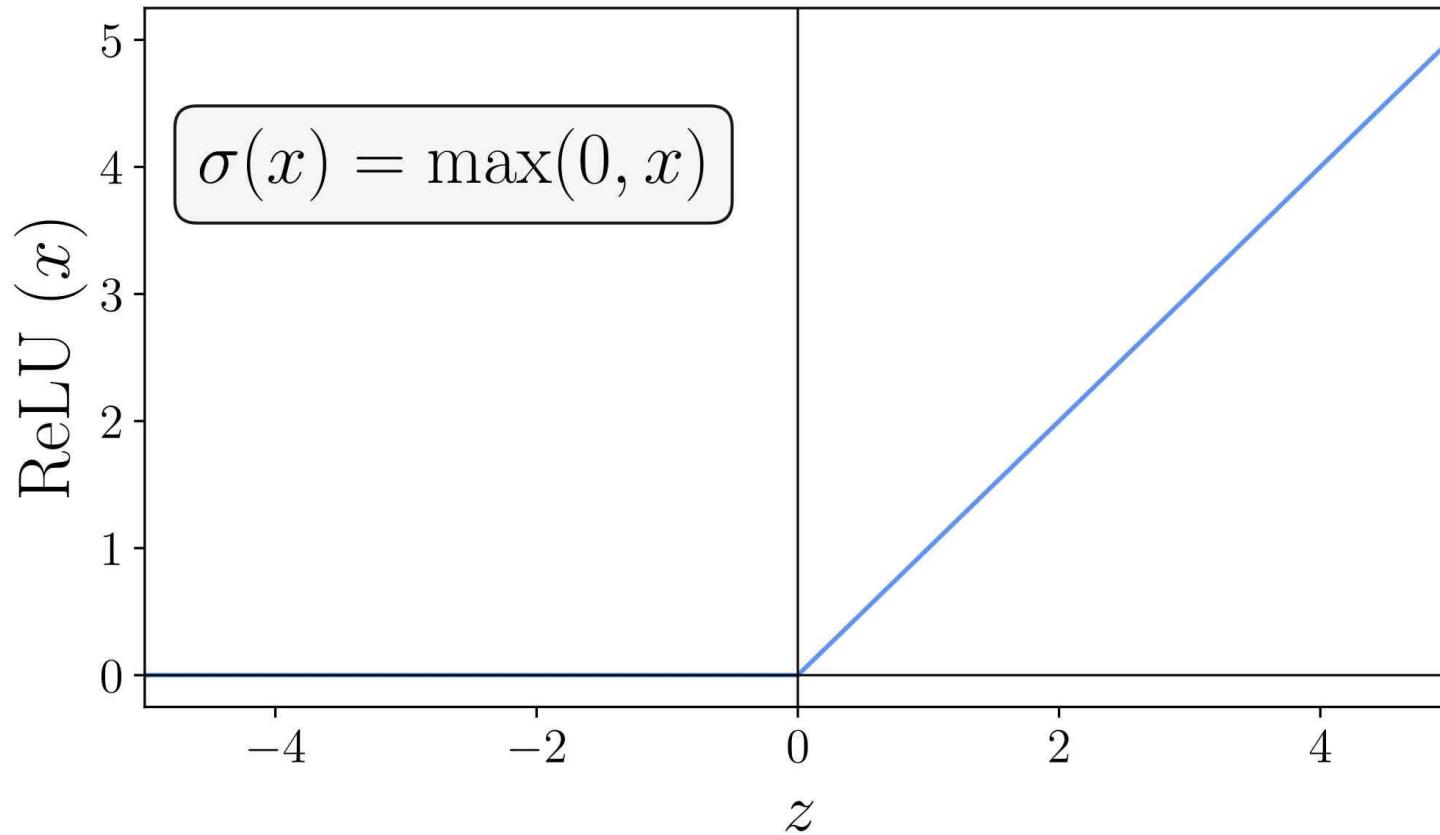
# Funciones de activación



# Funciones de activación

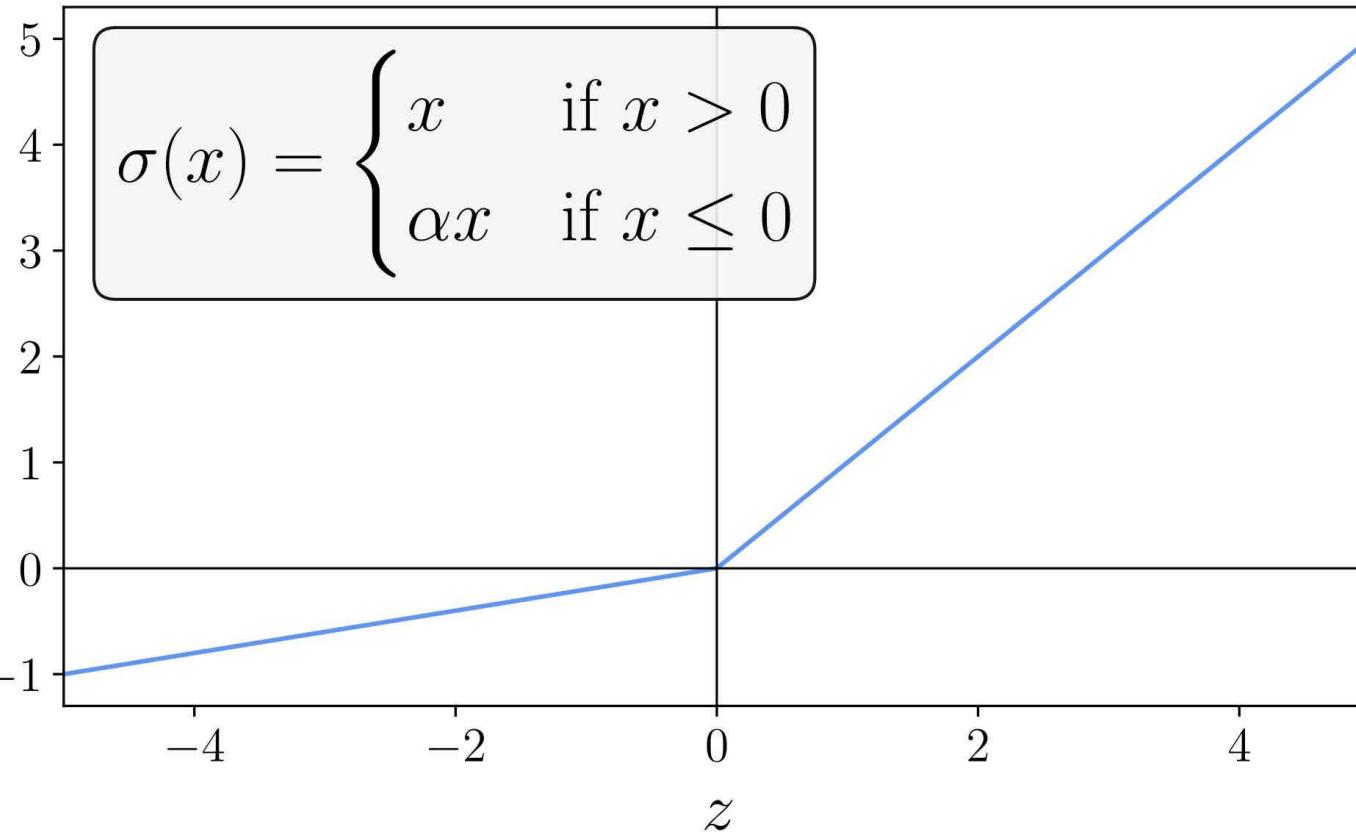


# Funciones de activación

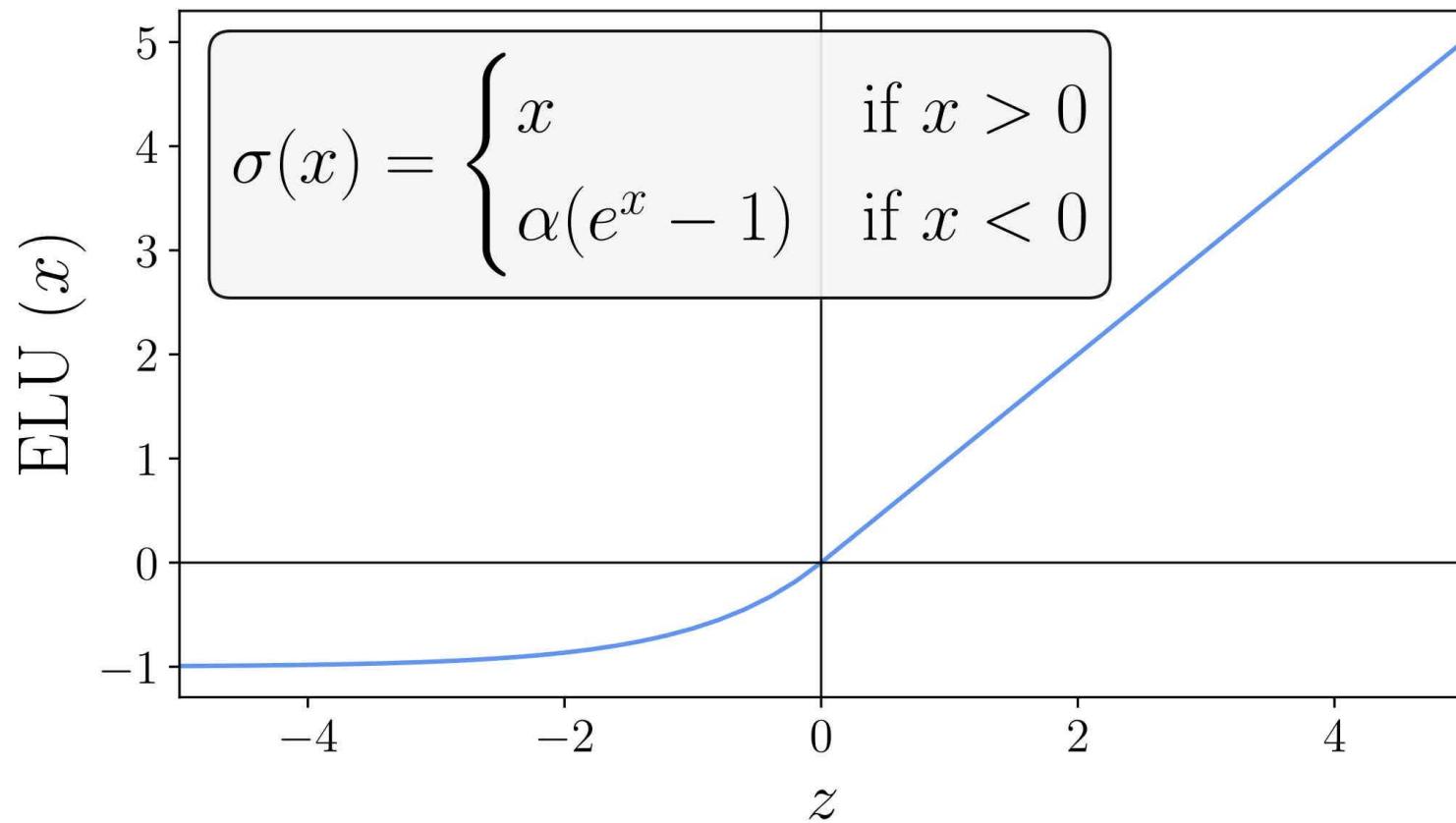


# Funciones de activación

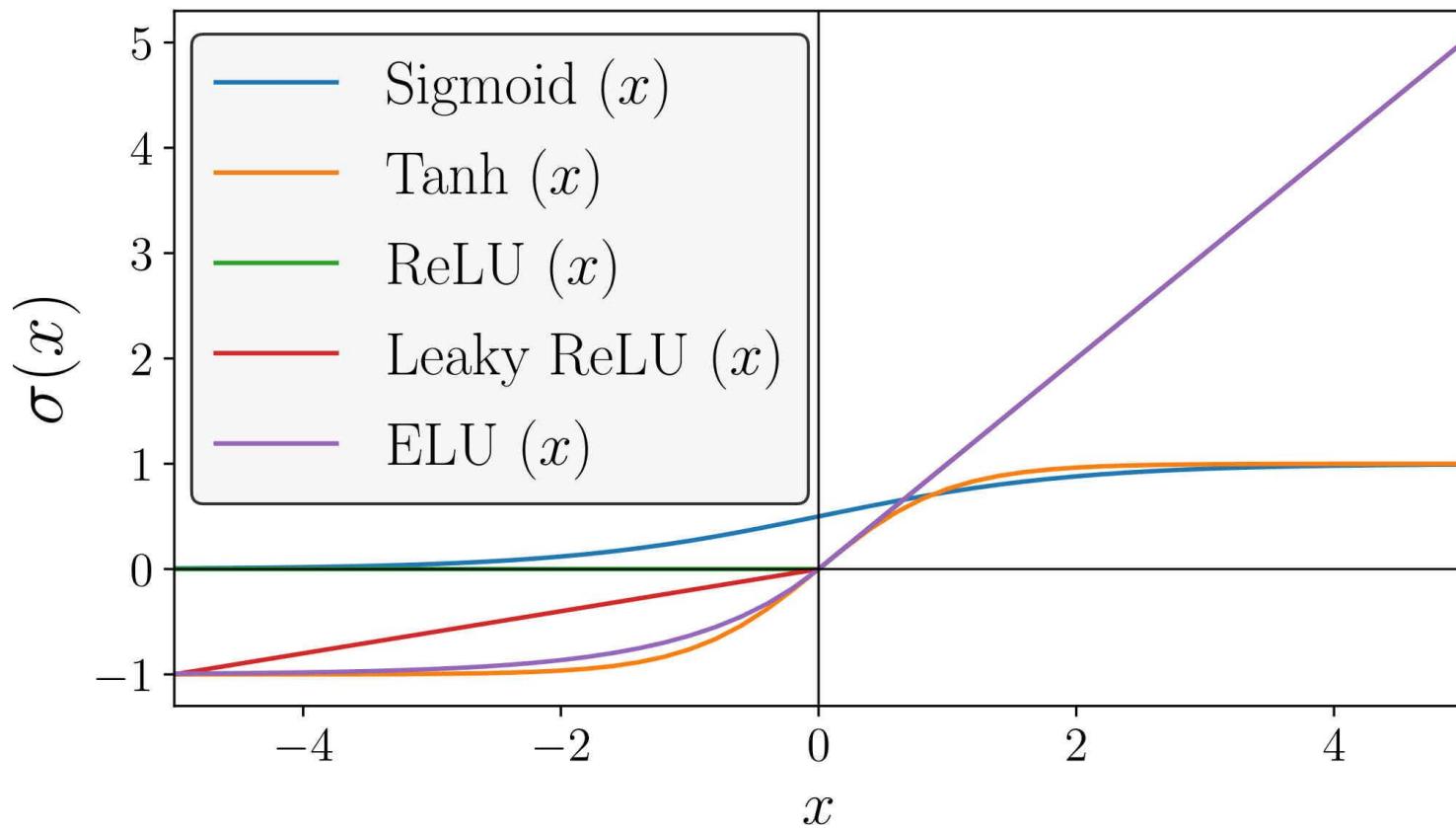
Leaky ReLU ( $x$ )



# Funciones de activación

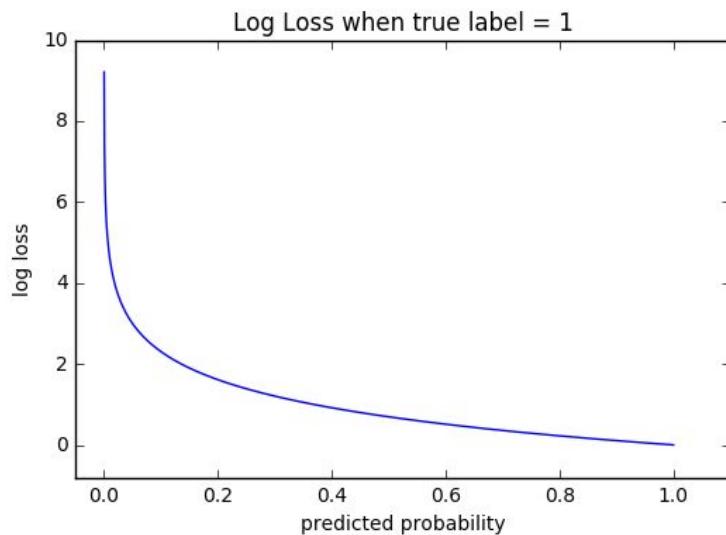


# Funciones de activación



# Funciones de costo: Cross Entropy

Mide la performance de un modelo de clasificación cuya salida es una probabilidad de 0 a 1. El Cross Entropy Loss aumenta cuando la probabilidad predicha diverge de la etiqueta actual. Un modelo perfecto debería dar una Cross Entropy loss = 0



$$H(P, Q) \stackrel{\text{def}}{=} \sum_j -P(j) \log Q(j)$$

$$\mathcal{L}_{CE}(y, \hat{y}) = - \sum_{j=1}^q y_j \log \hat{y}_j$$

# Funciones de costo: Cross Entropy

Softmax: Multi o binary classification

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Sigmoid: Binary classification

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

Al final de todas las “layers” de la red neuronal se ubica una función de activación llamada “Softmax” o ‘Sigmoid’ que arroja valores entre 0 y 1 a modo de probabilidad siendo las etiquetas 0 y 1. Los pesos de la red se aprenderán de manera tal que la salida de la softmax o sigmoid coincida lo mejor posible con las etiquetas reales de cada muestra.

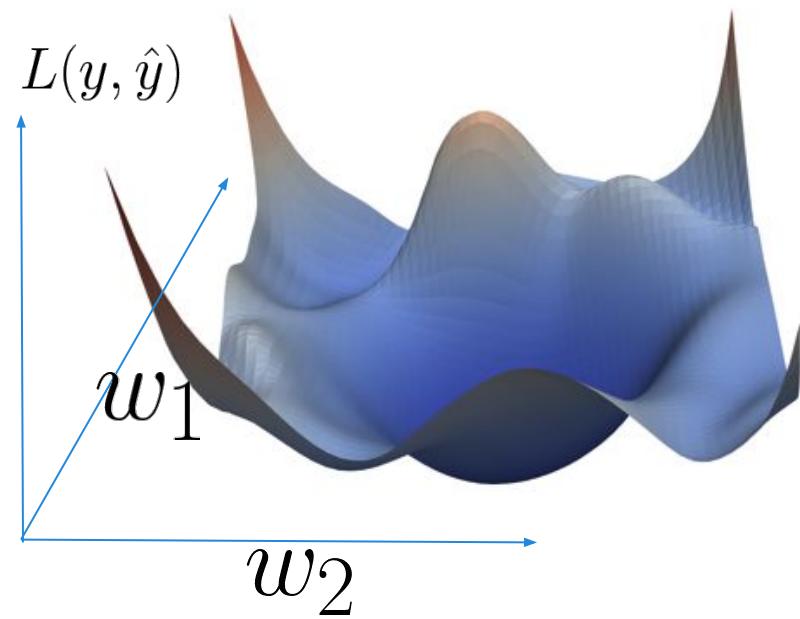
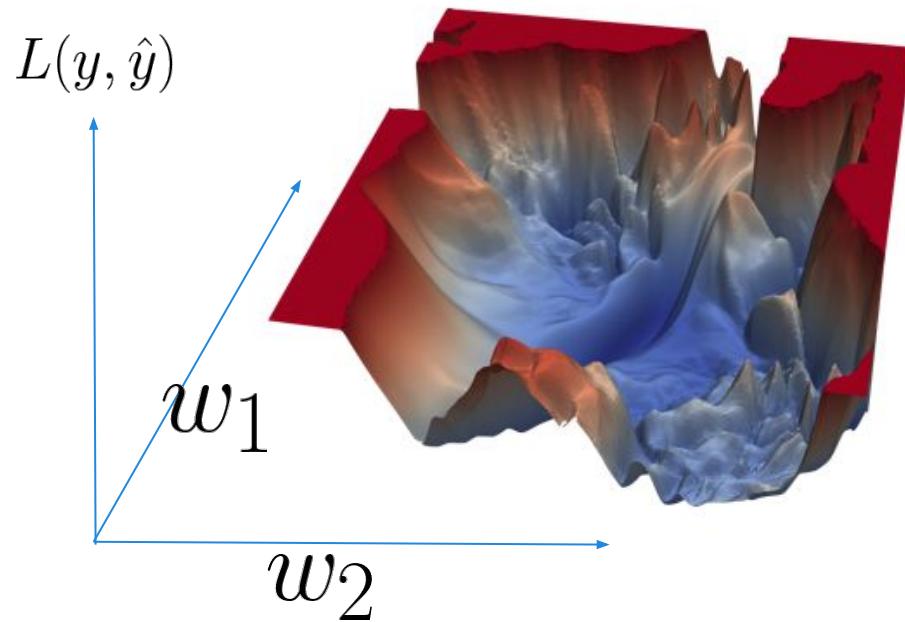
# Funciones de costo: Mean Squared Error

Mide la performance de un modelo de **regresión** cuya salida es la diferencia al cuadrado de los residuos. El MSE además de usarse en problemas de regresión puede usarse en autoencoders donde se busca reconstruir un vector d-dimensional.

$$\text{MSE} = \|y_i - y_j\|^2$$

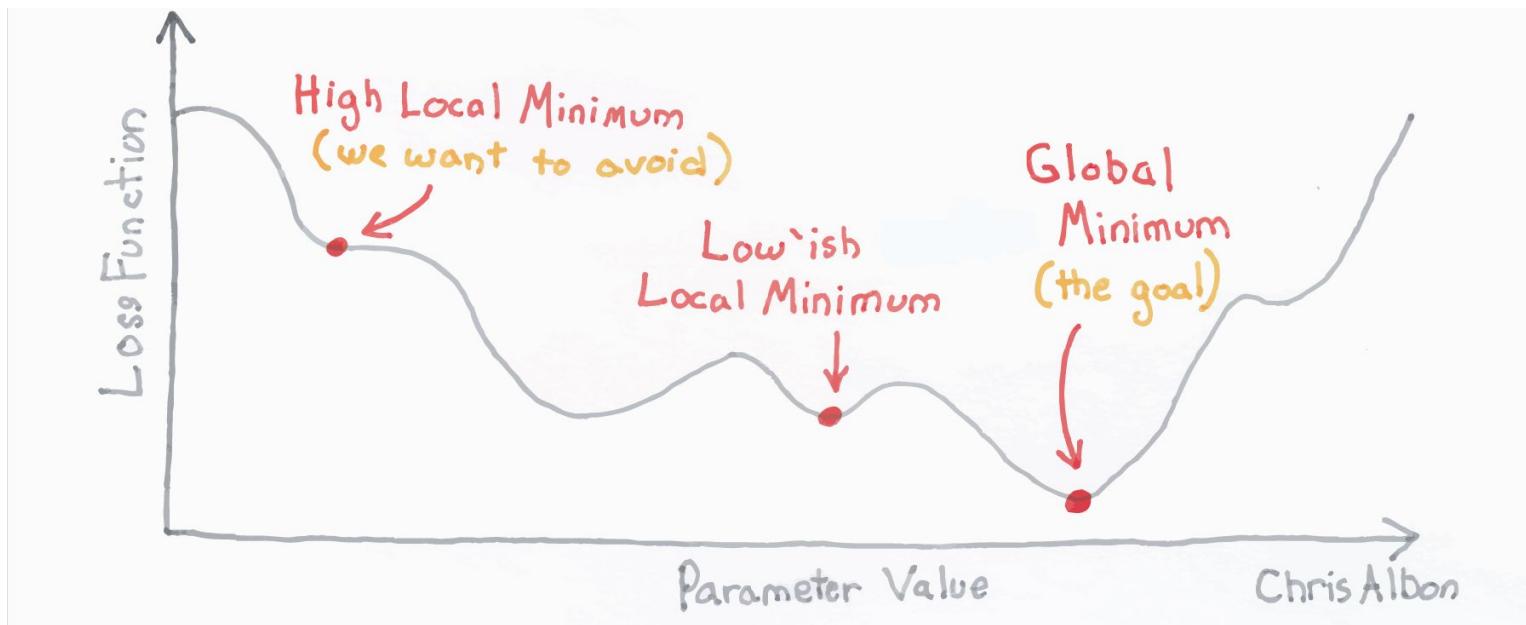
# Funciones de costo

Las funciones de costo dependen de los parámetros  $w$  (pesos) del modelo y del dataset en cuestión. Para encontrar los parámetros  $w$  que minimicen el error de predicción se debe resolver un problema de optimización no-lineal. El objetivo es encontrar un vector de parámetros que se encuentre en un mínimo local aceptable de la función de costo.

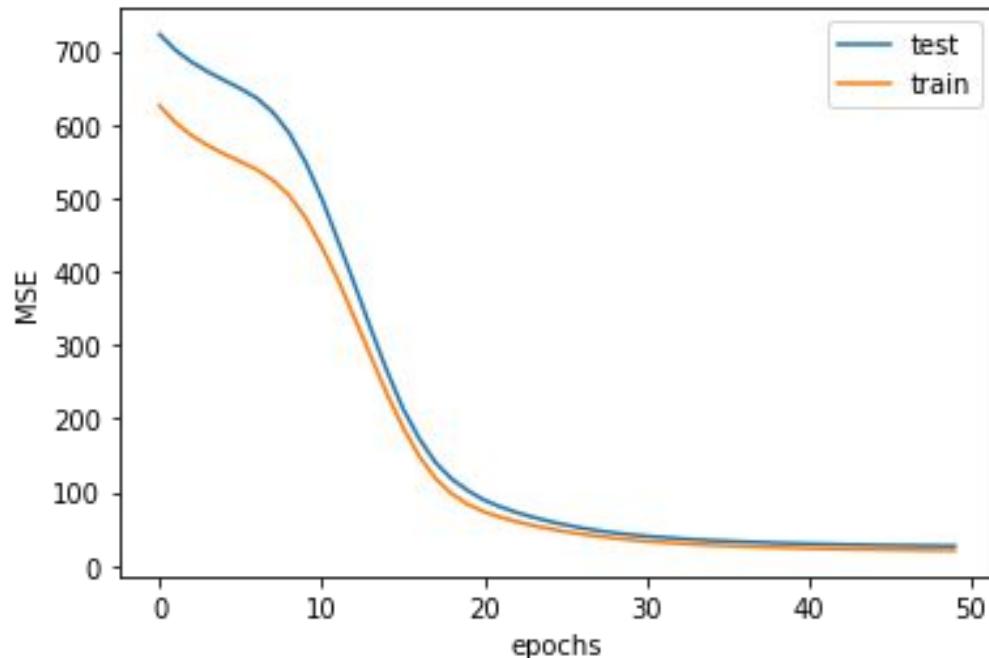


# Funciones de costo

Nuestro deseo es encontrar la configuración de pesos  $w$  que minimiza globalmente la función de costo. A veces no conseguimos ese objetivo ya que la función  $L$  puede tener múltiples mínimos locales, aunque podemos llegar a mínimos locales que permiten una performance de clasificación aceptable.



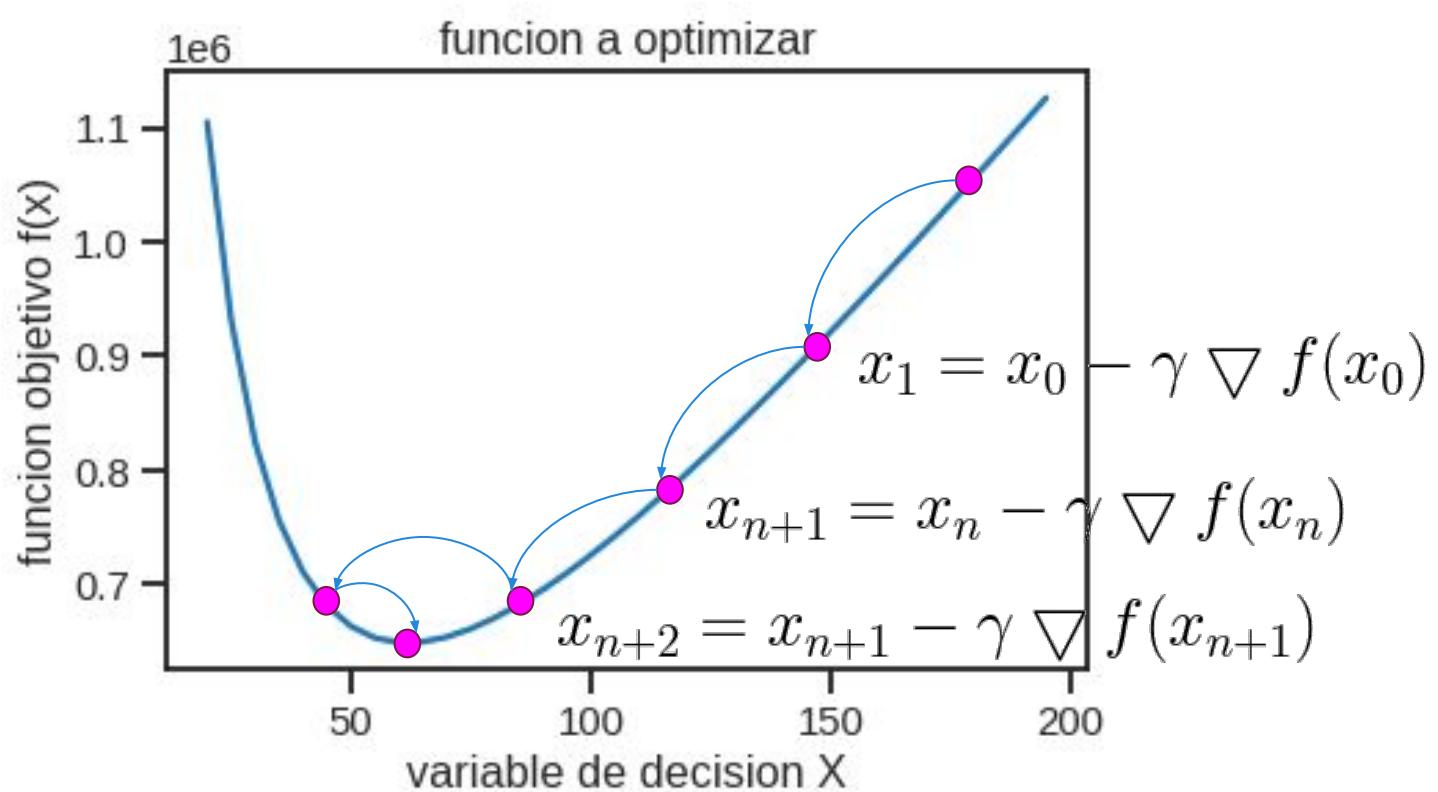
# Training Loss



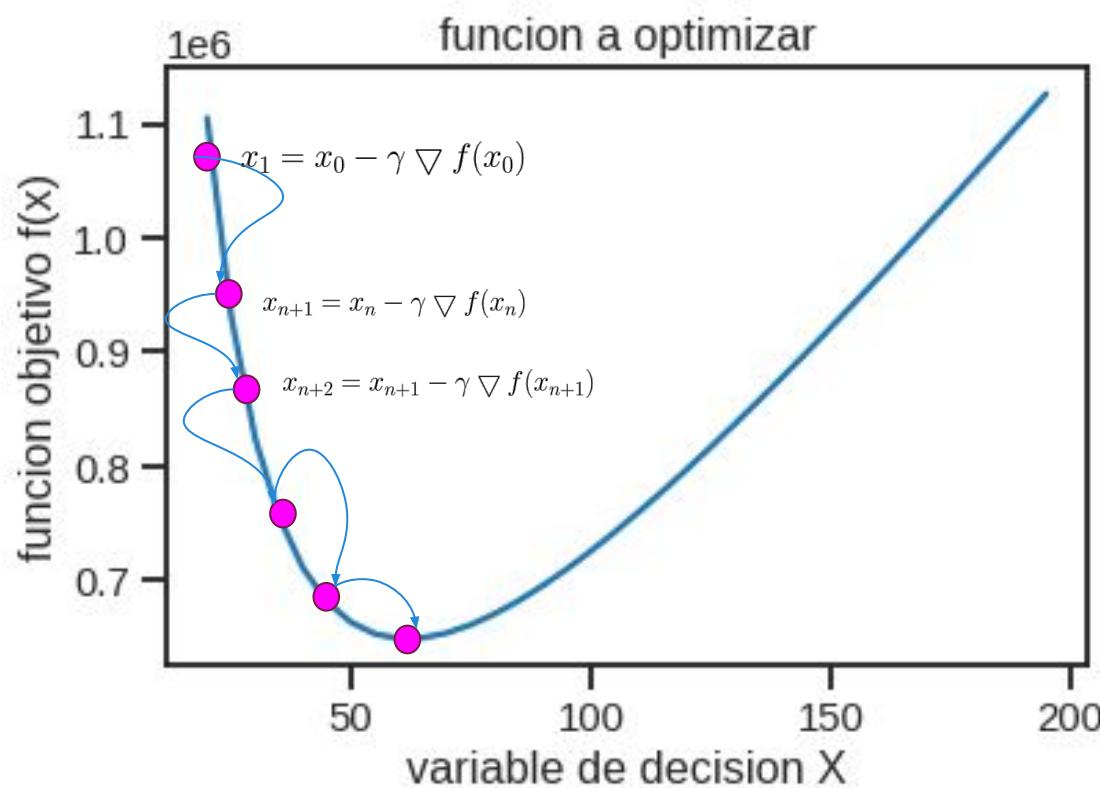
# Descenso por gradiente

Metodo para resolver: optimización no lineal

# Gradiente descendente



# Gradiente descendente



# Optimización por gradiente descendiente

$$x \in \mathbb{R}^d$$

$$\min_x f(x)$$

$$\nabla f(x_0)$$

$$\gamma$$

Gradiente de F(x)

Paso de optimización

$$x_1 = x_0 - \gamma \nabla f(x_0)$$

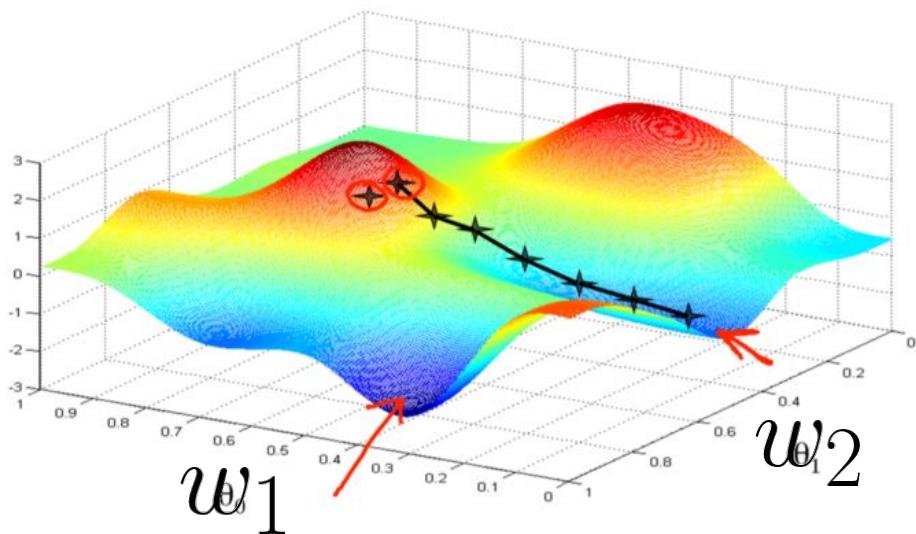
$$x_{n+1} = x_n - \gamma \nabla f(x_n)$$

$$f(x_n) > f(x_{n+1})$$

$$x_{n+2} = x_{n+1} - \gamma \nabla f(x_{n+1})$$

Supongamos que queremos optimizar la función no lineal, continua y diferenciable  $f(x)$  definida en el dominio de X. Se partirá desde la solución inicial  $x_0$  (llamado semilla) actualizando iterativamente la solución del problema en dirección al gradiente descendiente con un paso *gamma* de manera tal que la nueva solución presente un valor de la función objetivo menor.

# Learning Rate y Loss Function



En este caso tenemos una función de costo determinada por dos parámetros (pesos). El Learning Rate se visualiza como los “pasos” que se dan hacia la búsqueda del mínimo. Un Learning rate bajo significa tardar mucho en llegar a un mínimo local. Un LR alto puede significar pasar “por alto” un mínimo y no converger.

# Backpropagation & Gradient Descent

**Gradient Descent** o Gradiente Descendiente es una técnica de optimización para explorar la función de costo y buscar el mínimo que optimizará nuestra red. El punto de mínimo costo en la función de error implica la combinación de parámetros (pesos) que minimiza el error o la mal clasificación con las muestras de train. Una vez que el gradiente descendiente encuentra un mínimo, detendrá la búsqueda y se quedará en dicha región de la función de costo. La manera en que se busca

$$g = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\Theta} \mathcal{L}(z(x_i, \Theta_t), y_i)$$

$$\Theta_{t+1} = \Theta_t - \eta g$$

El **Learning Rate** (alpha) es un hiperparámetro que indica cuán rápido se actualizan los pesos en búsqueda de un mínimo.

# Backpropagation

nature

Explore content ▾ About the journal ▾ Publish with us ▾ Subscribe

nature > letters > article

Published: 09 October 1986

## Learning representations by back-propagating errors

David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Nature 323, 533–536 (1986) | [Cite this article](#)

75k Accesses | 11068 Citations | 239 Altmetric | [Metrics](#)

### Abstract

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods

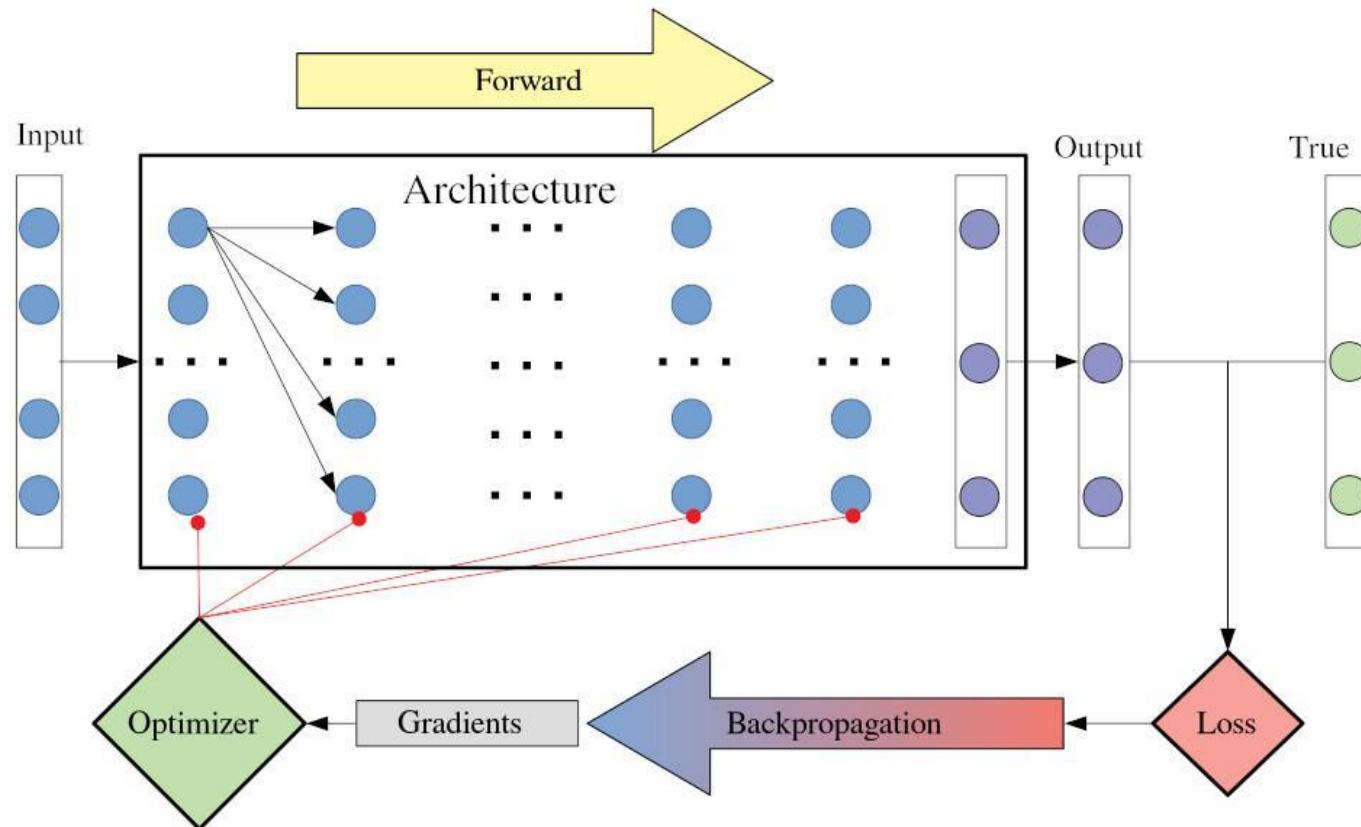
**Backpropagation** es el **algoritmo** de actualización de pesos de una red neuronal para moverse a favor del gradiente descendiente. Sucede luego que la información fluye desde la entrada input a la salida output (forward propagation) y se conoce si el modelo clasificó bien o mal a cada muestra. Mediante la **regla de la cadena** se calculan los gradientes de la función de costo en función de los parámetros  $w$ , es decir que se actualizan los pesos que aproximan en dirección al mínimo (local).

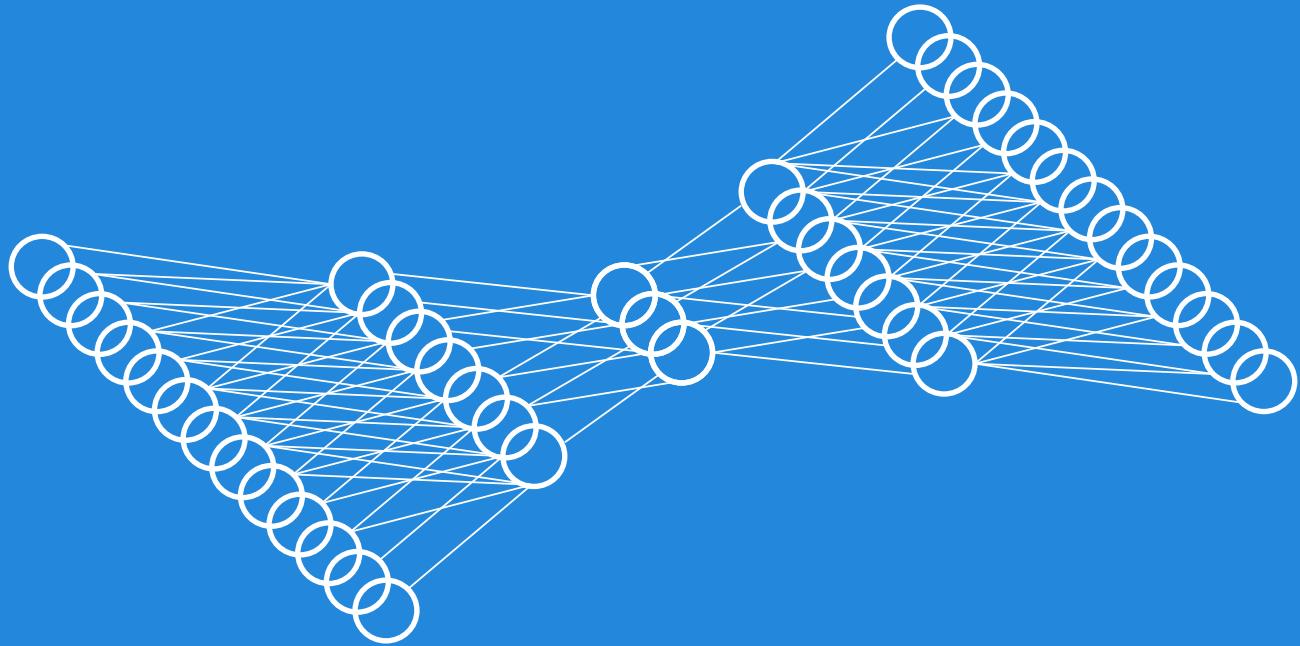
# Neural Networks: Components

- Activation functions
- Network architecture
- Loss Function
- Weight optimization technique
- Learning rate

# Neural Networks: Diagrama de entrenamiento

## Diagrama de Entrenamiento de Redes Neuronales





# Autoencoders

# Sample-to-feature ratio

“n” samples

“d” features

$$\mathbf{X}_{(n,d)} = \begin{bmatrix} x_{00} & x_{01} & \dots & x_{0d} \\ x_{10} & x_{11} & \dots & x_{1d} \\ \dots & \dots & \dots & \dots \\ x_{n0} & x_{n1} & \dots & x_{nd} \end{bmatrix}$$

$\frac{n}{d} < 1$  

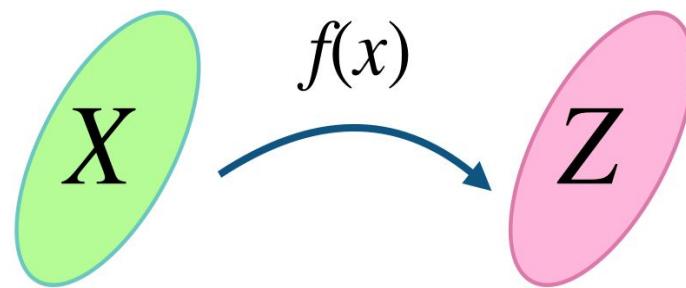
$\frac{n}{d} > 1$  

# Dim. Reduction: Feature extraction

$$\mathcal{X} \in \mathbb{R}^d$$

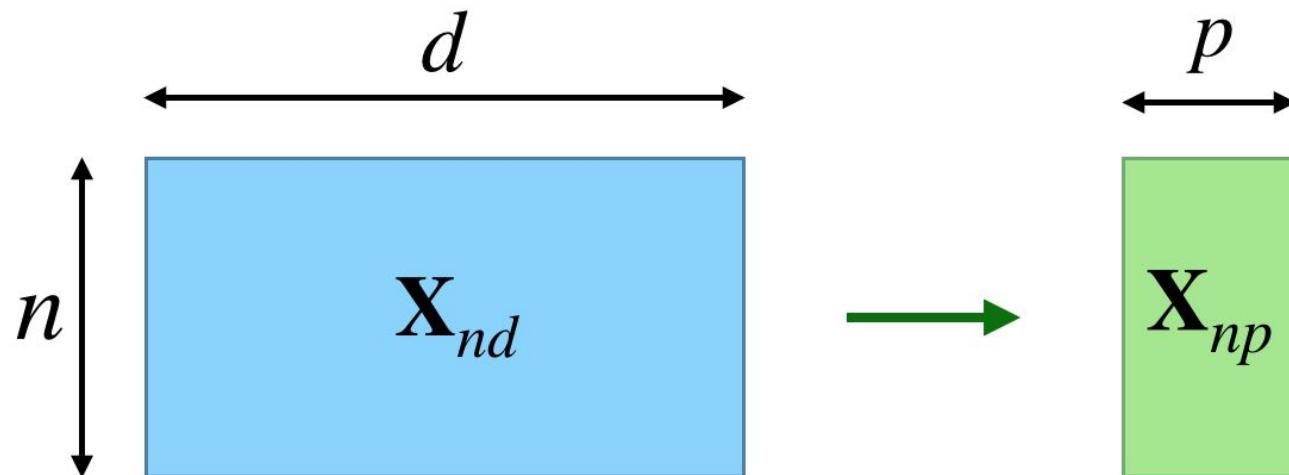
$$\mathcal{Z} \in \mathbb{R}^p$$

$$p < d$$

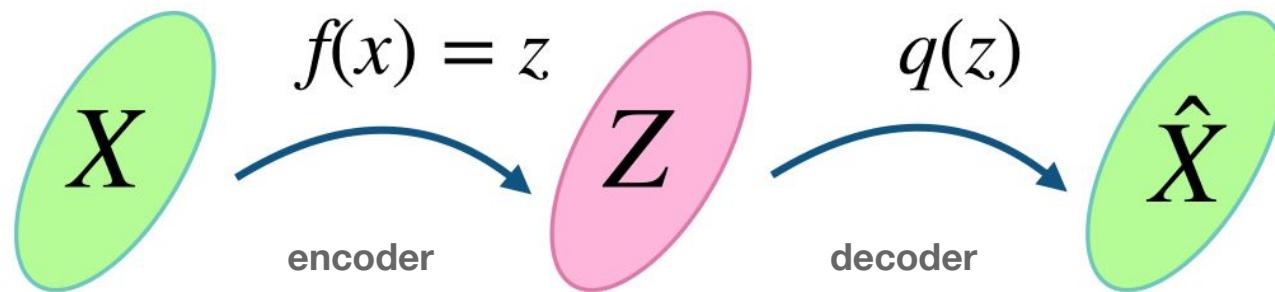


Una forma de reducir la dimensionalidad es mediante métodos de extracción de características (PCA, kPCA, Autoencoders). Este método aprende una transformación  $f(x)=z$ , donde  $\mathbf{Z}$  es un subespacio de baja dimensionalidad en el que se proyectan los datos  $\mathbf{X}$ . Posteriormente, se puede realizar el clustering sobre este subespacio reducido.

# High dimensional data



# Autoencoder



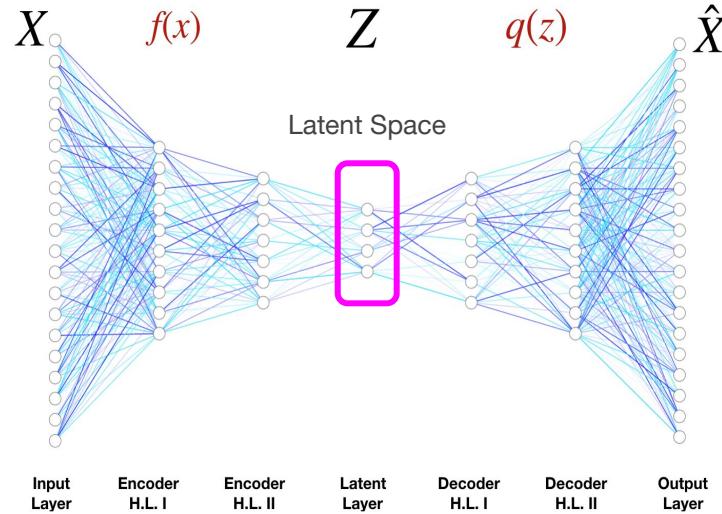
El objetivo es aprender dos funciones:

- el **encoder**  $f(x) = z$  mapea los vectores de entrada de alta dimensión  $X$  en un espacio latente de baja dimensión  $z$
- el **decoder**  $q(z) = x$  mapea los vectores latentes  $z$  al espacio de entrada original de alta dimensión.

## Autoencoder Loss Function: MSE

$$L(x, \hat{x}) = L(x, q(f(x))) = \|x - \hat{x}\|^2$$

# Autoencoder



$$\begin{aligned} \mathbf{z} &= f(\mathbf{x}, \mathbf{W}_f) = \sigma(\mathbf{W}_f \mathbf{x} + \mathbf{b}_f) \\ \tilde{\mathbf{x}} &= q(\mathbf{z}, \mathbf{W}_q) = \sigma(\mathbf{W}_q \mathbf{z} + \mathbf{b}_q) \end{aligned}$$

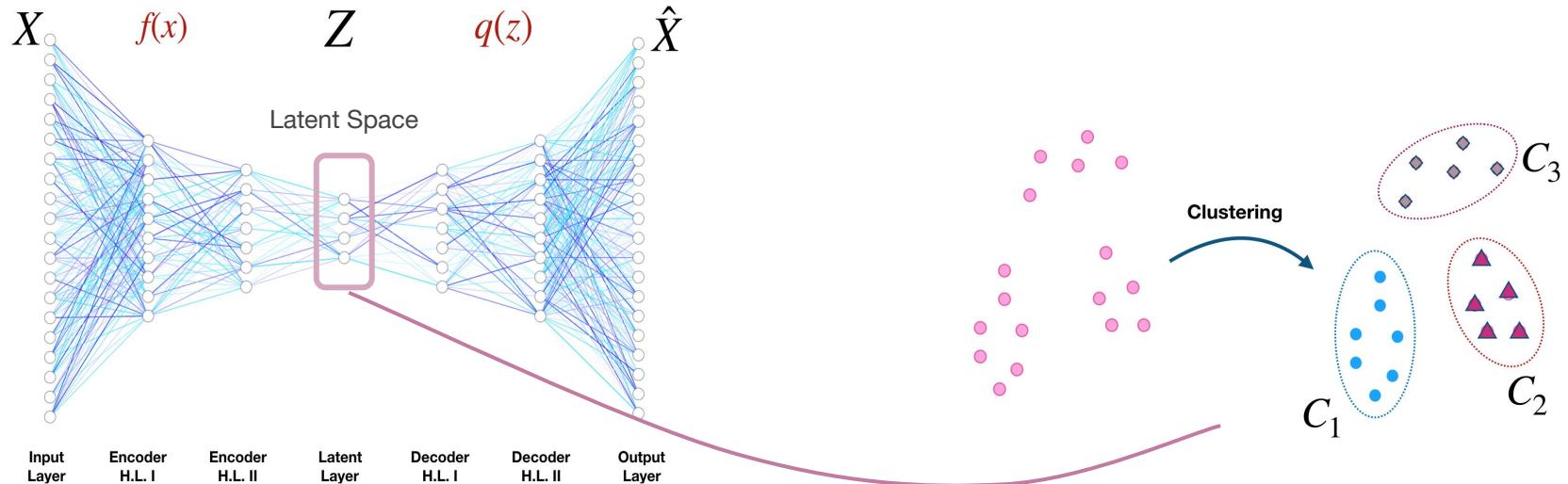
Mediante el uso de autoencoders es posible aprender representaciones de baja dimensionalidad de los datos (espacio latente). Estas representaciones ayudan a encontrar agrupaciones, aunque no sean interpretables en términos físicos.

[1] Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *science*, 313(5786), 504-507.

[2] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning Book. Cambridge: MIT press. (Chapter 14)

Figure: Dimension Reduction of tumor profiles using autoencoders (Palazzo M.)

# Autoencoder



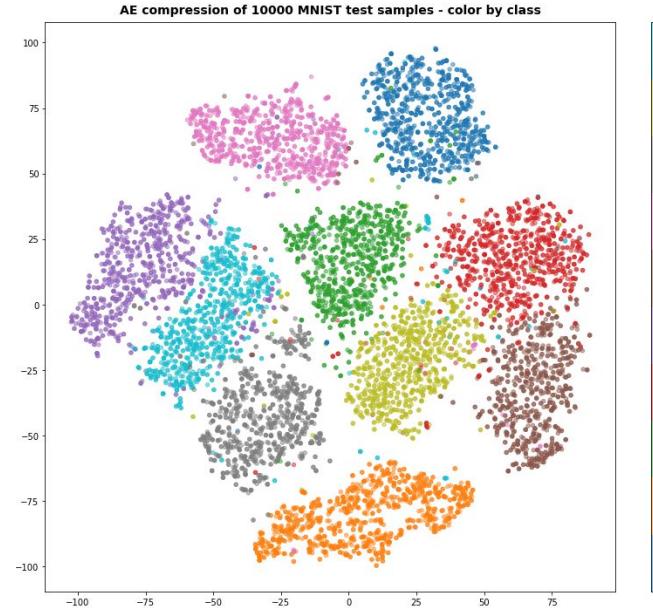
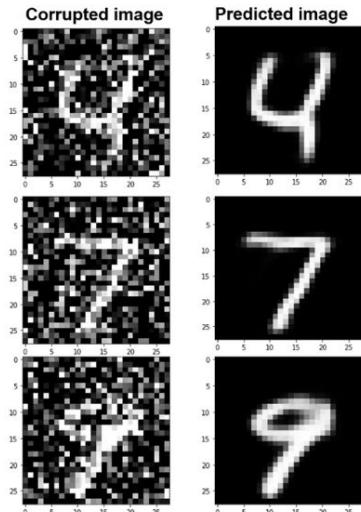
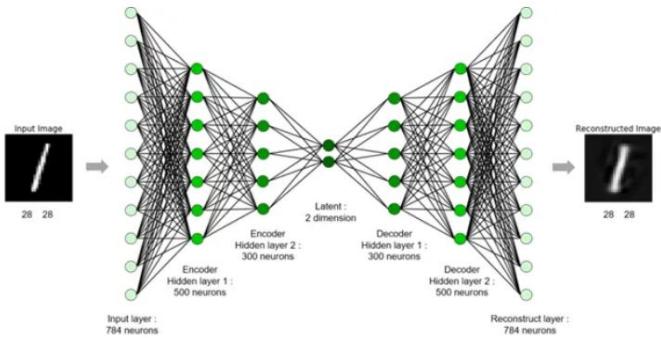
Mediante el uso de autoencoders es posible aprender representaciones de baja dimensionalidad de los datos (espacio latente). Estas representaciones ayudan a encontrar agrupaciones, aunque no sean interpretables en términos físicos.

[1] Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *science*, 313(5786), 504-507.

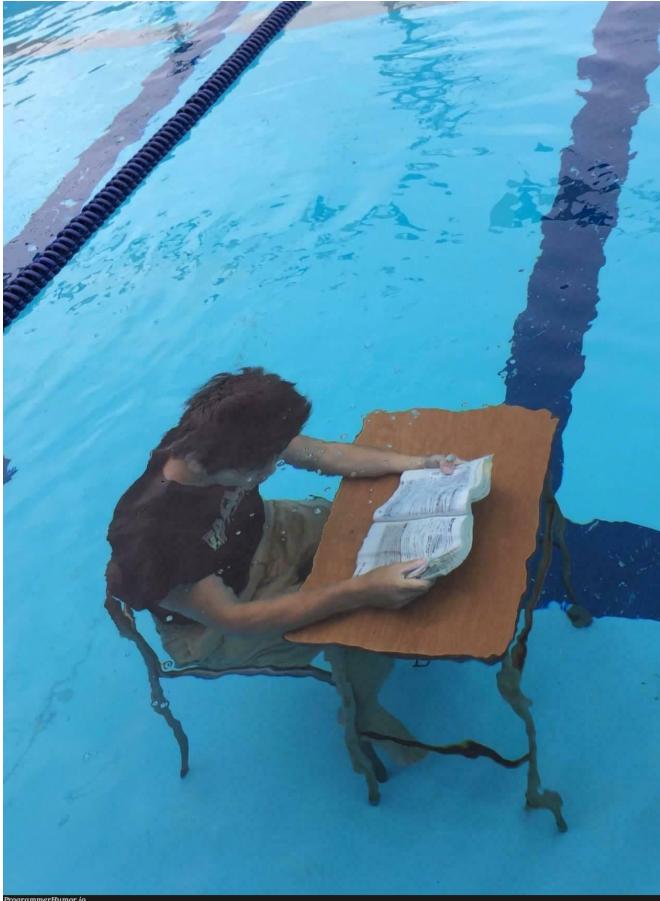
[2] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning Book. Cambridge: MIT press. (Chapter 14)

Figure: Dimension Reduction of tumor profiles using autoencoders (Palazzo M.)

# Autoencoders: Feature-Extraction



# Deep Learning



# Librerías para ANNs



Keras



TensorFlow



PyTorch