

# DATA LAB

## GUARDA AVANTI

**Big Data**, nuove competenze per nuove professioni.



"Anticipare la crescita con le nuove competenze sui Big Data - Edizione 3" Operazione Rif. PA 2021-16029/RER approvata con DGR n° 927 del 21 giugno 2021 e co-finanziata dal Fondo Sociale Europeo PO 2014-2020 Regione Emilia-Romagna

The background of the image is an underwater scene. A scuba diver in a black wetsuit and yellow fins is positioned in the lower-left foreground, facing right and holding a camera. To the right of the diver is a massive, dense school of small, silvery-blue fish swimming in a circular pattern. The water is a deep blue.

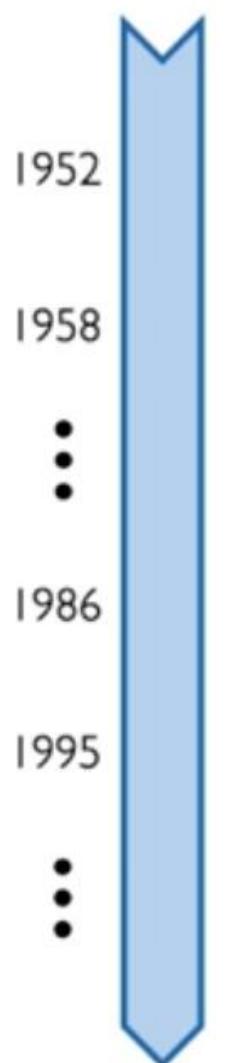
Deep Learning

DATA LAB.

# **Introduzione alle Reti Neurali**

# Why Now?

Neural Networks date back decades, so why the resurgence?



## 1. Big Data

- Larger Datasets
- Easier Collection & Storage



WIKIPEDIA  
The Free Encyclopedia



## 2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable



TensorFlow

## 3. Software

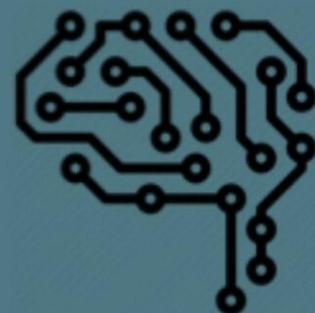
- Improved Techniques
- New Models
- Toolboxes

# ARTIFICIAL INTELLIGENCE

IS NOT NEW

## ARTIFICIAL INTELLIGENCE

Any technique which enables computers to mimic human behavior



1950's

1960's

1970's

1980's

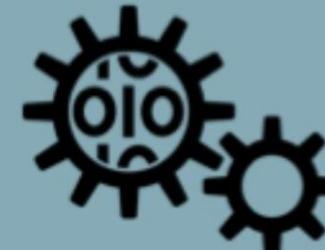
1990's

2000's

2010s

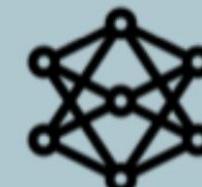
## MACHINE LEARNING

AI techniques that give computers the ability to learn without being explicitly programmed to do so

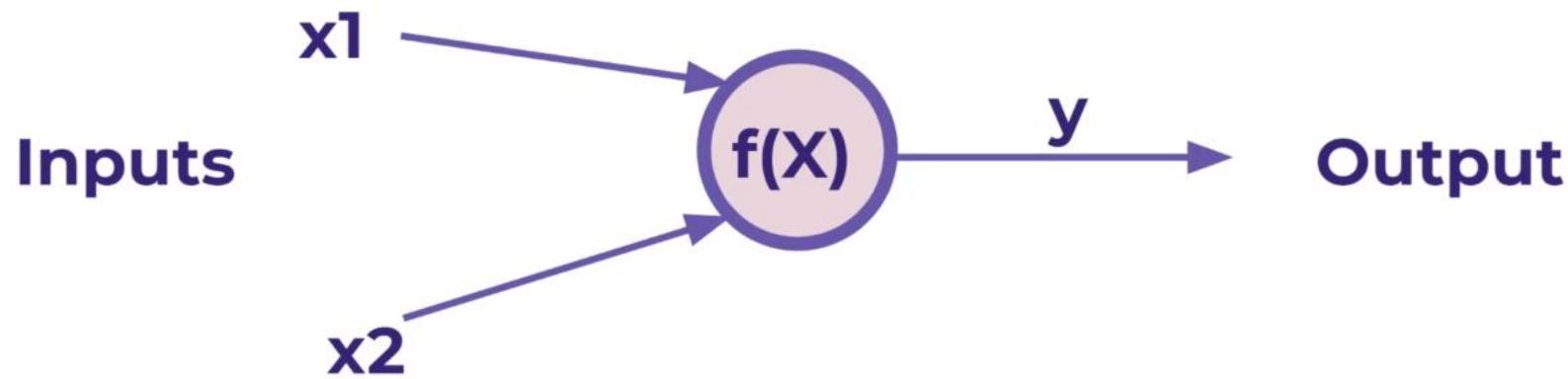
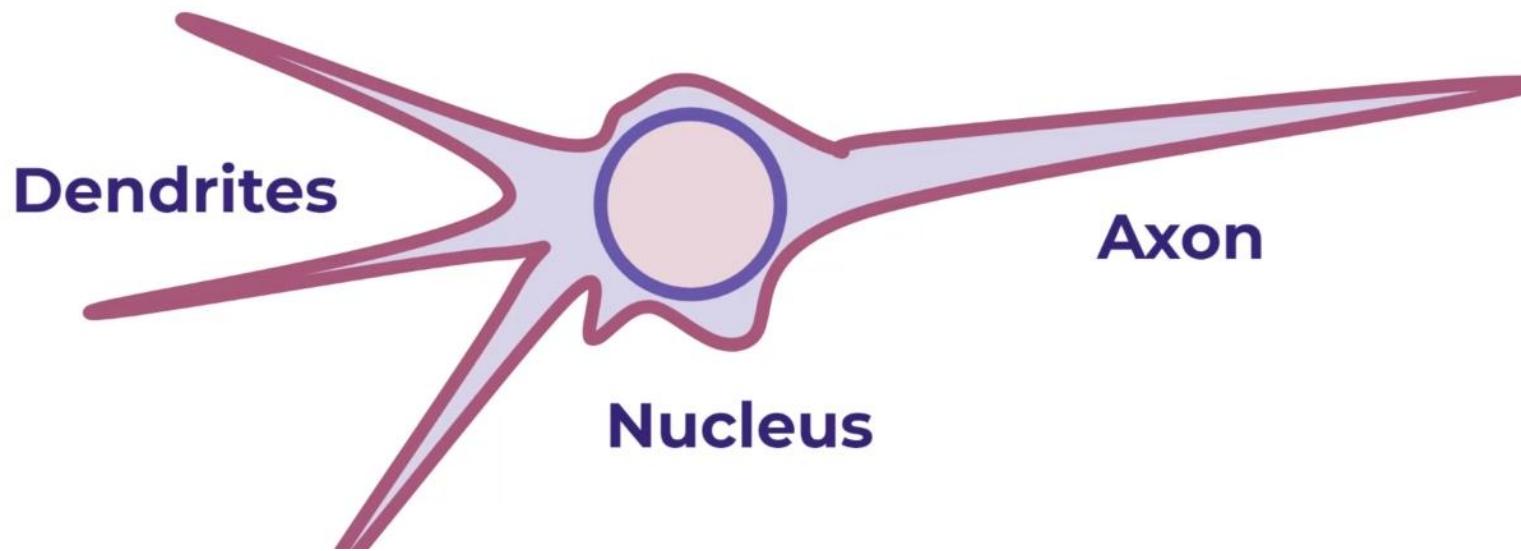


## DEEP LEARNING

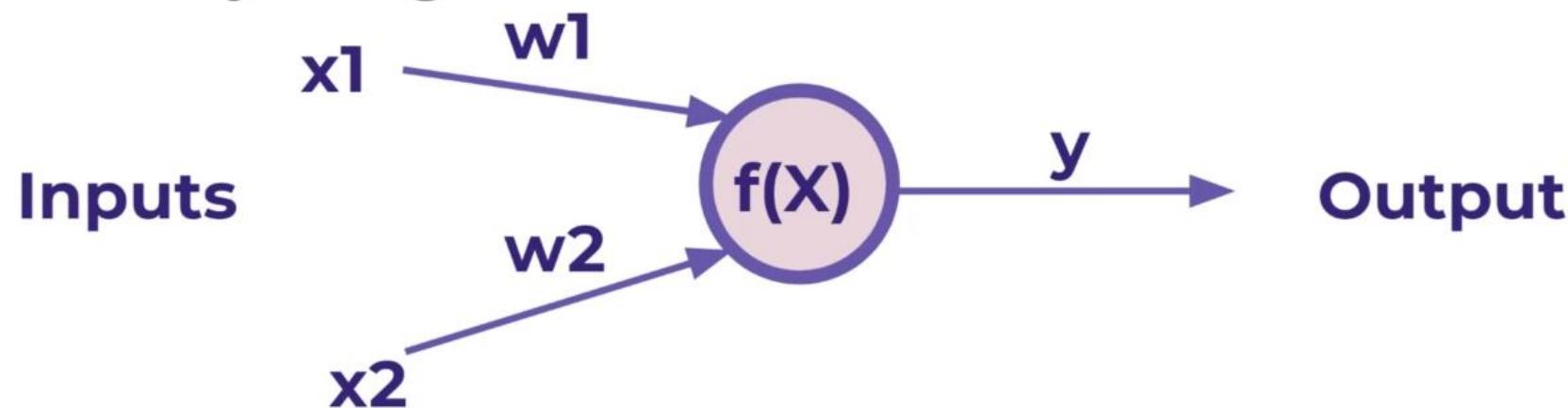
A subset of ML which make the computation of multi-layer neural networks feasible



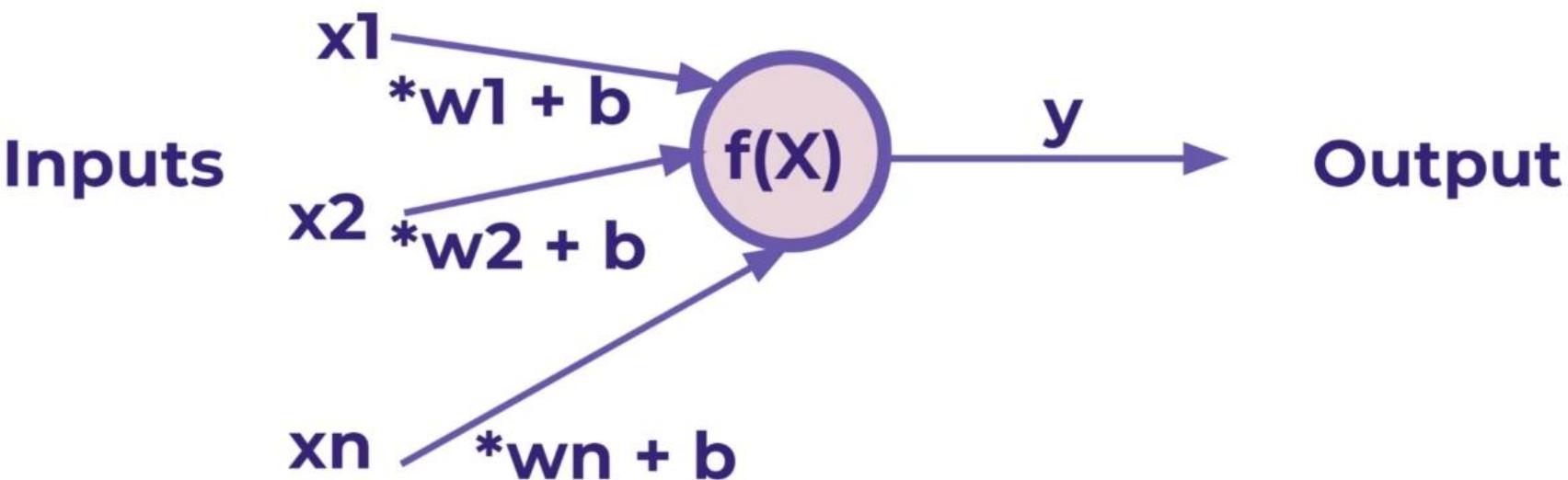
# PERCEPTRON



- But what if an  $\mathbf{x}$  is zero?  $\mathbf{w}$  won't change anything!



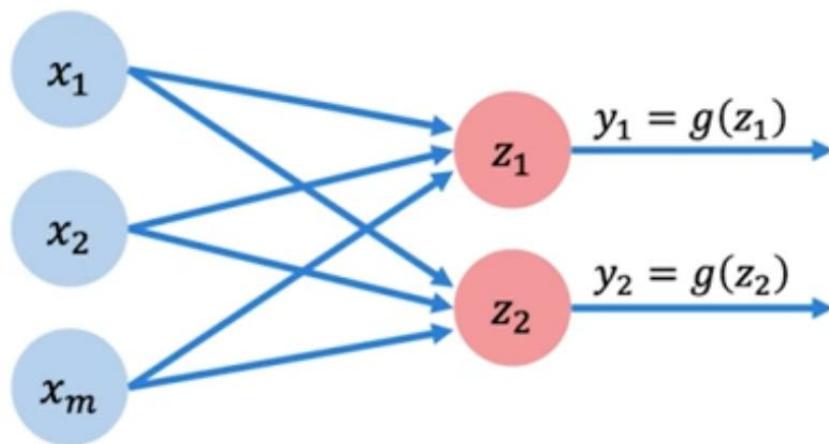
- We can expand this to a generalization:



$$\hat{y} = \sum_{i=1}^n x_i w_i + b_i$$

# Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called **Dense** layers



$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

# Dense layer from scratch

```
class MyDenseLayer(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim):
        super(MyDenseLayer, self).__init__()

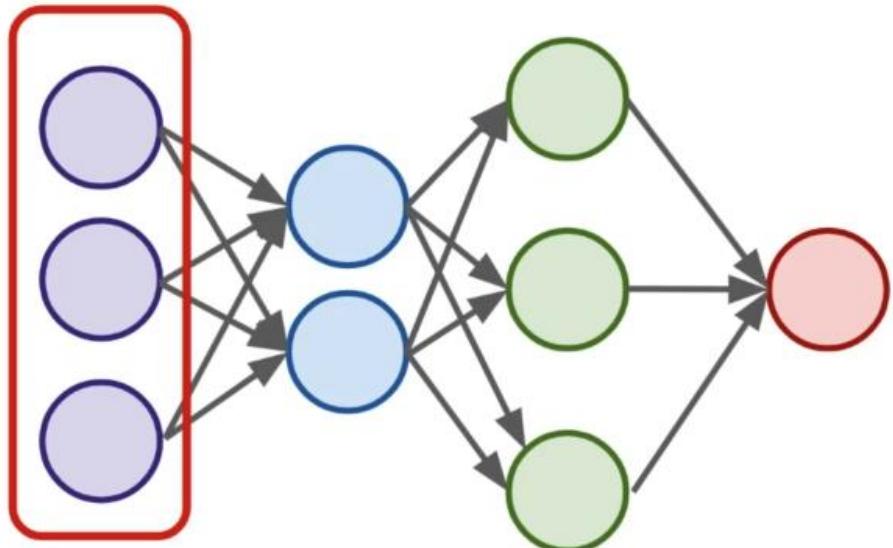
        # Initialize weights and bias
        self.W = self.add_weight([input_dim, output_dim])
        self.b = self.add_weight([1, output_dim])

    def call(self, inputs):
        # Forward propagate the inputs
        z = tf.matmul(inputs, self.W) + self.b

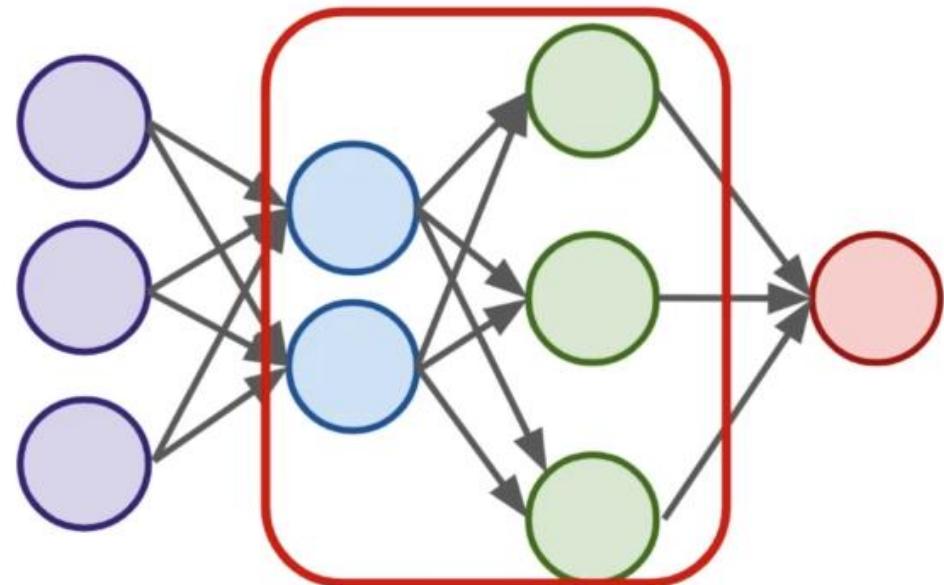
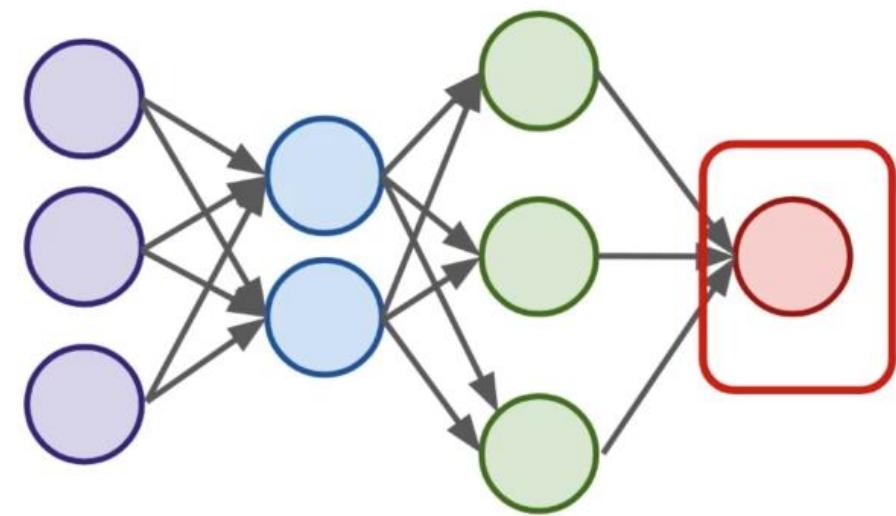
        # Feed through a non-linear activation
        output = tf.math.sigmoid(z)

    return output
```

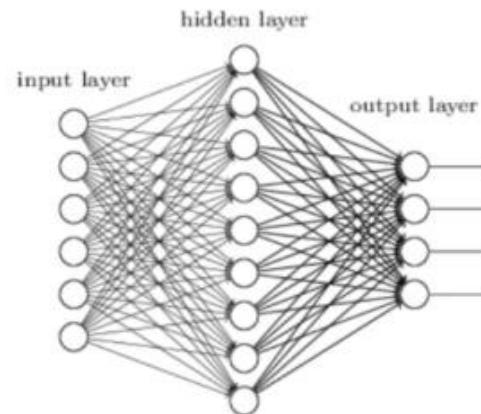
# input layer



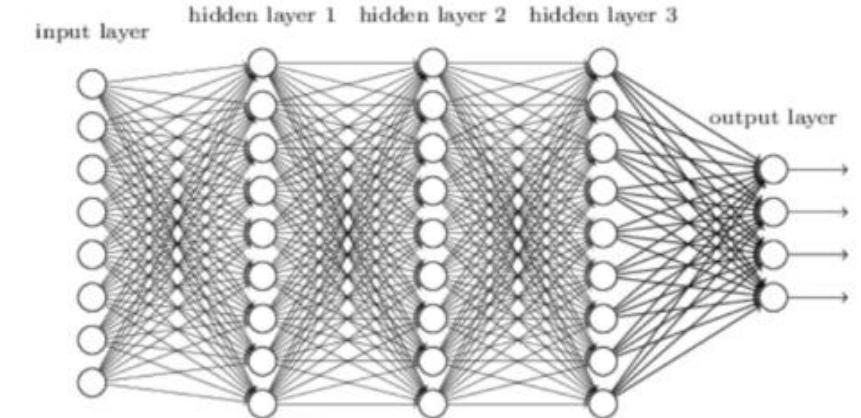
# output layer



"Non-deep" feedforward  
neural network



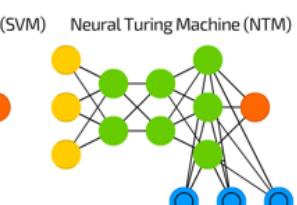
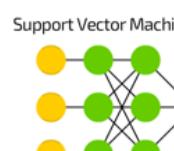
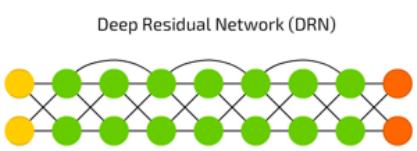
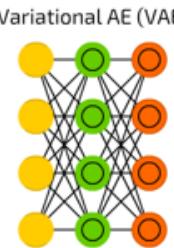
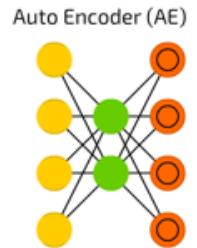
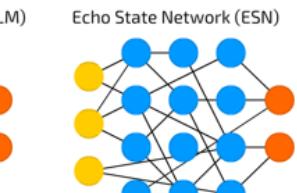
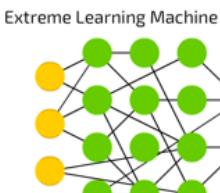
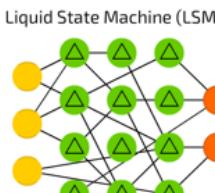
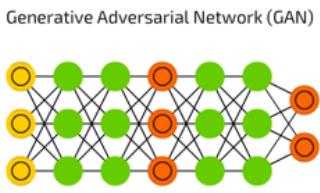
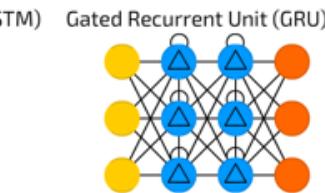
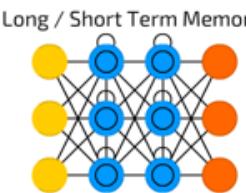
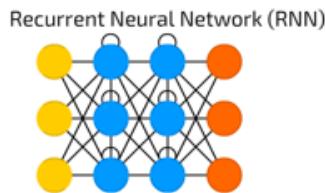
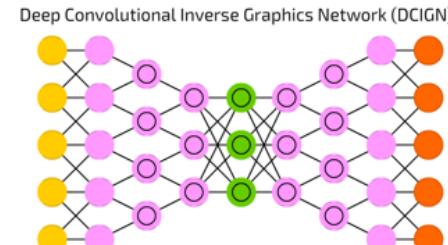
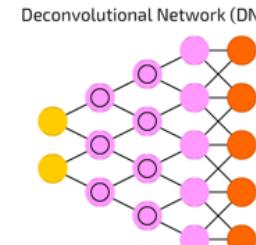
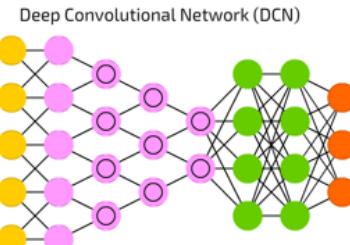
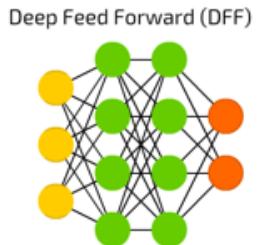
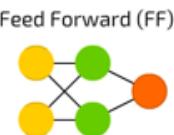
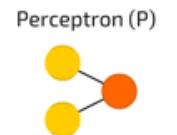
Deep neural network



*A mostly complete chart of*  
**Neural Networks**

©2016 Fjodor van Veen - asimovinstitute.org

- Backfed Input Cell
- Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- △ Different Memory Cell
- Kernel
- Convolution or Pool



Markov Chain (MC)



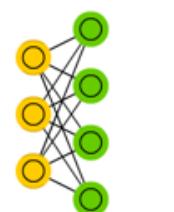
Hopfield Network (HN)



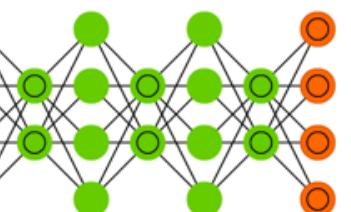
Boltzmann Machine (BM)



Restricted BM (RBM)



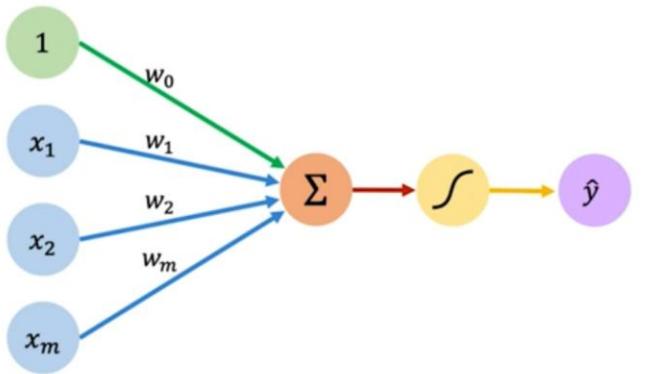
Deep Belief Network (DBN)



- Terminology:
  - Input Layer: First layer that directly accepts real data values
  - Hidden Layer: Any layer between input and output layers
  - Output Layer: The final estimate of the output.
- What is incredible about the neural network framework is that it can be used to approximate any function.
- Zhou Lu and later on Boris Hanin proved mathematically that Neural Networks can approximate any convex continuous function.

- Previously in our simple model we saw that the perceptron itself contained a very simple summation function  $f(x)$ .
  - For most use cases however that won't be useful, we'll want to be able to set constraints to our output values, especially in classification tasks.
- In a classification tasks, it would be useful to have all outputs fall between 0 and 1.
  - These values can then present probability assignments for each class.
  - In the next lecture, we'll explore how to use **activation functions** to set boundaries to output values from the neuron.

# The Perceptron: Forward Propagation



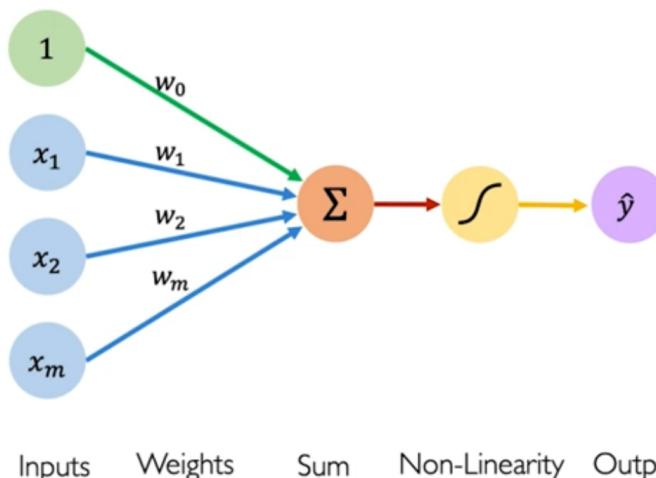
Inputs    Weights    Sum    Non-Linearity    Output

Output  
 $\hat{y} = g \left( w_0 + \sum_{i=1}^m x_i w_i \right)$

Linear combination of inputs  
Bias

Non-linear activation function

# The Perceptron: Forward Propagation



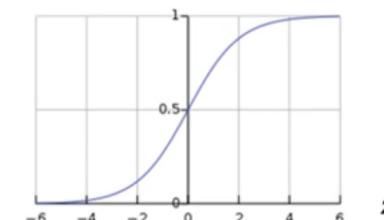
Inputs    Weights    Sum    Non-Linearity    Output

## Activation Functions

$$\hat{y} = g(w_0 + X^T W)$$

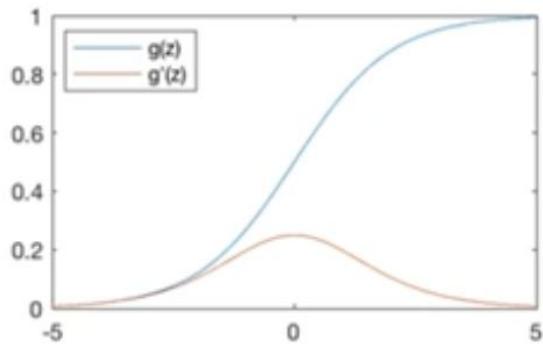
- Example: sigmoid function

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



# Common Activation Functions

Sigmoid Function

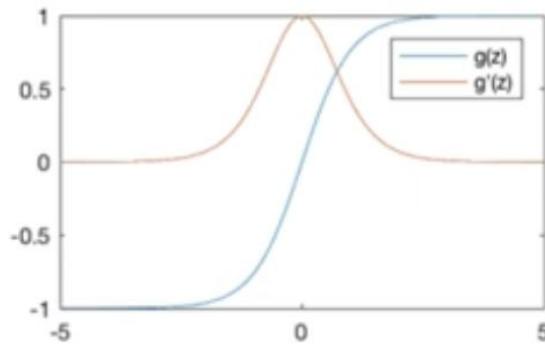


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

 `tf.math.sigmoid(z)`

Hyperbolic Tangent

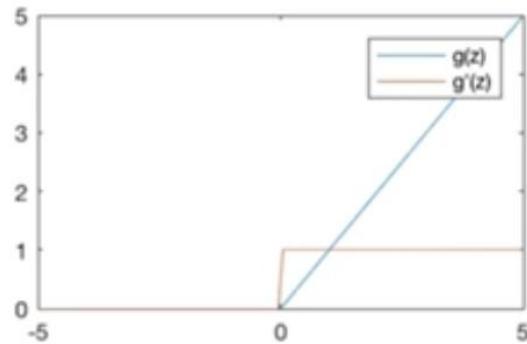


$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

 `tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



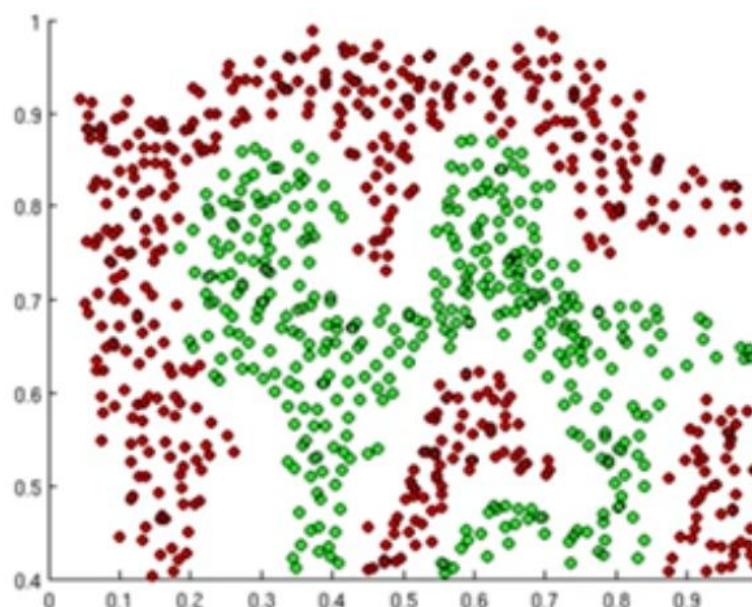
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

 `tf.nn.relu(z)`

# Importance of Activation Functions

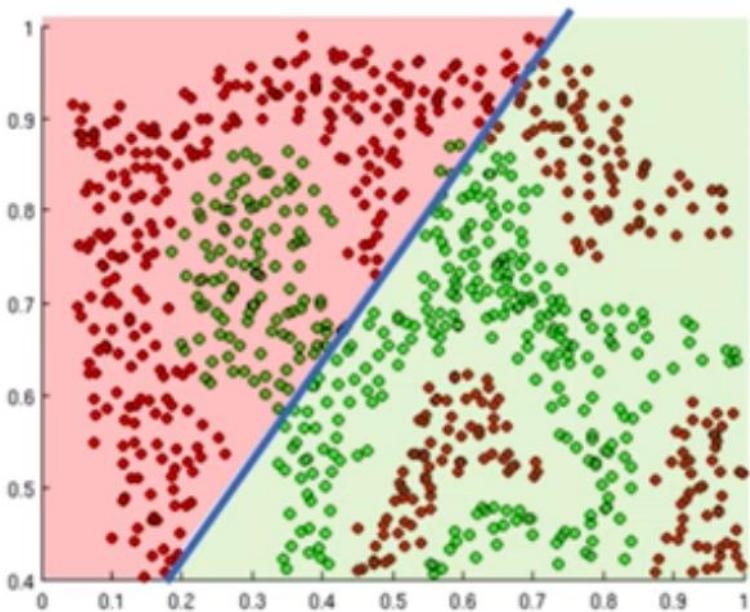
The purpose of activation functions is to *introduce non-linearities* into the network



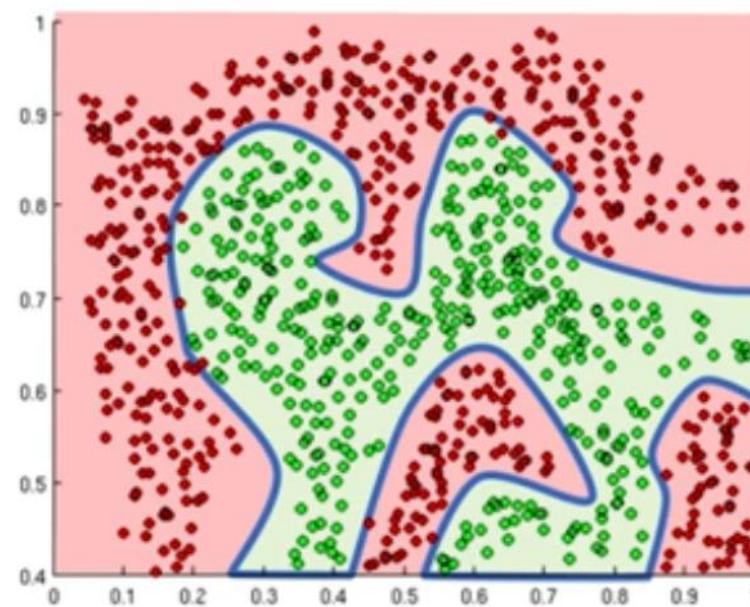
What if we wanted to build a neural network to distinguish green vs red points?

# Importance of Activation Functions

The purpose of activation functions is to **introduce non-linearities** into the network

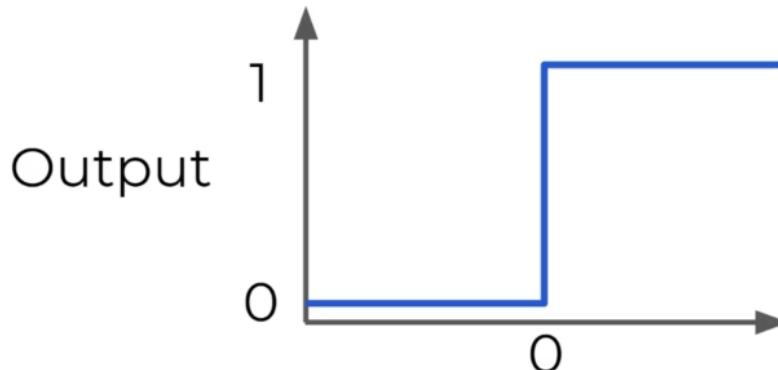


Linear activation functions produce linear decisions no matter the network size



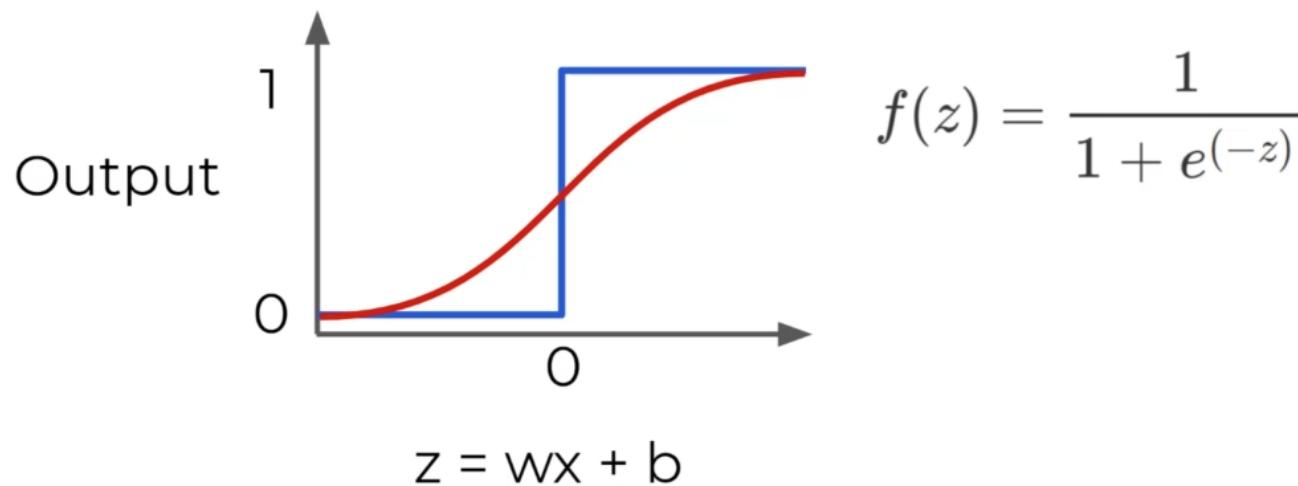
Non-linearities allow us to approximate arbitrarily complex functions

- The most simple networks rely on a basic **step function** that outputs 0 or 1.

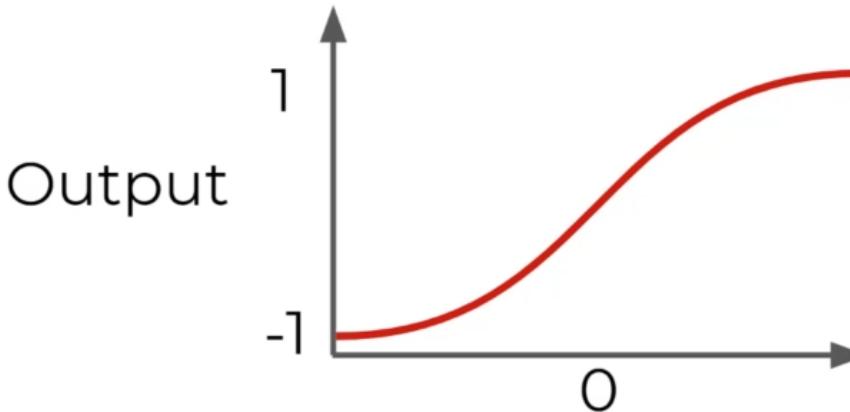


$$z = wx + b$$

This still works for classification, and will be more sensitive to small changes.



- Hyperbolic Tangent:  $\tanh(z)$
- Outputs between -1 and 1 instead of 0 to 1

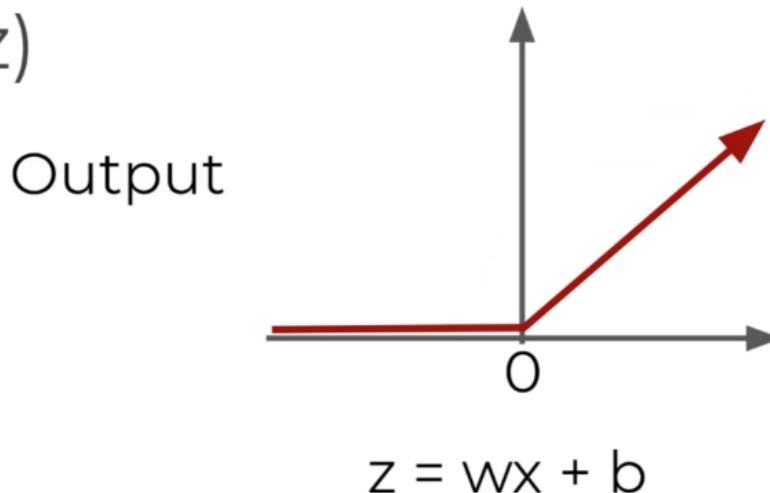


$$\cosh x = \frac{e^x + e^{-x}}{2}$$

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

$$\tanh x = \frac{\sinh x}{\cosh x}$$

- Rectified Linear Unit (ReLU): This is actually a relatively simple function:  
 $\max(0, z)$



# Deep Learning

## Activation Functions: Dance Moves

Sigmoid



$$y = \frac{1}{1+e^{-x}}$$

Tanh



$$y = \tanh(x)$$

Step Function



$$y = \begin{cases} 0, & x < n \\ 1, & x \geq n \end{cases}$$

Softplus



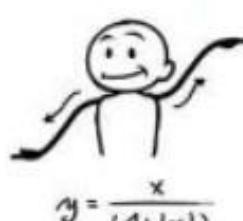
$$y = \ln(1+e^x)$$

ReLU



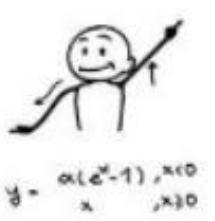
$$y = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

Softsign



$$y = \frac{x}{(1+|x|)}$$

ELU



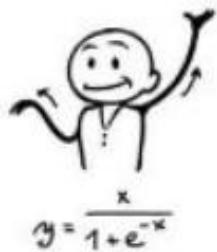
$$y = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$$

Log of Sigmoid



$$y = \ln\left(\frac{1}{1+e^{-x}}\right)$$

Swish



$$y = \frac{x}{1+e^{-x}}$$

Sinc



$$y = \frac{\sin(x)}{x}$$

Leaky ReLU



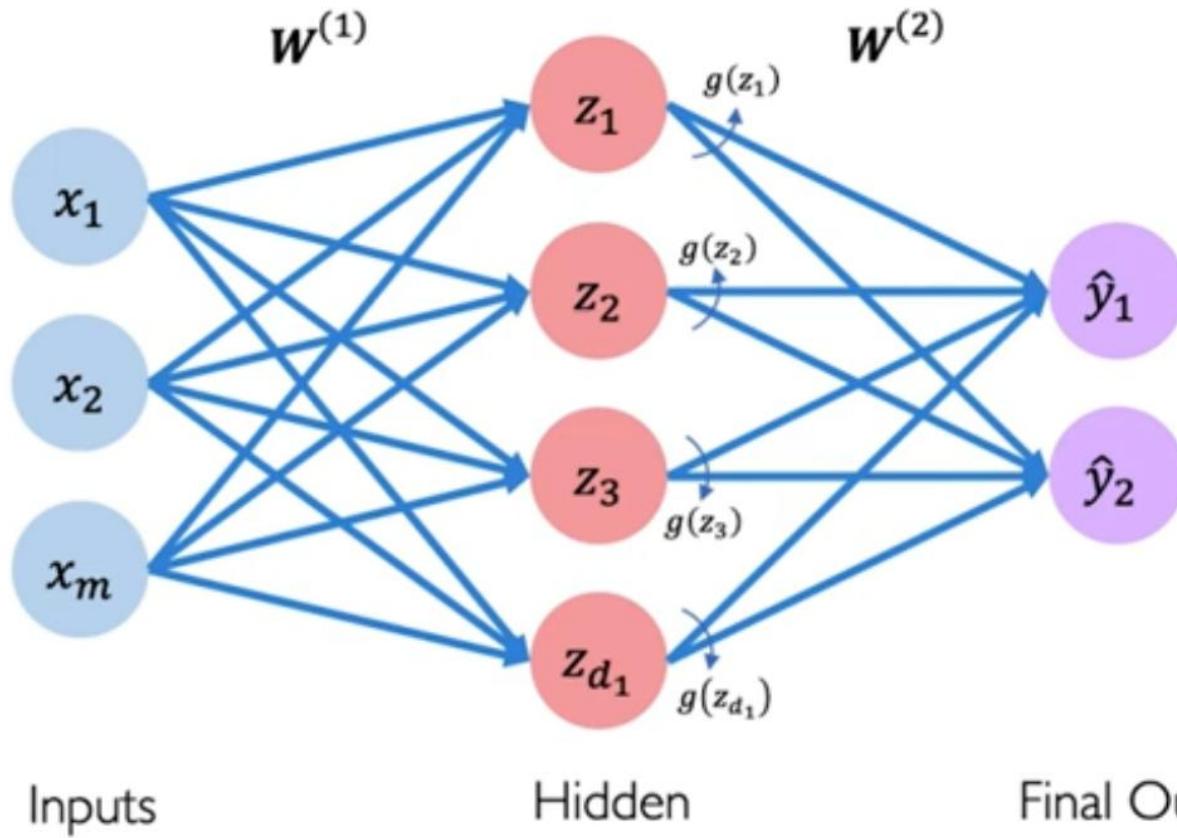
$$y = \max(0.01x, x)$$

Mish



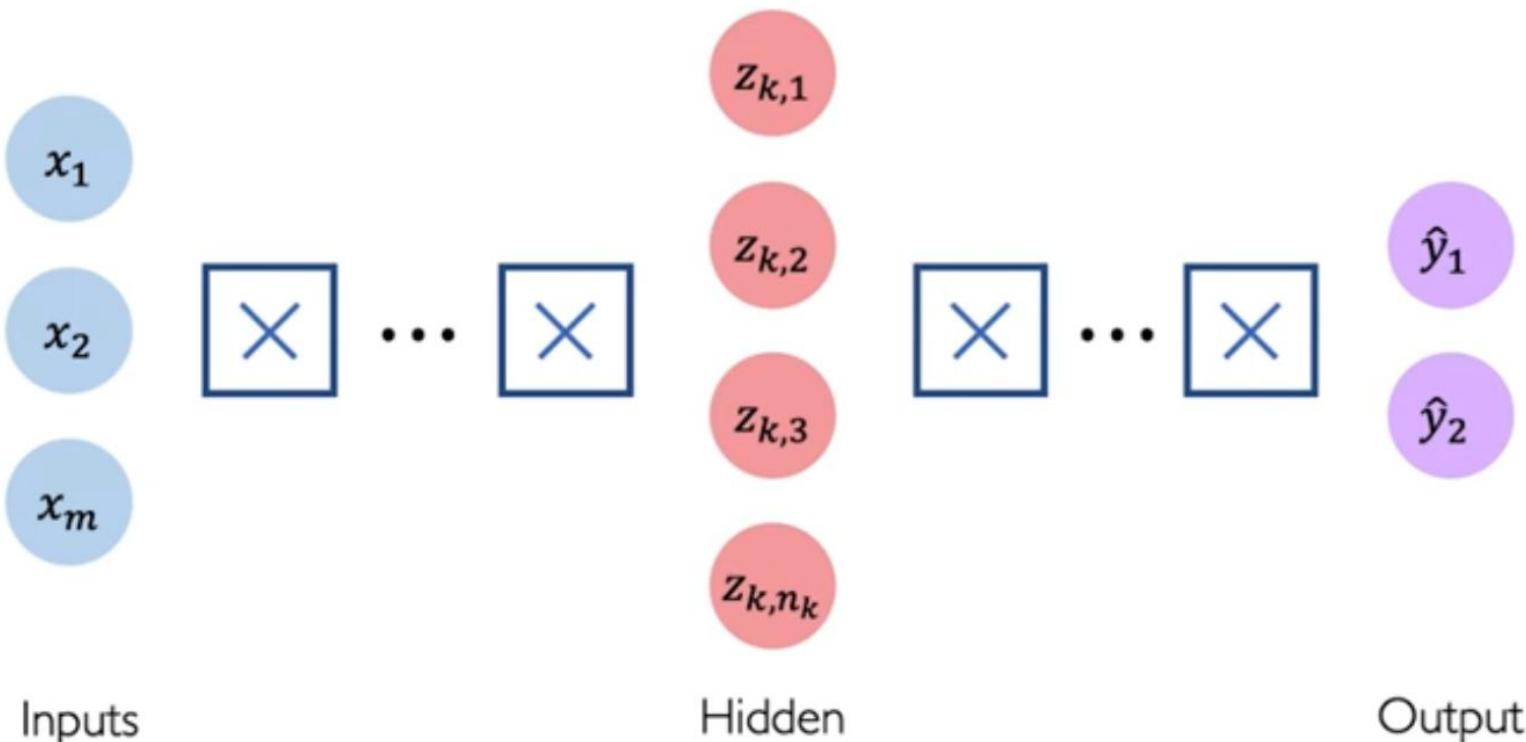
$$y = x(\tanh(\text{softplus}(x)))$$

# Single Layer Neural Network



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)} \quad \hat{y}_i = g \left( w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)} \right)$$

# Deep Neural Network

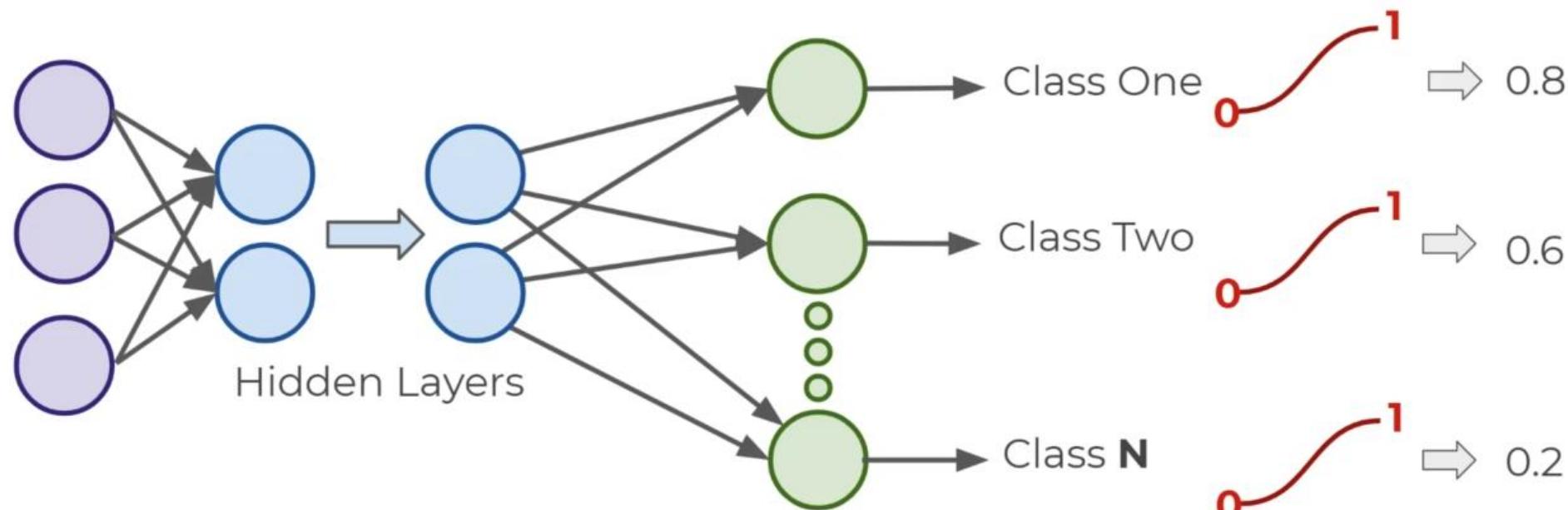


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$

 import tensorflow as tf  
  
model = tf.keras.Sequential([  
 tf.keras.layers.Dense(n<sub>1</sub>),  
 tf.keras.layers.Dense(n<sub>2</sub>),  
 :  
 tf.keras.layers.Dense(2)  
])

# Multiclass Classification

- Sigmoid Function for Non-Exclusive Classes



- Mutually Exclusive Classes
  - But what do we do when each data point can only have a single class assigned to it?
  - We can use the **softmax function** for this!

Softmax Function

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad \text{for } i = 1, \dots, K$$

- Mutually Exclusive Classes
  - The range will be 0 to 1, and **the sum of all the probabilities will be equal to one.**
  - The model returns the probabilities of each class and the target class chosen will have the highest probability.

### Mutually Exclusive Classes

- The probabilities for each class all sum up to 1. We choose the highest probability as our assignment.
  - [Red , Green , Blue]
  - [ 0.1 , 0.6 , 0.3 ]

# LA DERIVATA

$$f(x) = \frac{1}{4}x^2 - 2x + 5$$

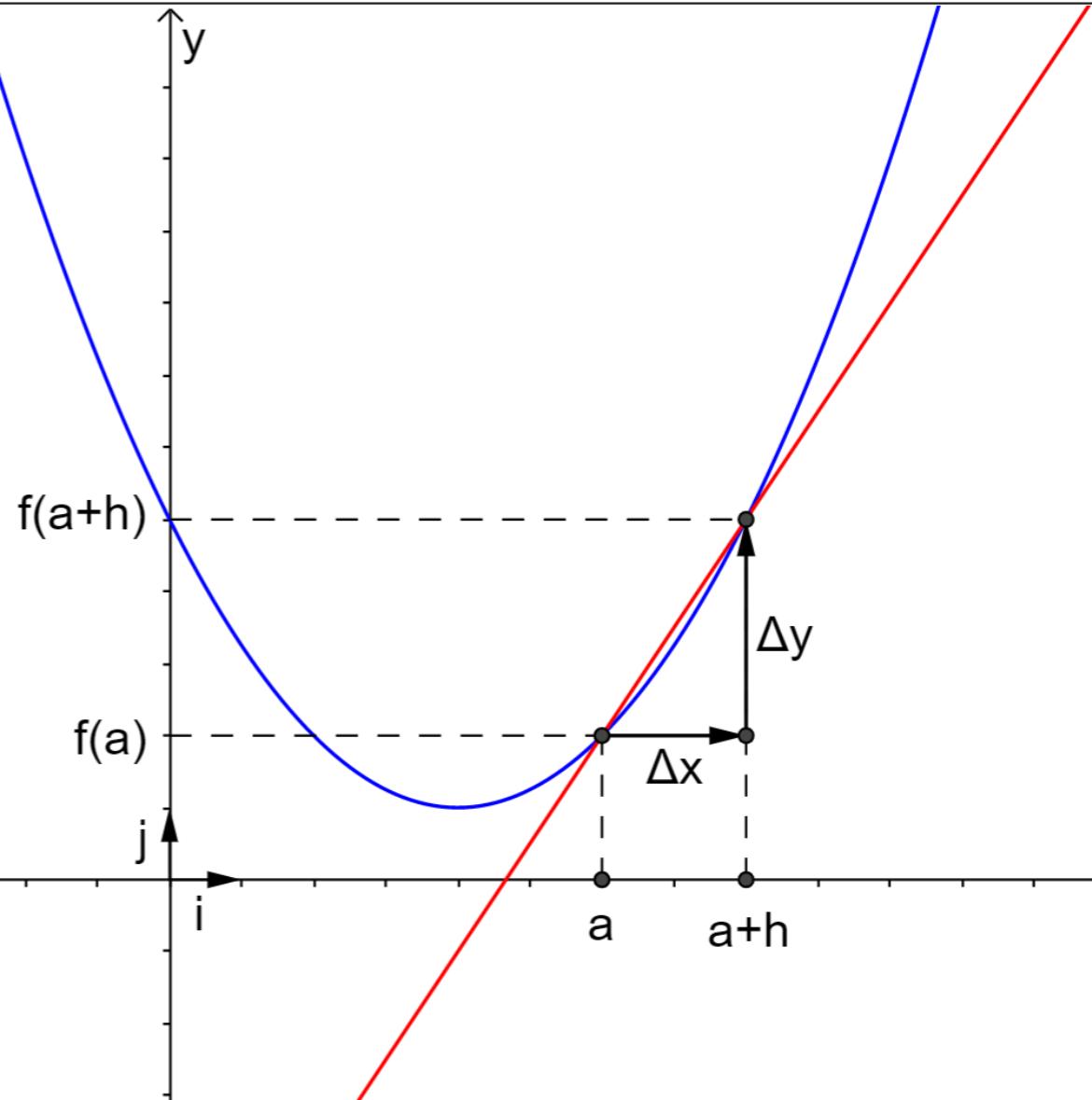
$$\Delta x = h = 2$$

$$\Delta y = f(a+h) - f(a) = 3$$

$$\frac{\Delta y}{\Delta x} = 1.5$$

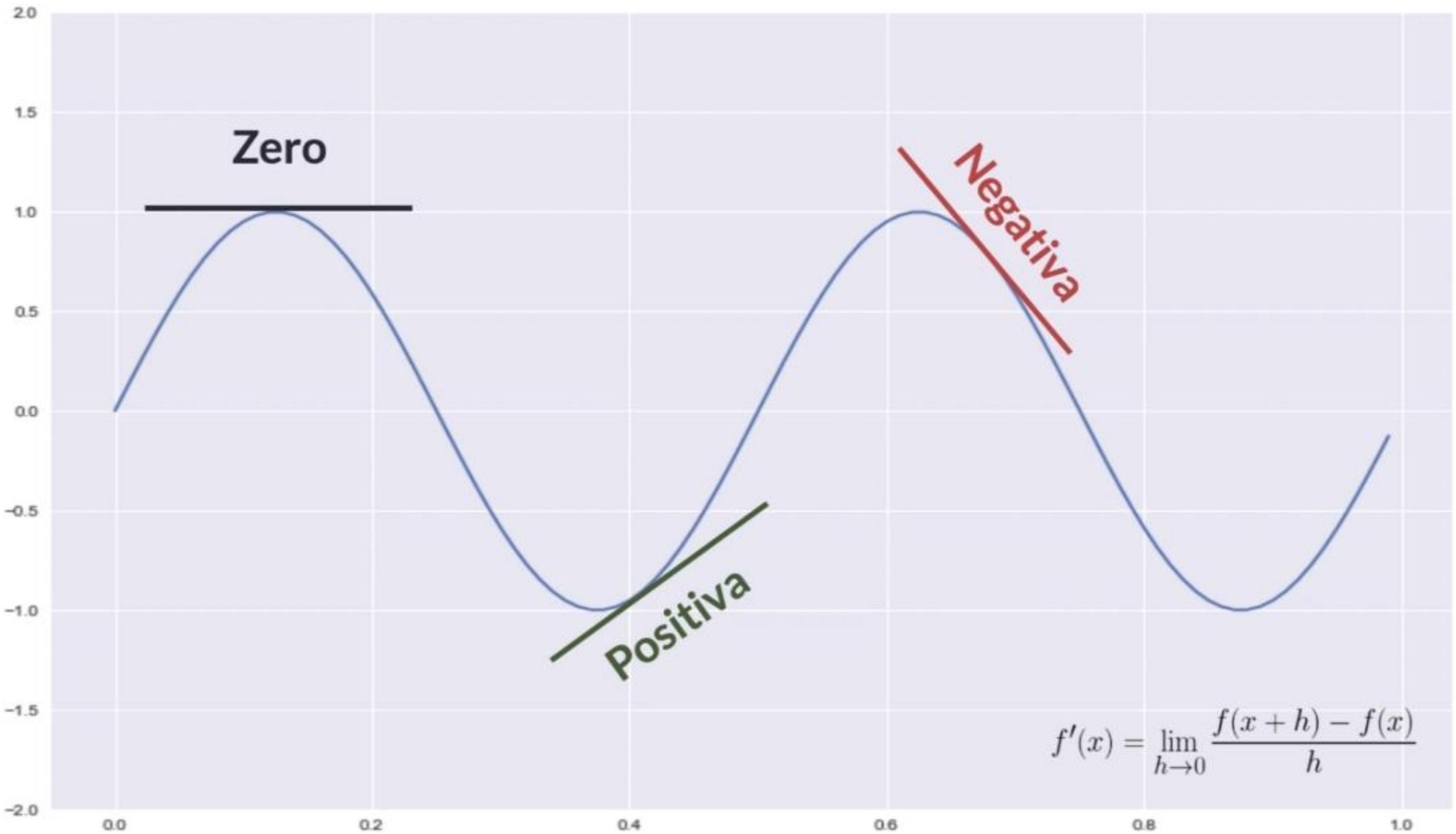
$$f'(a) = \lim_{h \rightarrow 0} \frac{\Delta y}{\Delta x} = 1$$

□ retta tangente

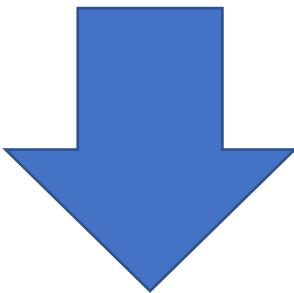


# Derivate fondamentali

	<b>Funzione</b>	<b>Derivata</b>
costante	$f(x) = n$	$f'(x) = 0$
lineare	$f(x) = x$	$f'(x) = 1$
potenza	$f(x) = x^n$	$f'(x) = nx^{n-1}$
esponenziale	$f(x) = e^x$	$f'(x) = e^x$
logaritmica	$f(x) = \ln x$	$f'(x) = \frac{1}{x}$



How do we evaluate the performance of a Neural network?



## COST FUNCTION

Cost Function/Loss Function quantifies the error between predicted values and expected values and presents it in the form of a single real number

- The cost function (often referred to as a loss function) must be an average so it can output a single value.
- We can keep track of our loss/cost during training to monitor network performance.

**punishes** large errors!

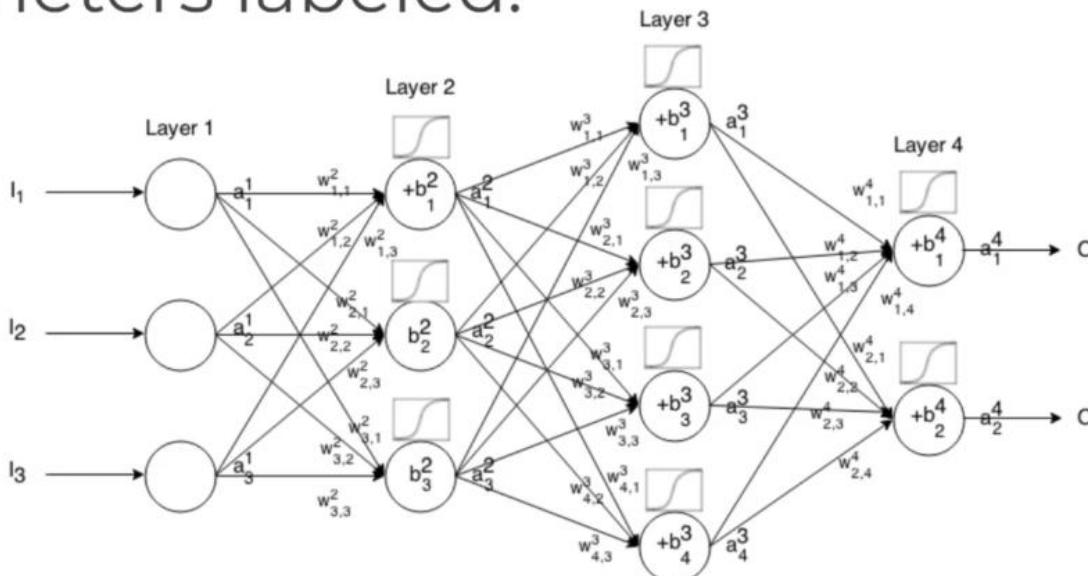
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

- **W** is our neural network's weights, **B** is our neural network's biases, **S<sup>r</sup>** is the input of a single training sample, and **E<sup>r</sup>** is the desired output of that training sample.

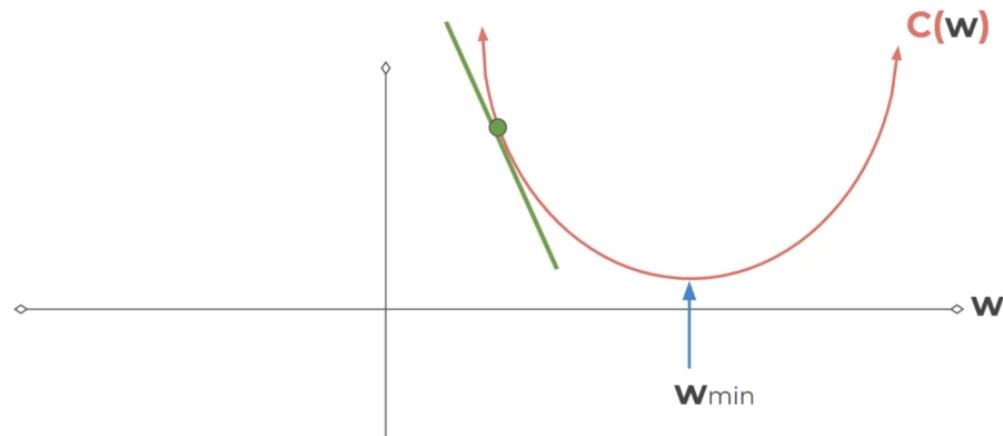
$$C(W, B, S^r, E^r)$$

- For simplicity, let's imagine we only had one weight in our cost function  $\mathbf{w}$ .
- We want to **minimize** our loss/cost (overall error).
- Which means we need to figure out what value of  $\mathbf{w}$  results in the minimum of  $\mathbf{C}(\mathbf{w})$

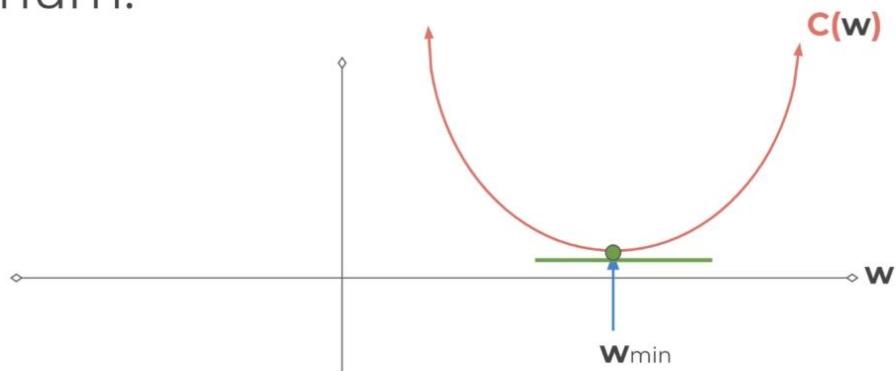
Here is a small network with all its parameters labeled:



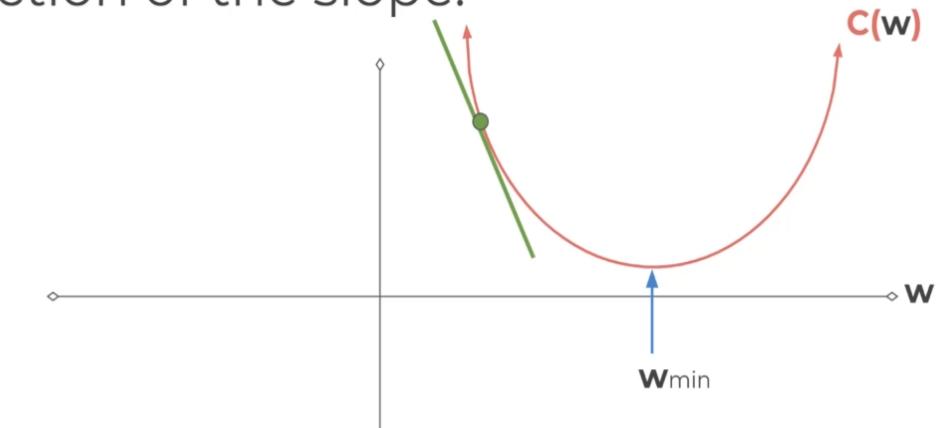
We can calculate the slope at a point



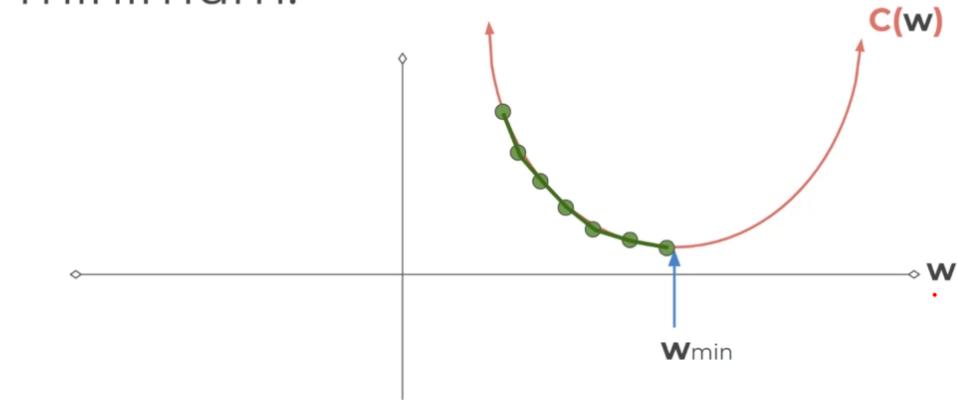
Until we converge to zero, indicating a minimum.



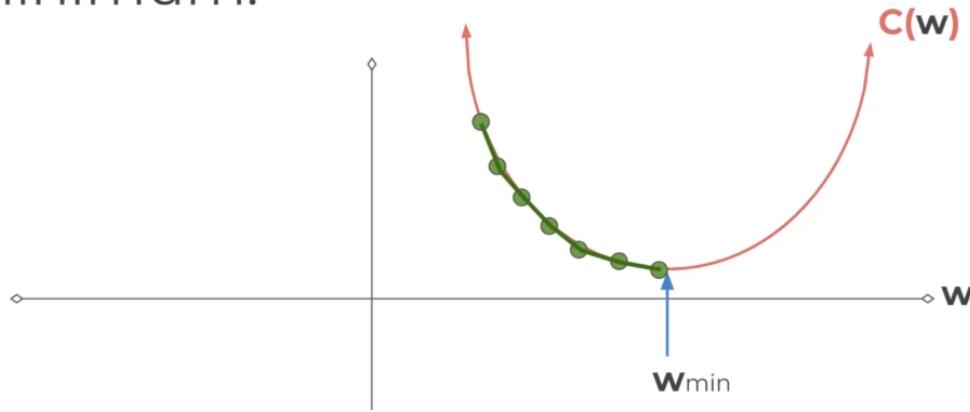
Then we move in the downward direction of the slope.



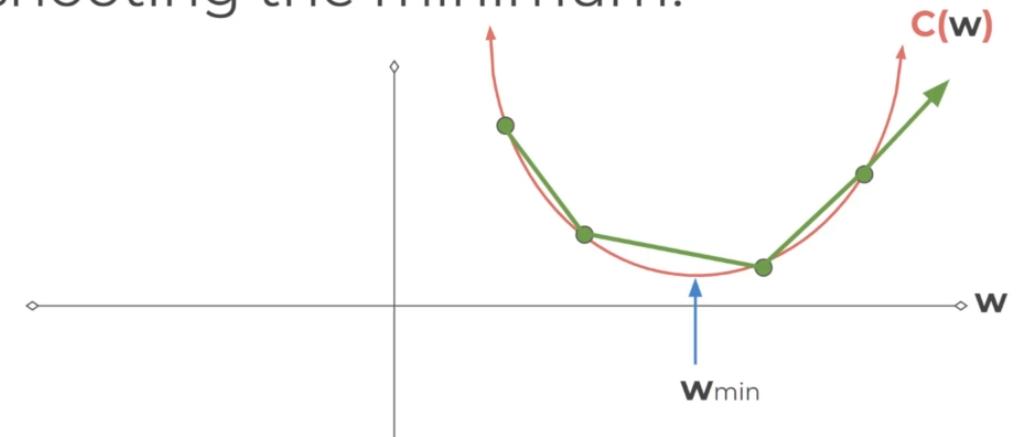
Smaller steps sizes take longer to find the minimum.



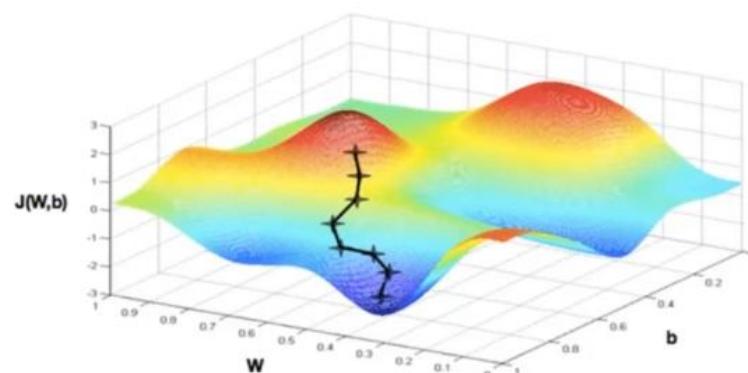
Smaller steps sizes take longer to find the minimum.



Larger steps are faster, but we risk overshooting the minimum!



Realistically we're calculating this descent in an n-dimensional space for all our weights.



## Derivate parziali

$$\frac{df}{dx}, \quad \frac{df}{dy}, \quad \frac{df}{dz}$$

Si ottengono calcolando la derivata rispetto ad una variabile per volta e trattando le altre come costanti

## Gradiente di una funzione

$$\nabla f(x, y, z) = \left[ \frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz} \right]$$

E' un vettore composto dalle derivate parziali della funzione

## Come si calcola il gradiente della funzione di costo ?

**Regressione** Somma dei quadrati residui

$$J(w, b) = \frac{1}{2} \sum_{i=1}^N (y^{(i)} - (w^{(i)}x^{(i)} + b))^2$$

**Classificazione** Log Loss

$$J(w, b) = - \sum_{i=1}^N (y^{(i)} \log(\phi(w^{(i)}x^{(i)} + b)) - (1 - y) \log(1 - \phi(w^{(i)}x^{(i)} + b)))$$

Differenziando le equazioni otteniamo le stesse derivate parziali

Derivata della funzione di costo rispetto al peso

$$\frac{dJ}{dw} = \sum_{i=1}^N (\phi(w^{(i)}x^{(i)} + b) - y^{(i)})x^{(i)}$$

Derivata della funzione di costo rispetto al bias

$$\frac{dJ}{db} = \sum_{i=1}^N (\phi(w^{(i)}x^{(i)} + b) - y^{(i)})$$

... insieme costituiscono il gradiente della funzione di costo

$$\nabla J(w, b) = \left[ \frac{dJ}{dw}, \frac{dJ}{db} \right]$$

```
def gradient_descent(X,y,alpha=0.001,epochs=100):  
  
    w = rand(X.shape[1],1)  
    b = 0  
  
    for _ in range(epochs):  
        z = dot(X,w)+b  
        a = activation(z)  
        dw = X.T.dot(a-y)  
        db = sum(a-y)  
        w = w - alpha*dw  
        b = b - alpha*db  
  
    return w,b
```

```
def stochastic_GD(X,y,epochs=100):  
  
    w = rand(X.shape[1],1)  
    b = rand(1,1)  
  
    for _ in range(epochs):  
        for x_i,y_i in zip(X,y):  
            z = dot(x_i,w)+b  
            a = activation(z)  
            dw = (a-y_i)*x_i  
            db = a-y_i  
            w = w - alpha*dw  
            b = b - alpha*db  
  
    return w,b
```



# Gradient Descent

## Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights,  $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

```
import tensorflow as tf

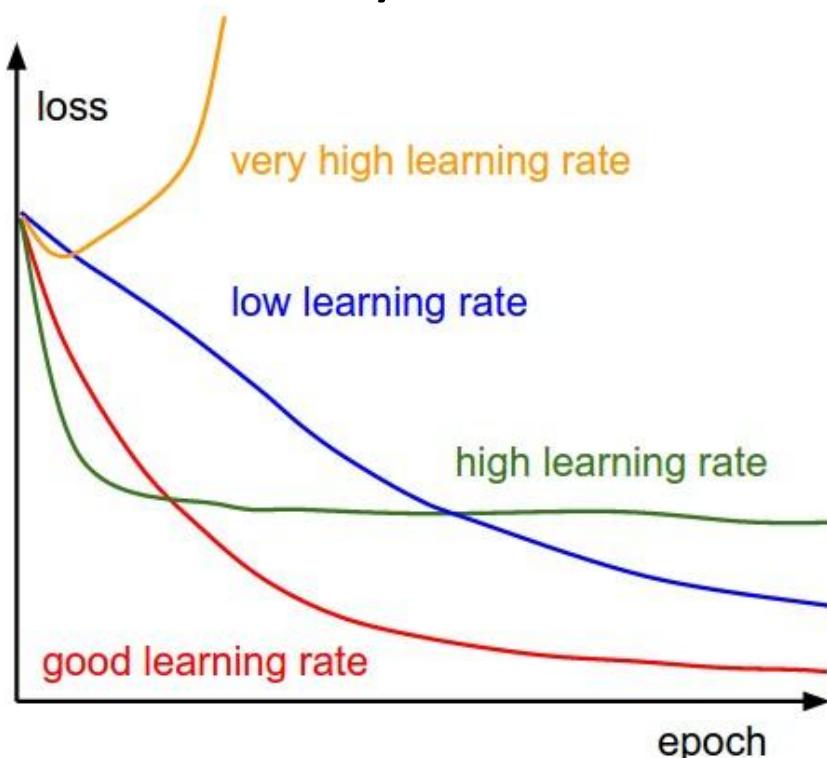
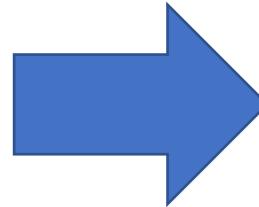
weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

# EPOCHS/ LEARNING RATE

One Epochs is when an ENTIRE dataset is passed forward and backward through the neaural network only once



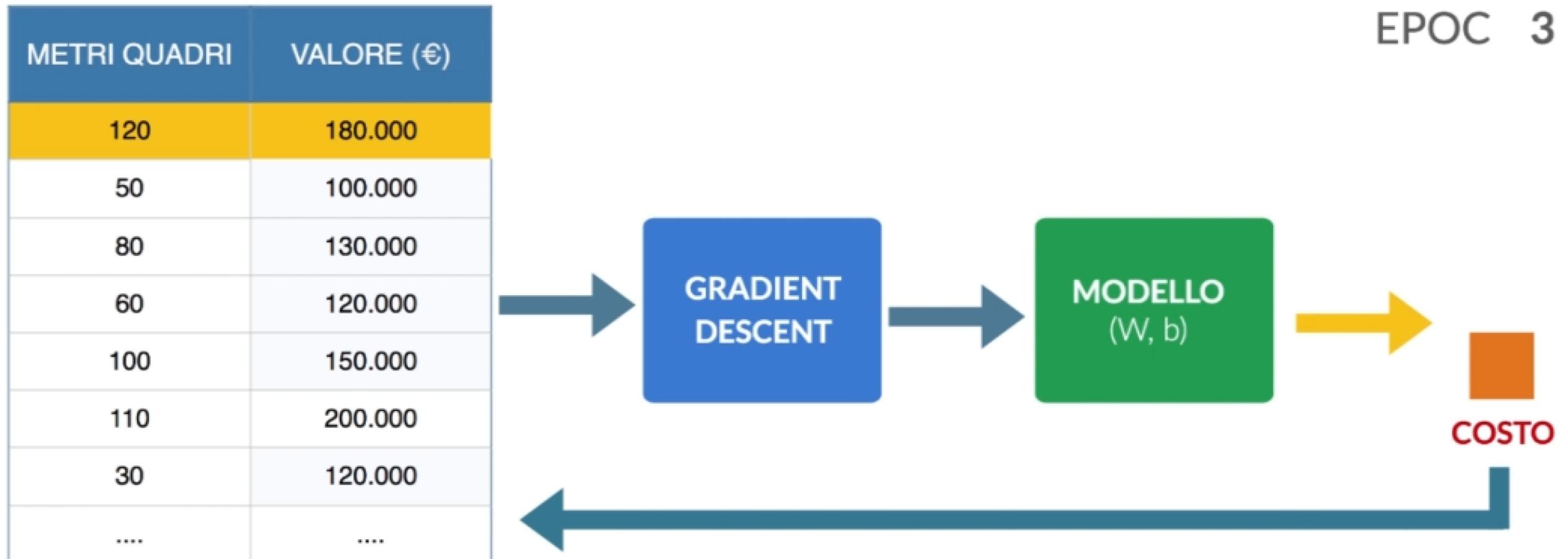
## Why Multiple Epochs?

Passing the entire dataset through a neural network is not enough.

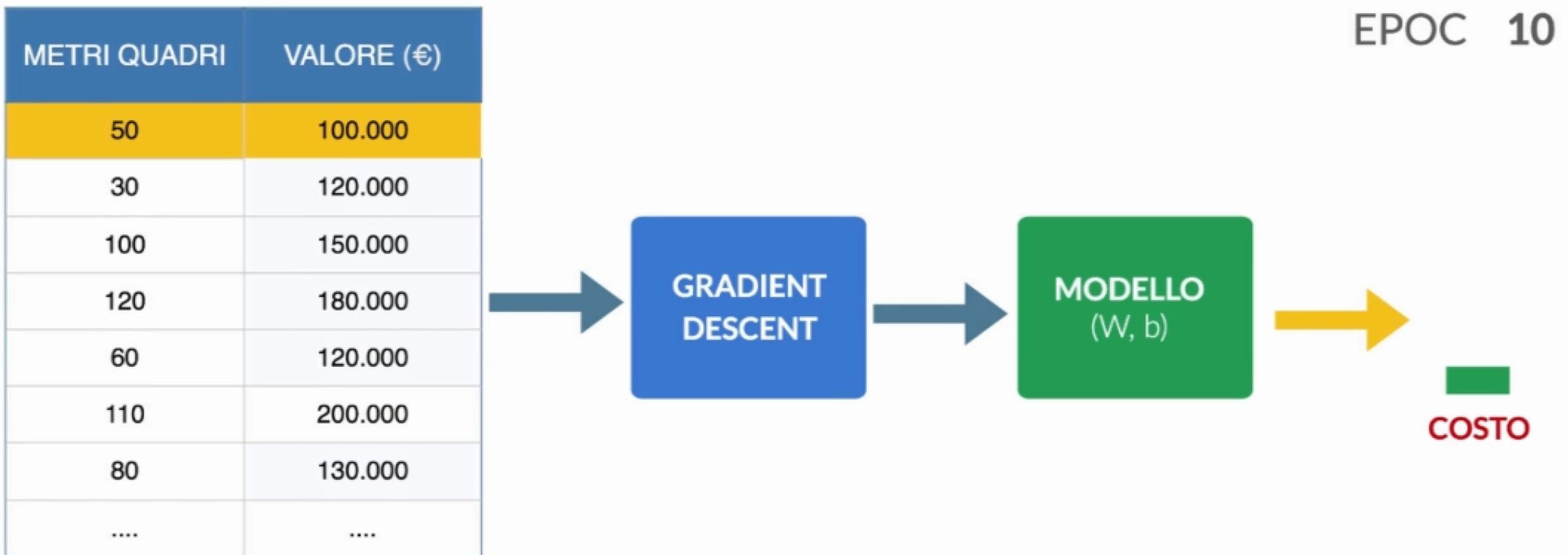
To optimise the learning rate, Stochastic Gradient Descent is used, which is an interative process. So, updating the weights with single pass or one epoch is not enough!

The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs.

# Stochastic Gradient Descent



# Stochastic Gradient Descent



## Stochastic Gradient Descent

### VANTAGGI

#### Pesa poco in memoria

E' sufficiente caricare un esempio per volta

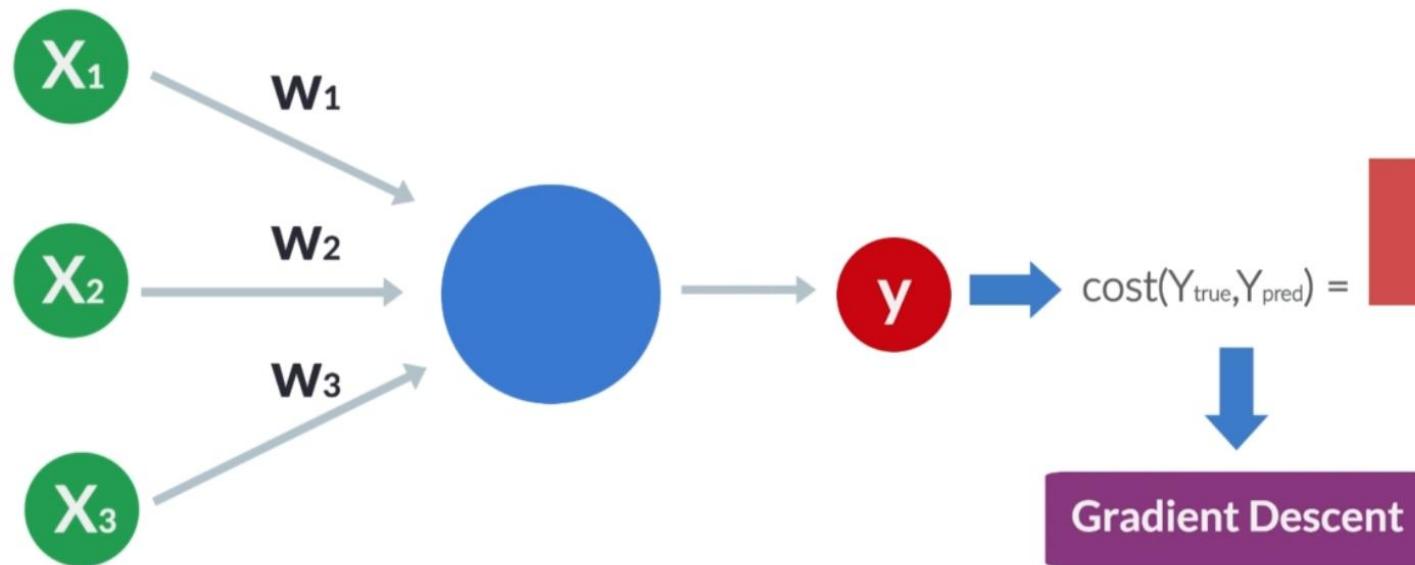
#### Molto dinamico

Per aggiornare il modello con nuovi dati è possibile eseguire uno step del SGD solo su questi dati

#### Meno soggetto al problema dei minimi locali

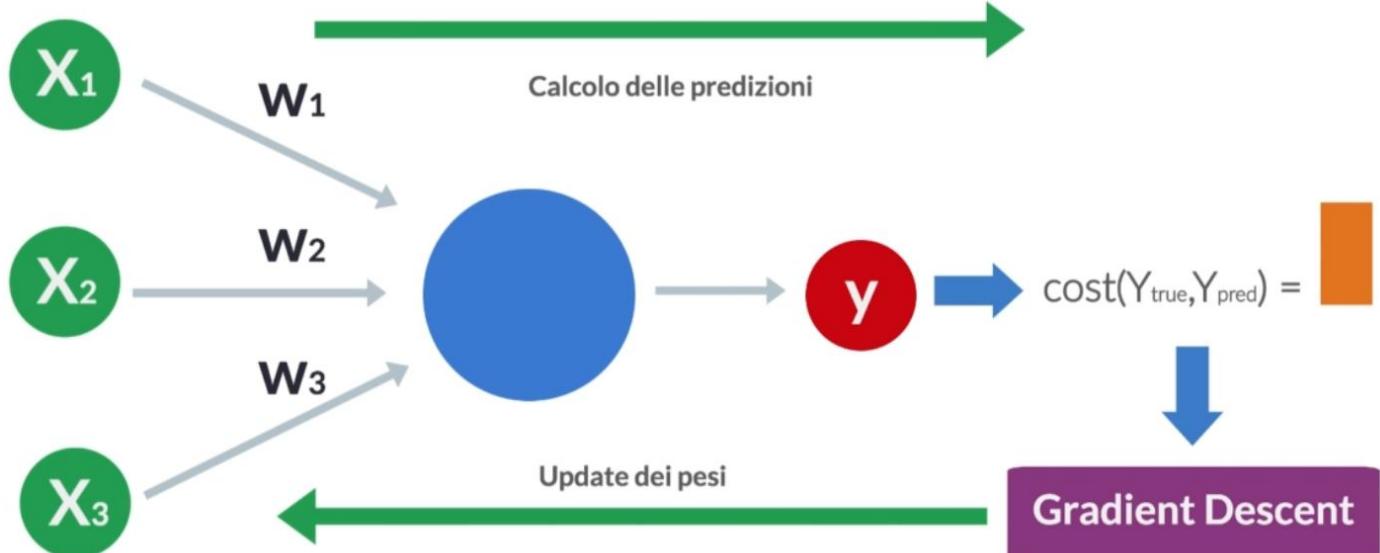
Grazie alle fluttuazioni della funzione di costo potrebbe sfuggire ai minimi locali

## Gradient Descent in azione



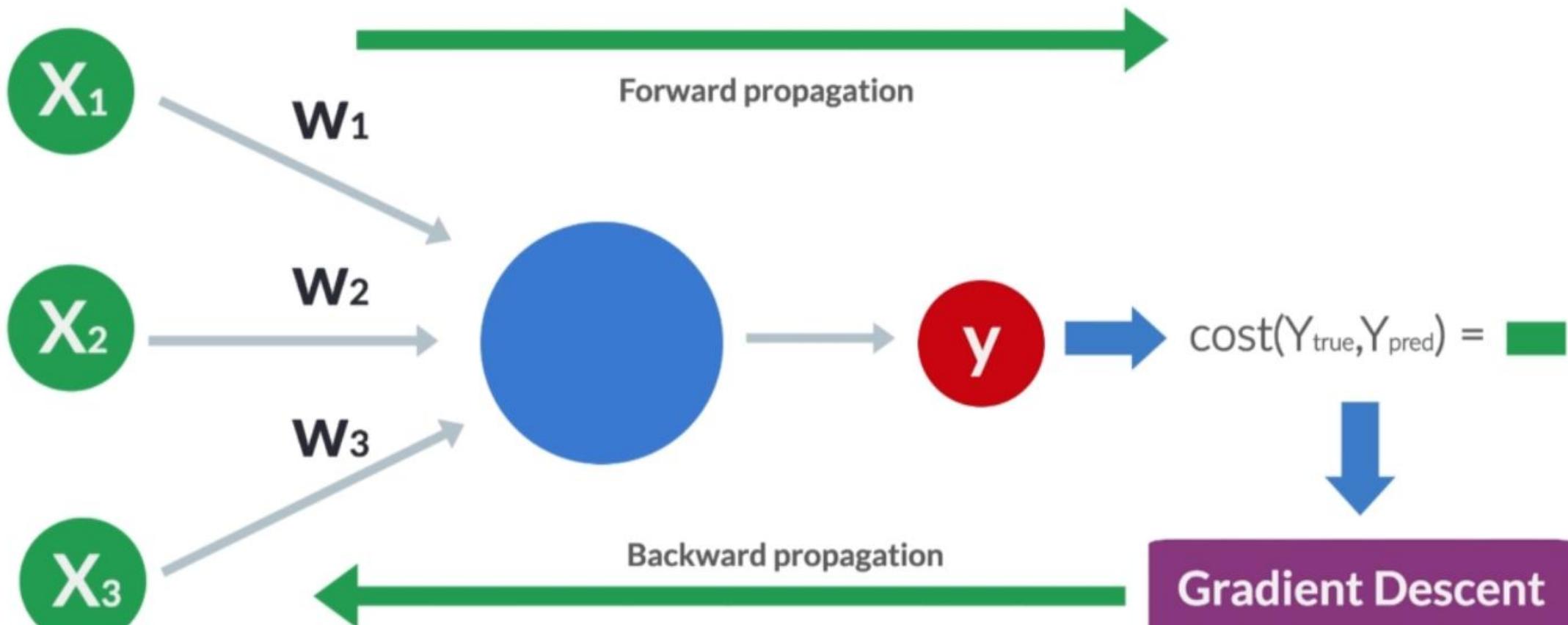
Gradient Descent

## Gradient Descent in azione



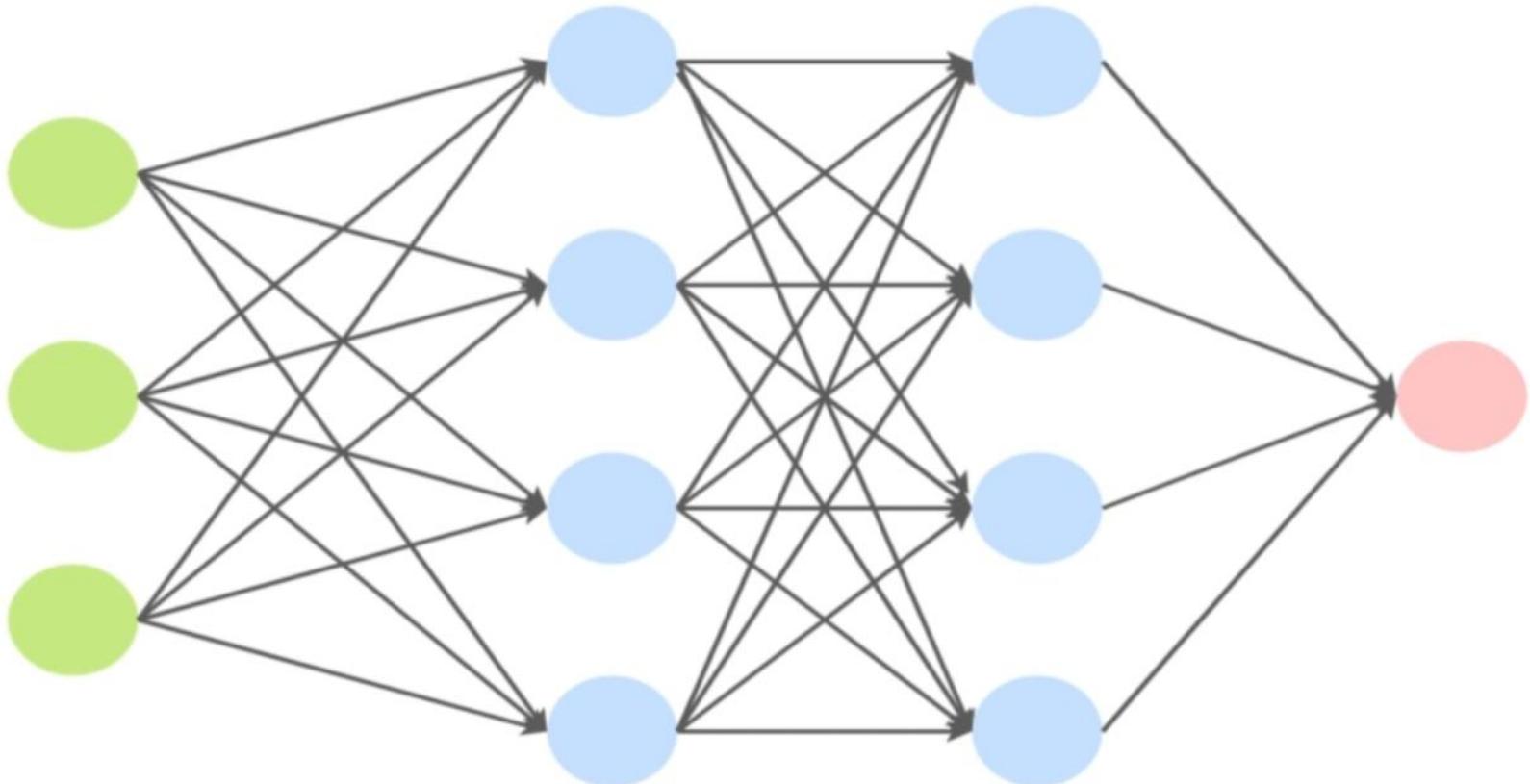
Gradient Descent

# Gradient Descent in azione

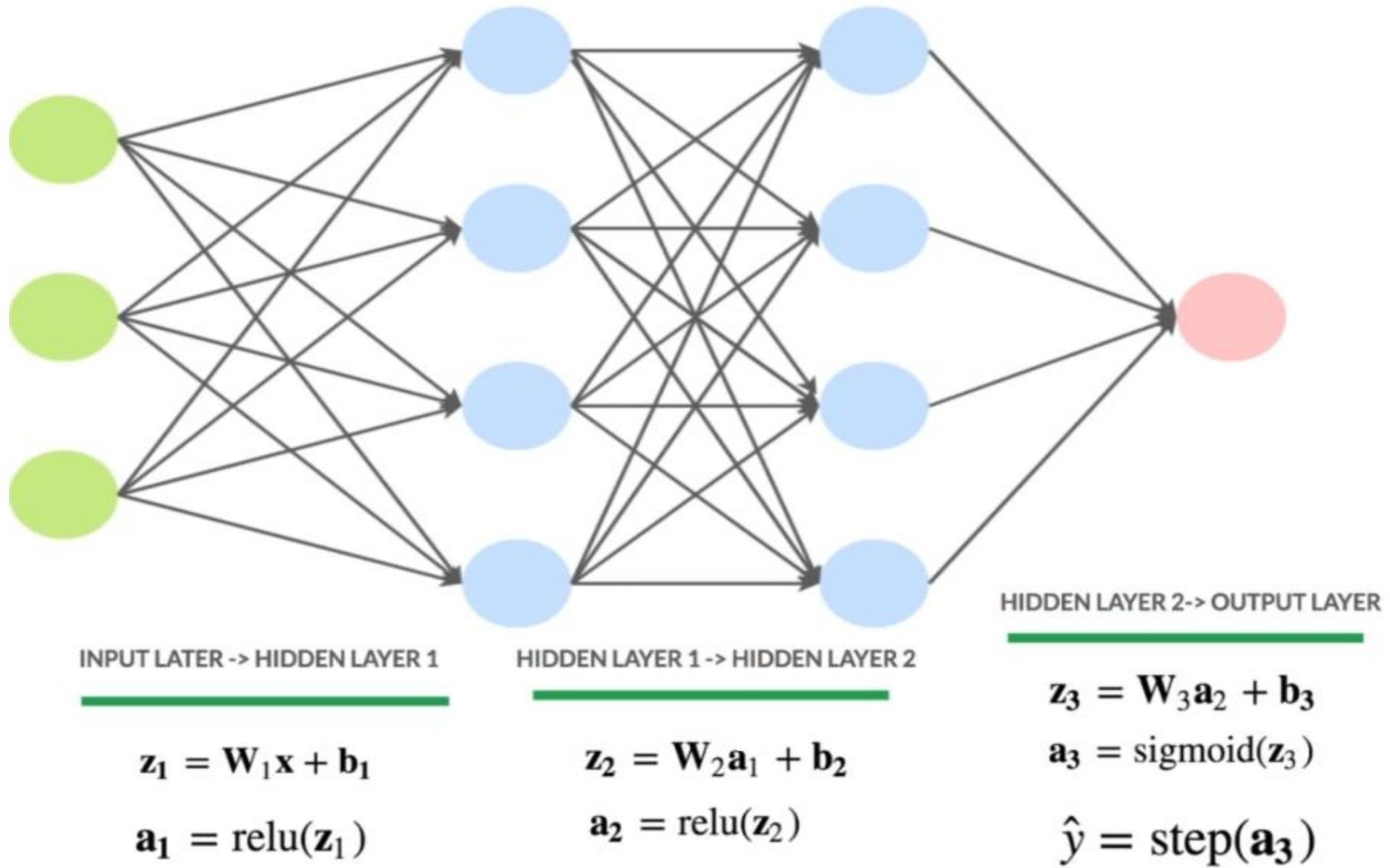


## Backpropagation

Permette di sapere quanto ogni peso di ogni strato  
ha contribuito all'errore del modello  
propagando l'errore dell'ultimo strato all'indietro.



Una rete neurale è formata da più funzioni composte,  
ognuna con i propri coefficienti, che mappano uno strato al successivo



$$\mathbf{z}_1 = \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1$$

$$\mathbf{a}_1 = \text{relu}(\mathbf{z}_1)$$

$$\mathbf{z}_2 = \mathbf{W}_2 \mathbf{a}_1 + \mathbf{b}_2$$

$$\mathbf{a}_2 = \text{relu}(\mathbf{z}_2)$$

$$\mathbf{z}_3 = \mathbf{W}_3 \mathbf{a}_2 + \mathbf{b}_3$$

$$\mathbf{a}_3 = \text{sigmoid}(\mathbf{z}_3)$$

## Chain rule

Permette di calcolare derivate di funzioni annidate

$$f(x) = f(g(x))$$

$$\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dx}$$

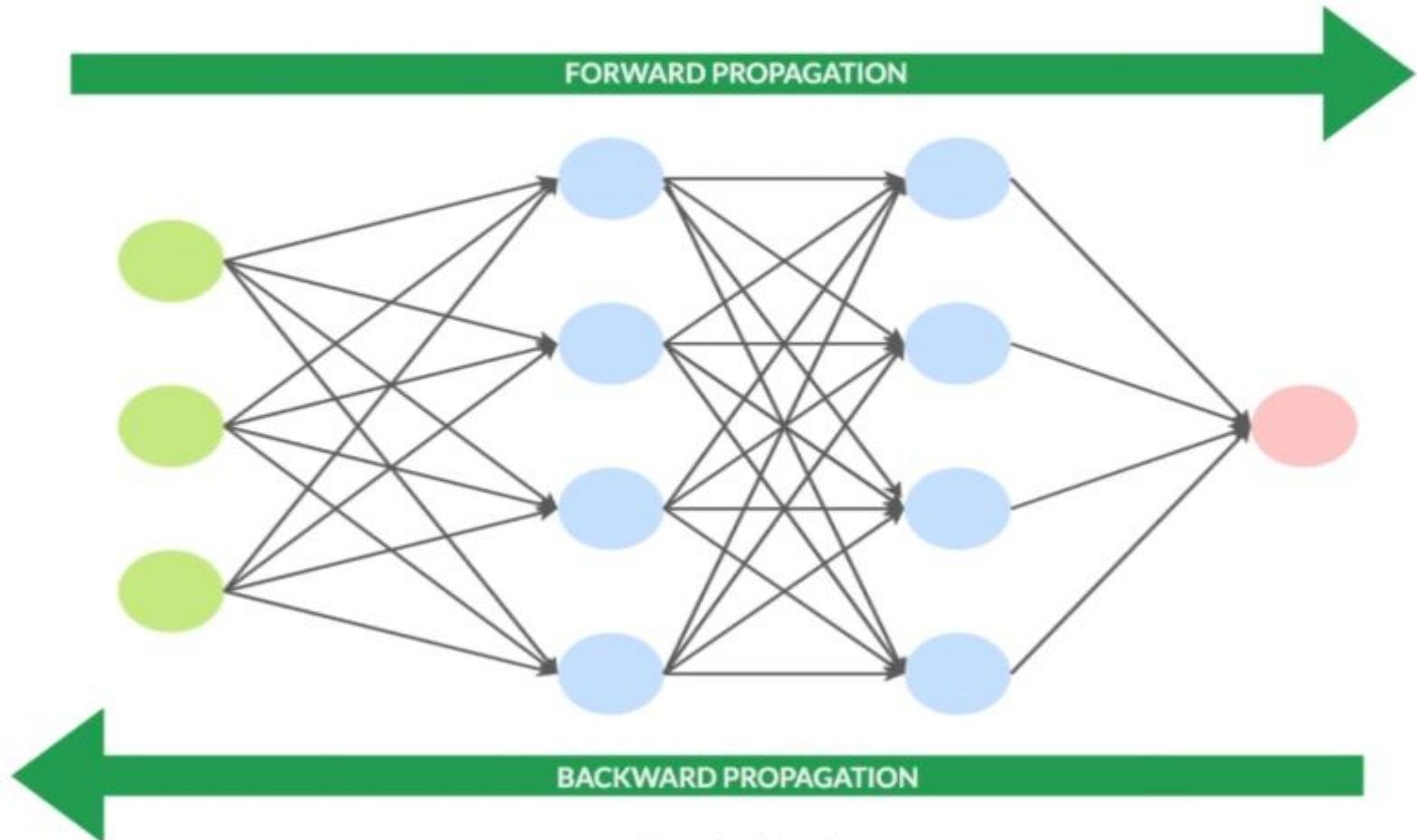
$$J(w, b) = J(\phi(wx + b))$$

## Backpropagation in azione

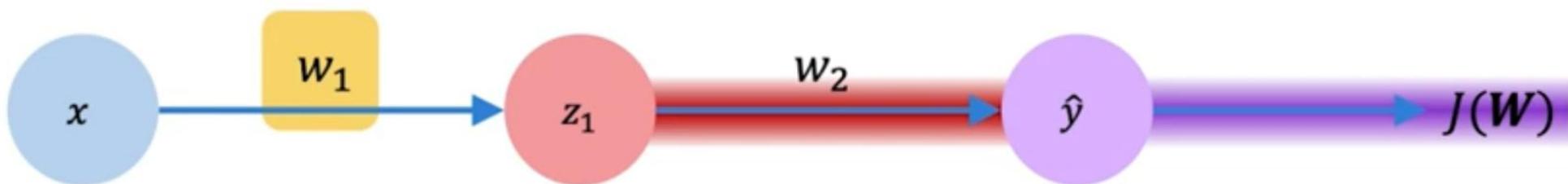
$$\nabla J(w, b) = \left[ \frac{dJ}{dw}, \frac{dJ}{db} \right]$$

$$\frac{dJ}{dw} = \frac{dJ}{d\phi} \frac{d\phi}{dz} \frac{dz}{dw}$$

$$\frac{dJ}{db} = \frac{dJ}{d\phi} \frac{d\phi}{dz} \frac{dz}{db}$$



# Computing Gradients: Backpropagation

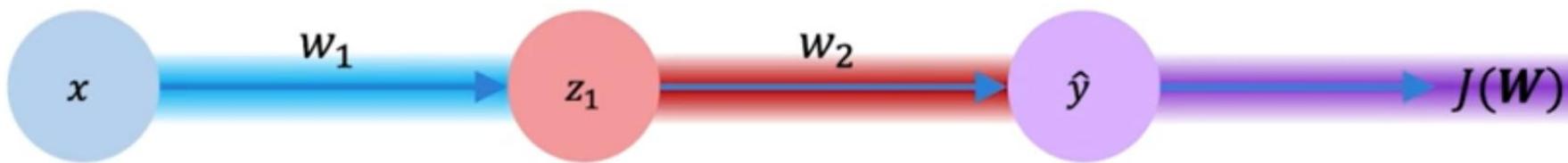


$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

# Computing Gradients: Backpropagation



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \underbrace{\frac{\partial J(\mathbf{W})}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

Repeat this for **every weight in the network** using gradients from later layers

```
def forward_propagation(X):  
  
    z1 = dot(self.W1,X.T)+self.b1  
    a1 = relu(z1)  
    z2 = dot(self.W2,a1)+self.b2  
    a2 = sigmoid(z2)  
  
    return a2
```

```
def backward_propagation(X,a2,y):  
  
    m = a1.shape[1]  
    dz2 = a2-y  
    dW2 = dot(a1.T,dz2)/m  
    db2 = sum(dz2, axis=0) ↴  
  
    m = X.shape[1]  
    dz1 = dot(dz2, self.W2.T)*relu_derivative(z1)  
    dW1 = dot(X.T,dz1)/m  
    db1 = sum(dz1, axis=0)  
  
    return dW1, db1, dW2, db2
```

# Gradient Descent Algorithms

## Algorithm

- SGD
- Adam
- Adadelta
- Adagrad
- RMSProp

## TF Implementation



`tf.keras.optimizers.SGD`



`tf.keras.optimizers.Adam`



`tf.keras.optimizers.Adadelta`



`tf.keras.optimizers.Adagrad`



`tf.keras.optimizers.RMSProp`

## Reference

Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.



**ADAM Optimizer**

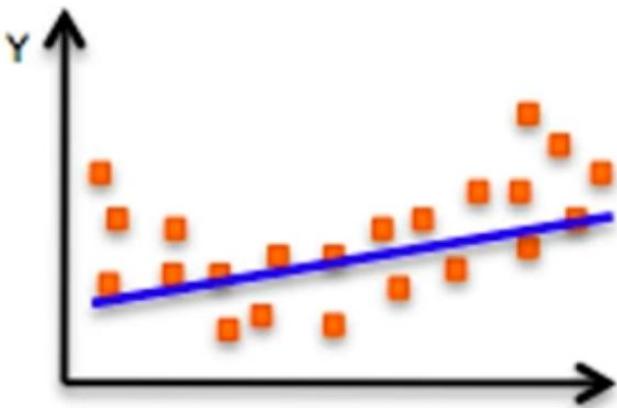
Will you please listen? I am not the messiah.



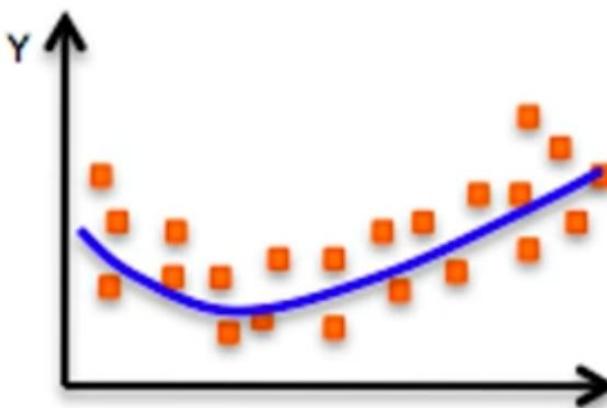
**Literally any sort of Neural Network**

HE IS THE MESSIAH!

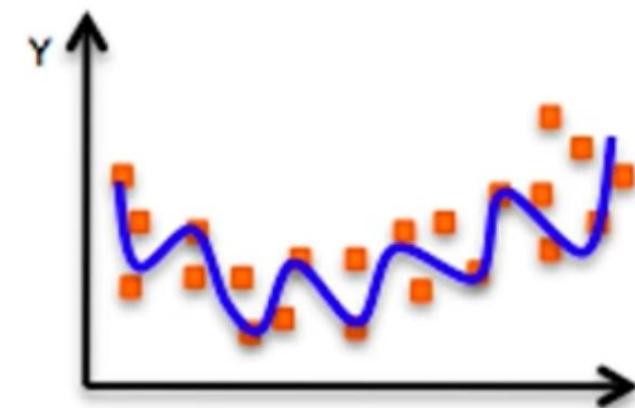
# The Problem of Overfitting



**Underfitting**  
Model does not have capacity  
to fully learn the data



←      **Ideal fit**      →



**Overfitting**  
Too complex, extra parameters,  
does not generalize well