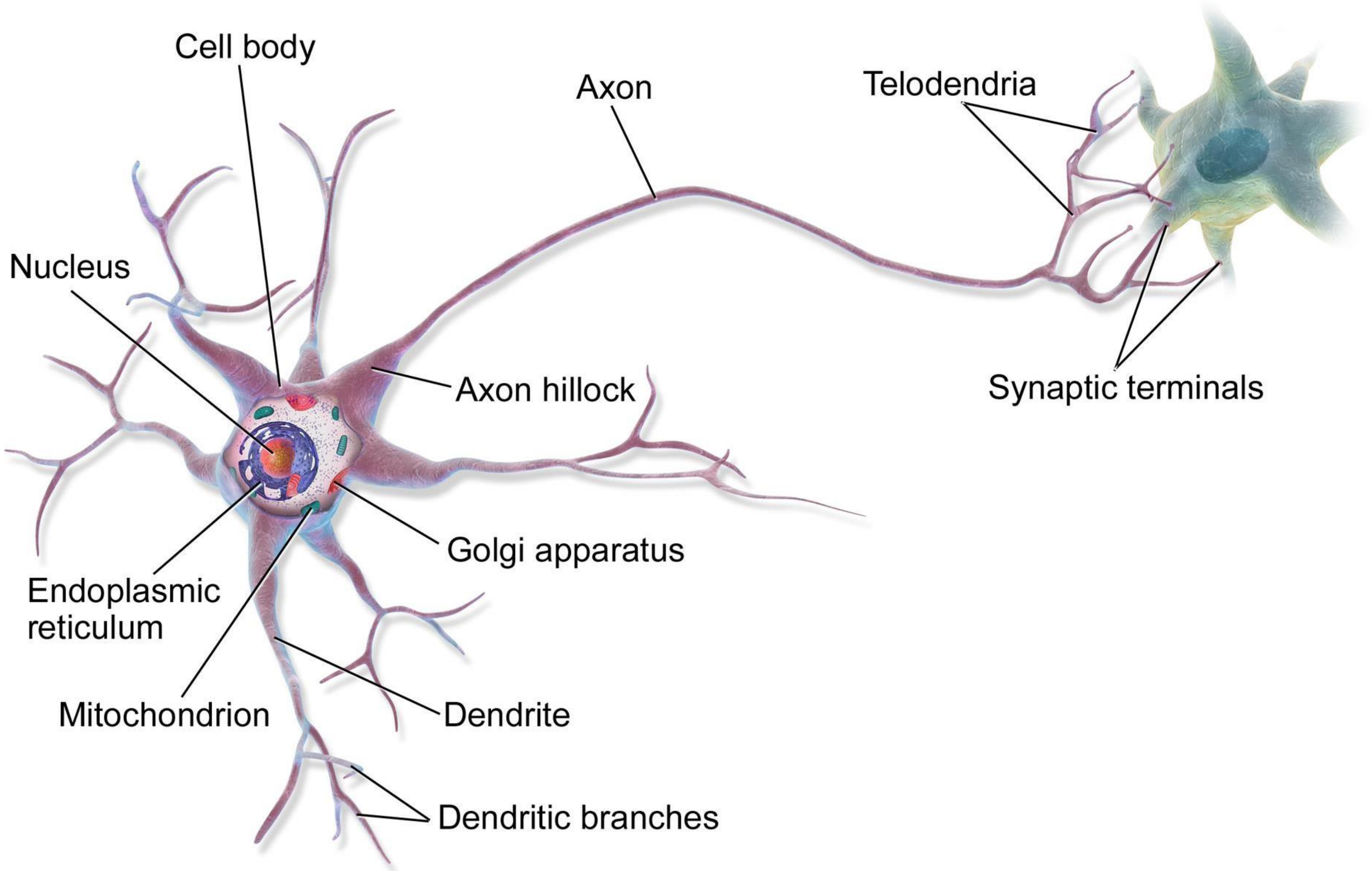
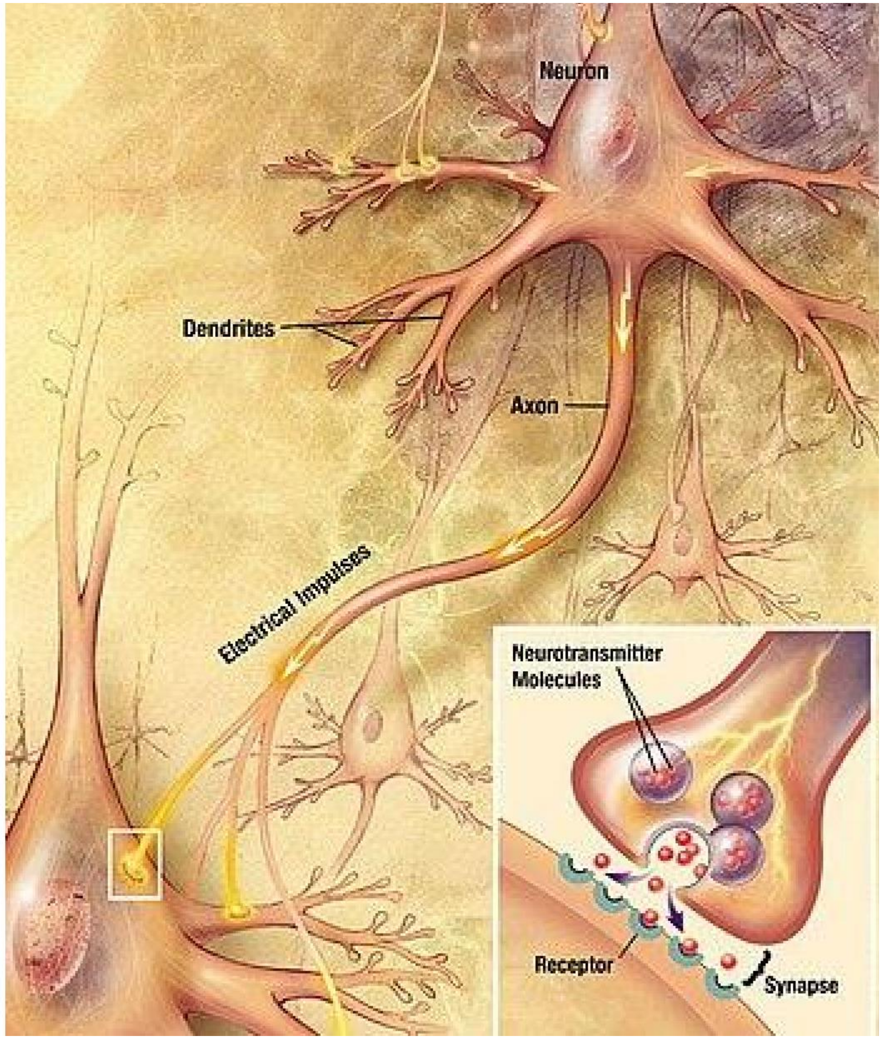
UNIVERSITÀ DELLA CALABRIA

Introduction to Neural Networks

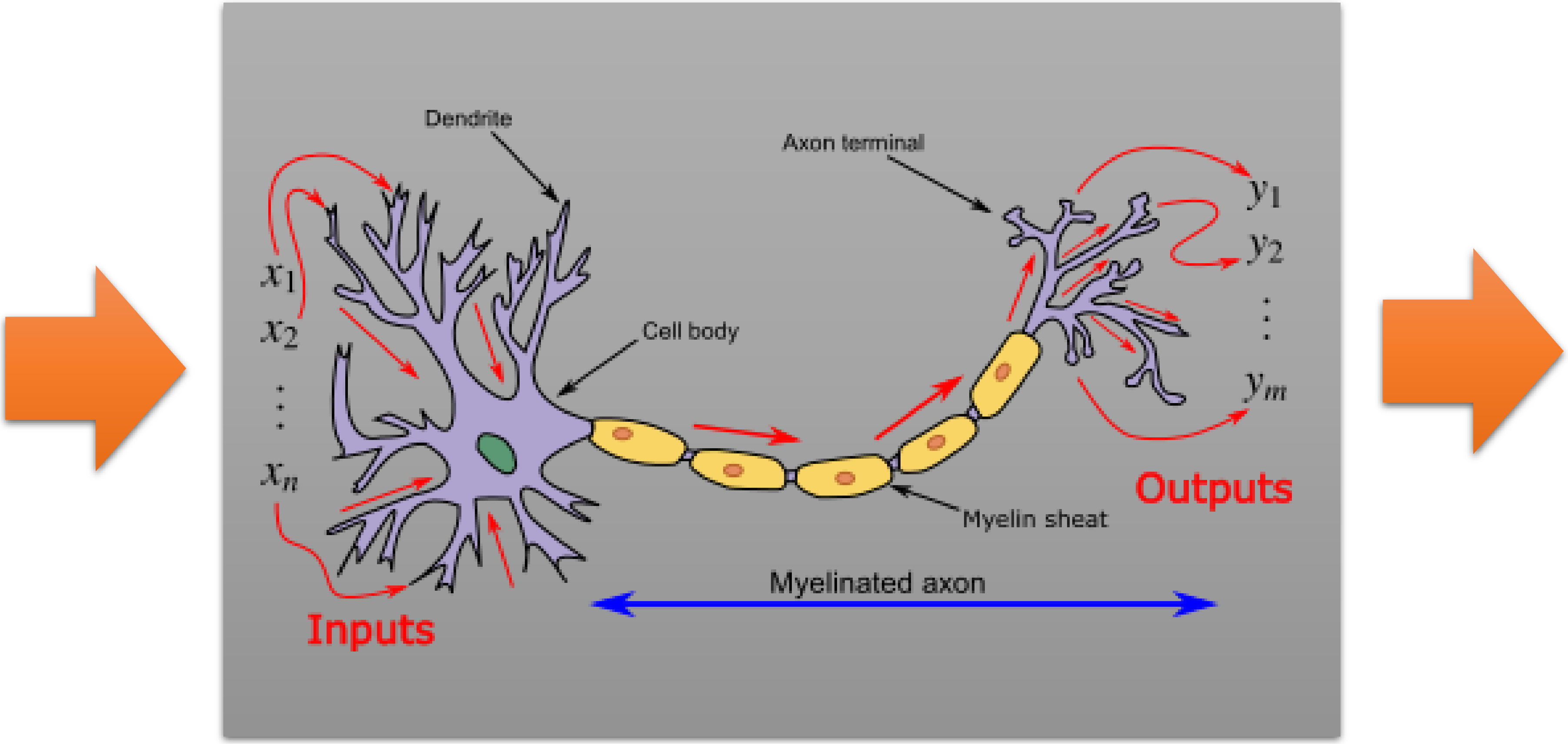
Gianluigi Greco

Biological Neurons



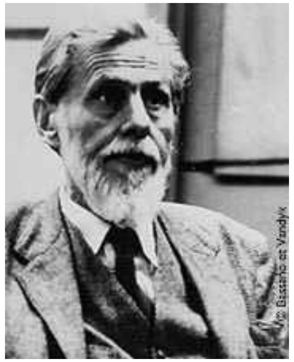


Biological Neurons



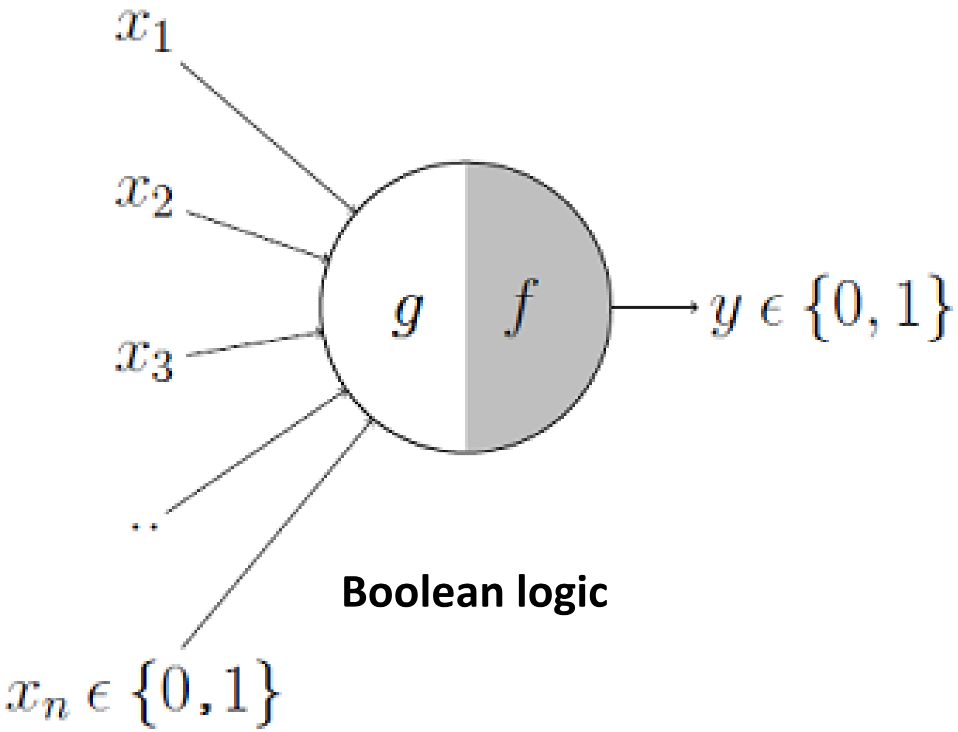
McCulloch & Pitt’s Neuron Model

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115-133.



Warren McCulloch

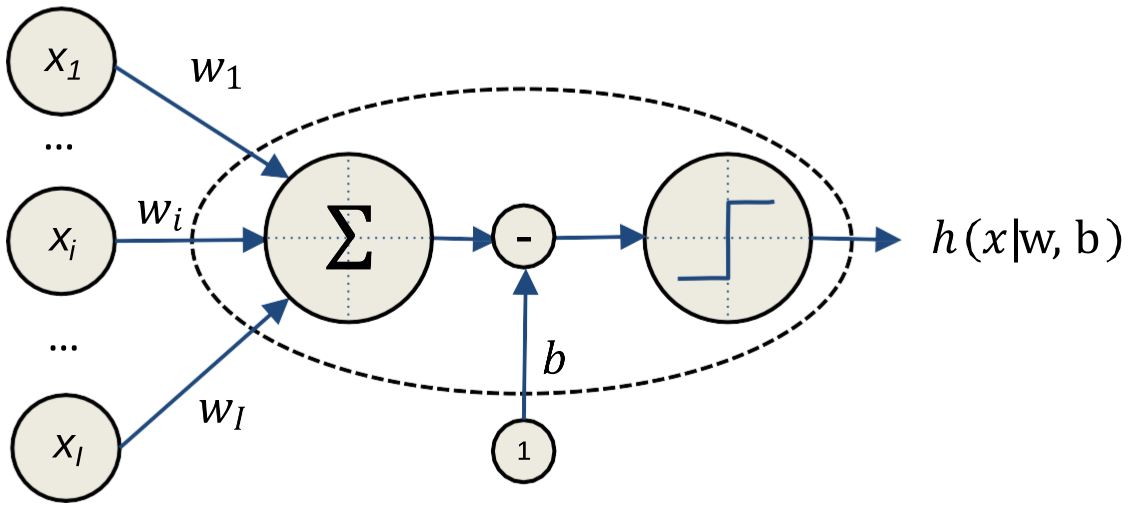
Walter Pitts



Boolean logic

https://www.i-programmer.info/babbages-bag/325-mcculloch-pitts-neural-networks.html

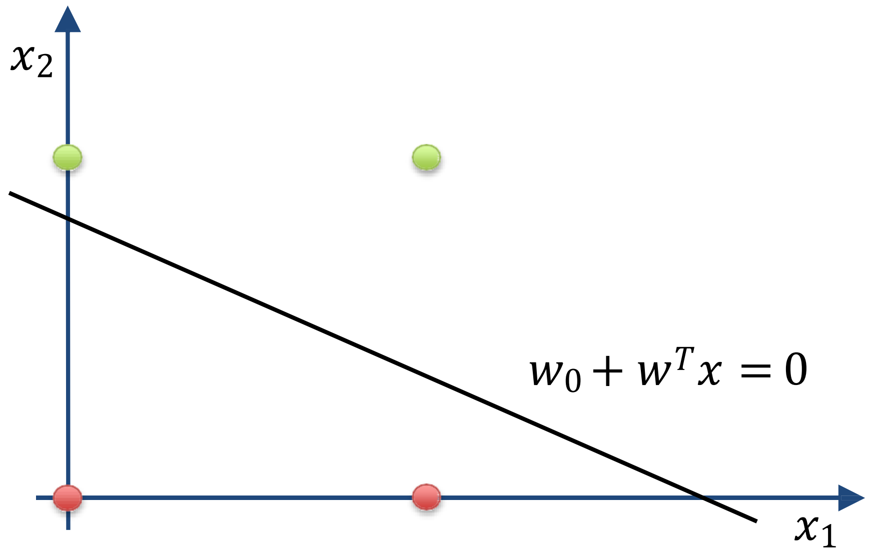
Frank Rosenblatt's Perceptron (1957)



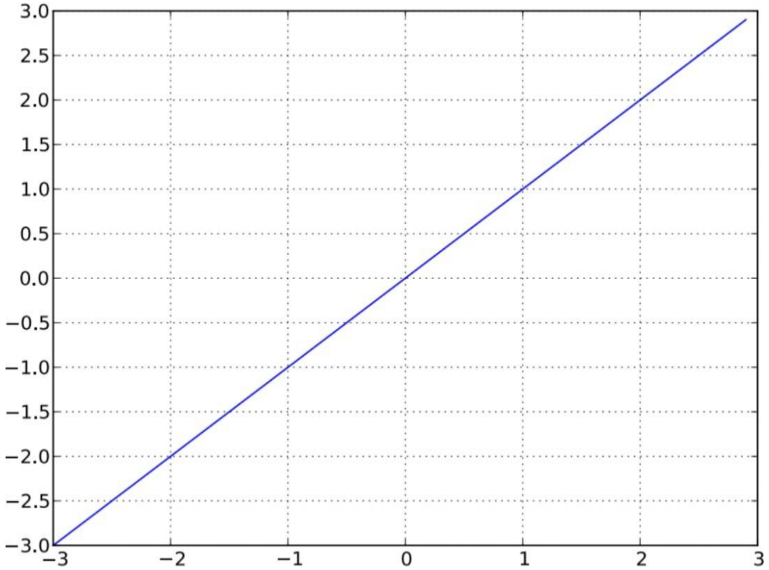
$$h(x|\mathrm{w}, \mathrm{b}) = h\left(\Sigma_{i=1}^{I} w_i \cdot x_i - b\right) = h\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = sign(w^T x)$$

The Math of the Perceptron

$$h\ (x|\mathrm{w}) = \mathrm{h}\ \left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = \ sign(w_0 + w_1 \cdot x_1 + \cdots + w_I \cdot x_I\ \ )$$

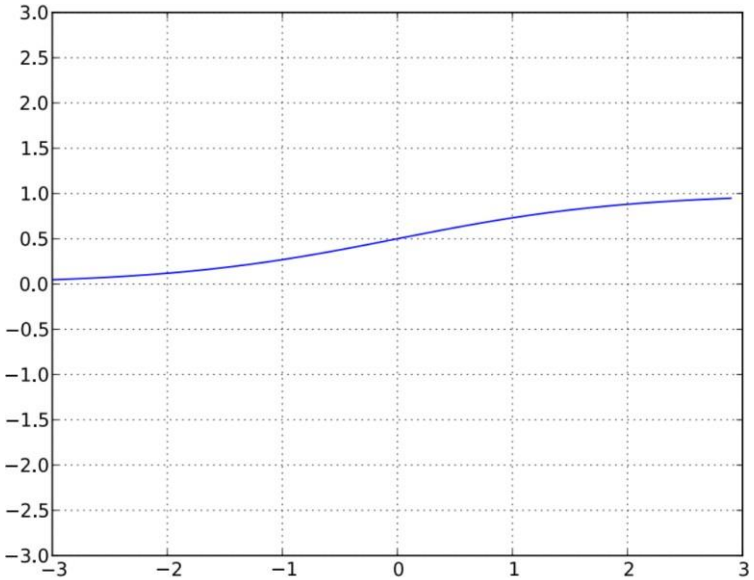


Activation Functions - Beyond «*sign*»



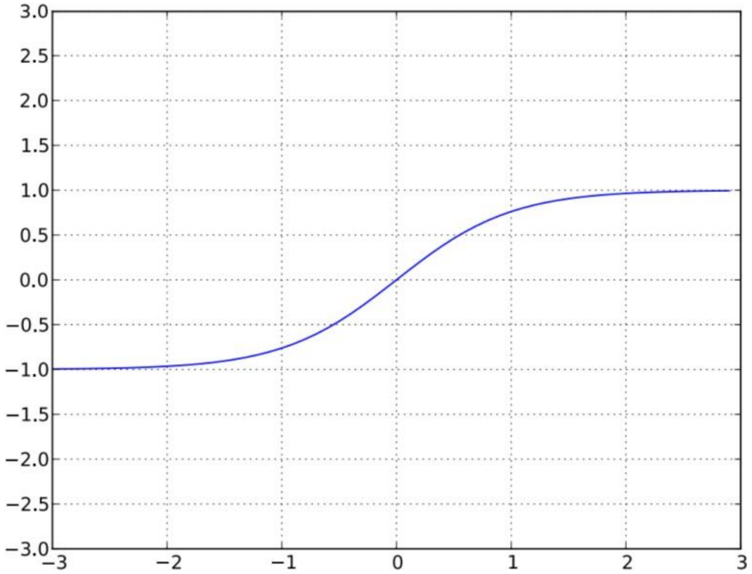
Linear activation function

$$g(a) = a$$



Sigmoid activation function

$$g(a) = \frac{1}{1 + \exp(-a)}$$



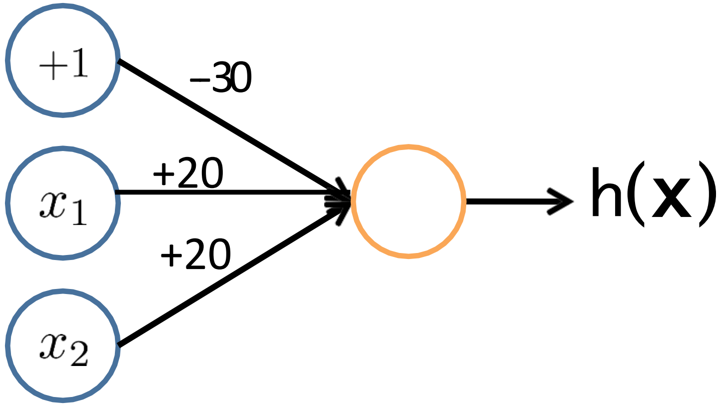
Tanh activation function

$$g(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$

Representing Boolean Functions: AND

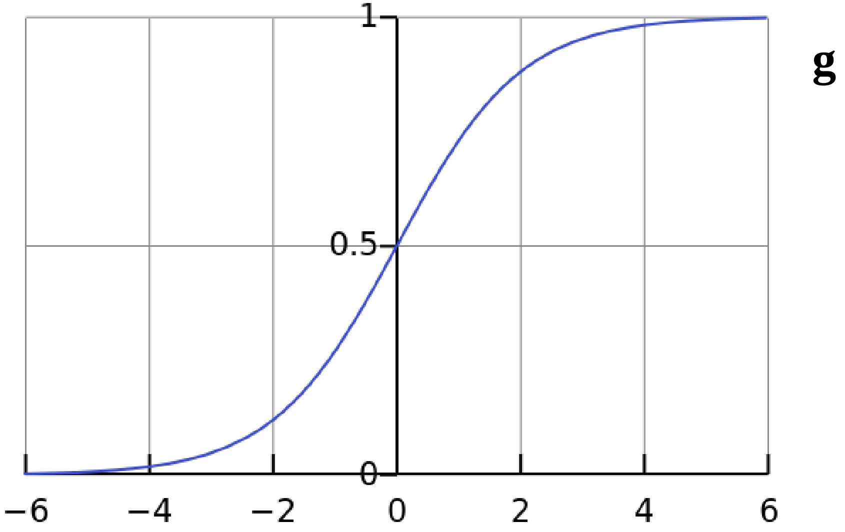
Simple example: AND

$$x_1, x_2 \in \{0, 1\}$$
$$y = x_1 \text{ AND } x_2$$



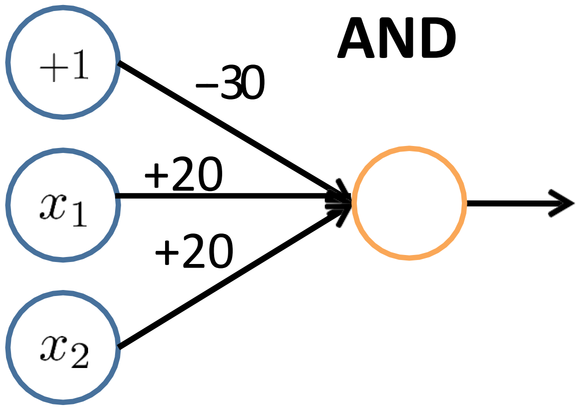
$$\mathrm{h}(\mathbf{x}) = \mathrm{g}(-30 + 20x_1 + 20x_2)$$

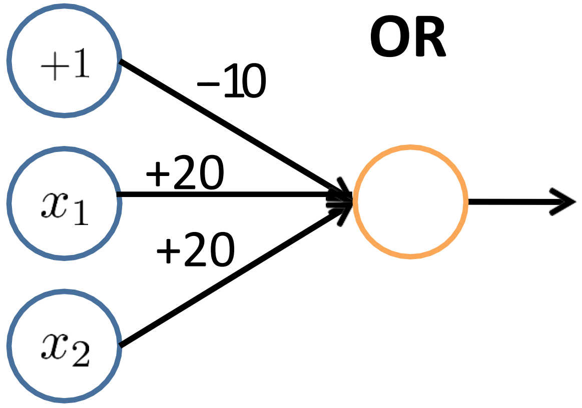
Logistic / Sigmoid Function

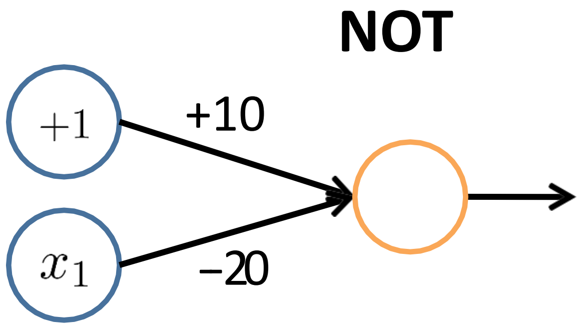


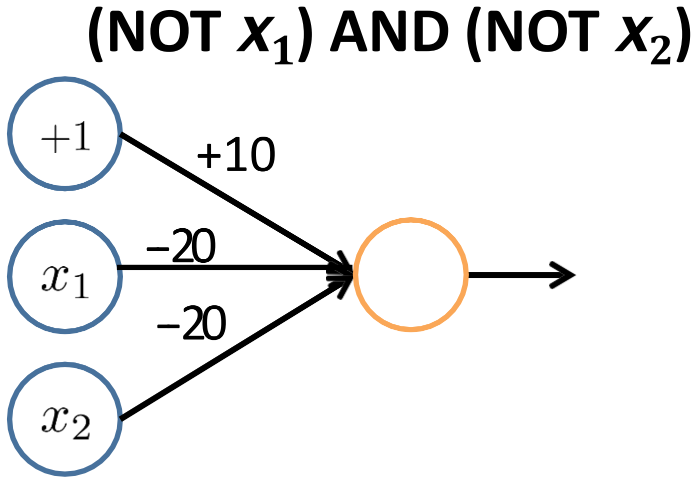
x_1	x_2	h(**x**)
0	0	g(−30) ≈ 0
0	1	g(−10) ≈ 0
1	0	g(−10) ≈ 0
1	1	g(10) ≈ 1

Other Boolean Functions





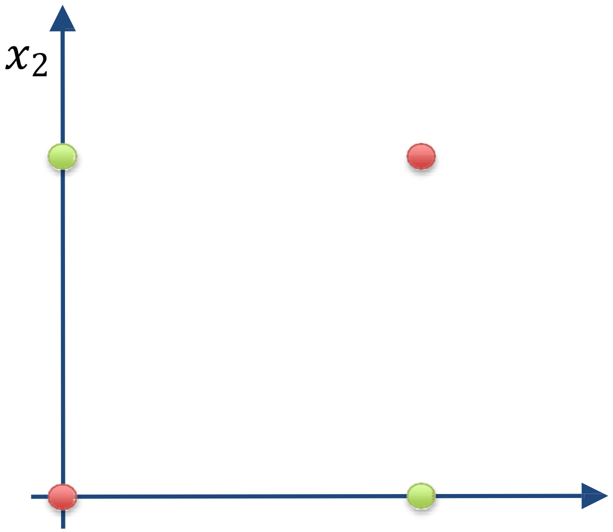




The XOR Problem

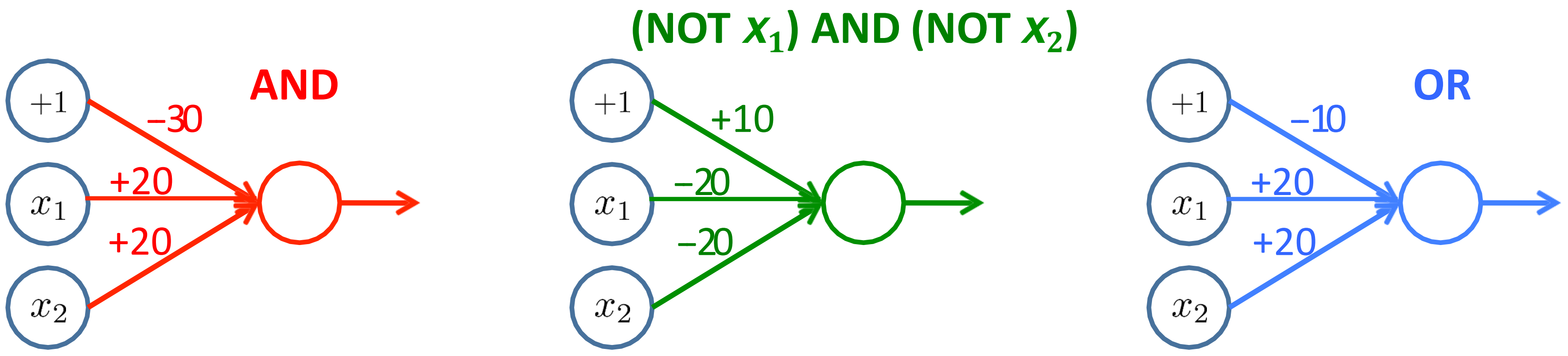
- The perceptron cannot learn regions that do not have linear boundaries

x_0	x_1	x_2	XOR
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	-1



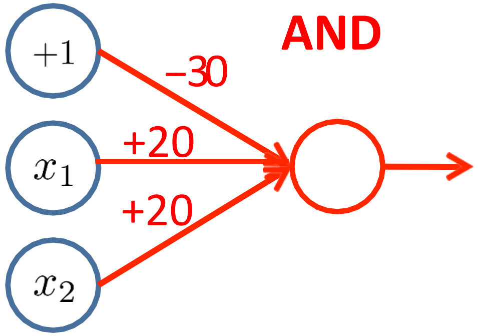
Marvin Minsky, Seymour Papert
"*Perceptrons: an introduction to computational geometry*" 1969.

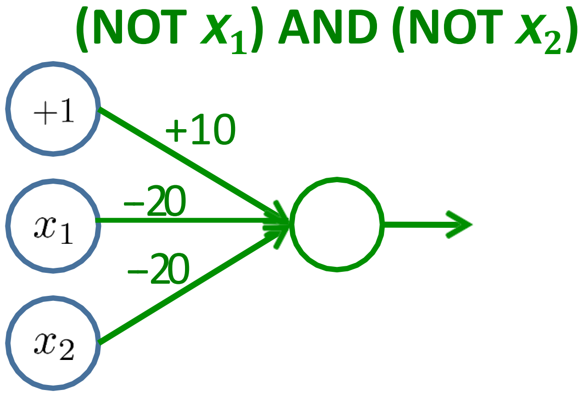
Non-Linearity by Adding Layers

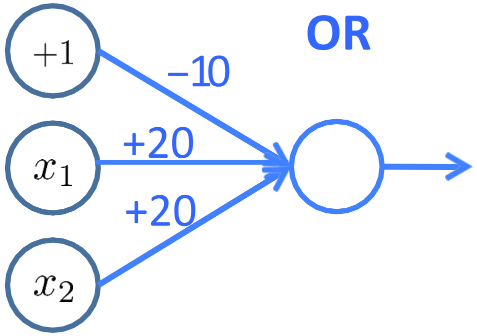


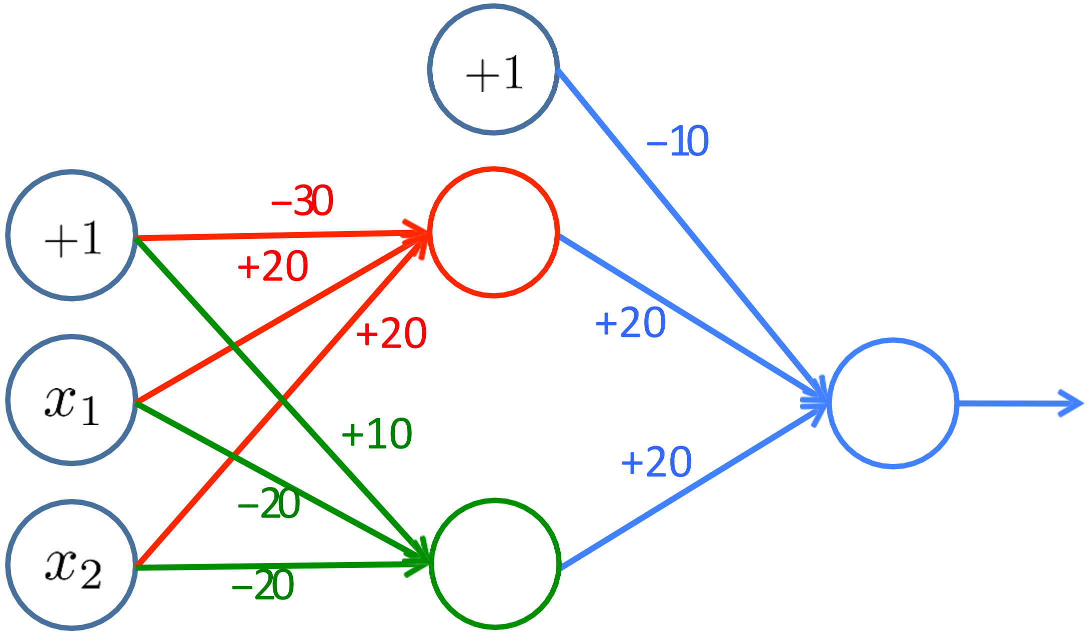
NOT (A XOR B) = (A AND B) OR ((NOT A) AND (NOT B))

Non-Linearity by Adding Layers

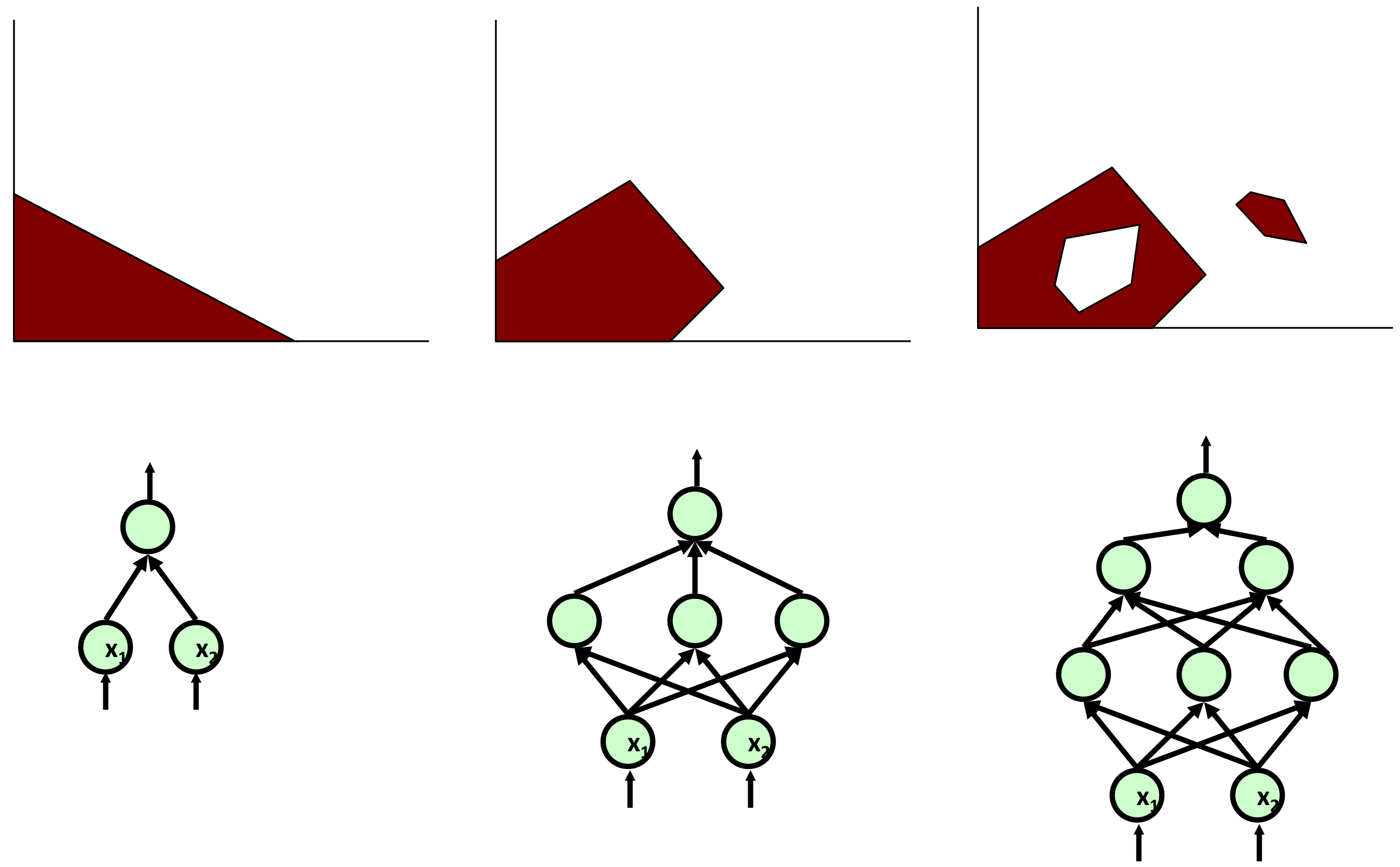




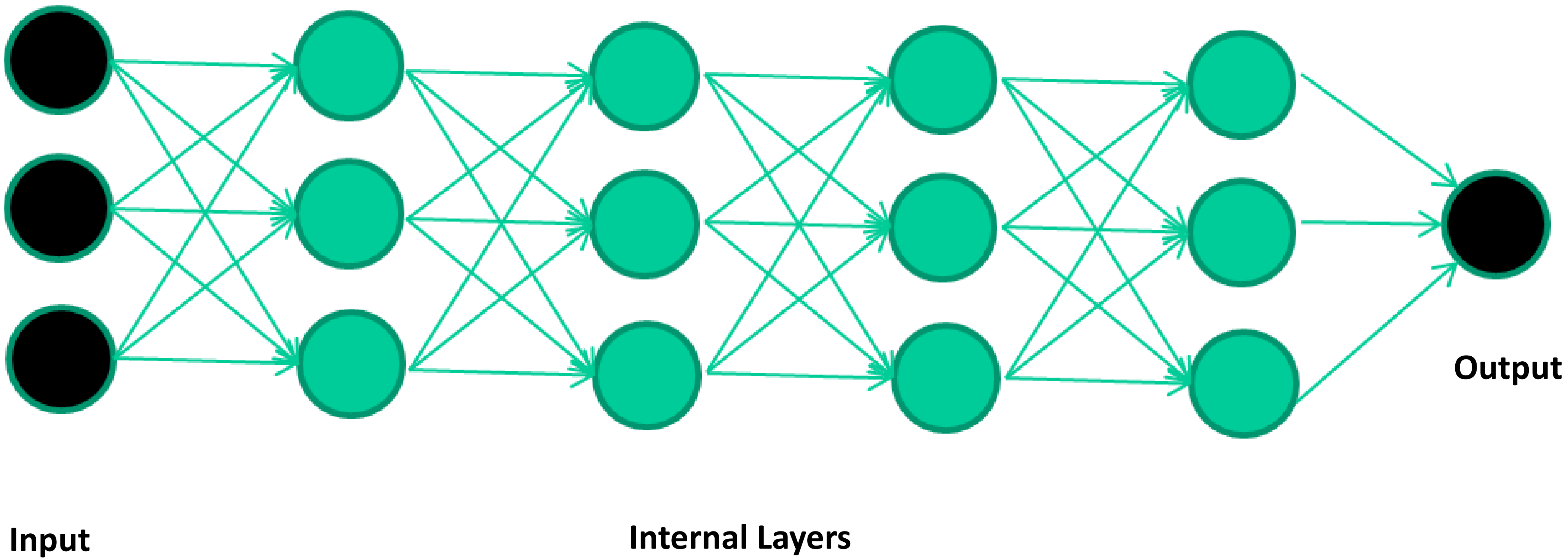




Representation Power...



...Deep Learning!



UNIVERSITÀ DELLA CALABRIA

Basic Ingredients: Graph

Gianluigi Greco

g the graph

- $g = \{N, E\}$ is a weighted labeled directed graph

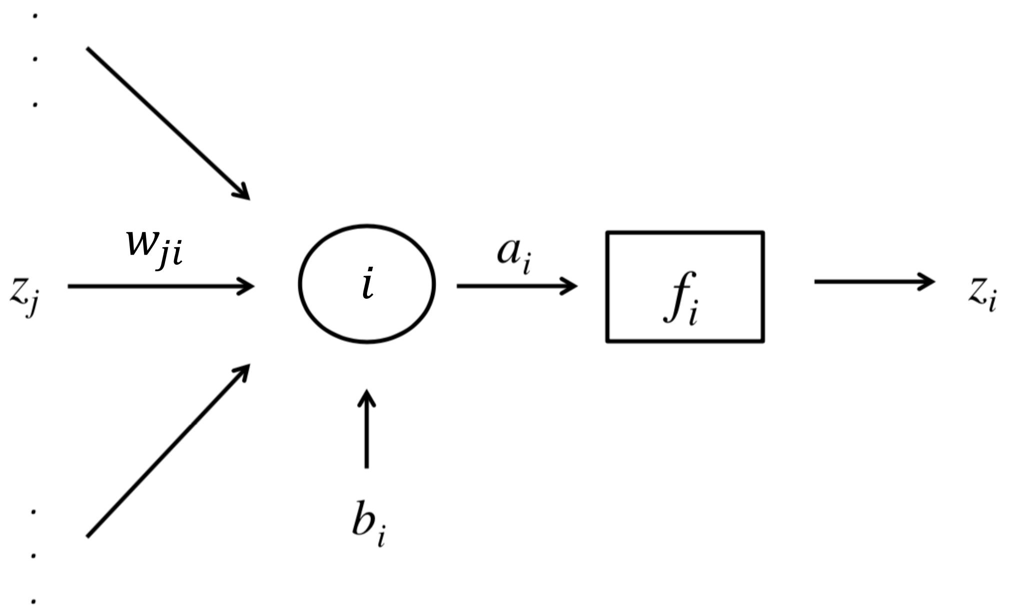
- Each node $i \in N$ is also called neuron or perceptron
  - It is equipped with two labels
    - A value a_i, that will be called “activation” in the next
    - An activation function, f_i, that, applied to the activation, produces an output $\boldsymbol{z_i}$

- Each edge $e = \{j \in N \rightarrow i \in N\} \in E$ is equipped with a weight w_{ji}

- Each node i is involved in an additional special edge, with a ghost node, whose weight is called bias, b_i

g the graph

- Each neuron is a calculus unit



$$z_i = f_i(a_i)$$

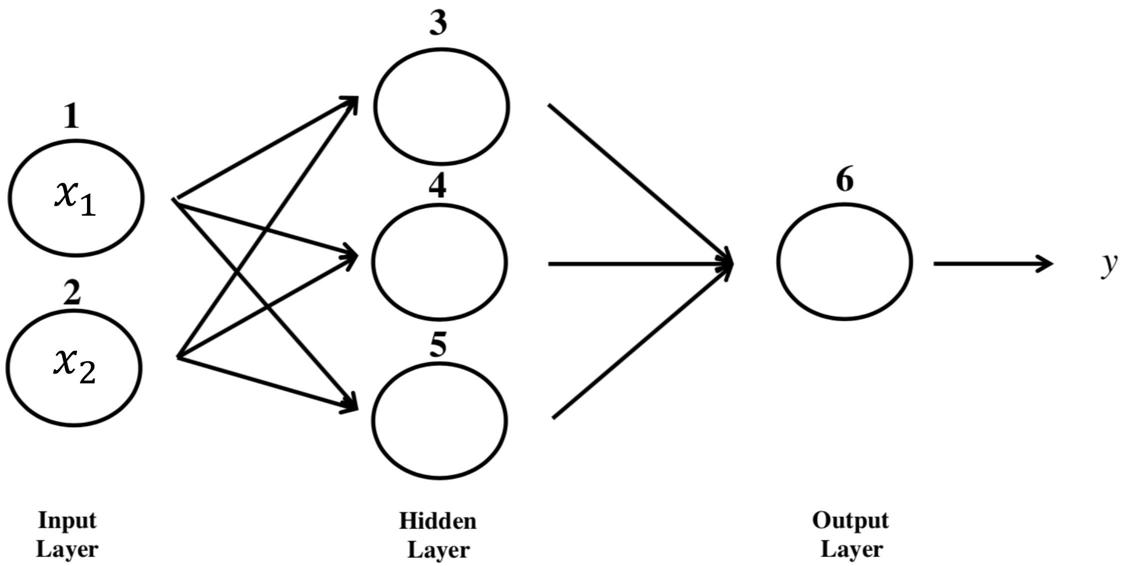
$$a_i = b_i + \sum_{j:j \to i \in E} w_{ji} z_j$$

g the graph

- Nodes in N belong to three categories:
  - Input nodes
    - Nodes whose value can be overwritten from the outside
  - Hidden nodes
    - Nodes that perform as calculus units
  - Output nodes
    - Nodes whose value is provided to the outside

g the graph

- A combination of connected neurons builds the graph up
  - Nodes that share the same input are grouped into layers



$$z_1 = x_1$$

$$z_2 = x_2$$

$$z_3 = f_3\left(b_3 + \sum_{j:j\rightarrow 3\in E} w_{j3} z_j\right)$$

$$z_4 = f_4\left(b_4 + \sum_{j:j\rightarrow 4\in E} w_{j4} z_j\right)$$

$$z_5 = f_5\left(b_5 + \sum_{j:j\rightarrow 5\in E} w_{j5} z_j\right)$$

$$y = z_6 = f_6\left(b_6 + \sum_{j:j\rightarrow 6\in E} w_{j6} z_j\right)$$

$$y = f_6\left(b_6 + w_{5,6} f_5(b_5 + w_{1,5}x_1 + w_{2,5}x_2) + w_{4,6} f_4(b_4 + w_{1,4}x_1 + w_{2,4}x_2) + w_{3,6} f_3(b_3 + w_{1,3}x_1 + w_{2,3}x_2)\right)$$

g the graph

- A combination of connected neurons builds the graph up
  - Nodes that share the same input are grouped into layers

- The resulting expression is defined in terms of all the parameters of the graph, and is also determined by the different activation functions
  - Very large flexibility
  - Can be used to approximate a given function **F**

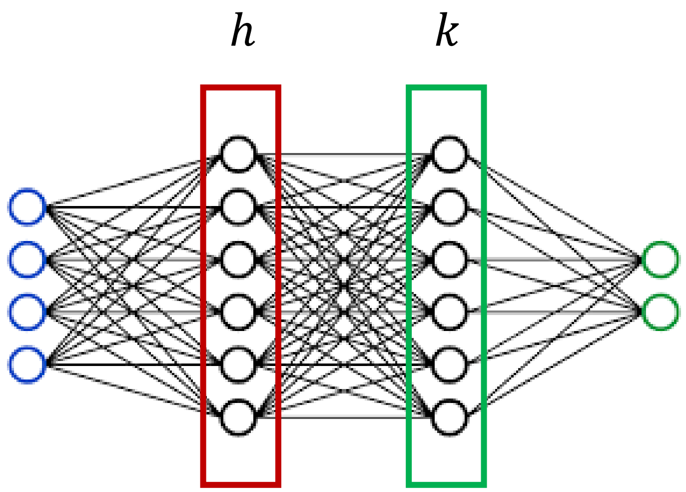
$$y = f_6\left(b_6 + w_{5,6} f_5(b_5 + w_{1,5}x_1 + w_{2,5}x_2) + w_{4,6} f_4(b_4 + w_{1,4}x_1 + w_{2,4}x_2) + w_{3,6} f_3(b_3 + w_{1,3}x_1 + w_{2,3}x_2)\right)$$

g the graph: Feedforward Networks

- Compact notation:
  - Given two consecutive layers k and h:

$$\vec{z}_k = f_k\left(\vec{b}_k + W_k \vec{z}_h\right)$$

- We are assuming that all the nodes in k share the same activation function f_k
- $\vec{b}_k$ contains all the biases of the nodes in k
- W_k is the matrix containing all the w_{hk} weights
- $\vec{z}_t$ is the vector containing the values of the node in the layer $t \in \{k, h\}$



UNIVERSITÀ DELLA CALABRIA

Basic Ingredients: Loss Function

Gianluigi Greco

l the loss function

- The graph g is actually a non-linear algebraic operator
  - $\boldsymbol{g}\left(\vec{\boldsymbol{x}}\,|\boldsymbol{W},\boldsymbol{B}\right)$

$$Example{:}\ f_6\left(b_6 + w_{5,6}f_5\left(b_5 + w_{1,5}x_1 + w_{2,5}x_2\right) + w_{4,6}f_4\left(b_4 + w_{1,4}x_1 + w_{2,4}x_2\right) + w_{3,6}f_3\left(b_3 + w_{1,3}x_1 + w_{2,3}x_2\right)\right)$$

- The operator is composed by a-priori unknown variables:
  - The weights ***W*** and the biases ***B***
- The *learning phase* of a neural network aims at finding the “best” ***W*** and ***B***
  - What does “best” mean?

l the loss function

- Finding the “best” values needs to optimize an objective function that expresses the semantics of the analysis goals
  - For what purpose are we using the neural networks?
  - What is the input?
  - What is the desired output?
  - How far is the produced output from the desired output?

- Find ***W*** and ***B*** such that the neural network approximates **F**

$$g(\vec{x}\,|\boldsymbol{W},\boldsymbol{B})$$

l the loss function

- In neural networks the objective function is called loss function
- The loss function represents the error in producing an output as close as possible to the desired one, by applying its operator $\boldsymbol{g}$ on the input
- The objective of a network is:

$$\underset{W,B}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} loss[\vec{y}_i, g(\vec{x}_i | W, B)]$$

l the loss function

Loss functions for Regression

- Mean Absolute Error (MAE)

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}\|\vec{y}_i - g(\vec{x}_i|W,B)\|_1 = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\left|y_{i,j} - g(\vec{x}_i|W,B)_j\right|$$

- **MAE:**
  - **considers equally all the errors**
  - **is not differentiable**

- Mean Squared Error (MSE)

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\|\vec{y}_i - g(\vec{x}_i|W,B)\|_2 = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\left[y_{i,j} - g(\vec{x}_i|W,B)_j\right]^2$$

- **MSE:**
  - **strongly penalizes big errors**
  - **is cautious with the small ones**
  - **suffers from outliers**
  - **is smooth and differentiable**

l the loss function

Loss functions for Regression

- Smooth Absolute Error (SAE) *- Tries to merge MAE and MSE*

$$\text{SAE} = \begin{cases} \frac{1}{2n}\sum_{i=1}^{n} \|\vec{y}_i - g(\vec{x}_i|W,B)\|_2 & \text{if } \|\vec{y}_i - g(\vec{x}_i|W,B)\|_1 < 1 \\ -\frac{1}{2} + \frac{1}{n}\sum_{i=1}^{n} \|\vec{y}_i - g(\vec{x}_i|W,B)\|_1 & \text{otherwise} \end{cases}$$

l the loss function

Loss functions for Classification (essentially, binary outputs)

- Binary Cross Entropy (BCE), appropriate if $y_i \in \{0; 1\}$, $g(\vec{x}_i|W,B) \in [0; 1]$

$$\mathrm{BCE} = -\frac{1}{n}\sum_{i=1}^{n} y_i \ln g(\vec{x}_i|W,B) - (1-y_i)\ln[1-g(\vec{x}_i|W,B)]$$

- Categorical Cross Entropy (CCE), appropriate with K classes, $y_{i,k} \in \{0; 1\}$, $g(\vec{x}_i|W,B) \in [0; 1]$

$$\mathrm{CCE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{i,k} \ln g(\vec{x}_i|W,B)_k$$

UNIVERSITÀ DELLA CALABRIA

Basic Ingredients: Optimizer

Gianluigi Greco

o the optimizer

- How to solve this problem?

$$\underset{W,B}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} loss[\vec{y}_i, g(\vec{x}_i|W, B)]$$

o the optimizer

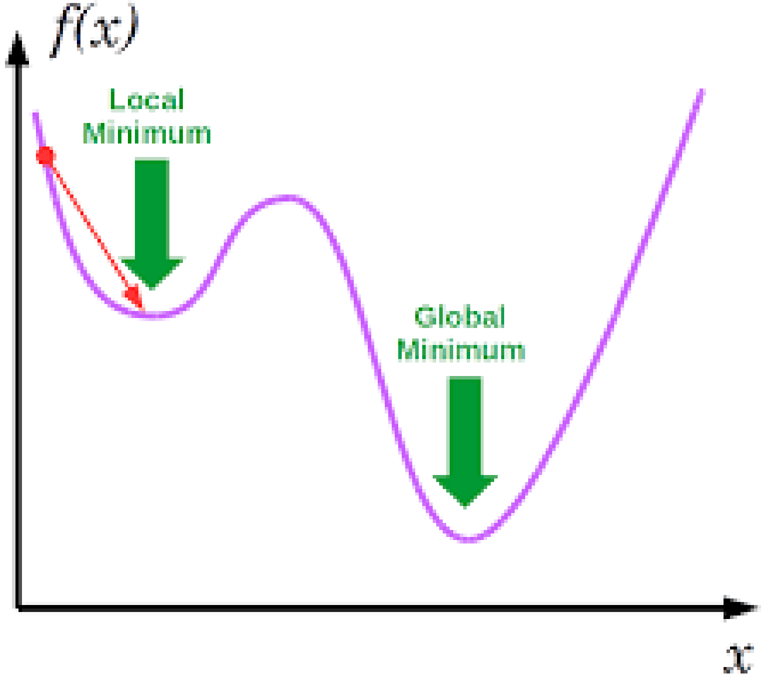
- How to solve this problem?

$$\underset{W,B}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} loss[\vec{y}_i, g(\vec{x}_i | W, B)]$$

- We can compute the gradient of the loss function, put it equal to zero and check if the solutions are minima, maxima or saddle points

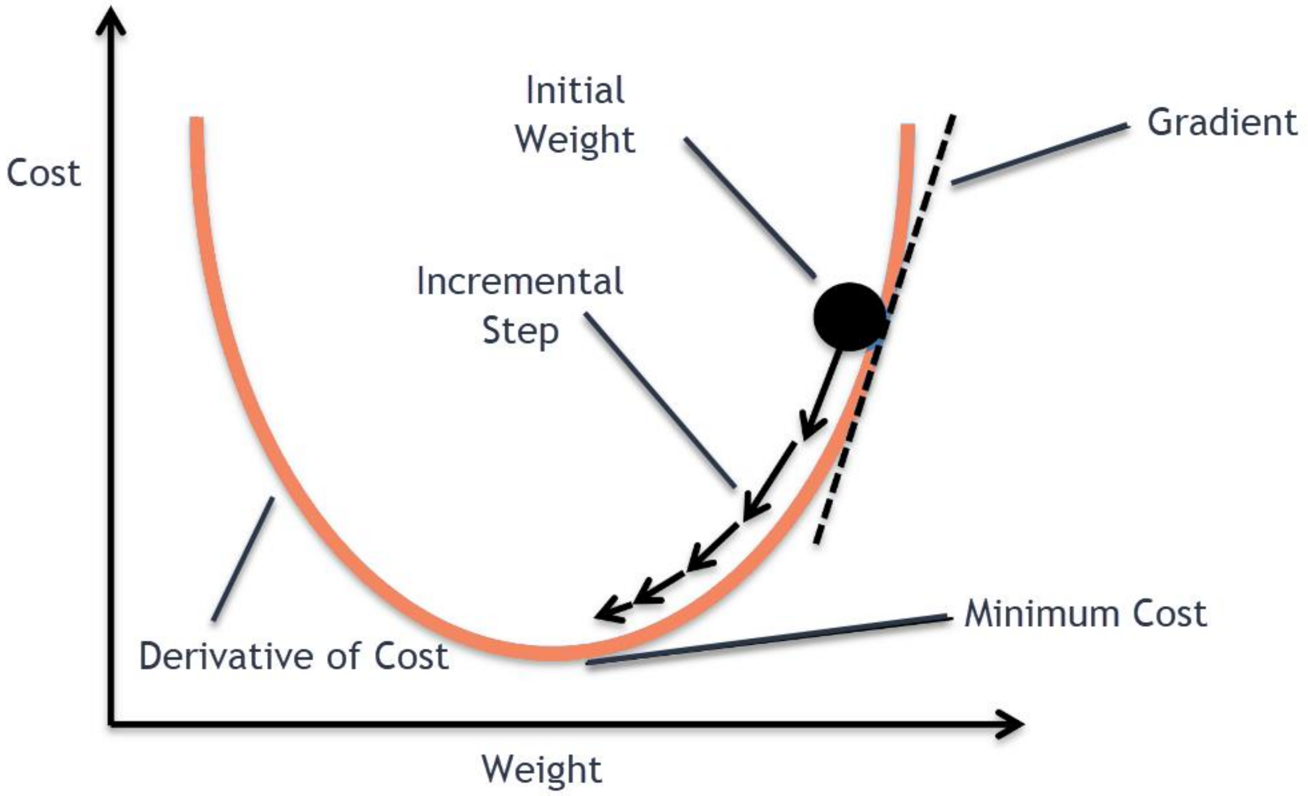
o the optimizer

- Dealing with a lot of (noisy) data and parameters makes hard to find an analytical solution
- We need to define an approximation, a heuristic…
- Typically, we must be happy with local minima
  - Gradient Descent (Finding local minima)



o the optimizer: Gradient Descent

- How does it work?

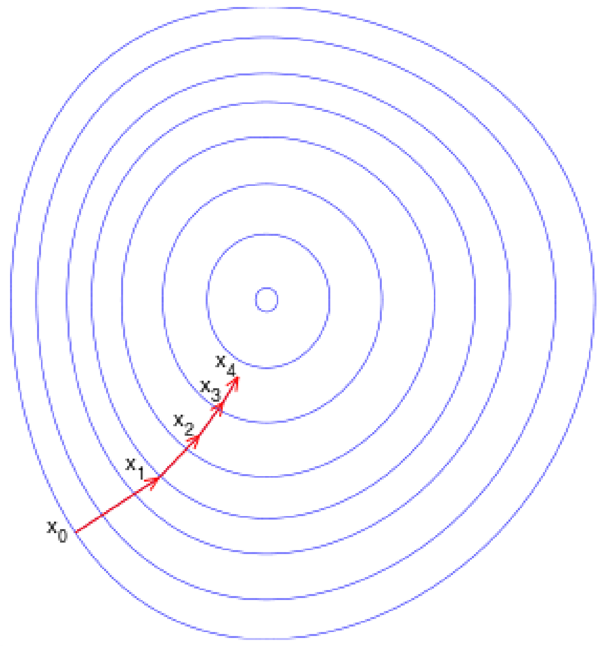


o the optimizer: Gradient Descent

- Gradient Descent is an iterative algorithm for searching for minimal points
- Let $F(\bar{x})$ be a multivariate and differentiable in a neighborhood of a point $\bar{a}$
  - $F(\bar{x})$ decreases *fastest* if one goes from $\bar{a}$ in the direction of the negative gradient
  - The algorithm is: update $\bar{a}$ until convergence:

$$\bar{a}^{new} = \bar{a}^{old} - \eta \nabla F\left(\bar{a}^{old}\right)$$

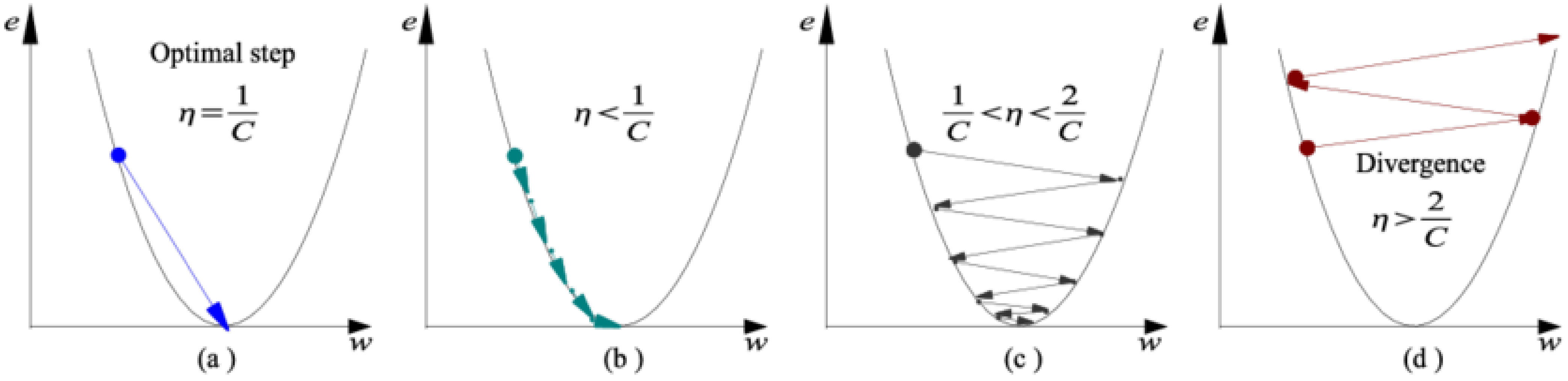
The parameter η is called learning rate and determines the behavior of the optimization



o the optimizer: Gradient Descent

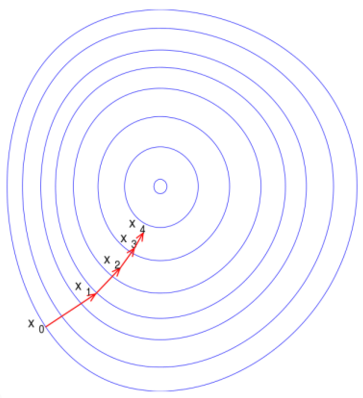
- Possible behaviors according η
  - Let assume that our loss function is:

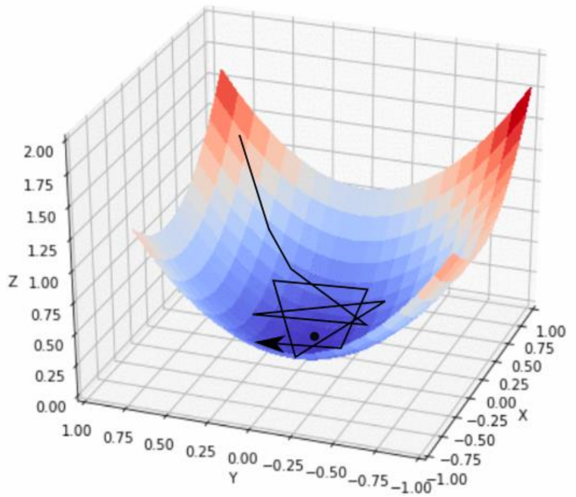
$$e(w) = \frac{1}{2} \cdot C \cdot w^2 \qquad (where\ C\ is\ a\ constant)$$

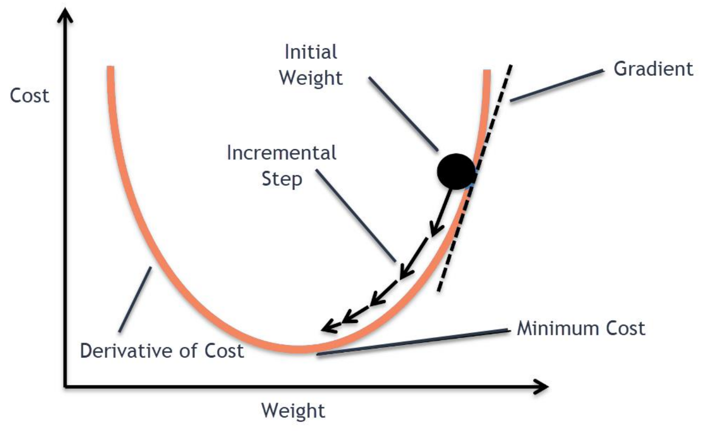


o the optimizer : Gradient Descent

- A fix point procedure:
  - Start from a random point of the function to optimize
  - Compute the gradient on that point
  - Follow the gradient to find another point of the function that is closer to the optimum
  - Repeat until convergence







o the optimizer : Gradient Descent

$$\overline{\boldsymbol{a}}^{\boldsymbol{new}} = \overline{\boldsymbol{a}}^{\boldsymbol{old}} - \boldsymbol{\eta}\nabla\boldsymbol{F}\left(\overline{\boldsymbol{a}}^{\boldsymbol{old}}\right)$$

$$[W,B]_{t+1} = [W,B]_t - \eta\frac{1}{n}\sum_{i=1}^{n}\nabla_{W,B}loss[\vec{y}_i, g(\vec{x}_i|W,B)]$$

where:

- $[W,B]$ *is the concatenation of the unknown parameters*
- $\nabla_{W,B}$ *is the gradient operator*
- t *is the current step*
- η *is the learning rate*

o the optimizer: Gradient Descent

- Typical rewriting

$$\vec{z}_k = f_k\left(\vec{b}_k + W_k \vec{z}_h\right)$$

$$W_k^* = \begin{bmatrix}\vec{b}_k & W_k\end{bmatrix} \text{ and } \vec{z}_h^* = \begin{bmatrix}1 \\ \vec{z}_h\end{bmatrix}$$

$$W_k^* \vec{z}_h^* = \vec{b}_k + W_k \vec{z}_h$$

$$\Downarrow$$

$$W_{t+1}^* = W_t^* - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla loss[\vec{y}_i, g(\vec{x}_i | W^*)]$$

o the optimizer: Computing derivatives

- We have also singular loss function…
- …but, singularities are very few…
  - The probability of reaching exactly the singularities is practically zero
- …and, in any case, we can compute an approximation of the gradient

$$\frac{d}{dx}f(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} = \lim_{h\to 0}\frac{f(x+h)-f(x-h)}{2h}$$

o the optimizer: Stochastic Approach

- Let’s speed up the convergence:

$$\boldsymbol{W^*_{t+1} = W^*_t - \eta \frac{1}{n}\sum_{i=1}^{n} \nabla loss[\vec{y}_i, g(\vec{x}_i|W^*)]}$$

  - Stochastic Gradient Descent (SGD)

$$\forall i \in \{1, \dots, n\}: W^*_{t+1} = W^*_t - \eta \nabla loss[\vec{y}_i, g(\vec{x}_i|W^*)]$$

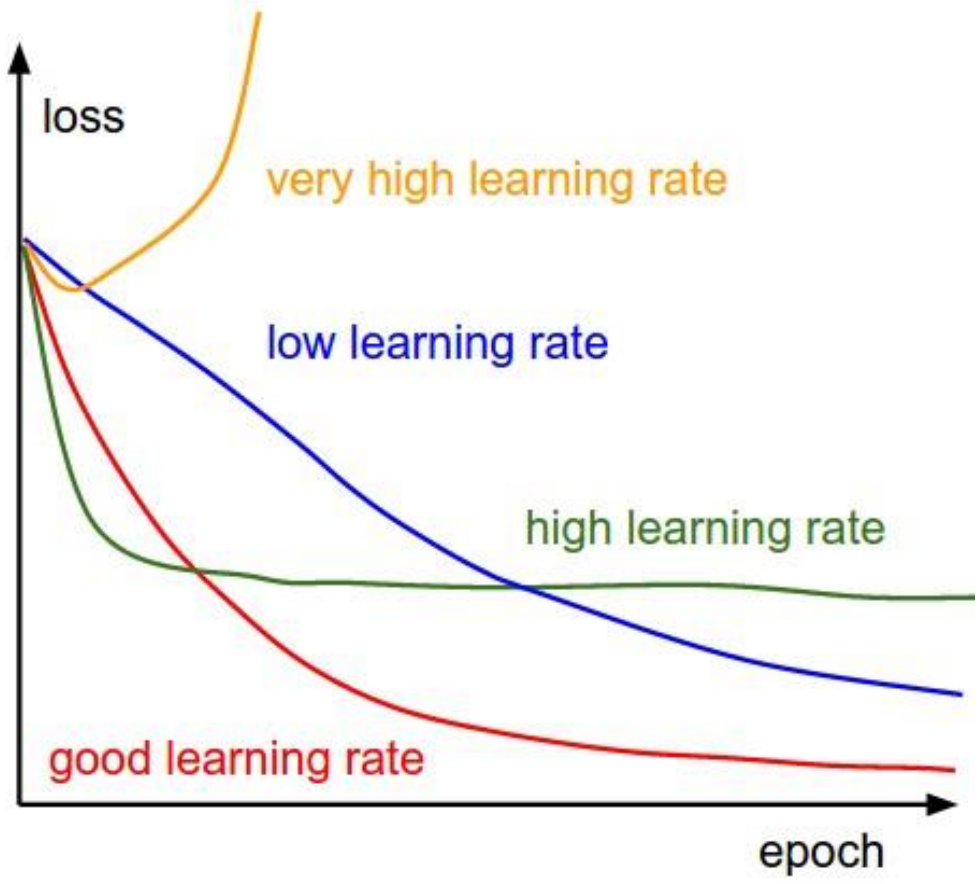
or

$$\forall i \in \{1, \dots, n\}: W^*_{t+1} = W^*_t - \eta \nabla l_i$$

    - The weight update is performed for each entry (or for a batch of them)
    - Where $\vec{l}_i = loss[\vec{y}_i, g(\vec{x}_i|W^*)]$

o the optimizer

- Guideline on learning rate



(Source: Stanford class CS231n,
MIT License, Image credit: Alec Radford)

o the optimizer

- Stochastic Gradient Descent Variants
  - **AdaGrad**

$$W_{t+1}^{*} = W_{t}^{*} - \frac{\eta}{\sqrt{\epsilon \cdot I + \mathrm{diag}\left(\nabla l_i \cdot \nabla l_i^T\right)}} \nabla l_i$$

  - Weights that receive high gradients will have their effective learning rate reduced
  - Weights that receive small or infrequent updates will have their effective learning rate increased

o the optimizer

- Stochastic Gradient Descent Variants
  - **RMSprop**

$$\zeta_{\mathrm{t+1}} = \alpha \cdot \zeta_t + (1-\alpha) \cdot (\nabla l_i)^2$$

$$W_{t+1}^{*} = W_t^{*} - \frac{\eta}{\sqrt{\epsilon \cdot I + \zeta_{t+1}}} \nabla l_i$$

  - It adjusts AdaGrad via a decreasing learning rate

o the optimizer

- Stochastic Gradient Descent Variants
  - **Adam**

$$\zeta_{t+1} = \alpha \cdot \zeta_t + (1-\alpha) \cdot (\nabla l_i)^2 \qquad \rightarrow \qquad \zeta_{t+1}^* = \frac{\zeta_{t+1}}{1-(\alpha)^{t+1}}$$

$$m_{t+1} = \beta \cdot m_t + (1-\beta) \cdot \nabla l_i \qquad \rightarrow \qquad m_{t+1}^* = \frac{m_{t+1}}{1-(\beta)^{t+1}}$$

$$W_{t+1}^* = W_t^* - \frac{\eta \cdot m_{t+1}}{\sqrt{\epsilon \cdot I + \zeta_{t+1}}} \nabla l_i$$

  - It is RMSprop with smoothing
  - The update is a function of the iteration as well as the other parameters

o the optimizer: Backpropagation

- Each layer $k \in \{1, \dots, K\}$ has a weight matrix (with biases) W_k^*
- The gradient can be expressed as follows:

$$\nabla loss\big(\vec{y}_i, g(\vec{x}_i | W^*)\big) = \nabla loss\left(\vec{y}_i, f_K\left(W_K^* \cdot f_{K-1}\left(W_{K-1}^* \cdot f_{K-2}\big(W_{K-2}^* \cdot f_{K-3}(\dots)\big)\right)\right)\right)$$

o the optimizer: Backpropagation

- By the chain rule of derivatives, we know:

$$\frac{d}{dx} f_1(f_2(x)) = f_1'\big(f_2(x)\big) \cdot f_2'(x)$$

- So:

$$\nabla loss\big(\vec{y}_i, g(\vec{x}_i | W^*)\big)$$
$$= loss'\big(\vec{y}_i, f_K(\dots)\big) \cdot f_K'\big(W_K^* \cdot f_{K-1}(\dots)\big) \cdot W_K^* \cdot f_{K-1}'\big(W_{K-1}^* \cdot f_{K-2}(\dots)\big) \cdot W_{K-1}^* \cdot f_{K-2}'\big(W_{K-2}^* \cdot f_{K-3}(\dots)\big) \cdot \dots$$

o the optimizer: Backpropagation

- By the chain rule of derivatives, we know:

$$\frac{d}{dx} f_1(f_2(x)) = f_1'\left(f_2(x)\right) \cdot f_2'(x)$$

- So:

$$\nabla loss\left(\vec{y}_i, g(\vec{x}_i | W^*)\right)$$
$$= loss'\left(\vec{y}_i, f_K(\dots)\right) \cdot f_K'\left(W_K^* \cdot f_{K-1}(\dots)\right) \cdot W_K^* \cdot f_{K-1}'\left(W_{K-1}^* \cdot f_{K-2}(\dots)\right) \cdot W_{K-1}^* \cdot f_{K-2}'\left(W_{K-2}^* \cdot f_{K-3}(\dots)\right) \cdot \dots$$

o the optimizer: Backpropagation

- We can distribute the gradient on the layers
  - Each layer k contributes to the gradient:

$$\delta_k \stackrel{\text{def}}{=} f_k'(\dots) \cdot W_{k+1}^* \cdot f_{k+1}'(\dots) \cdot \dots \cdot W_K^* \cdot f_K'(\dots) \cdot loss'\left(\vec{y}_i, f_K(\dots)\right)$$

- So:

$$\nabla loss\left(\vec{y}_i, g(\vec{x}_i | W^*)\right)$$
$$= loss'\left(\vec{y}_i, f_K(\dots)\right) \cdot f_K'\left(W_K^* \cdot f_{K-1}(\dots)\right) \cdot W_K^* \cdot f_{K-1}'\left(W_{K-1}^* \cdot f_{K-2}(\dots)\right) \cdot W_{K-1}^* \cdot f_{K-2}'\left(W_{K-2}^* \cdot f_{K-3}(\dots)\right) \cdot \dots$$

o the optimizer: Backpropagation

- We can distribute the gradient on the layers
  - Each layer k contributes to the gradient:

$$\delta_k \stackrel{\text{def}}{=} f_k'(\dots) \cdot W_{k+1}^* \cdot f_{k+1}'(\dots) \cdot \dots \cdot W_K^* \cdot f_K'(\dots) \cdot loss'\left(\vec{y}_i, f_K(\dots)\right)$$

- So:

$$\nabla loss\left(\vec{y}_i, g(\vec{x}_i|W^*)\right)$$
$$= loss'\left(\vec{y}_i, f_K(\dots)\right) \cdot f_K'\left(W_K^* \cdot f_{K-1}(\dots)\right) \cdot W_K^* \cdot f_{K-1}'\left(W_{K-1}^* \cdot f_{K-2}(\dots)\right) \cdot W_{K-1}^* \cdot f_{K-2}'\left(W_{K-2}^* \cdot f_{K-3}(\dots)\right) \cdot \dots$$

$$\delta_{k-1} = f_{k-1}'(\dots) \cdot W_k^* \cdot \delta_k$$

o the optimizer: Backpropagation

- Each layer can be updated backwardly:

$$W^*_{k;t+1} = W^*_{k;t} - \eta\delta_k \cdot W^*_k \cdot f'_{k-1}(\dots)$$

UNIVERSITÀ DELLA CALABRIA

Basic Ingredients: Initialization

Gianluigi Greco

i the initialization

- In Gradient Descent, edges weights and biases need an **initial value**
- The initialization strategy may **strongly** change the network behavior
- Since we are searching for optimal solution, the **starting point** of the fix point procedure (Gradient Descent) is **crucial**

i the initialization

- Zero initialization
  - Bad solution
    - All the nodes have the same initial gradient
    - There is no diversification of the nodes
      - Hidden nodes becomes symmetric

- Constant initialization?
  - It has the same problems

i the initialization

- Rationale:
  - If weights are initialized with very high values, asymptotical activation functions (e.g. sigmoid, tanh) produce a gradient that is practically equal to 0
    - Low gradient → learning takes a lot of time
  - If weights are initialized with low values, the gradient goes to 0, which is the case is the same as before

i the initialization

- Simplest initialization
  - Random: $W_k \sim \mathrm{Uniform}()$
  - Random: $W_k \sim N(0, I)$

- (He et al., 2015) initialization
  - The weights of a layer h are:

$$W_k = \epsilon \cdot \sqrt{\frac{2}{|\rightarrow k|}} \quad \text{with} \quad \epsilon \sim N(0, I)$$

  - $\rightarrow k$ is the set of all incoming connections (fan-in)

i the initialization

- Xavier initialization (aka Glorot)
  - Similar to He et al. initialization

$$W_k = \epsilon \cdot \sqrt{\frac{2}{|h| + |k|}} \quad \text{with} \quad \epsilon \sim N(0, I)$$

  - With the variant

$$W_k = \epsilon \cdot \sqrt{\frac{6}{|h| + |k|}} \quad \text{with} \quad \epsilon \sim \text{Uniform}()$$

  - Where h is the previous layer connected to k