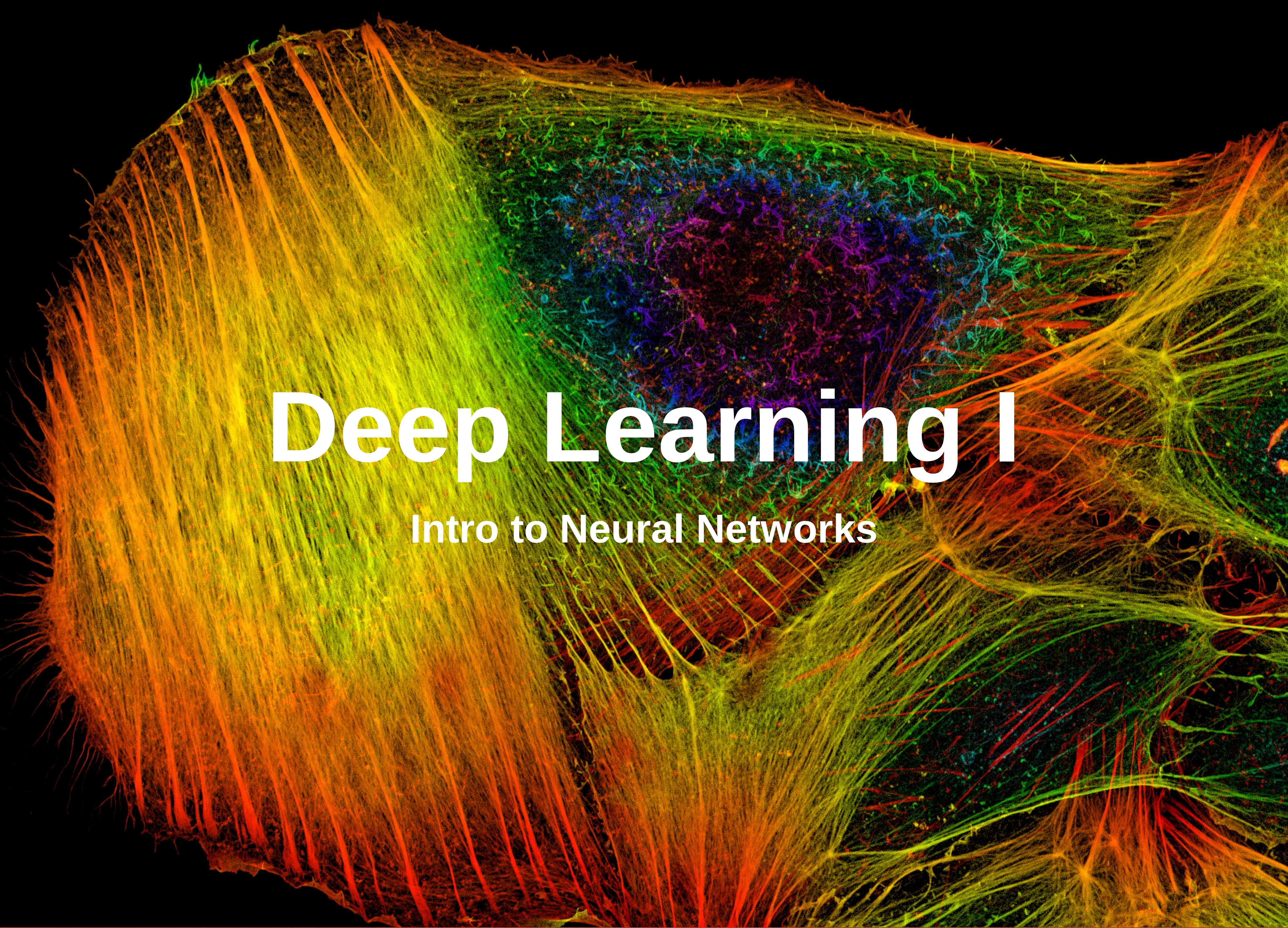
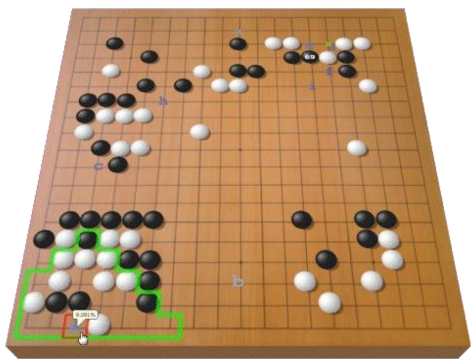


Deep Learning I

Intro to Neural Networks

Andrea Azzini, Giovanni Conserva

Today, Deep Learning
is everywhere…
What makes it so effective?



Example from computer vision: traditional approach

Start from raw data (image pixels)

Preprocess the image with handcrafted features (i.e. edges)

Run a learning algorithm (i.e. SVM) based on the new features, to perform tasks such classification, segmentation etc.

Hand crafted features

Pros:

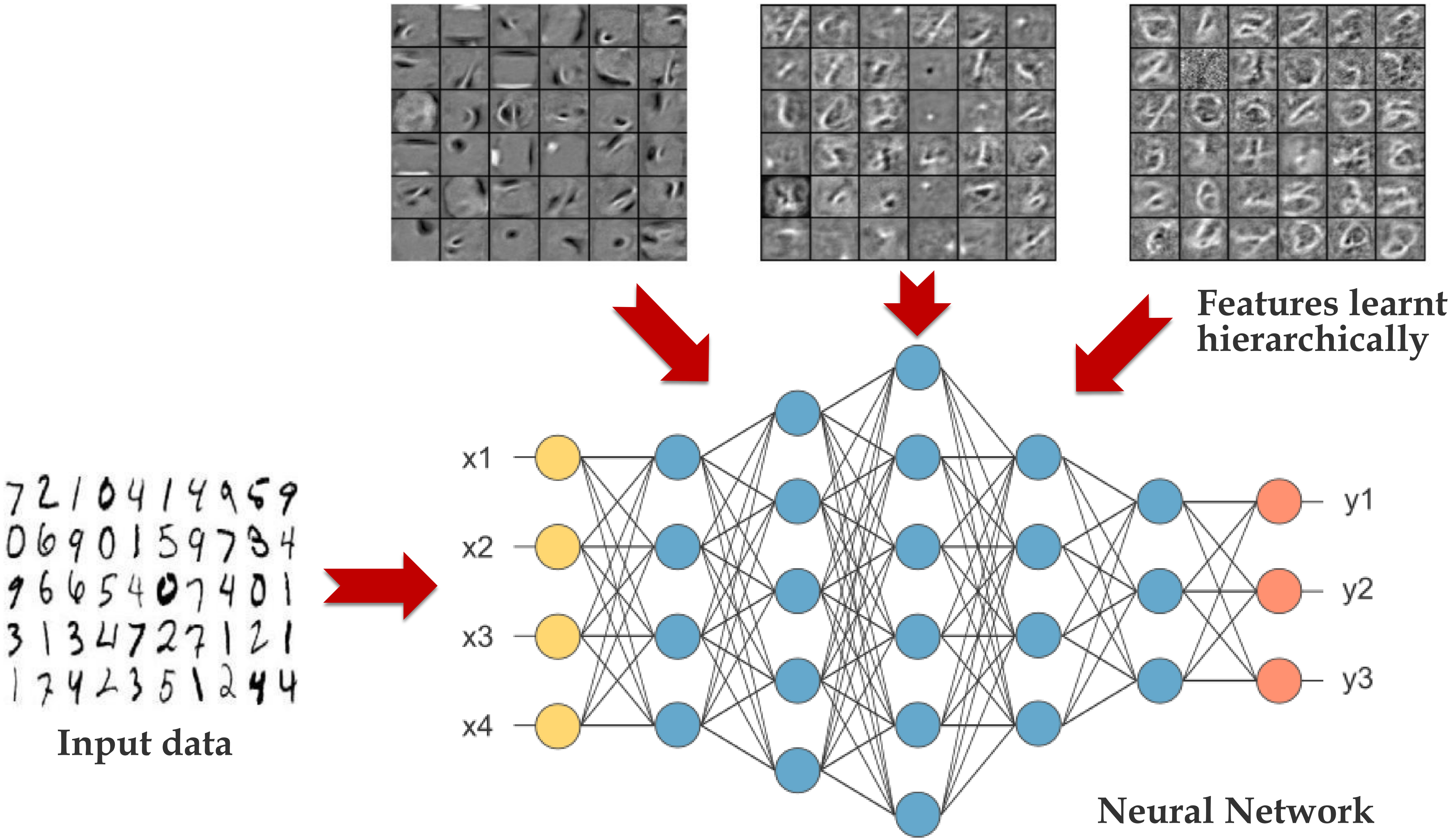
- Don’t need huge datasets to devise them (human intellect)
- Don’t need computational power

Cons:

- In many contexts, features are hard to hand-engineer
- Bad performance and bad context generalization

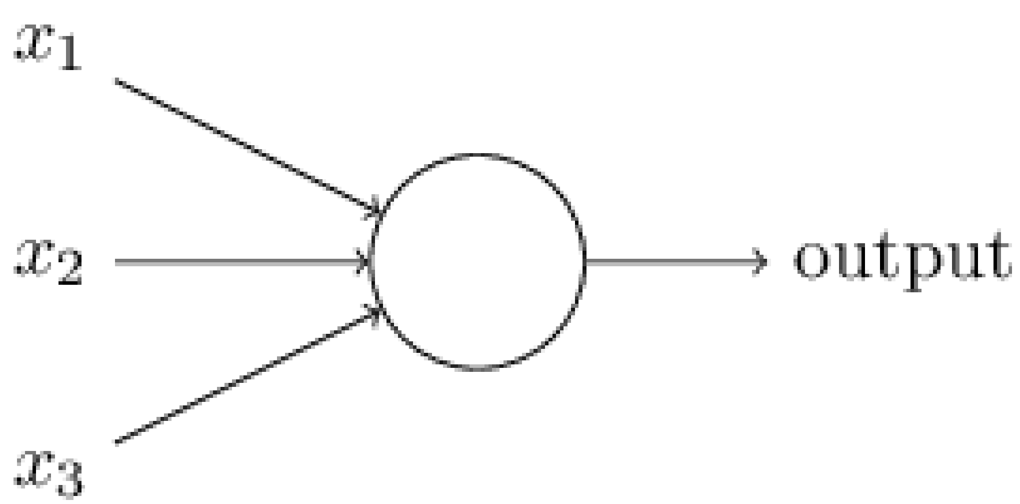
Alternative: **learn features from data!**

How do we design an architecture able to learn useful features?



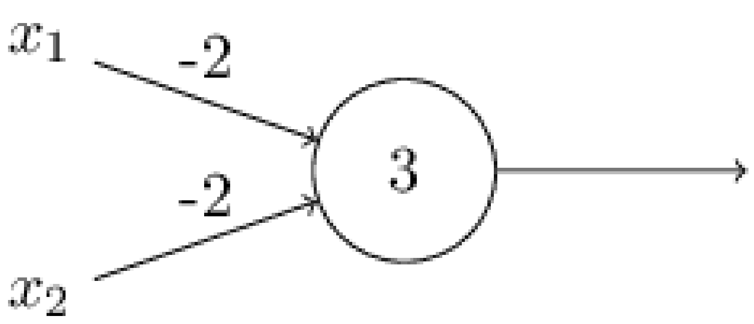
Basic component: the Perceptron

- Inspired by **human brain**
- Can implement **any logical function**
- The **«activation function»** shown here tells us whether the neuron is «firing». The neuron’s output is the input of its connected neurons



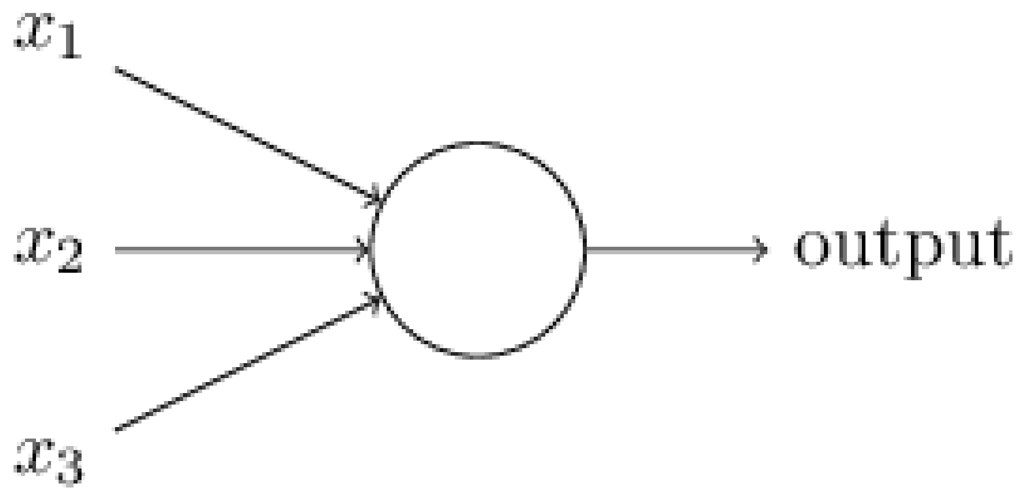
$$\begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Example: NAND



INPUTS		OUT
X	Y	output
0	0	1
0	1	1
1	0	1
1	1	0

From the Perceptron to the Sigmoid Neuron



$$\begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

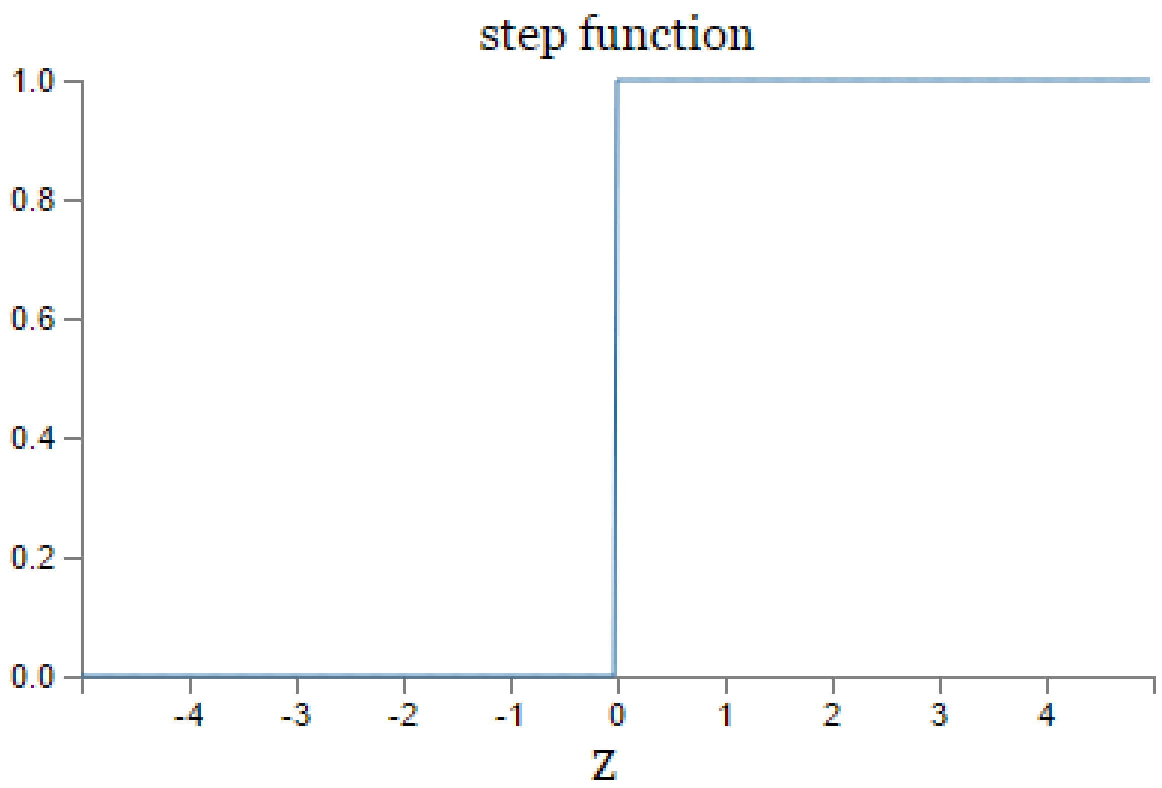
Given that the output of the perceptron is **boolean**, its activation function is **not differentiable**. Since an effective learning algorithm is based on Gradient Descent, which works on derivatives, we need another function to tell us how a neuron should fire. We choose the **Sigmoid function:**

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \qquad z \equiv w \cdot x + b$$

Perceptron vs Sigmoid Neuron

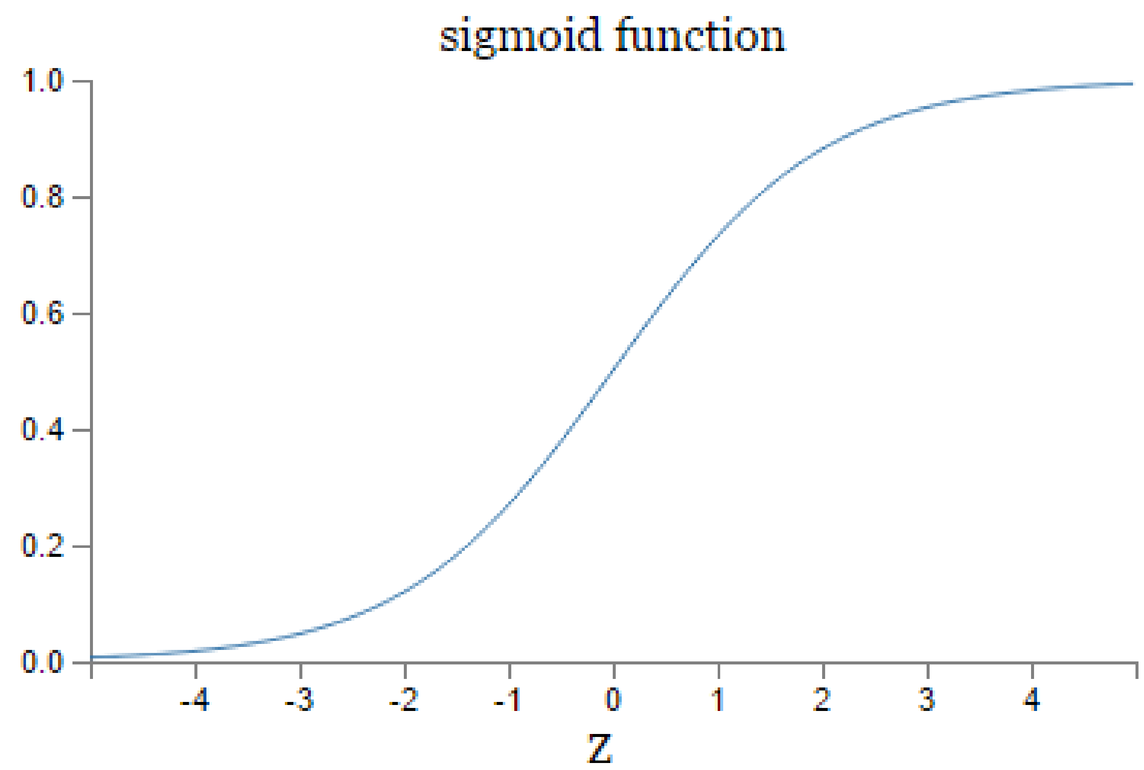
Not differentiable

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$



Differentiable

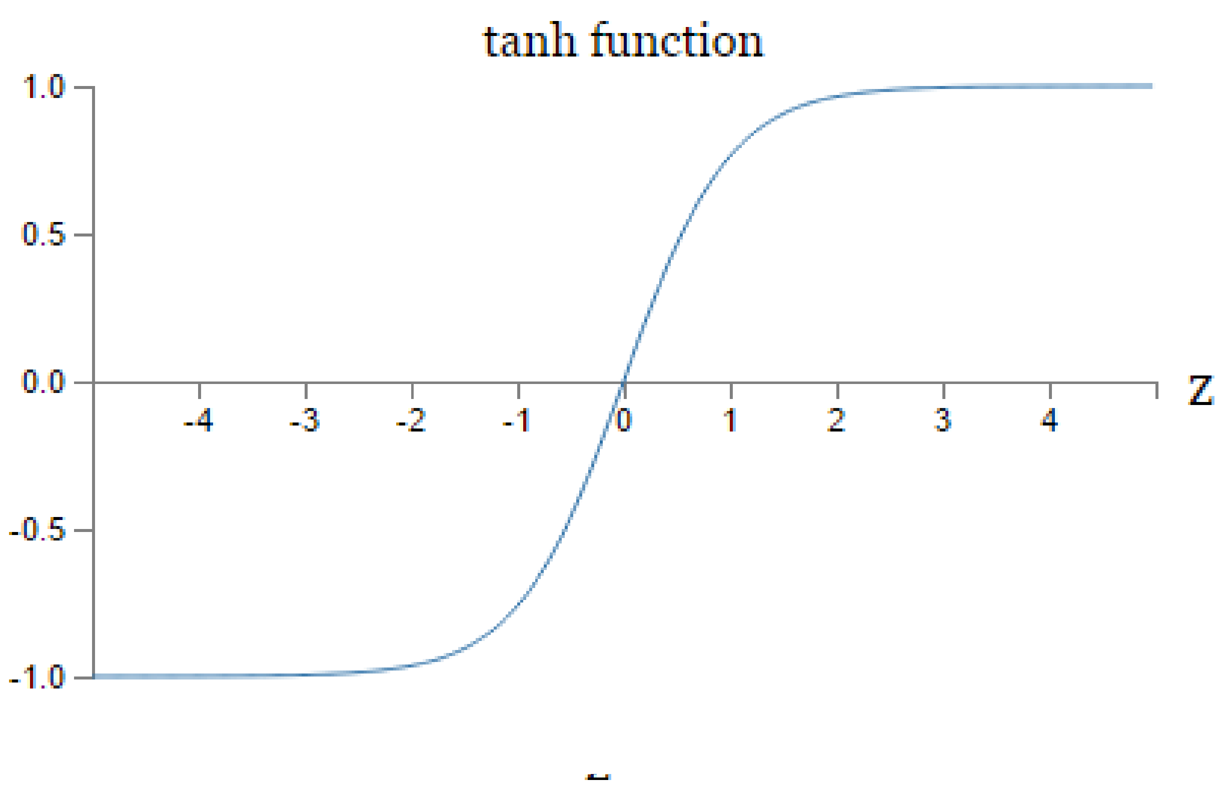
$$\text{output} = \frac{1}{1 + \exp(-\sum_j w_j x_j - b)}$$



Other activation functions

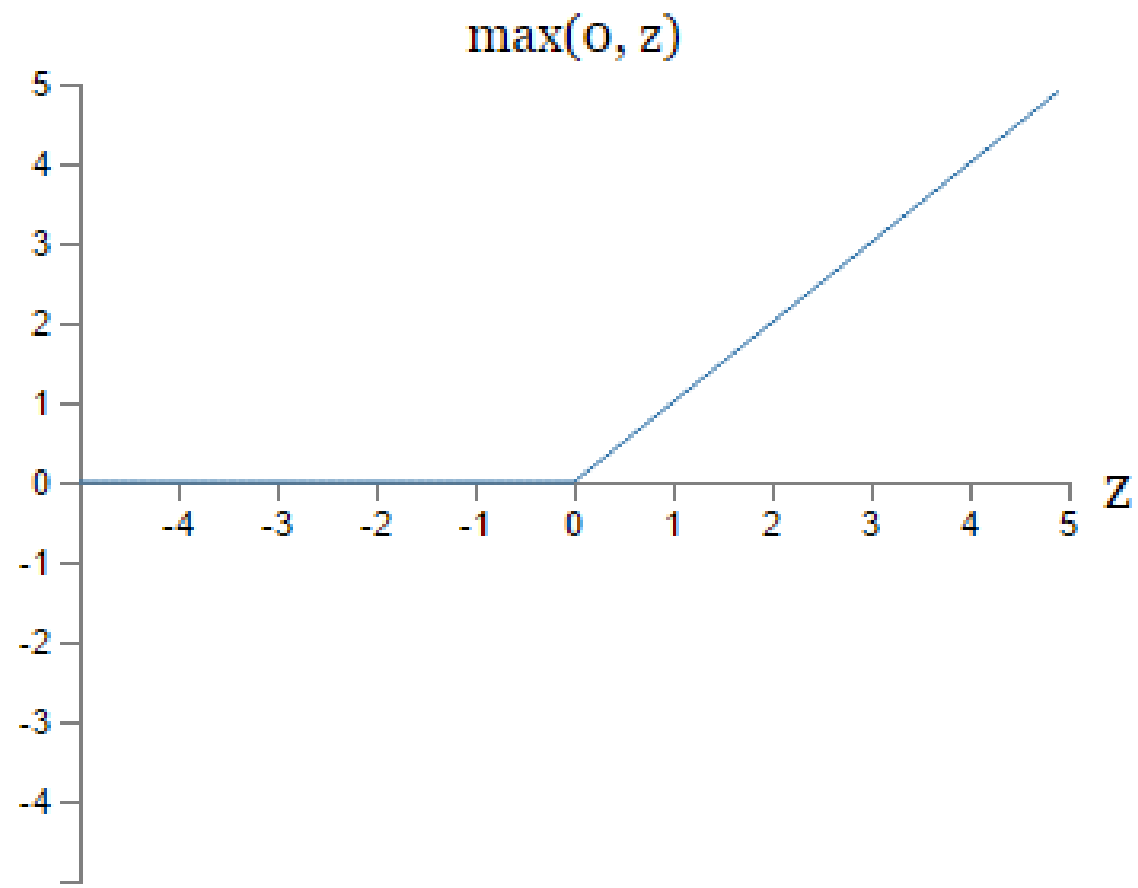
Tanh

$$\text{output} = \tanh(w \cdot x + b)$$



Rectified linear units (RELU)

$$\text{output} = \max(0, w \cdot x + b)$$



We have seen the **basic components** of a neural network. Now we see how neurons **combine** together to form an actual network

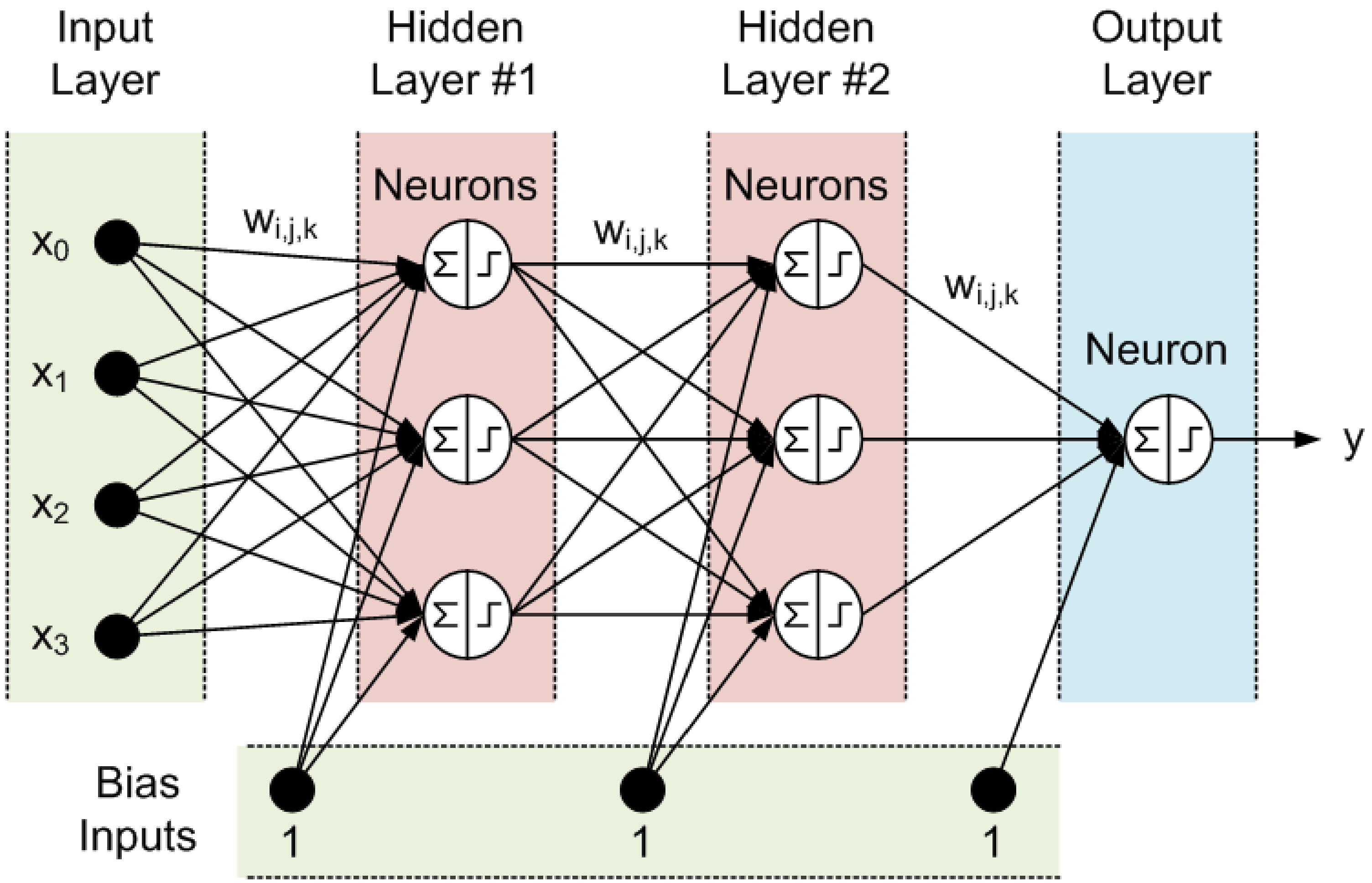
Architecture of a Neural Network

Input and output layers are shaped **based on the problem's constraints**

Example of input layers: 224*224*3 neurons for a 224*224 RGB pixel images

Example of output layer: 10 neurons for a 10 class classification problem

Hidden layers need to be designed according to the **computational power** available, overfitting issues and by chosing the **best architecture** for the given problem (CNN, RNN, FC, etc.)



Now we need a **loss function**, to evaluate how well the model is performing and to make it learn. Which one should we choose?

Quadratic loss function

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

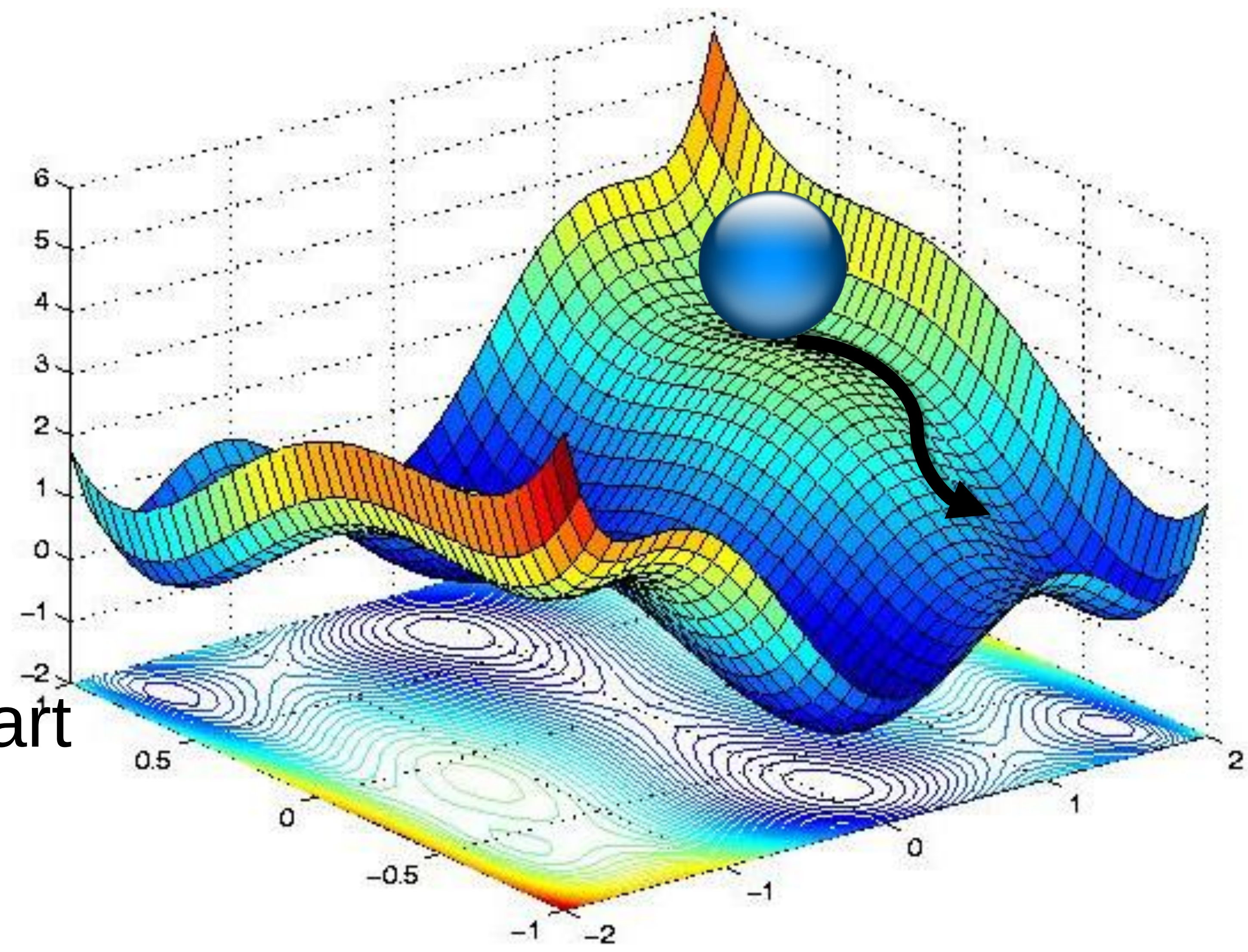
The sum is over all the elements in the training set.
«a» is the value predicted based on the last layer activations, while «y(x)» is its actual corresponding real value.
Our goal is to change the network parameters in order to minimize the difference between the predicted and actual values.
We do so through the **Gradient Descent algorithm**

Toward learning: Gradient Descent

Goal: Find the minimum of the loss function, wrt weights and biases

Problem: Find the global minimum analitically is computationally expensive

Solution: Use an heuristic. Start from a random point (random initialization) and move toward the decreasing direction, until you reach a local minimum



Toward learning: Gradient Descent

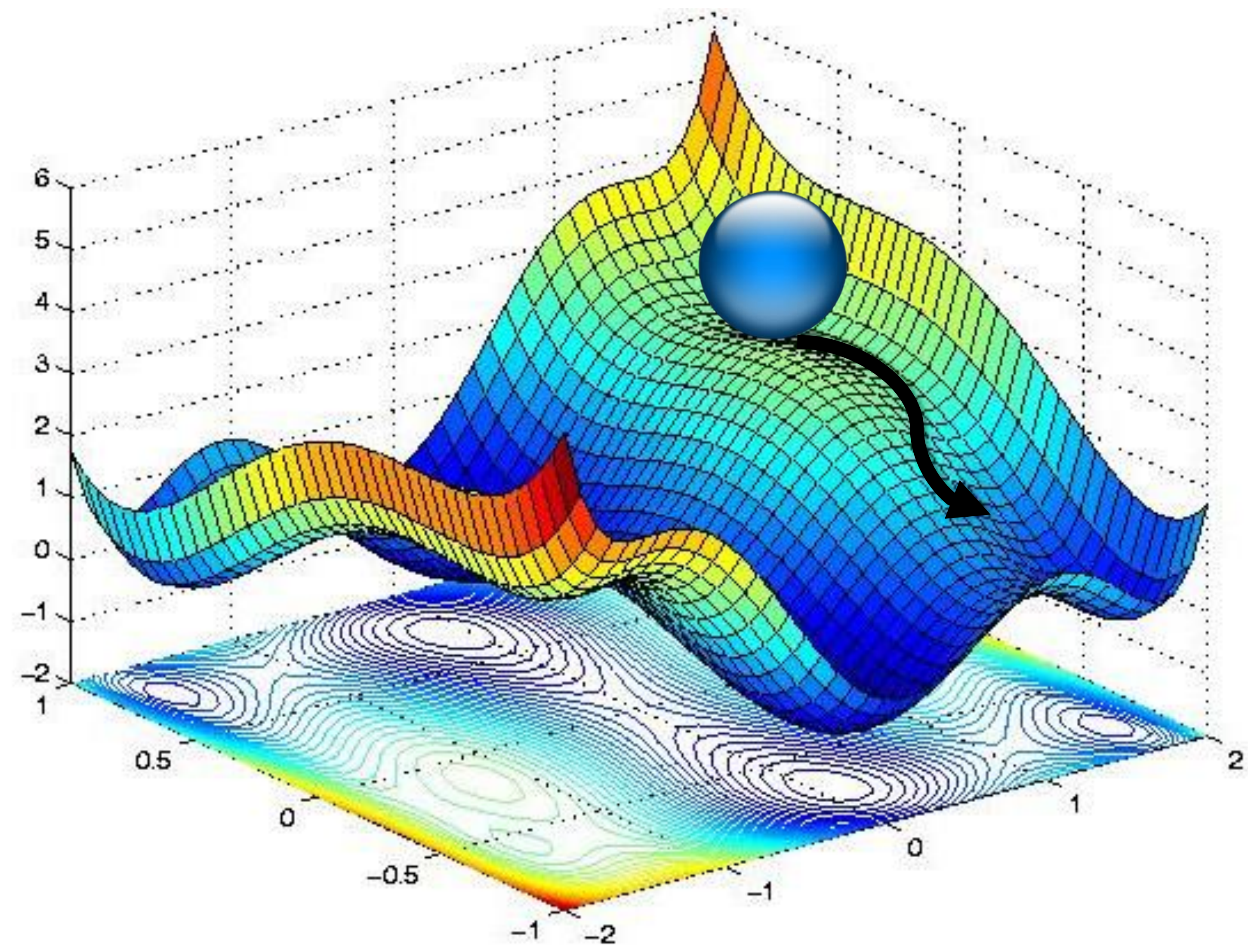
The **decreasing direction** is given by the derivatives wrt weights and biases:

$$\frac{\partial C}{\partial w_k} \qquad \frac{\partial C}{\partial b_l}$$

At each **iteration**, we move in that direction by a step η :

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$



To make Gradient Descent work in a Neural Networks, we need only a last ingredient: a way to find the partial derivatives:

$$\frac{\partial C}{\partial w_k} \qquad \frac{\partial C}{\partial b_l}$$

This is done through the **Backpropagation algorithm**

The partial derivatives that we need can be expressed in term of a quantity called **«error»**:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

Where z is the **weighted sum:**

$$z^l \equiv w^l a^{l-1} + b^l$$

Given so, Backpropagation is based on 4 equations:

- 2 equations for computing the **«error»** from quantities available in the forward pass
- 2 equations for computing the **gradient** from the **«error»**

Toward learning: Backpropagation

Compute the error from the input

BP 1: Error in the output (final) layer

$$\boxed{\delta^L = \nabla_a C \odot \sigma'(z^L).}$$

BP 2: Error in a layer as a function of the error in its next layer:

$$\boxed{\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)}$$

Compute the gradient from the error

BP 3: Rate of change of the error wrt to any weight

$$\boxed{\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j}$$

BP 4: Rate of change of the error wrt to any bias

$$\boxed{\frac{\partial C}{\partial b^l_j} = \delta^l_j}$$

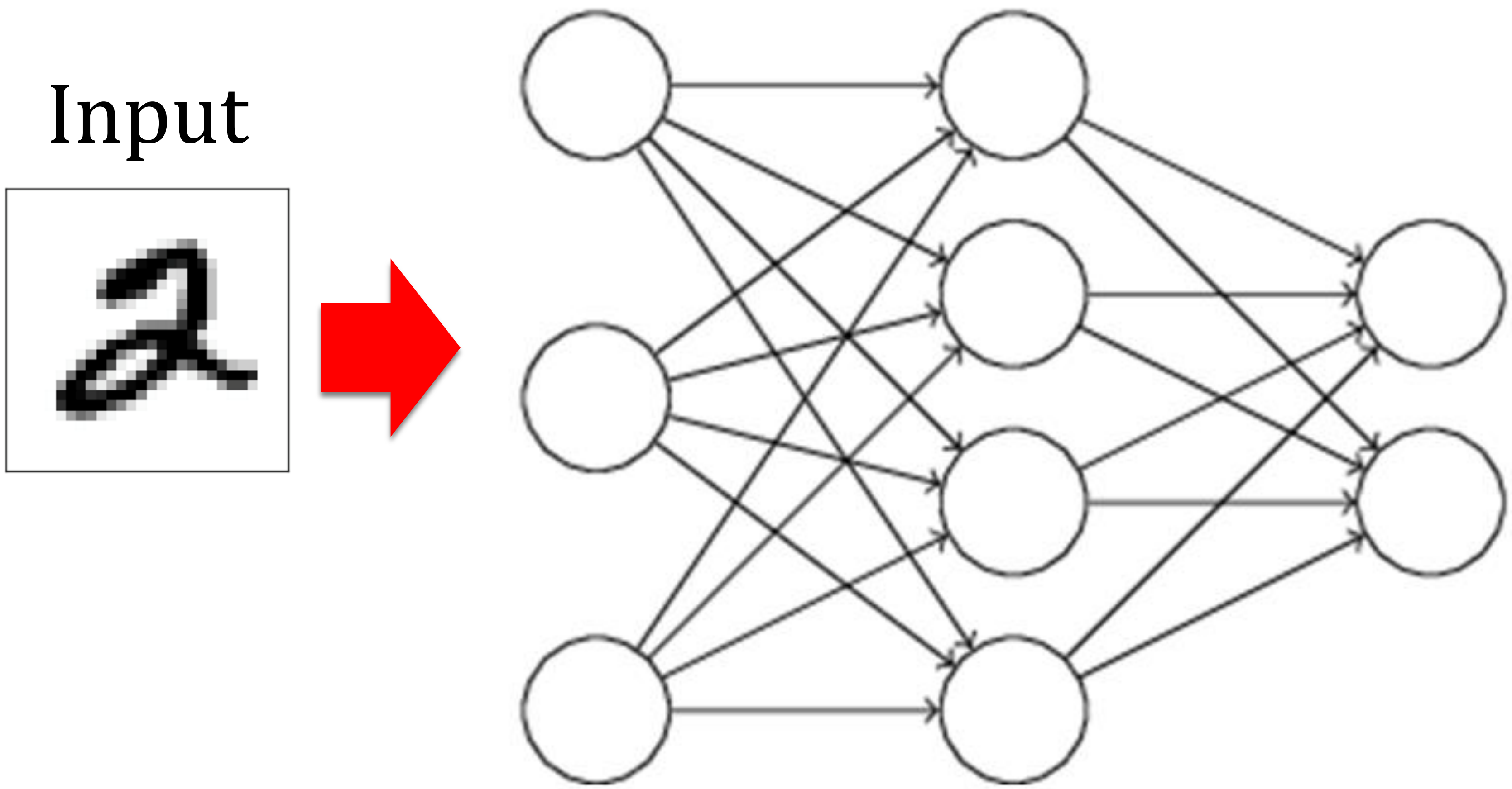
Too many equations?
Let’s grasp them through an intuitive visualization of a **forward** and **backward** pass

Forward pass / input

	layer 1	*layer2*	*layer F*
δ	?	?	?
a / z	f (input)	?	?

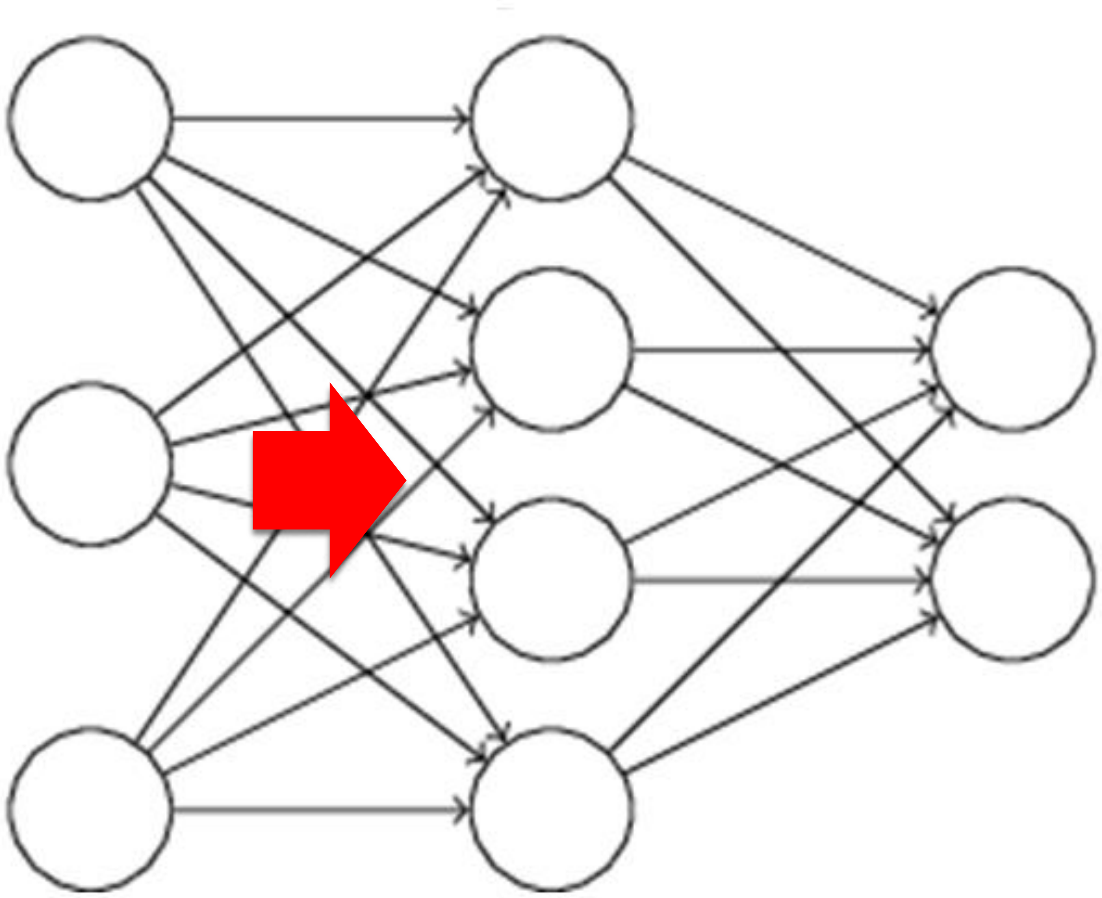
$$z^l \equiv w^l a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

Input



Forward pass / a1 to a2

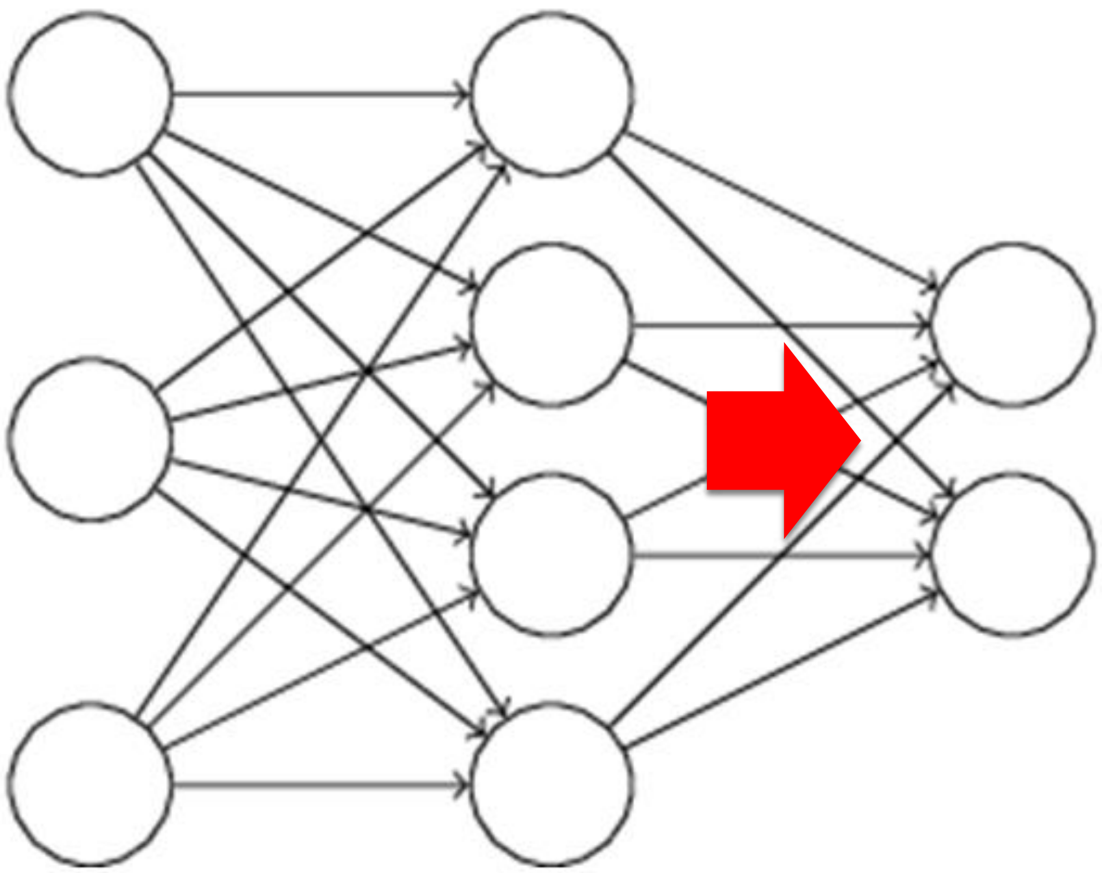
	layer 1	*layer*2	*layer F*
δ	?	?	?
a / z	f (input)	f(a1)	?



$$z^l \equiv w^l a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

Forward pass / a2 to F

	layer 1	*layer2*	*layer F*
δ	?	?	?
a / z	f (input)	f(a1)	f(a2)



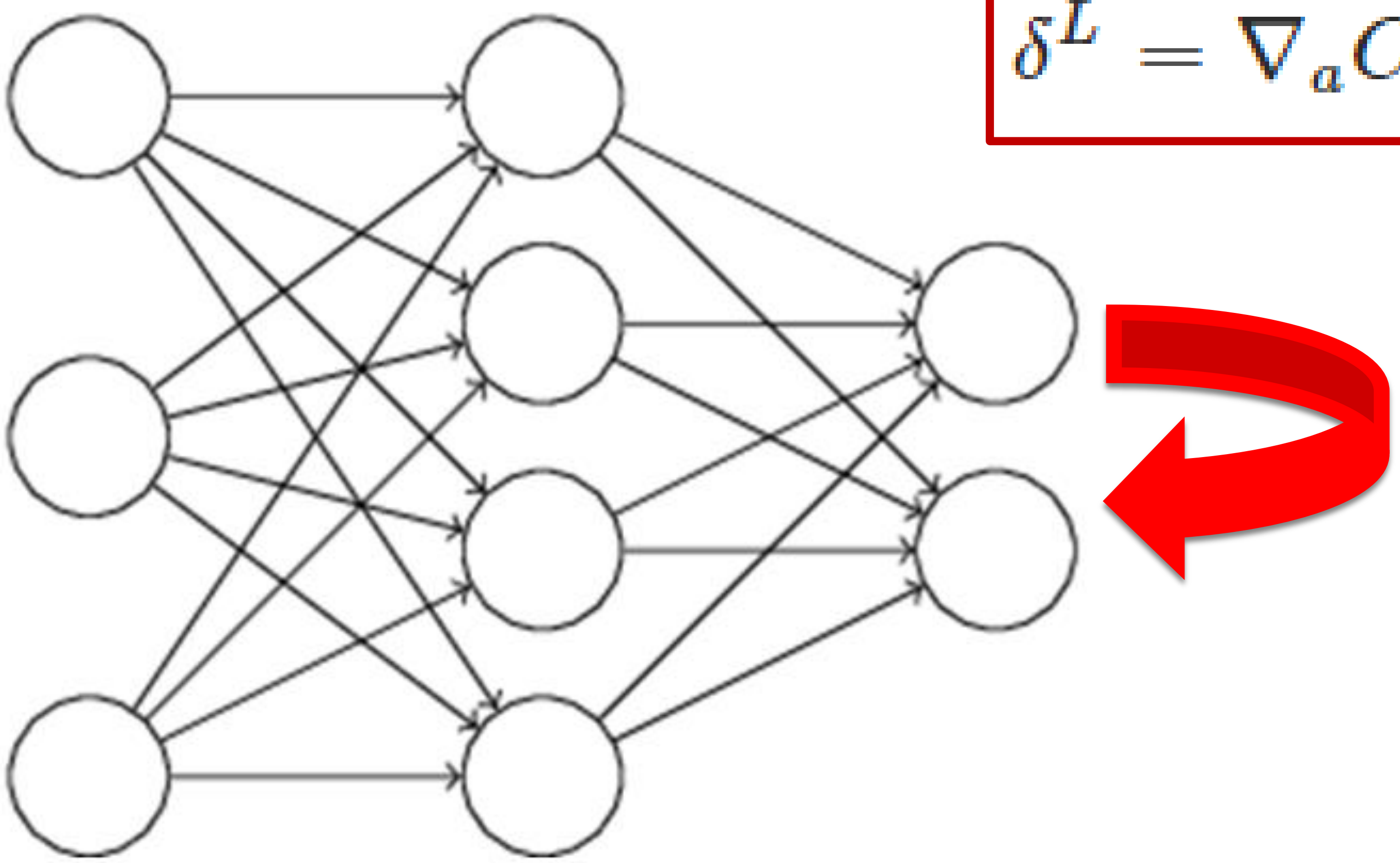
$$z^l \equiv w^l a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

Backward pass / af to δf

	$layer\ 1$	$layer2$	$layer\ F$
δ	?	?	BP1
a / z	f (input)	f(a1)	f(a2)

BP1

$$\boxed{\delta^L = \nabla_a C \odot \sigma'(z^L).}$$

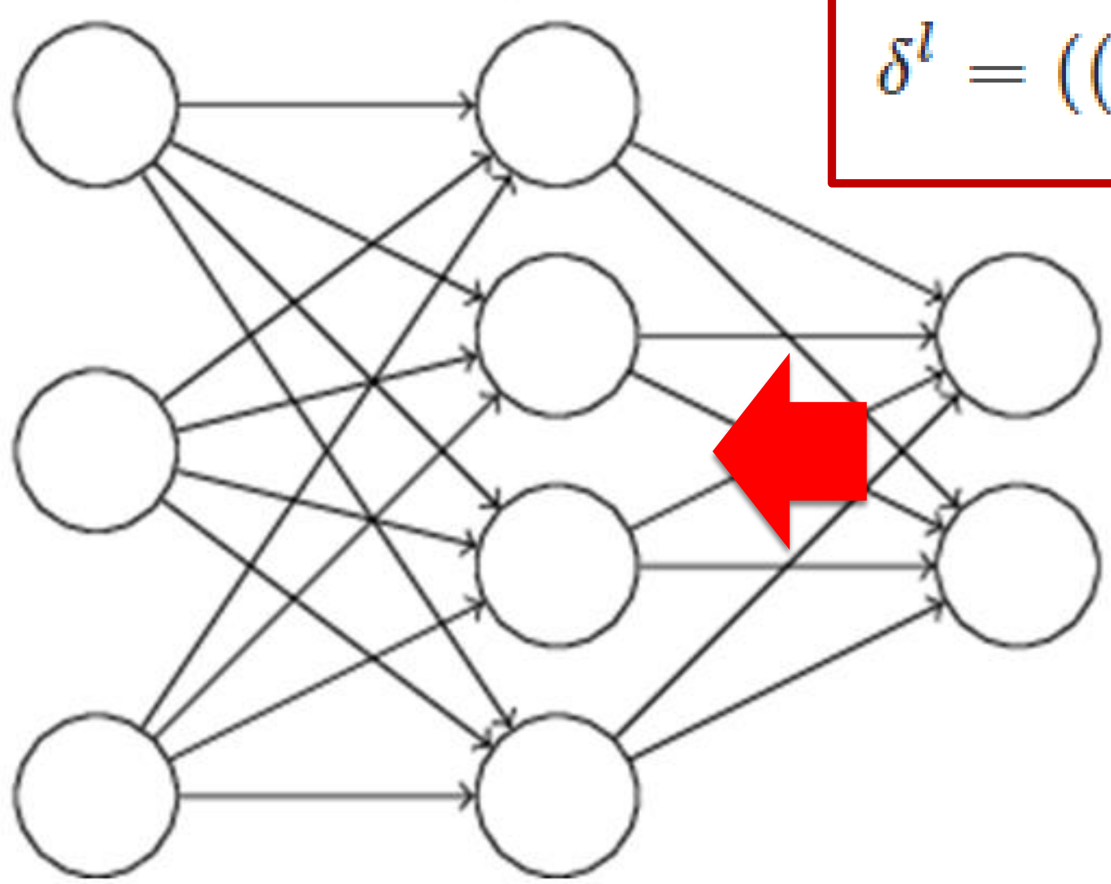


Backward pass / $\delta f \ to \ \delta 2$

	$layer\ 1$	$layer2$	$layer\ F$
δ	?	BP2 / f(aF)	BP1 / f(aF)
a / z	f (input)	f(a1)	f(a1)

BP2

$$\boxed{\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)}$$

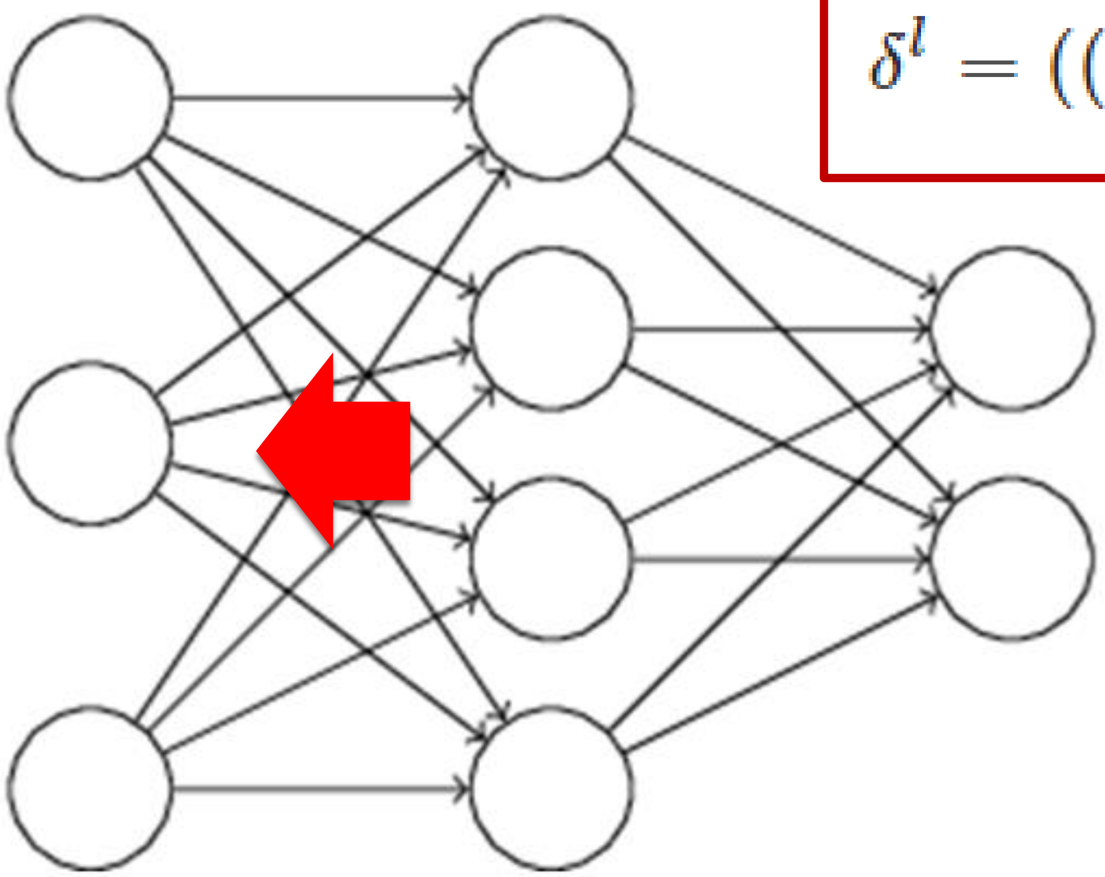


Backward pass / $\delta 2\ to\ \delta 1$

	$layer\ 1$	$layer2$	$layer\ F$
δ	BP2 / f(aF)	BP2 / f(aF)	BP1 / f(aF)
a / z	f (input)	f(a1)	f(a2)

BP2

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$



Updating the gradient

	$layer\ 1$	$layer2$	$layer\ F$
δ	BP2 / f(aF)	BP2 / f(aF)	BP1 / f(aF)
a / z	f (input)	f(a1)	f(a2)

BP3

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

$\frac{\partial C}{\partial w_k}$	f (δ1)	f (δ2)	f (δ3)
$\frac{\partial C}{\partial b_l}$	f (δ1)	f (δ2)	f (δ3)

BP4

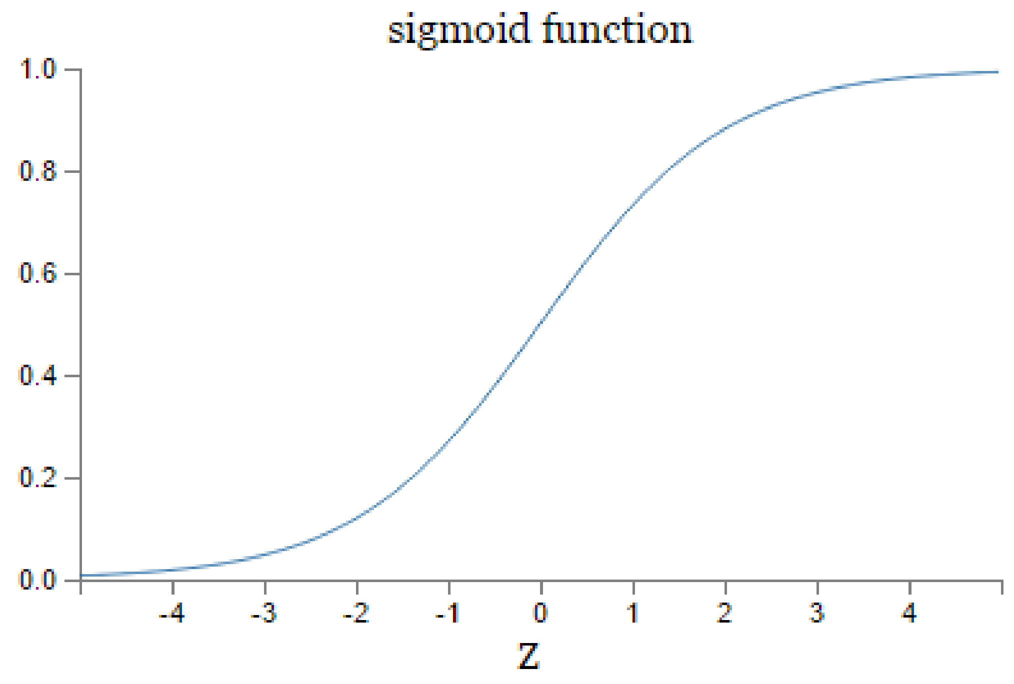
$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Issues with the Quadratic loss function

Depending on the inizialization, the network might learn slowly, because it learns by changing the weight and bias at a rate determined by the partial derivatives of the loss function, ∂C/∂w∂C/∂w and ∂C/∂b

Quadratic loss:

$$C = \frac{(y-a)^2}{2}$$



From Quadratic to Cross-Entropy loss function

Solution: change the loss function!

Cross-entropy is non negative, close to 0 when the output is similar to the input.

Its derivative depends on the difference between expected and predicted value.

The larger the error, the faster the neuron will learn!

Cross entropy:

$$C = -\frac{1}{n}\sum_x \left[y \ln a + (1-y)\ln(1-a)\right]$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$$

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z) - y)$$

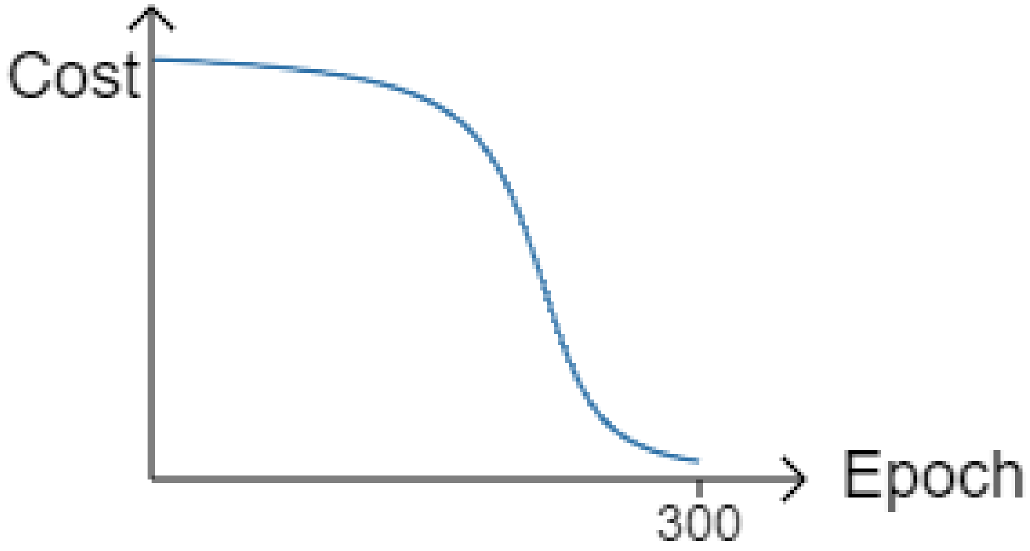
Quadratic vs Cross-Entropy

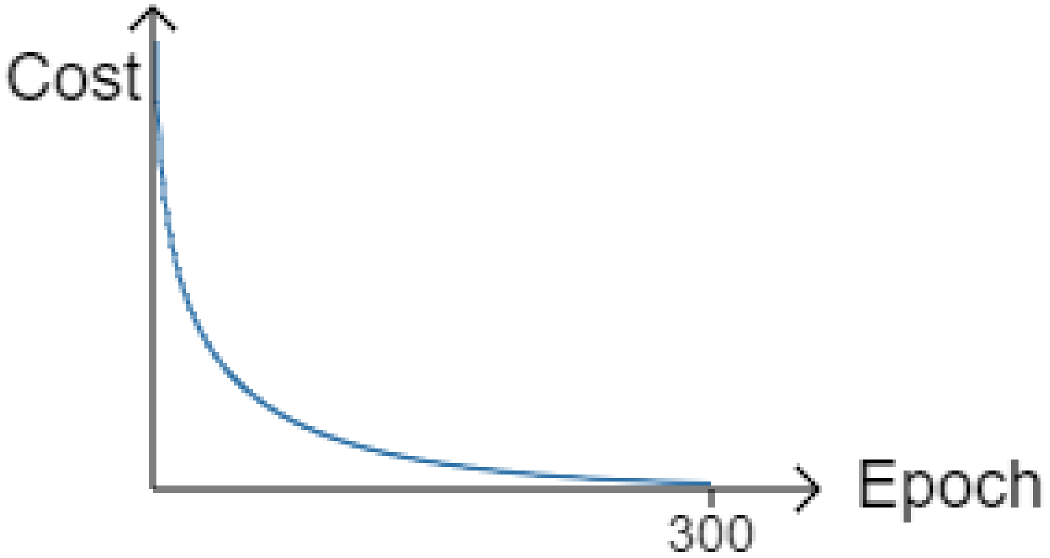
$$C = \frac{(y-a)^2}{2}$$

$$C = -\frac{1}{n}\sum_x \left[y \ln a + (1-y)\ln(1-a)\right]$$

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z)-y)$$





Enough theory.

Let's get our hands dirty with some **TensorFlow**