



Politecnico
di Torino
SmartData@PoliTO



From Zero to Neural Networks

**Data Science and Machine Learning
for Engineering Applications**

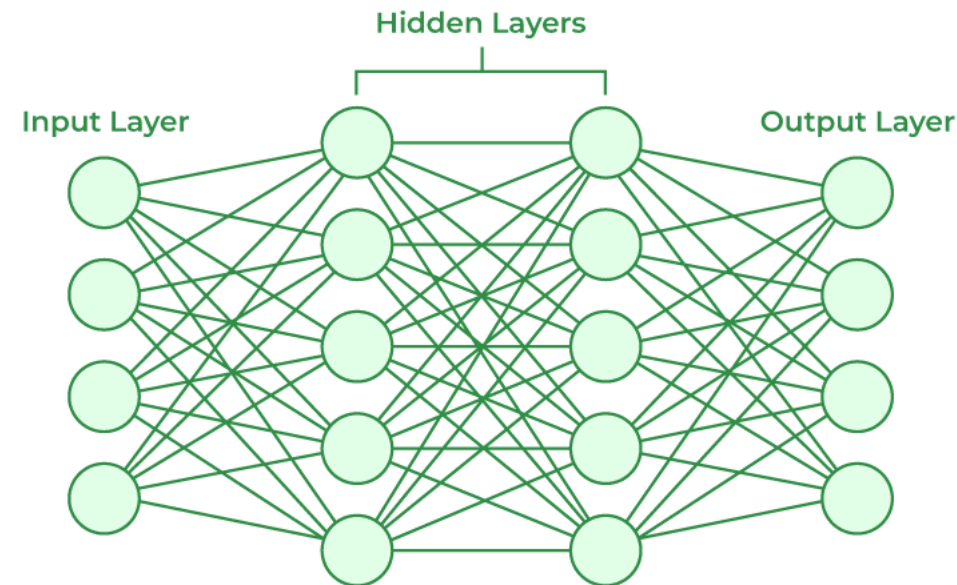
Lab 10

Giordano Paoletti

What Are Neural Networks?

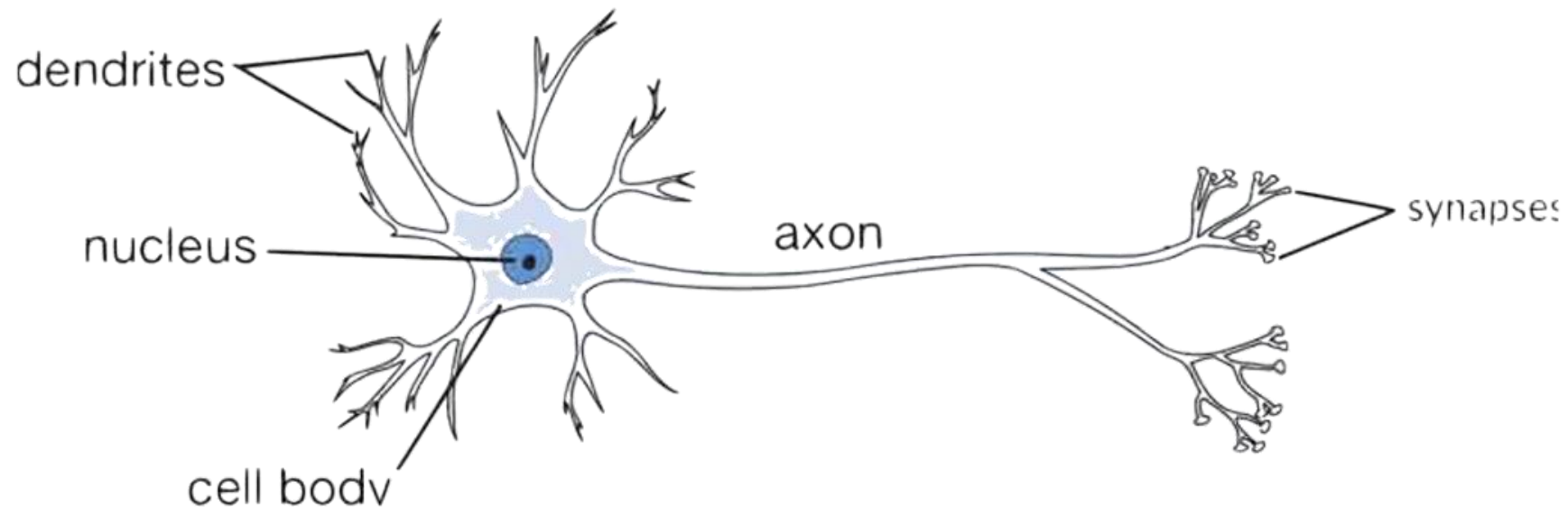


- Neural Networks are computational models inspired by the human brain, used for tasks like image recognition, translation, and more.



- Brains are made of neurons that fire when they receive signals above a threshold. Neural networks mimic this behavior.

Biological Neuron

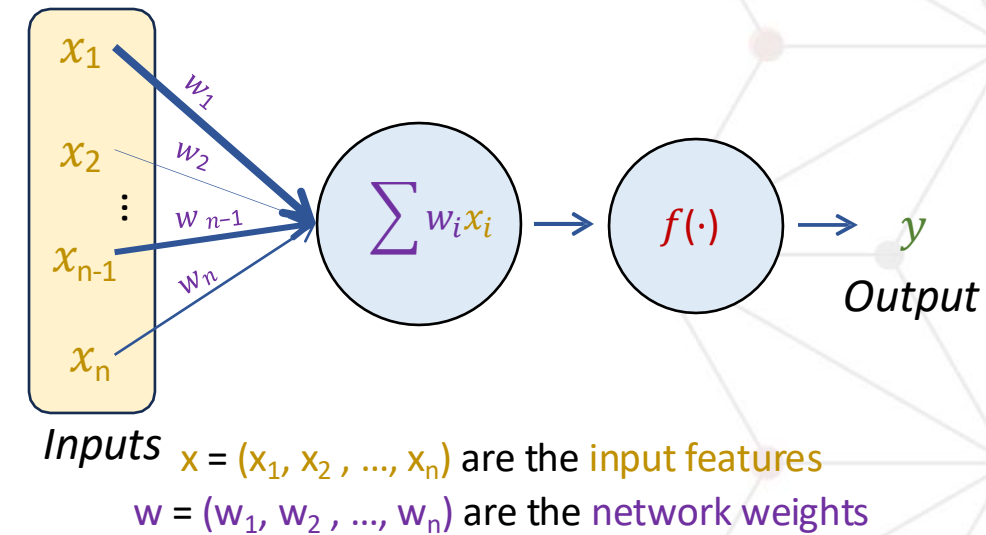


The perceptron: The Artificial Neuron



- The perceptron is the simplest unit of neural networks
- It takes an input with *multiple features*, and:
 - It weights each input *feature* with a given *weight*
 - It produces a weighted sum of the inputs, and
 - It applies an *activation function* to the weighted sum and produces an *output*

$$y = f(w_1x_1 + w_2x_2 + \dots + w_nx_n)$$



Perceptron Generalizes Classical Models



The perceptron is a linear model. Depending on its activation function:

- With *linear activation* $f(x)=x$, it behaves like **Linear Regression**:

$$y = w^t x + b$$

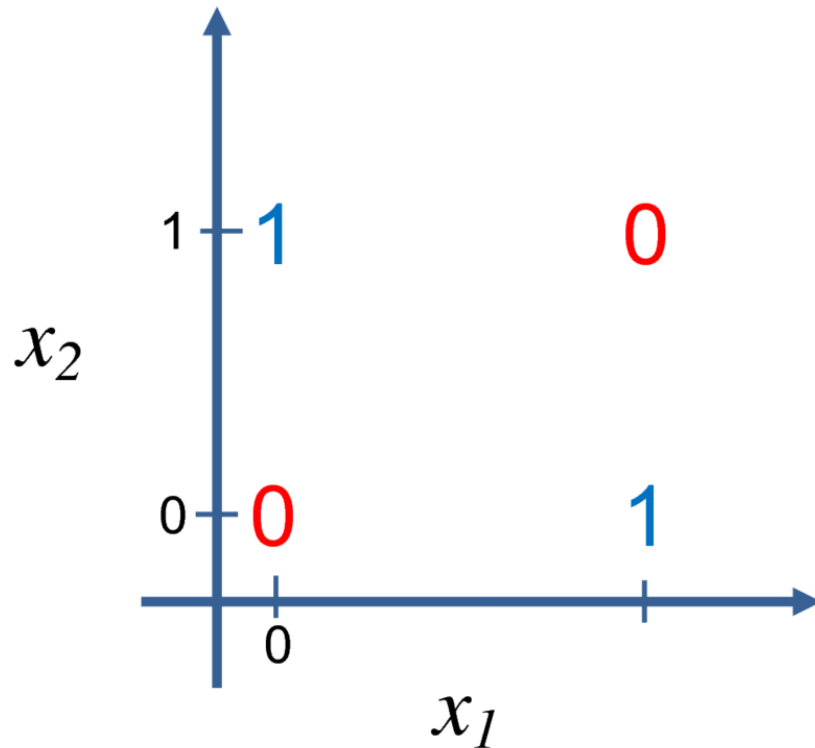
- With a *sigmoid activation*, it becomes equivalent to **Logistic Regression**

$$y = \sigma(w^t x + b), \text{ where } \sigma(z) = 1 / (1 + \exp(-z))$$

- The original perceptron use the *step activation*:

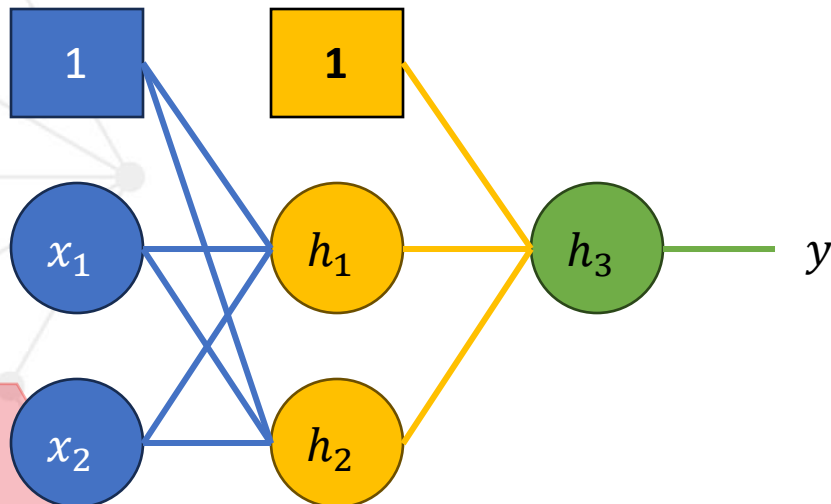
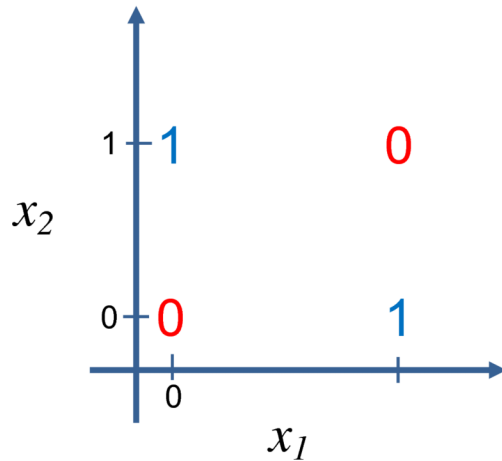
$$y = \delta(w^t x + b), \text{ where } \delta(z)=1 \text{ if } z>0, \text{ else } 0$$

When Linearity fails: XOR Problem



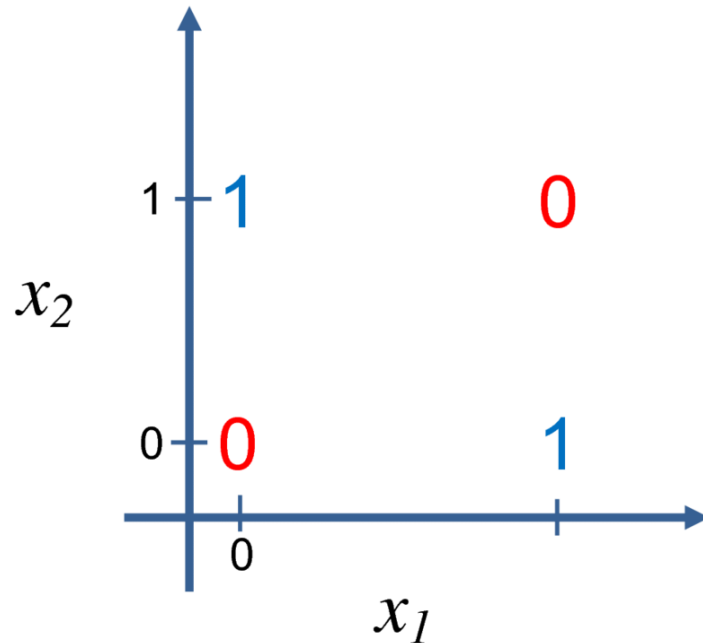
- XOR is not linearly separable -> a single perceptron cannot solve the problem
- Requires at least one hidden layer

When Linearity fails: XOR Problem



- XOR is not linearly separable -> a single perceptron cannot solve the problem
- Requires at least one hidden layer
- 2 input neurons (x_1 and x_2)
- 1 hidden layer with 2 neurons
- 1 output neuron

When Linearity fails: XOR Problem



Let us define:

- Hidden neuron h_1 computes $h_1 = \delta(w_{11} x_1 + w_{12} x_2 + b_1)$
- Hidden neuron h_2 computes $h_2 = \delta(w_{21} x_1 + w_{22} x_2 + b_2)$
- Output neuron computes: $y = \delta(w_{31} h_1 + w_{32} h_2 + b_3)$

If:

- $w_{11} = w_{12} = w_{21} = w_{22} = 1$,
- $b_1 = -1.5$, $b_2 = -0.5$,
- $w_{31} = -1$, $w_{32} = 1$, $b_3 = -0.5$

The XOR is rocked

- $(0,0): h_1 = \delta(-1.5) = 0, h_2 = \delta(-0.5) = 0 \rightarrow y = \delta(-0.5) = 0$
- $(0,1): h_1 = \delta(-0.5) = 0, h_2 = \delta(0.5) = 1 \rightarrow y = \delta(0.5) = 1$
- $(1,0): h_1 = \delta(-0.5) = 0, h_2 = \delta(0.5) = 1 \rightarrow y = \delta(0.5) = 1$
- $(1,1): h_1 = \delta(0.5) = 1, h_2 = \delta(1.5) = 1 \rightarrow y = \delta(-0.5) = 0$

Basic Elements in a FF-NN



- You have just seen the simplest form of a **Feed-Forward Neural Network** — one capable of solving a non-linear problem like XOR.
- Generally they are composed of:
 - An **input layer**: Receives raw features (e.g., pixels, numerical values).
 - **One or more Hidden Layers**: Perform transformations to capture non-linear relationships. The “deep” in deep learning. **They learn new features combining the previous one.**
 - An **output layer** that makes the final decision

Basic Elements in a FF-NN



- **Weights and Biases:**

Weights determine importance of inputs. Biases shift activations. Both are learned during training.

- **Activation Functions:**

Introduce non-linearity. Enable networks to approximate complex functions.

Why does Non-Linearity matter?

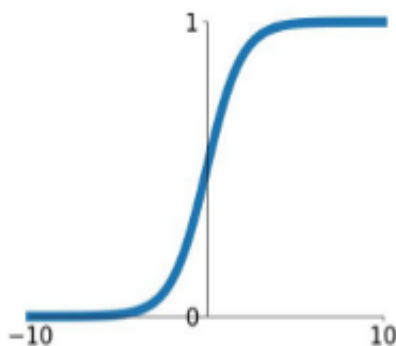
Without them, any number of layers behaves like a single linear transformation.

$$f(g(x)) = W_2(g(x)) + b_2 = W_2(W_1x + b_1) + b_2 = (W_2W_1)x + (W_2b_1 + b_2) = h(x)$$

Activation Functions

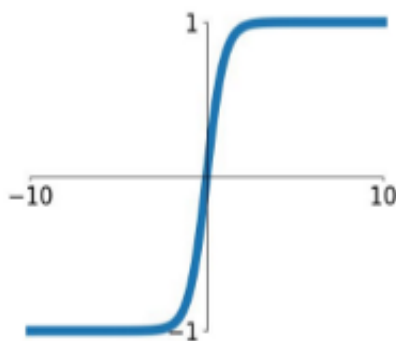
Sigmoid

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



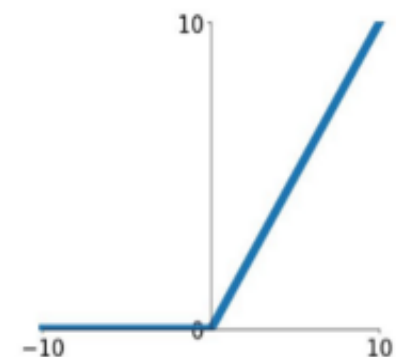
tanh

$$\tanh(x)$$



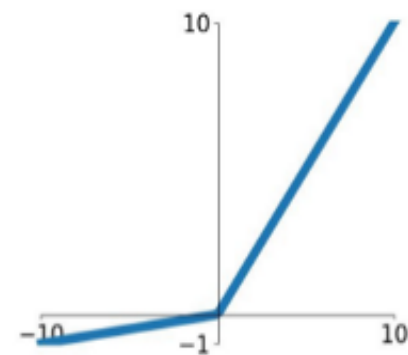
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

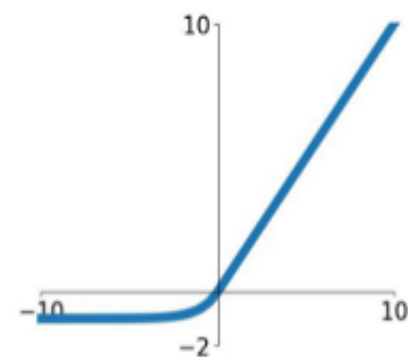


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



Activation Functions



- **Sigmoid**
Smooth, maps input to (0,1). Useful for binary classification, but suffers from vanishing gradients.
- **Tanh**
Maps input to (-1,1). Zero-centered. Better than sigmoid in practice.
- **ReLU**
 $\text{ReLU}(x) = \max(0, x)$. Fast, simple, and widely used. Can lead to dead neurons.
- **Other Activations**
Leaky ReLU, ELU, GELU – variants to mitigate ReLU issues.

How Wrong Is the Model? – Loss Function



- The **loss function** measures how far the model's predictions are from the true values. It is the objective minimized during training.
- **Common Loss Functions:**
 - **Mean Squared Error (MSE)**
 - Used for regression tasks
 - Sensitive to large errors
 - **Cross-Entropy Loss (Log Loss)**
 - Used for classification tasks
 - Compares predicted and true probability distributions

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$$\text{CE} = - \sum_{i=1}^n \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c})$$

Gradient Descent



- **What Are We Optimizing?**

We adjust weights to **minimize the loss function**. Optimization is the core of training.

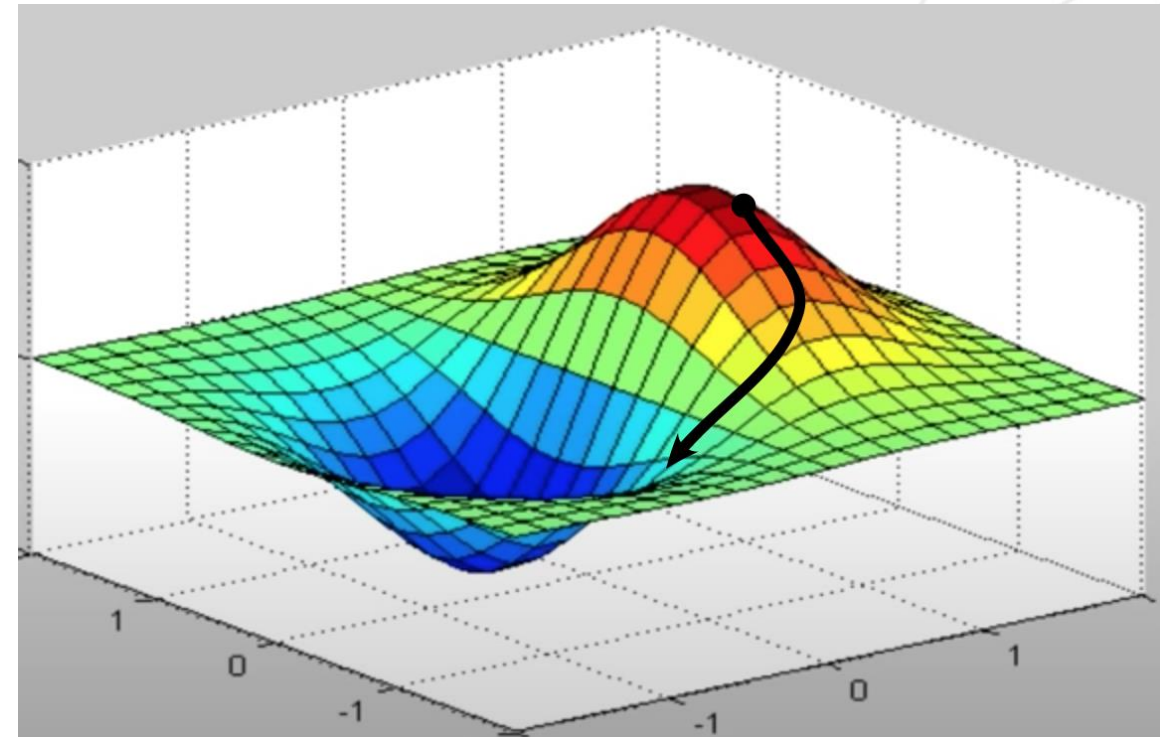
- **Gradient Descent**

A method to find the minimum of a function by moving in the **opposite direction of the gradient**.

- **How It Works**

- Compute gradient of loss with respect to weights
- Update rule:

$$w' = w - \eta \cdot \nabla L(w)$$



Backpropagation



- In a neural network, weights in earlier layers affect the output **indirectly**, through multiple transformations.
- To update those weights, we must know:
 "How does changing this weight change the final loss?"
- But the loss depends on the output, the output depends on activations, and activations depend on previous weights.
- We can't compute this influence in one step...
 → But we can decompose it!
- The **chain rule** allows us to break down the total derivative:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Backpropagation uses this rule recursively, layer by layer, to compute all gradients **from output to input** — efficiently and exactly.

Mini-Batch SGD and Learning Rate



- **Why is the Learning Rate Important?**
 - Too small \rightarrow slow convergence
 - Too large \rightarrow divergence or instability
 - Needs tuning \rightarrow can be decayed or adapted (e.g., Adam)
- Neural networks are trained using **Mini-Batch Stochastic Gradient Descent (SGD)**:
 - Updates weights using a small batch of samples
 - Faster and more memory-efficient than full-batch GD
 - Introduces noise \rightarrow helps escape **local minima**

Training a Neural Network

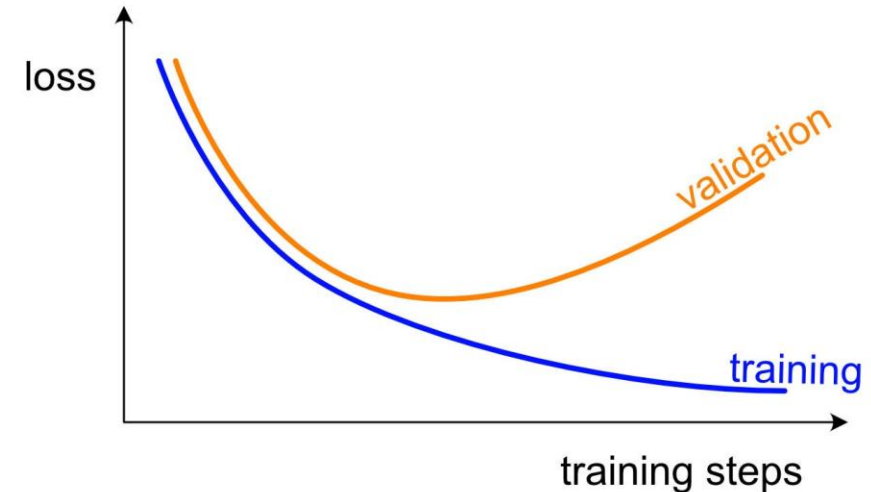


1. Initialize weights
2. Forward pass
3. Compute loss
4. Backpropagate
5. Update weights
6. Repeat

How to Know When to Stop Training?



- **Training Loss ↓**: Model is learning to fit the data. If the model is big enough, it always decrease.
- **Validation Loss ↓ then ↑**: Indicates **overfitting**—model starts memorizing.
- **Ideal Stop**: When validation loss **stops improving**, even if training loss keeps decreasing.
- **Why It Matters**
Early stopping prevents overfitting and improves generalization > We need to find how many **epochs** (full passes over the dataset) optimize learning **without overfitting**.



Neural Networks - More complex architectures

Deep neural networks

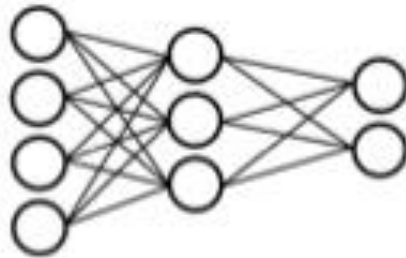


- Deep neural networks are neural networks with “many” layers (up to billions of neurons)
- They often allow to use raw input
- The multiple layers are used to progressively extract higher-level features from the raw input
- Often deep learning uses more advanced layers than the one we have seen in feed-forward neural networks

Artificial neural networks

- Different tasks, different architectures

numerical vectors classification/regression:
feed forward NN (what we have seen so far)



time series analysis: **recurrent NN**
(RNN)

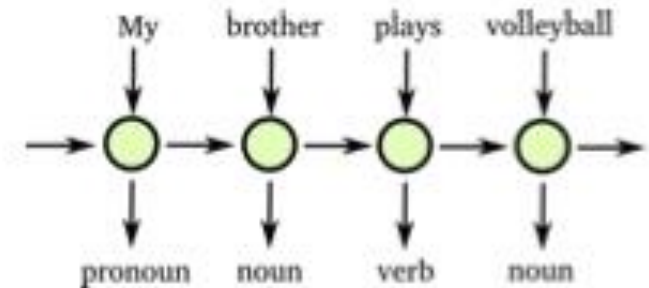
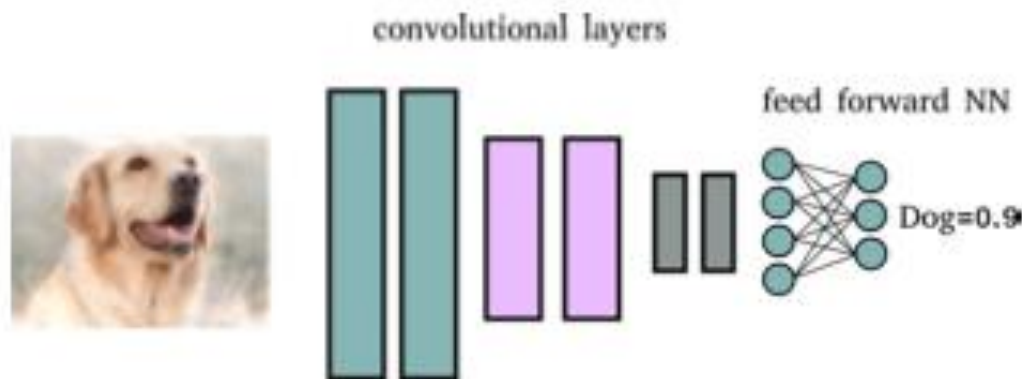
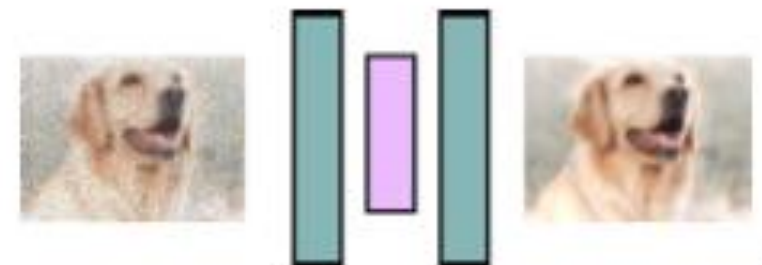


image understanding: **convolutional NN** (CNN)



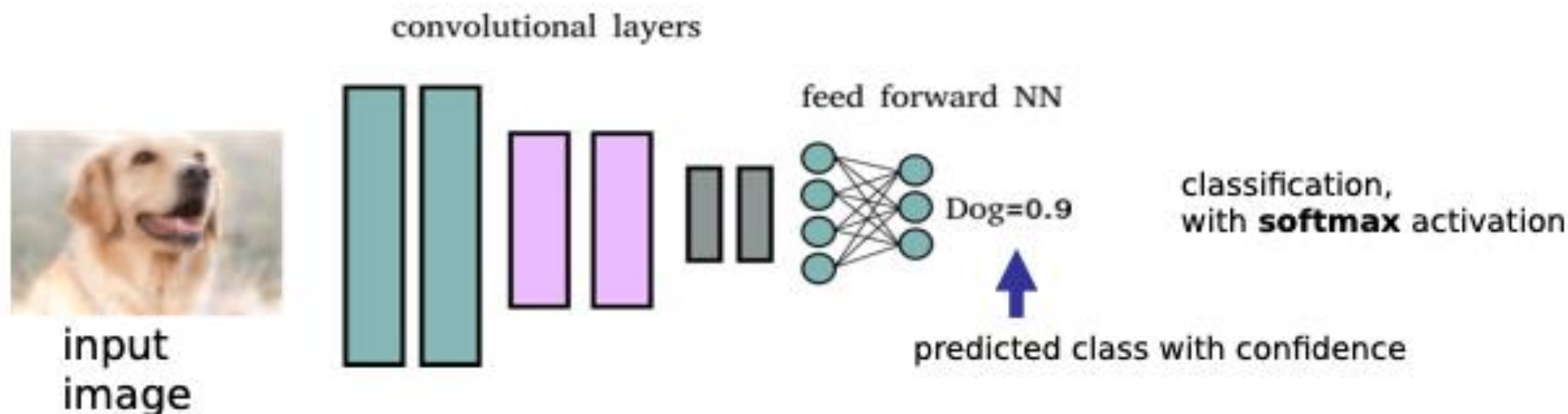
denoising: **auto-encoders**



Convolutional neural networks

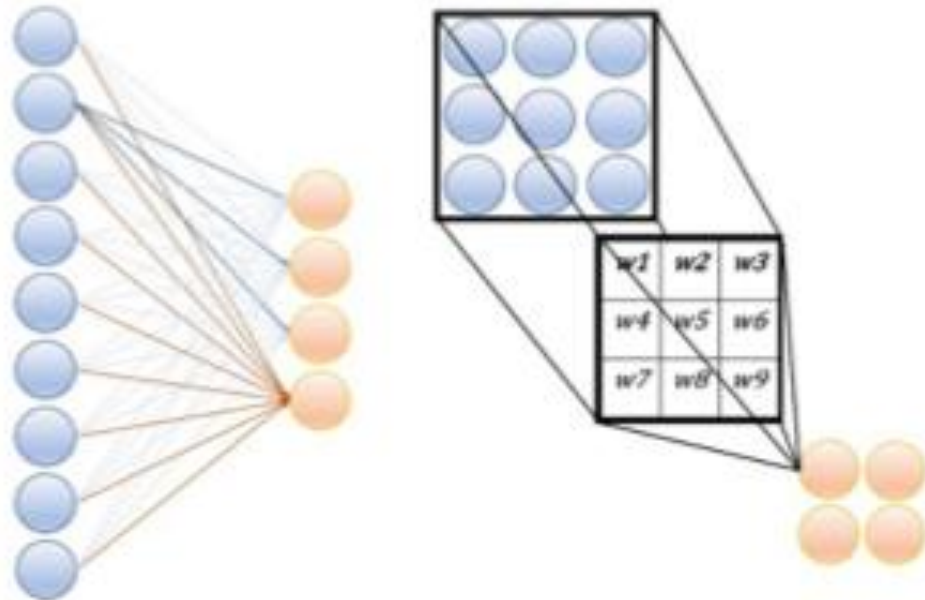
- Allow automatically extracting features from images and performing classification

Convolutional Neural Network (CNN) Architecture



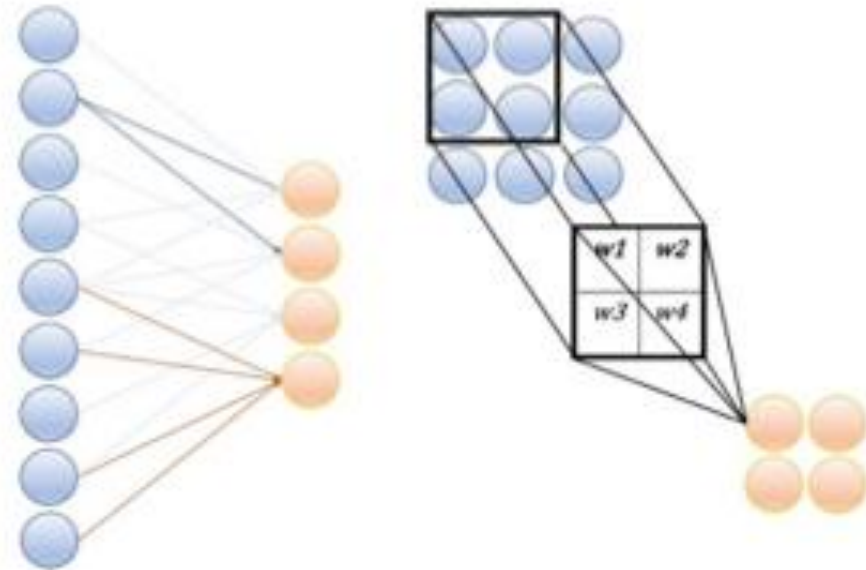
Convolutional layer

Dense layer



*Weights of the different neurons
are different!*

Convolutional layer

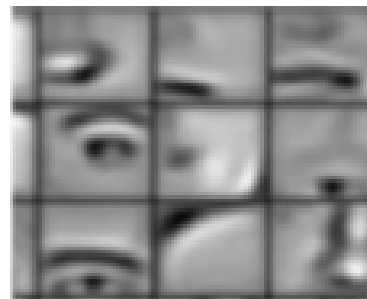
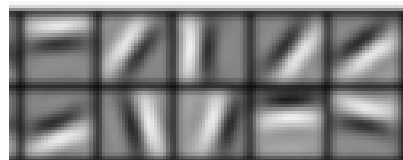


*Weights of the different neurons
are the same!*

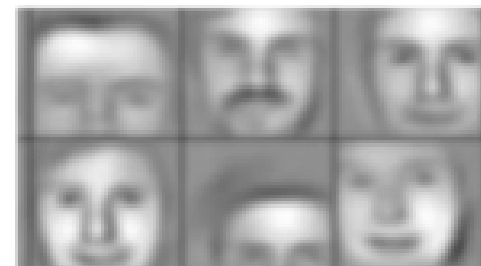
Convolutional neural networks

- Convolutional layers training
 - during training each sliding filter learns to recognize a particular pattern in the input tensor
 - filters in shallow layers recognize textures and edges
 - filters in deeper layers can recognize objects and parts (e.g. eye, ear or even faces)

shallow filters

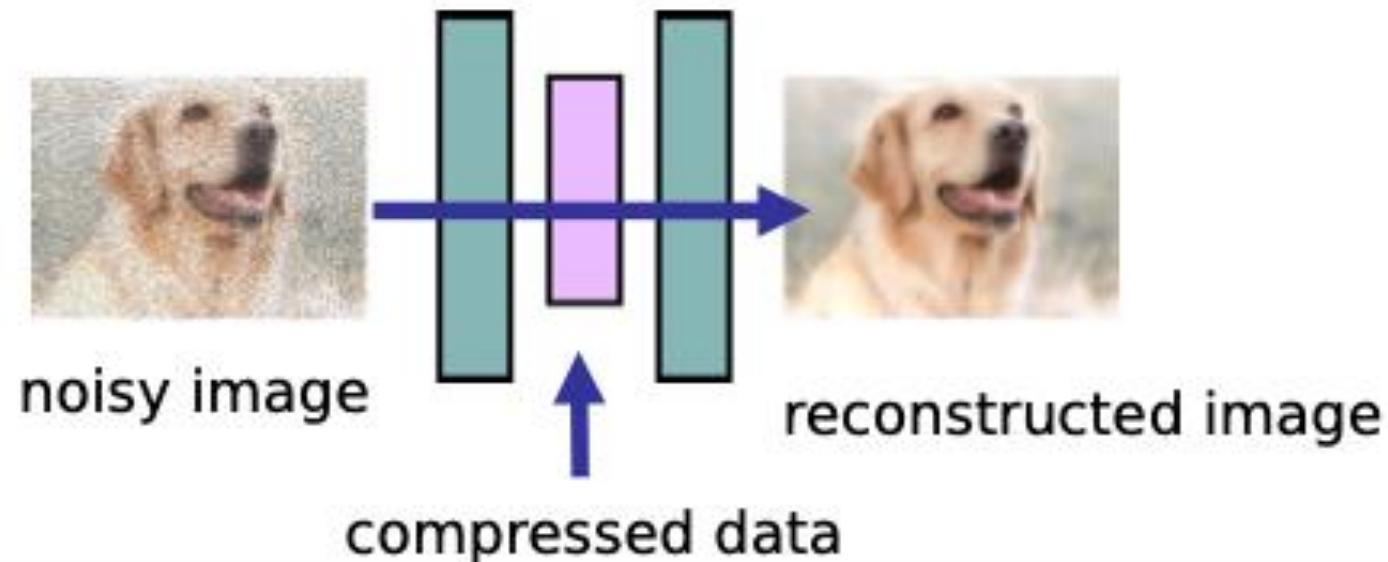


deeper filters



Autoencoders

- Autoencoders allow compressing input data by means of compact representations (embeddings) and from them reconstructing the initial input
 - for feature extraction: the compressed representation can be used as significant set of features representing input data
 - for image (or signal) denoising: the image reconstructed from the abstract representation is denoised with respect to the original one



Deep learning is advancing quickly...



- Long Short Term Memories (LSTM)
- Generative Adversarial Networks (GAN)
- Transformers
- Language models (LM) and large language models (LLM) Graph
- neural networks (GNN)

...



NN in Pytorch

What is PyTorch?



- Open source machine learning library
- Developed by Facebook's AI Research lab
- It leverages the power of GPUs
- Automatic computation of gradients
- Makes it easier to test and develop new ideas.

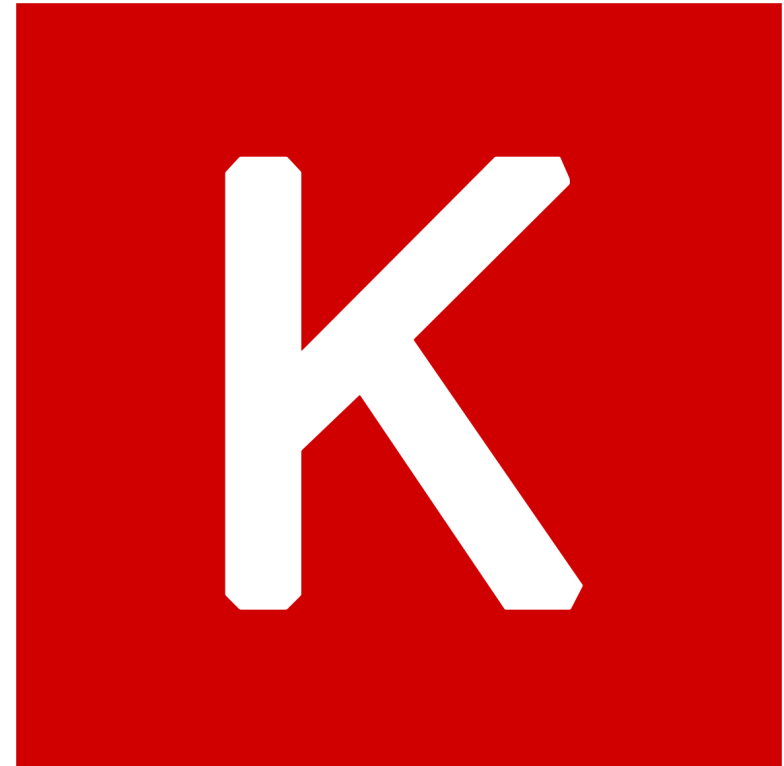
Other libraries?



Politecnico
di Torino
SmartData@PoliTO



TensorFlow



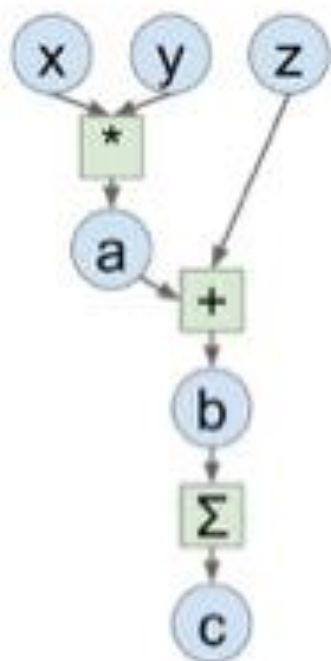
Why PyTorch?



- It is pythonic- concise, close to Python conventions
- Strong GPU support
- Autograd- automatic differentiation
- Many algorithms and components are already implemented
- Similar to NumPy

Why PyTorch?

Computation Graph



Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                    feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch

N, D = 3, 4

x = torch.randn((N, D), requires_grad=True)
y = torch.randn((N, D), requires_grad=True)
z = torch.randn((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
```




Lab10_NN_tutorial.ipynb