AI & Machine Learning for Genomic Data Science

Master in Genomic Data Science – Università di Pavia

Barbara Tarantino

AA 2025-2026

Course Outline

- Day 1 → Python Foundations & AI in Medicine
- Day 2 → Core Machine Learning for Genomics
- Day 3 → Deep Learning Foundations (PyTorch)
- Day 4 → Computer Vision for Medicine
- Day 5 → Large Language Models & Clinical Text

Objective: to acquire theoretical and practical bases to apply AI to genomics, clinical images, medical text.

Program - Day 3

Deep Learning Foundations (PyTorch)

- From classic ML to deep learning
- Fundamentals of neural networks (neurons, layers, activations, losses, backprops)
- Leak Functions and Optimizers
- Training and regularization techniques
- Python Ecosystem for Deep Learning

Practical organization – Day 3

Morning (9:30 – 12:30)

- Theoretical foundations of Deep Learning
  - From classic ML to DL
  - Neurons, layers, activation functions
  - Backpropagation, loss functions, optimizers
  - Training and regularization techniques (dropout, batch norm, early stopping)

Afternoon (14:30 – 17:00)

- Switching to the ecosystem in Python:
  - Tensors, models, and training loops in PyTorch
  - Notebook pratico su Breast Cancer dataset

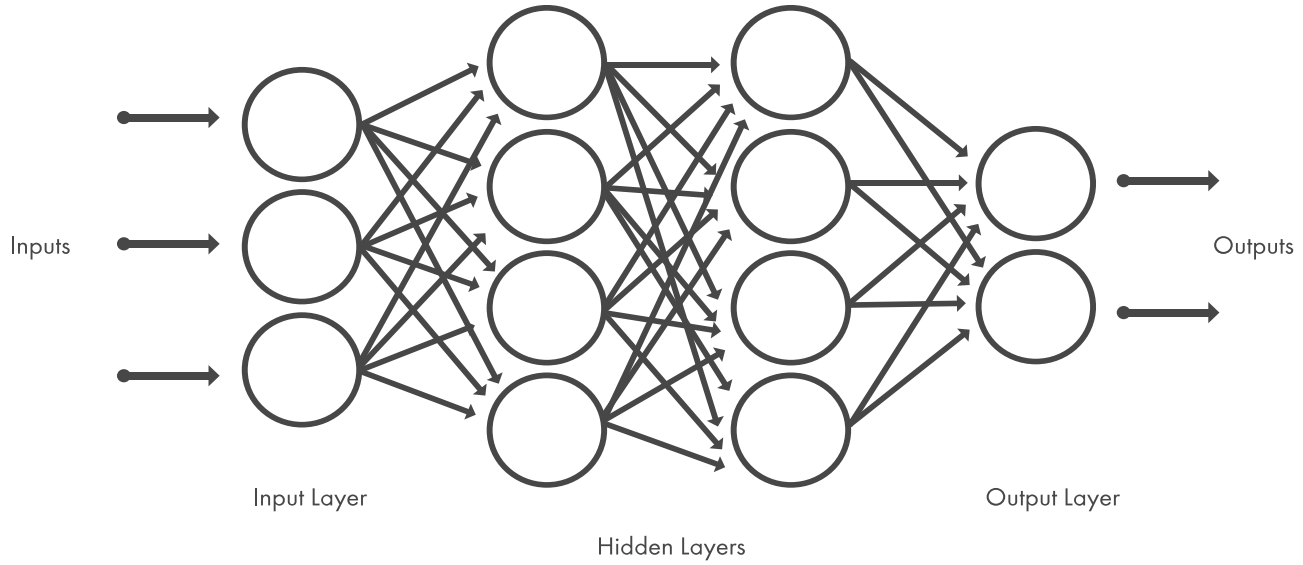
Recap + Q&A

Section 1

Introduction to Deep Learning

Dal Machine Learning al Deep Learning

- Classic ML: *knowledge-driven* approach, the analyst manually chooses the relevant variables (*feature engineering*).
- Deep Learning: *data-driven* approach, the network autonomously learns useful representations from raw data.
- Each layer builds increasingly abstract and complex representations.



Key difference: in ML knowledge is designed, in DL it is learned.

Why Deep Learning

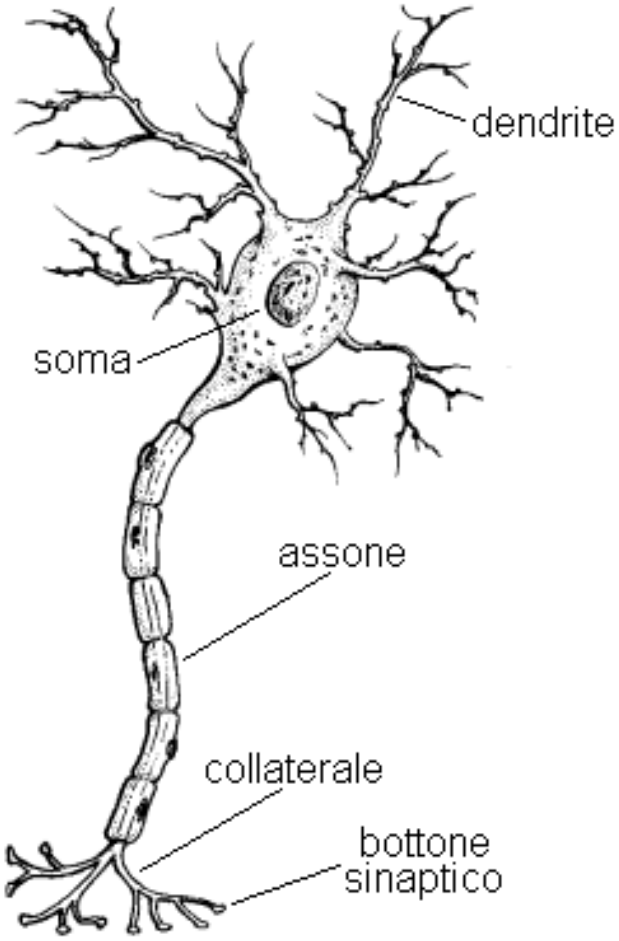
- Biomedical data is often complex and high-dimensional.
- Linear or tree-based models do not always capture nonlinear relationships.
- Neural networks allow you to build flexible and powerful models.
- It all starts with a fundamental element: the artificial neuron.

Key idea: before understanding the network, we need to understand how a single artificial neuron works.

Biological neuron

- The human brain is made up of billions of neurons.
- Each neuron:
  - receives input signals through dendrites,
  - integrates signals into the cell body (soma),
  - if the activation exceeds a threshold → transmits an impulse along the axon to the synapses,
  - the impulse becomes the output to other neurons.

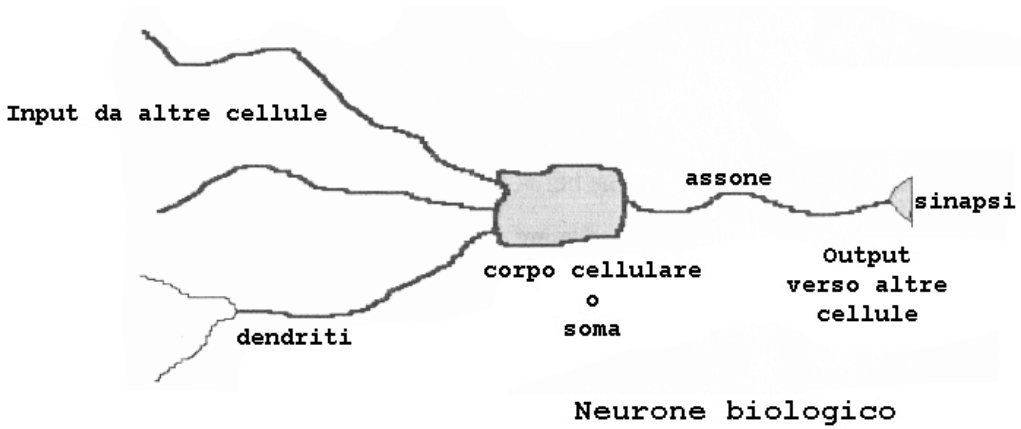
Key idea: the biological neuron adds signals and decides whether to activate.

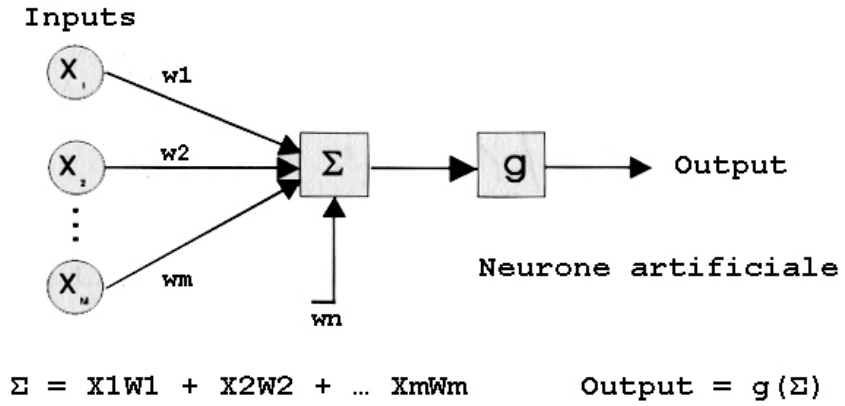


From the biological neuron to the artificial neuron

- The artificial neuron is a mathematical abstraction that replicates the scheme of the biological neuron:
  - **Input** x_i= signals from dendrites,
  - **Weights** w_i= strength/effectiveness of synapses,
  - **Bias** b= trigger threshold,
  - **Activation function** $f(\cdot)$= decision whether to activate,
  - **Output** = signal transmitted to the next neuron.

$$z = \sum_{i=1}^{d} w_i\, x_i + b, \qquad y = f(z)$$





From linear models to the artificial neuron

- In linear regression the relationship between inputs and outputs is:

$$\hat{y} = \sum_{i=1}^{d} w_i \, x_i + b$$

which represents a purely linear model.

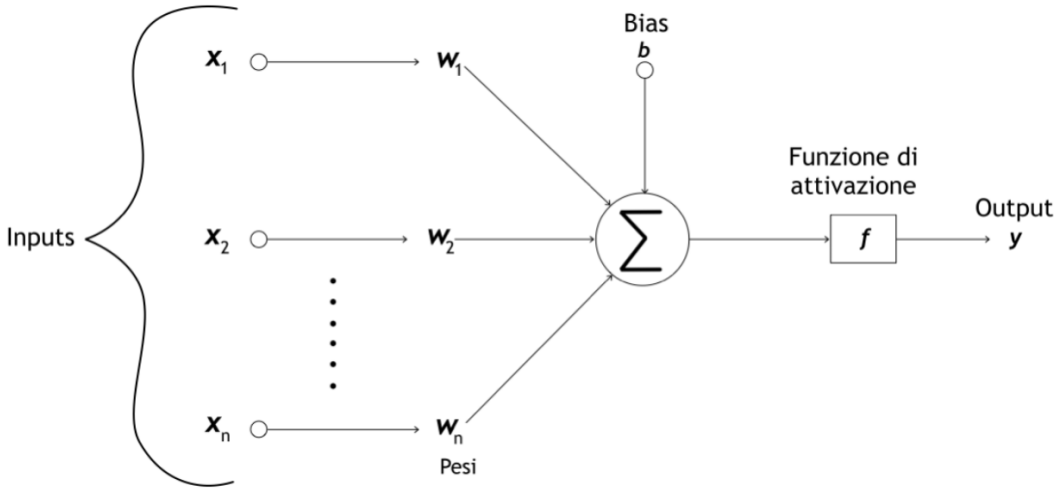
- An artificial neuron extends this pattern by introducing an activation function: $f(\cdot)$

$$\hat{y} = f(\sum_{i=1}^{d} w_i \, x_i + b)$$

Activation allows nonlinear relationships between variables to be modeled, making the neuron more expressive than a simple regression.

The artificial neuron

$$z = \sum_{i=1}^{d} w_i \, x_i + b, \qquad y = f(z)$$



- x_i: input (observed variables).
- w_i: The weight → importance of each input.
- b: Bias → shifts the trigger threshold, makes the model more flexible.
- $f(\cdot)$: Activation function → introduces nonlinearity.
- y: Neuron output → used as input into the next layer.

This is the fundamental formula behind all neural networks.

From the single unit to the neural network

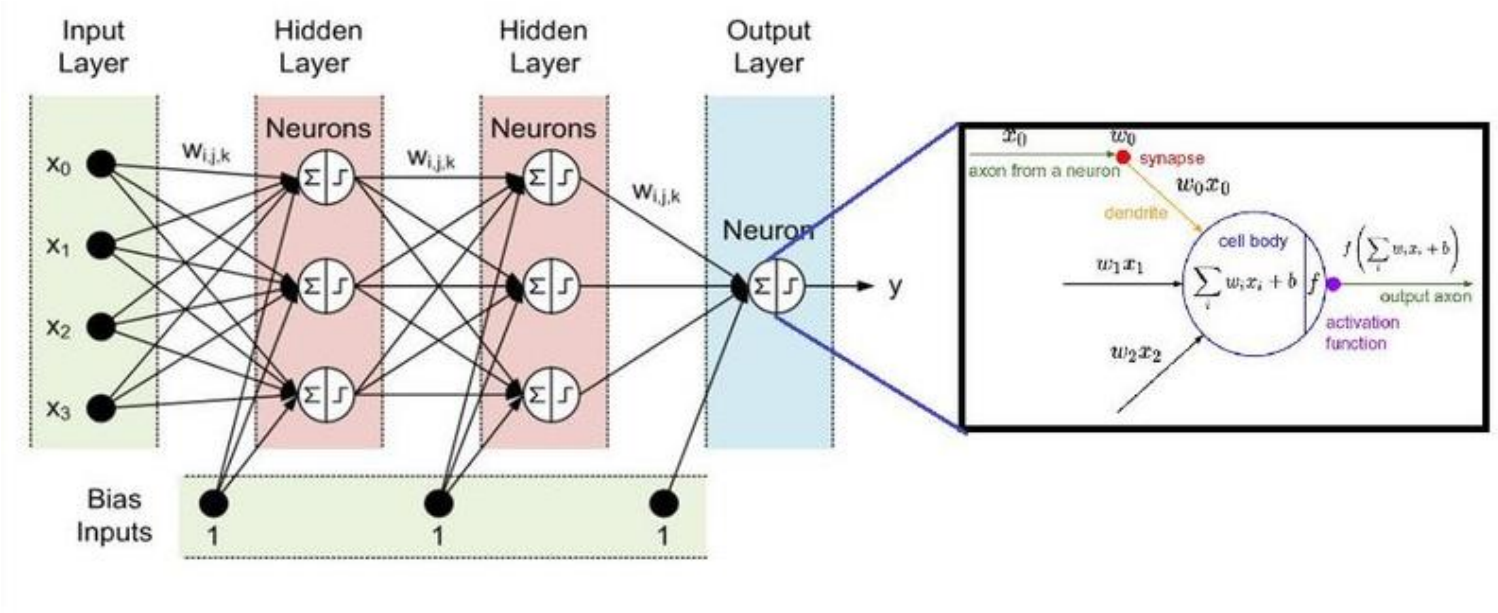
A neural network connects many neurons in layers:

$$h^{(1)} = f\left(W^{(1)}x + b^{(1)}\right)$$
$$h^{(2)} = f\left(W^{(2)}h^{(1)} + b^{(2)}\right)$$
$$\hat{y} = g\left(W^{(3)}h^{(2)} + b^{(3)}\right)$$

- Input layer: Start vector $x = (x_1, \dots, x_d)$.
- Hidden layers: calculate intermediate representations.$h(l) = f(W^{(l)}h^{(l-1)} + b^{(l)})$
- Output layer: final prediction.$\hat{y} = g(W^{(L)}h^{(L-1)} + b^{(L)})$

Each layer transforms data into more abstract representations.

Feedforward architecture



- Input layer (green): Observed data ().x
- Hidden layers (pink): intermediate representations ($h^{(1)}, h^{(2)}$).
- Output layer (light blue): Final prediction ($\hat{y}$).
- Every arrow is a weight.w_{ij}
- Each neuron applies an activation function.$f(\cdot)$

Key idea: A feedforward network processes data sequentially, without loops.

Section 2

Forward Pass

Structure of a neural network

- Input layer: receives the data (Each neuron represents a variable in the dataset.$x_1, x_2, \ldots, x_d$).
- Hidden layers: intermediate layers, transform data into more abstract features:

$$h_j^{(l)} = f(\sum_i w_{ij}^{(l)}\, h_i^{(l-1)} + b_j^{(l)})$$

- Output layer: last layer, produces the final prediction (); use functions such as Sigmoid (binary) or Softmax (multiclass).$\hat{y}$

Summary: input = raw data → hidden = hidden patterns → output = final result.

What is the forward pass

- Every piece of data traverses the network layer by layer.
- In each neuron:
  - The weighted combination of the inputs is calculated:

$$z = \sum_i w_i \, x_i + b$$

  - The activation function applies:

$$h = f(z)$$

- The output becomes input for the next layer.
- The forward pass is the "flow" that carries an input up to the prediction.

The role of activation functions

- Without : the network remains linear → unable to learn complex relationships. $f(\cdot)$
- With : we introduce nonlinearities → the network becomes a universal approximator. $f(\cdot)$
- The choice influences both the expressiveness of the model and the efficiency of learning. $f(\cdot)$

Activations allow for data space deformations → complex separations.

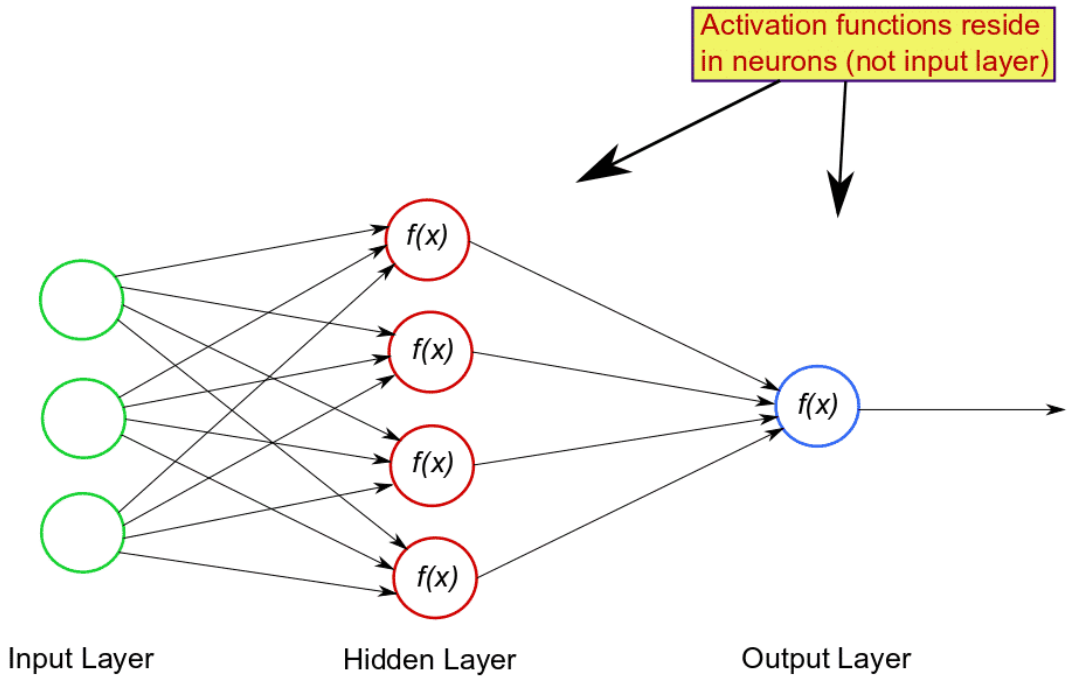
Universal approximation theorem

- A feed-forward network with ≥ 1 hidden layer and nonlinear activation functions can approximate any continuous function (on a compact domain).
- Meaning:
  - Hidden layers → nonlinear transformations that capture complex patterns.
  - Output layer → translation into interpretable values (e.g. probability).

Theoretical foundation of deep learning.

Where the activations work

- Input layer: No activation (raw data only).
- Hidden layers: simple and efficient activations (e.g. ReLU, Tanh).
- Output layer: problem-specific activations (e.g. binary Sigmoid, multiclass Softmax).



Forward pass: the flow of the calculation

- Step 1: The data enters from the input layer.
- Step 2: Each hidden layer combines the values () and transforms them with a trigger function $z = W^{(l)}h^{(l-1)} + b^{(l)}f(\cdot))$.
- Step 3: The output layer also applies an activation function, chosen according to the problem:$g(\cdot)$
  - regression → identity ($\hat{y} = z$)
  - binary classification → Sigmoid,
  - multiclass classification → Softmax.

- Each layer transforms the data into more abstract representations, until it produces an interpretable result. This complete process is called forward pass.

Section 3

Parameter learning

From Forward Pass to Learning

- The forward pass defines how an input is transformed into an output.
- But the goal of deep learning is to optimize parameters to reduce the error between prediction and truth.
- Learning consists of:
  - Define a loss function.$L(y, \hat{y})$
  - Calculate the gradient of loss with respect to weights.
  - Update the weights with an optimization rule.

This process allows the network to move from a static compute system to an adaptive model that can learn from data.

Learning loop of a network

1. Forward pass → I get the prediction. $\hat{y}$
2. Loss function → comparison with the real . $\hat{y}y$
3. Backpropagation → calculation of gradients. $\frac{\partial L}{\partial w}$
4. Updated weights → the weights are corrected.

Loss Function

- In a neural network, learning requires the definition of a quantitative criterion that measures the discrepancy between observed values () and predictions ($y\hat{y}$).
- This criterion is expressed by the loss function, which associates each parameter configuration with a numerical value proportional to the error:$L(y, \hat{y})\theta$

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, f_\theta(x_i)\right)$$

Summary: the loss provides the reference that guides the updating of the parameters.

Types of Loss Function

- The choice of loss depends on the type of task (regression vs classification).
  - Regression → *Mean Squared Error (MSE)*

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_j)^2$$

  - *Cross-Entropy* → Classification

$$L(\theta) = -\sum_{i=1}^{n}\sum_{k=1}^{K} y_{i,k}\ \log\left(\hat{y}_{i,k}\right)$$

Once the loss has been defined, we must reduce it with an optimization method.

Optimization

- Aim of the training: to find the parameters that minimize the loss function. θ

$$\theta^* = arg \min_{\theta} \mathcal{L}(\theta)$$

- Problem: The parameter space is high-dimensional, → impossible to explore exhaustively.
- Workaround: Use iterative algorithms that update parameters following the information provided by the gradient.

In summary: optimization translates the loss function into a learning process, driving the evolution of network weights.

The Gradient

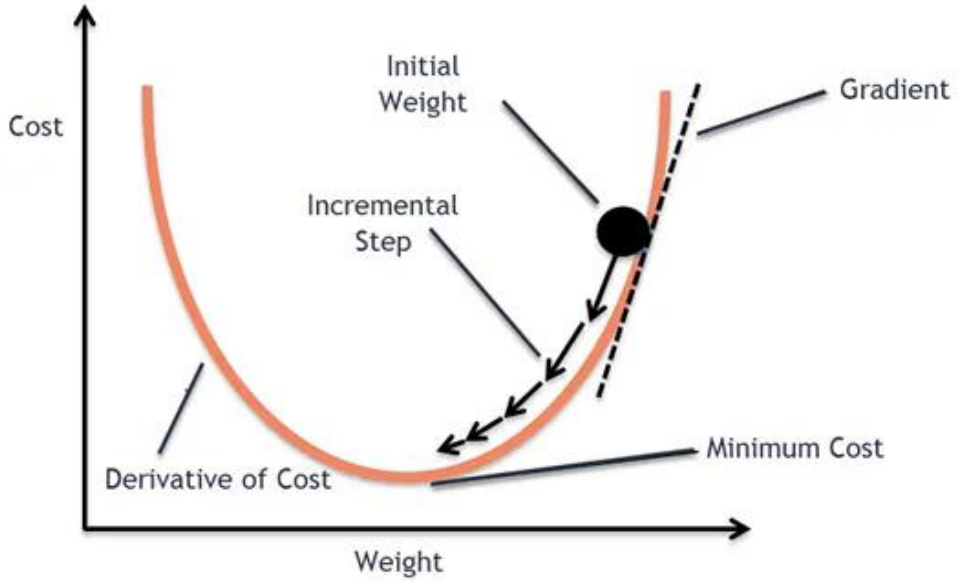
- In optimization, we can't explore the entire parameter space to solve. $\theta^* = arg\ \min_{\theta} \mathcal{L}(\theta)$
- So let's use the gradient, i.e. the vector of the partial derivatives of loss:

$$\nabla_\theta \mathcal{L}(\theta) = \left(\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \ldots, \frac{\partial \mathcal{L}}{\partial w_m}\right)$$

- $\mathcal{L}(\theta)$: loss function, depends on the parameters of the network.
- w_j: parameter (weight) of the network.
- $\frac{\partial \mathcal{L}}{\partial w_j}$: how much does the loss change if I change the weight slightly. w_j
- m: The total number of parameters (weights) of the network.

Geometric Intuition of the Gradient

- Loss can be seen as a curve (1D) or a surface (n-D).$\mathcal{L}(\theta)$
- At each point:
  - the gradient indicates the fastest growth direction;
  - Moving in means going down to lower values.$-\nabla\mathcal{L}$
- Upgrading is like going down a hill, step by step.
- The learning rate controls the width of the stride:η
  - too big → risk of skipping the minimum;
  - too small → slow descent.
- Summary: the gradient locally guides the loss descent towards the low.



Gradient Descent – Key Steps

1. Initialization: Randomly initialized parameters. θ

2. Gradient calculation:

$$\nabla_\theta \mathcal{L}\left(\theta^{(t)}\right)$$

Measure how the loss varies with respect to the parameters.

3. Parameter Update:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_\theta \mathcal{L}\left(\theta^{(t)}\right)$$

- $\theta^{(t)}$: parameters to iteration t
- η: learning rate (controls the width of the stride)
- $\nabla_\theta \mathcal{L}\left(\theta^{(t)}\right)$: gradient of the loss with respect to the parameters

Summary: Gradient Descent is an iterative process that updates parameters by moving in the opposite direction to the gradient to reduce the loss function.

Limitations of Gradient Descent

- Use *the entire dataset* to calculate the gradient at each iteration:

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \, \mathcal{L}(x_i, y_i, \theta)$$

- Problem: Very expensive on large datasets → slow and requires a lot of memory.

- Solution: Introduce more efficient variants:
  - Stochastic GD (SGD): Updates after *each example*.
  - GD Mini-Batch: Refreshes after *small groups* of examples.
  - Momentum: Adds a "memory" to reduce oscillations.

Summary: Variants are used to make optimization faster, more scalable and more stable.

Stochastic Gradient Descent (SGD)

- **Definition:** Update the parameters after each dataset example (x_i, y_i).

- **Upgrade formula:**

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \nabla_\theta \mathcal{L}\left(x_i, y_i, \theta^{(t)}\right)$$

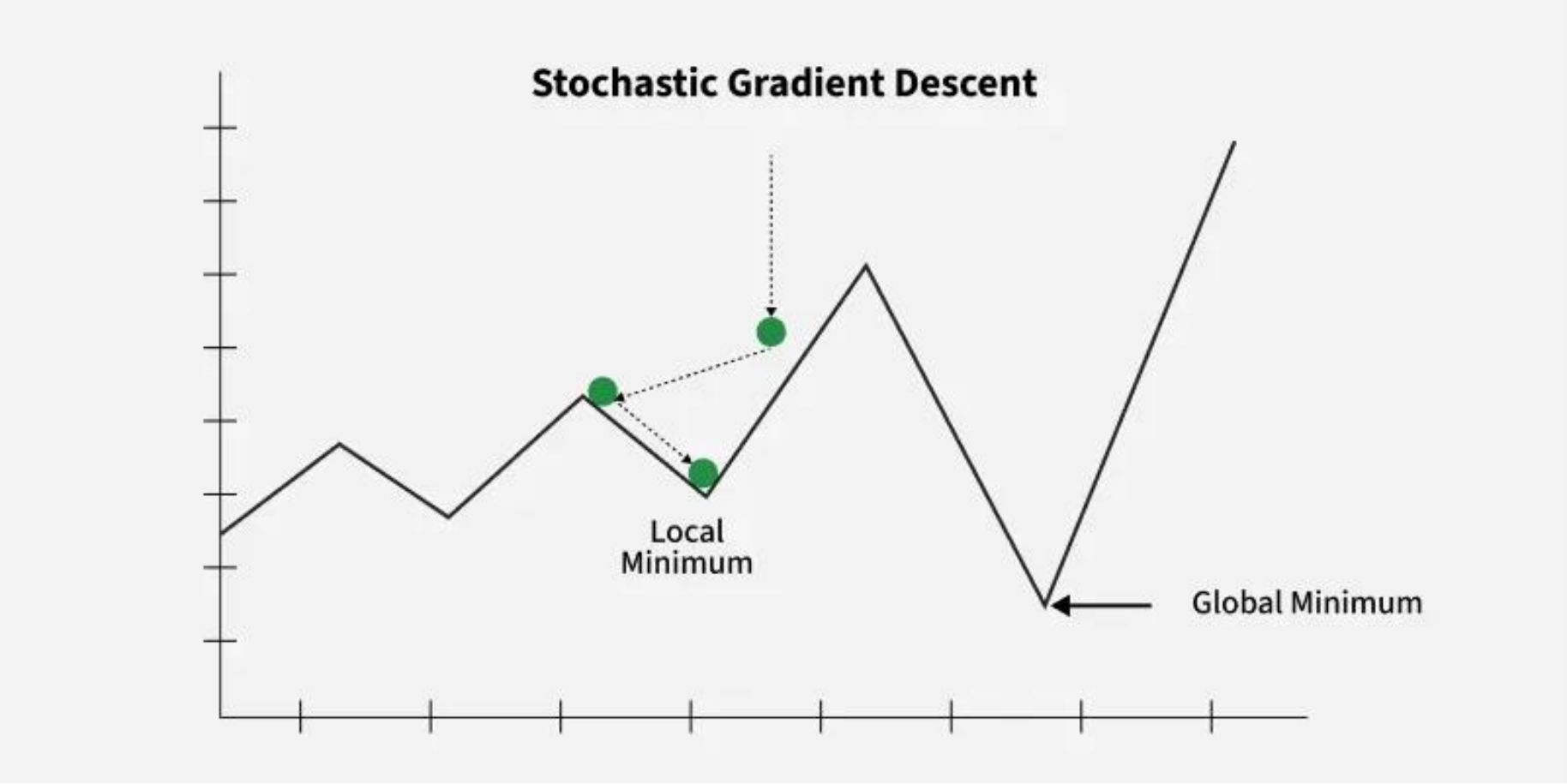
  - $\theta^{(t)}$: parameters at the pitch $.t$
  - η: learning rate.
  - $\nabla_\theta \mathcal{L}\left(x_i, y_i, \theta^{(t)}\right)$: loss gradient calculated only on the sample $.(x_i, y_i)$

- Each example provides an estimate of the gradient → very frequent updates.
- "Noisy" trajectory, but capable of escaping local minimums.

Advantages: low memory, better explore space.

Disadvantages: Very noisy, requires a lot of iterations.

Stochastic Gradient Descent (SGD)



Irregular oscillations, but progressive descent towards the minimum.

Mini-Batch Gradient Descent

- **Definition:** Update the parameters after calculating the gradient on a small group of examples (mini-batches).

- **Upgrade formula:**

$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta \, \mathcal{L}\left(x_i, y_i, \theta^{(t)}\right)$$

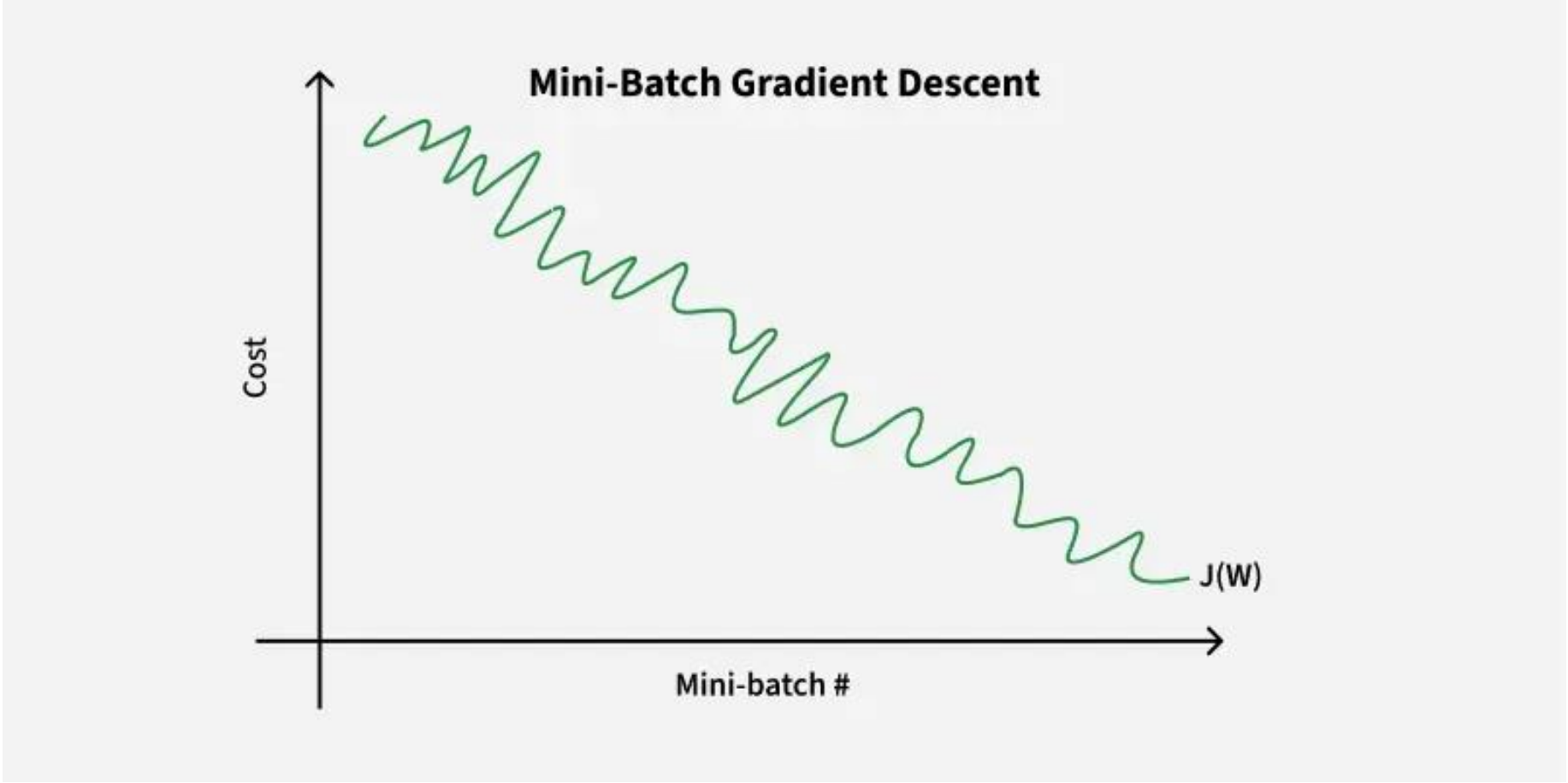
  - B: mini-batch size (e.g. 32, 64)
  - Average : Reduces noise compared to SGD$\frac{1}{B}\sum$:

- Averaging on samples reduces noise compared to SGD, but maintains frequent updates.
- Trajectory more stable than SGD, but still not as smooth as the whole batch.

Advantages: compromise between speed and stability.

Disadvantages: Can get stuck in local minimums.

Mini-Batch Gradient Descent



"Smooth zigzag" trajectory towards the bottom

SGD with Momentum

- **Definition:** accumulates information about gradients passed through a velocity .v_t
- **Upgrade formulas:**

$$v_{t+1} = \beta v_t + \nabla_\theta \mathcal{L}\left(\theta^{(t)}\right)$$
$$\theta^{(t+1)} = \theta^{(t)} - \eta \cdot v_{t+1}$$

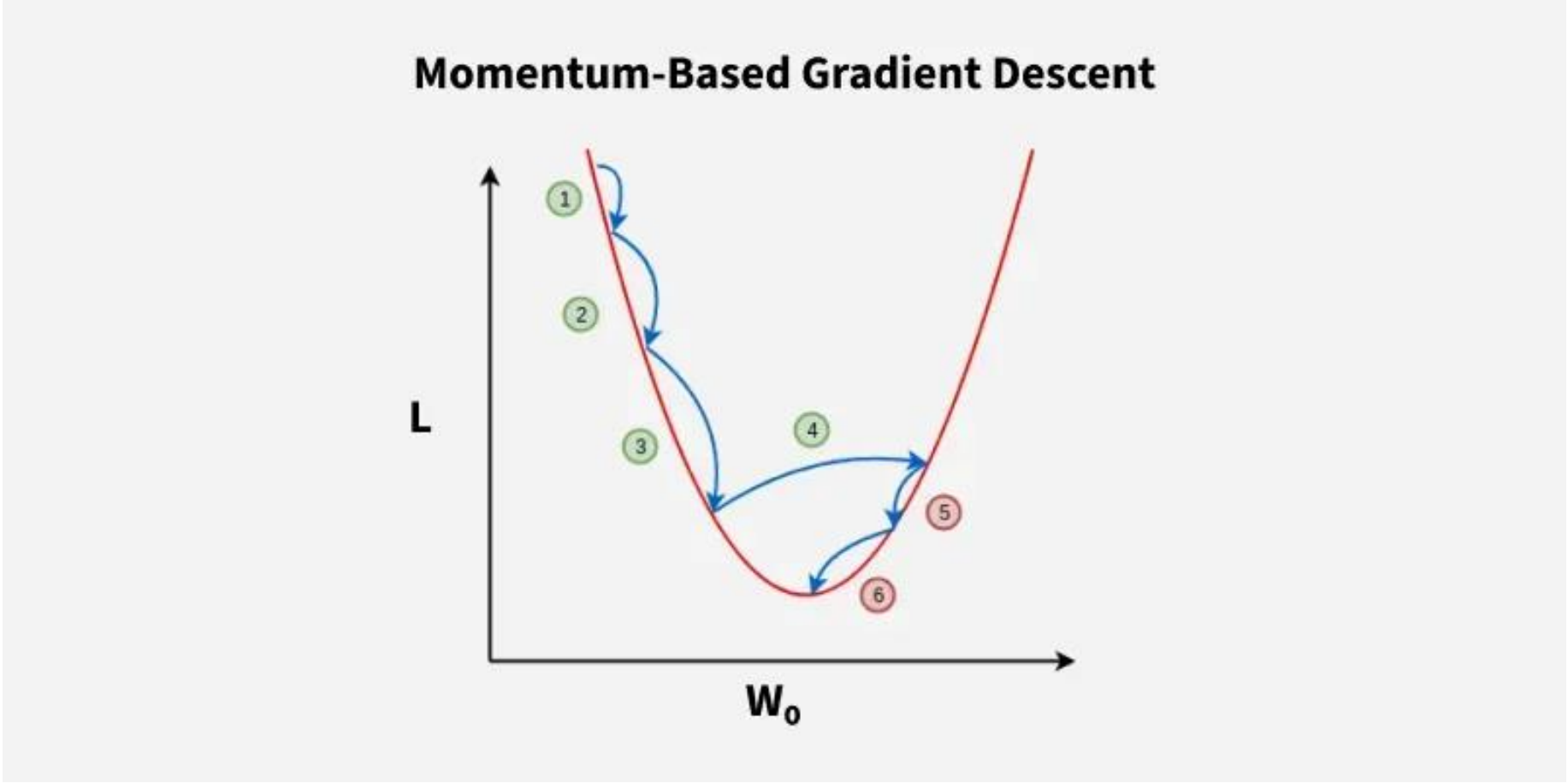
  - v_t: accumulated speed up to the wheelbase t
  - β: coefficient (0–1) → how much weight to give to past gradients
  - If : No memory, normal SGD$\beta = 0$
  - If: A lot of memory, the trajectory follows the average direction accumulated → smoother and more consistent updates. $\beta \to 1$

- Instead of just following the current gradient, it also takes into account those that have passed → smoother trajectory.
- Reduces oscillations and accelerates along consistent directions.

Advantages: Faster and more stable convergence.

Disadvantages: you have to choose well.β

SGD with Momentum



Smooth descent, less zig-zag, faster to the bottom

Limitations of Gradient Descent

Even when using **SGD**, **Mini-Batch** or **Momentum**, Gradient Descent has important limitations:

- **Fixed learning rate**
  - Too large → the loss explodes or fluctuates.
  - Too small → the descent is very slow.
- **A single step for all weights** → but not all parameters should update at the same speed.
- **Oscillations** → along "narrow" directions, the trajectory bounces back and forth.
- **Deep Networks**
  - **Vanishing gradient: Gradients** become **very small** as they propagate backwards → the first layers **stop learning**.
  - **Exploding gradient:** Gradients become **too large** → weights **grow unsteadily** and the model diverges.

You need a method that automatically adjusts pitch and direction, adapting to the network and data.

The idea of advanced optimizers

Advanced optimizers introduce **adaptive mechanisms**:

- The *learning rate* **is not fixed** but varies over time.
- Each parameter can have **a different pitch** based on its gradient "history".
- **Mean and variance of gradients** are used to stabilize the course of the descent.

Key examples:

- **AdaGrad** →automatically adapts the pitch of each parameter:
  weights that change often make smaller steps, rare weights larger.
- **RMSProp** → averages *recent gradients* to make steps more stable over time.
- **Adam** → combines the ideas of Momentum and RMSProp:
  he follows a smoother direction and adjusts the size of the steps himself.

Summary: The model *learns to learn*, adjusting the amplitude of the steps on its own.

AdaGrad (Adaptive Gradient)

- **Fixed problem:** In classic GD, all weights use the same pitch.

- **Idea:** AdaGrad automatically adjusts the *learning rate* of each parameter based on how often it has received large gradients.
  - Updated weights often → smaller wheelbase.
  - Rarely updated weights → larger wheelbase.

- **Pros:** Great for *sparse* data (e.g. rare words in texts).

- **Cons:** the steps get smaller and smaller → learning can "get stuck".

AdaGrad adapts the start of the descent well, but slows down too much over time.

RMSProp (Root Mean Square Propagation)

- **AdaGrad problem: it** accumulates all gradients → the step (*effective learning rate*) becomes progressively smaller until it almost stops.

- **Idea:** RMSProp introduces a **moving average** of squared gradients, which gradually "forgets" older gradients and gives **more weight to recent gradients**.
  - *Moving average* = time-weighted average that reduces the influence of the recent past with a decay factor ($\gamma \approx 0.9$).

- **Effect:**
  - Large recent gradients → smaller pitch (avoids divergence).
  - Small recent gradients → larger step (promotes updating).
  - The γ parameter controls **how much "passed" is kept in memory**.

Advantages:
More stable steps, continuous learning, and the basis for more advanced optimizers like **Adam**.

Adam (Adaptive Moment Estimation)

- Adam combines the ideas of **Momentum** and **RMSProp** to achieve faster and more stable learning.

- Calculate two pieces of information:
  - **Gradient averaging** → indicates the *average direction* in which to move (such as Momentum).
  - **Gradient variance** → measure *how large* the gradients were (such as RMSProp).

- Use both to **automatically adjust direction and stride width**.

- **Effects:**
  - Smoother movement → less oscillation.
  - Adaptively scaled steps → faster and more stable convergence.
  - Initial correction (*bias correction*) → more accurate updates in the early stages.

- **Result:**
  Fast, robust and stable optimizer, now **the default choice** in modern deep learning frameworks.

To recap: how the web learns

1. Forward pass
   Data passes through layers → we get a prediction.$\hat{y}$

2. Loss function
   Measures the error between and the real values .$\hat{y}\ y$

3. Gradients
   Calculate how the error varies with respect to each weight.

4. Optimizers
   Update weights, step by step, to reduce error.

We have seen from the basic Gradient Descent up to the advanced optimizers (AdaGrad, RMSProp, Adam).

Next steps on our journey

- **Backpropagation:** How we calculate gradients across the network efficiently.
- **Activation functions:** why they need non-linearities, and what problems they introduce (sigmoid, tanh, ReLU).
- **Practical problems and solutions:** regularization, normalization, dropout.

We move from updating parameters to designing neural networks capable of learning and generalizing.

Section 4

Backpropagation

The problem to be solved

- After the *forward pass* we get:
  - Network output $\hat{y}$
  - The **Leak** Function $\mathcal{L}(y, \hat{y})$

- To update weights with Gradient **Descent** you need:

$$\frac{\partial \mathcal{L}}{\partial W}, \frac{\partial \mathcal{L}}{\partial b}$$

for **every weight and bias** of the network.

- With millions of parameters, it is not possible to calculate everything by hand.

You need an efficient algorithm that automatically calculates all gradients:
Backpropagation.

What is Backpropagation

- It is the **algorithm that calculates the loss gradients** with respect to all the parameters of the network.

- It works in **two main steps**:
  - **Forward pass:** input → layer → output → loss calculation.
  - **Backward pass:** loss → calculation of gradients → updating of weights.

- Use the **chain rule** to propagate the error backwards, from output to input.

The result: fast, systematic and scalable gradient calculation for any architecture.

The Chain Rule

When a function consists of multiple steps, for example

$$y = f\big(g(x)\big),$$

The final change depends on y**how each intermediate step changes**:

$$\frac{dy}{dx} = f'\big(g(x)\big) \cdot g'(x)$$

To know how much the output changes, let's multiply the effects of each level.

In neural networks:

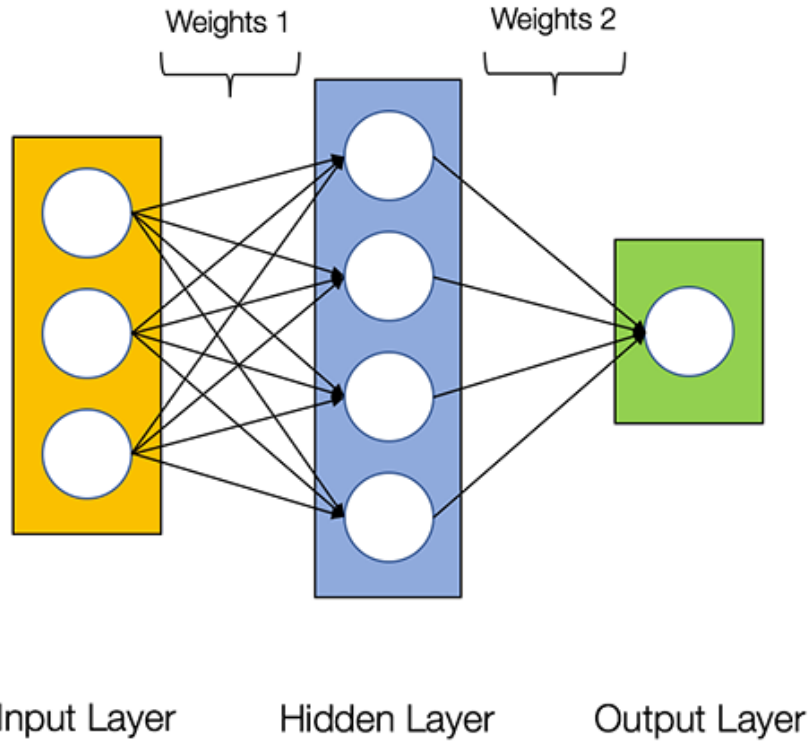
- Each **layer** is a function that transforms the input.
- The *chain rule* allows **the error to be transmitted backwards**, discovering **how much each layer contributed** to the final error.
- So the network can **update all weights consistently**.

The *chain rule* is the basis of **backpropagation**: it connects changes between layers and makes machine learning possible.

Example architecture

- Simple network with 1 hidden layer:
  - Input: $x \in \mathbb{R}^3$
  - Hidden layer (4 neuroni):
    $z^{[1]} = W^{[1]}x + b^{[1]}, h = f\left(z^{[1]}\right)$
  - Output layer (1 neurone):
    $\hat{y} = W^{[2]}h + b^{[2]}$
  - Loss (MSE):
    $\mathcal{L} = \frac{1}{2}\left(y - \hat{y}\right)^2$

Objective: to find $\frac{\partial \mathcal{L}}{\partial W^{[1]}}, \frac{\partial \mathcal{L}}{\partial b^{[1]}}, \frac{\partial \mathcal{L}}{\partial W^{[2]}}, \frac{\partial \mathcal{L}}{\partial b^{[2]}}$.



Layer Output - Local Error

- The local error measures **how much the prediction differs** from the target:

$$\delta^{[2]} = \frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y$$

- If → the network $\delta^{[2]} > 0$**has overestimated**, the weights must **decrease**.
- If → the network $\delta^{[2]} < 0$**has underestimated**, the weights must **increase**.

Indicates the direction and intensity of the correction for the output layer.

Layer Output – Updating Weights

- Each weight is updated in proportion to:
  - The **local** error $\delta^{[2]}$
  - The **activation** of the previous neuron h

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \delta^{[2]} h^T, \frac{\partial \mathcal{L}}{\partial b^{[2]}} = \delta^{[2]}$$

The output layer provides the "direct error signal" that will drive backpropagation.

Hidden layer - Error propagation

- The hidden layer **does not see** the target directly, but receives an error share from the output:

$$\delta^{[1]} = \left(W^{[2]}\right)^T \delta^{[2]} \odot f'\left(z^{[1]}\right)$$

- The error is "weighted" by the links and modulated by the derivative of the activation.$W^{[2]}$

Thus each hidden neuron receives *feedback proportionate* to its influence on the output.

Hidden layer - Updating weights

Each weight is updated based on:

- The back-propagated error $\delta^{[1]}$
- The original input x

$$\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \delta^{[1]} x^T, \frac{\partial \mathcal{L}}{\partial b^{[1]}} = \delta^{[1]}$$

So the inner layers also learn, even though they do not have direct access to the final error.

Backpropagation – Schema generale

Phase	Formula	Meaning
Local error	$\delta^{[L]} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \odot f'\left(z^{[L]}\right)$	Final detour
Output Update	$\frac{\partial \mathcal{L}}{\partial W^{[L]}} = \delta^{[L]} \left(h^{[L-1]}\right)^T$	Final weight correction
Backpropagated error	$\delta^{[l]} = \left(W^{[l+1]}\right)^T \delta^{[l+1]} \odot f'\left(z^{[l]}\right)$	Error Distribution
Hidden update	$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \delta^{[l]} \left(h^{[l-1]}\right)^T$	Internal fit

Conceptual interpretation

- **Output layer:** compares the prediction to the target.
- **Hidden layers:** receive "weighted" errors from subsequent layers.
- **Chain rule:** allows the error to propagate backwards, distributing the "responsibility" to each neuron.
- **Practical effect:** each weight is updated according to its contribution → *efficient and scalable training*.

Section 5

Activation functions

From gradient to trigger functions

- During error backpropagation, the contribution of each layer depends on the derivative of the trigger function.
- In backpropagation the derivative of the activation always appears:

$$\delta^{[l]} = \left(W^{[l+1]}\right)^T \delta^{[l+1]} \odot f'\left(z^{[l]}\right)$$

- If ≈ 0 → the gradient cancels out → the weights do not update → "locked" network.$f'(z)$
- If it is too large → explosive gradient → unstable updates.$f'(z)$
- The choice of activation conditions:
  - The stability of training.
  - The speed of convergence.
  - The ability to train deep networks.

Activation determines **how the information and error flow into the network**.

Role of activation functions in hidden layers

- Hidden *layers* progressively transform inputs into increasingly abstract intermediate representations.

- To do this, you need **nonlinear activation functions** that allow the network to model complex relationships.

- Activation functions affect:
  - Gradient propagation (**vanishing/exploding**).
  - The ability to learn nonlinear relationships and feature hierarchies.
  - The stability and speed of training.

Activation functions are therefore the main mechanism by which a neural network acquires nonlinear representational capabilities.

The problems of the gradient

- During *backpropagation*, the error gradient is propagated by multiplying the derivatives of the activation functions in each layer.

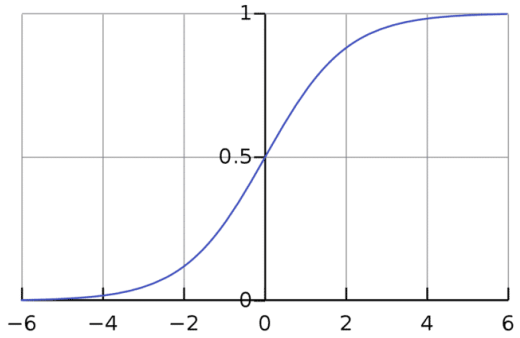
Phenomenon	Mathematical cause	Practical effect	Consequence
Vanishing gradient	**Derivatives ≈ 0** (flat functions at the edges, e.g. Sigmoid, Tanh)	Gradients are reduced at each layer	Initial layers stop learning
Exploding gradient	**Derivatives > 1** (successive multiplications)	Gradients explode	Unstable updates, chaotic training
Dying ReLU	**Derivative = 0** for negative inputs	"Off" neurons	Part of the network is not updating

To avoid these phenomena, the choice of activation becomes crucial in ensuring a stable propagation of information.

Visual comparison of key activation functions

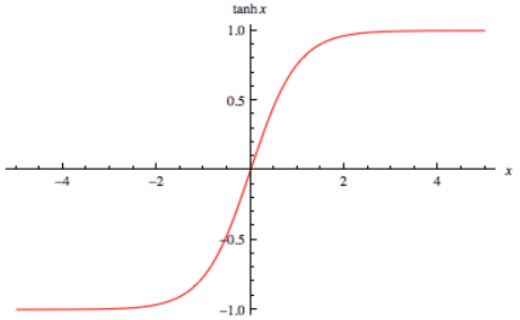
Sigmoid

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



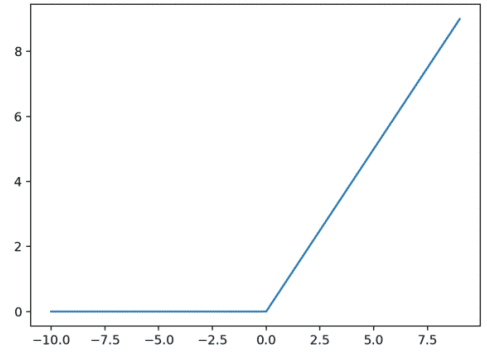
Tanh

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



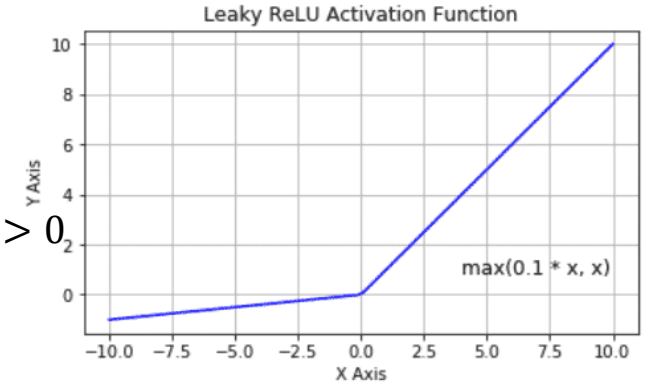
ReLU

$$f(x) = \max(0, x)$$



Leaky ReLU

$$f(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}, \alpha > 0$$



Curves show how functions respond to inputs: the flatter the curve, the more the gradient tends to cancel out

Comparison of the main activation functions

Function	Range output	Gradient behavior	Typical problems	Ideal use
Sigmoid	(0, 1)	Strong near 0, almost zero at the edges	Vanishing gradient, positive gradients only	Binary outputs (p ≈ probability)
Tanh	(–1, 1)	Strong close to 0, none at edges	Vanishing gradient, but more balanced	Small networks, classic RNNs
ReLU	[0, +∞)	Constant for $x>0$, zero for $x\leq 0$	Dying ReLU	Hidden layer standard
Leaky ReLU	(–∞, +∞)	Always >0	Slight distortion on negatives	Deep networks and stable training

In practice, ReLU and Leaky ReLU allow for a more stable gradient flow and are the dominant choices in modern deep networks.

Visual synthesis: gradient behavior

Function Type	Typical trend	Gradient	Effect on learning
Sigmoid / Tanh	Smooth, "flat" curves at the edges	Derivatives → 0	Slow or blocked learning
ReLU	Linear for x>0, flat for x≤0	Derivative = 1 or 0	Rapid, but risk of dead neurons
Leaky ReLU	Linear Anywhere	Always derived >0	Stable training, no blockage

"Flat" functions *stifle* the gradient, "broken" functions *keep it alive*.
→ For modern deep networks, **ReLU and variants** are the optimal choice.

From activation in hidden layers to output activation

- In *hidden layers* , activations control **the propagation of the gradient** (avoiding vanishing/exploding).
- In the **output layer**, on the other hand, activation translates the last representation of the network into a **format that can be interpreted** for the task.
- The choice of activation depends on the **type of problem and loss function** (regression, binary or multi-class classification).

Direct influence:

- The **shape** and **range** of the values produced.
- The **interpretability** of the result (e.g. probability).
- Consistency with the loss function (e.g. MSE, ECB, Cross-Entropy).

In hidden layers → we keep the gradient stable.

In the → output layer, we make the output comparable to the target.

Output Activation Types

Type	Formula	Output Range	Typical use	Consistent loss
Linear	$f(z) = z$	(–∞, +∞)	Regression	MSE / MFA
Sigmoid	$f(z) = \frac{1}{1 + e^{-z}}$	(0, 1)	Binary classification	Binary Cross-Entropy
Softmax	$f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$	(0, 1), sum = 1	Multi-class classification	Categorical Cross-Entropy

Sigmoid and Softmax flatten the gradient for extreme values, but in the output layer this does not compromise training: here the goal is to represent interpretable probabilities, not to optimize error propagation.

Section 6

Loss and optimization functions

From activations to loss function

- In a neural network, the activation of the output layer defines the space of predicted values:
  - Continue
  - probabilistic binary,
  - Multi-class deployments.
- The loss function formalizes the discrepancy between predicted output and target.$\mathcal{L}(y, \hat{y})\hat{y}\, y$
- The *activation-loss combination* guarantees:
  - statistical consistency (e.g. maximum likelihood),
  - derivability for backpropagation,
  - training stability.
- The combination is chosen based on the nature of the problem (regression, binary classification, multi-class).

Typical trigger–loss pairs

- **Linear Output → Mean Squared Error (MSE)**
  ↳ Regression() :Mean Squared Errors.$\hat{y} \in \mathbb{R}$

- **Sigmoid → Binary Cross-Entropy (BCE)**
  ↳ Binary classification (probability between 0 and 1.$\hat{y} \in (0,1)$):

- **Softmax → Categorical Cross-Entropy (CCE)**
  ↳ Multi-class classification (: probability distribution.$\hat{y} \in \Delta^{C-1}$)

Loss Functions: General Formulas

- **Mean Squared Error (MSE):** Used in regression → penalizes large errors in a quadratic fashion.

$$\mathcal{L}_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- **Binary Cross-Entropy (ECB):** Used with sigmoid → measure divergence between predicted and actual probability.

$$\mathcal{L}_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

- **Categorical Cross-Entropy (CCE):** Used with softmax → compares probability distributions (target vs predicted).

$$\mathcal{L}_{CCE} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

Summary of typical activation–loss combinations

Task	Output layer	Activation	Leak function
Regression	Real value	Linear	MSE
Binary classification	Probability ∈ (0.1)	Sigmoid	ECB
Multi-class classification	Odds Distributed	Softmax	EHR

Section 7

Training and regularization techniques

From loss to generalization

- Reducing training loss does not guarantee good performance on new data.
- It is necessary to estimate the capacity for generalization, i.e. the expected error on data never seen.

Type	Cause	Practical effect
Underfitting	Too simple a model	Doesn't learn the structure → errors on training and testing
Overfitting	Too complex model	"Stores" training → low error on training, high on test

Objective: to find the right balance between the ability to learn and simplicity → *generalization*.

How to prevent overfitting in neural networks

Technique	How it works	Main effect	When to use it
L2 (Weight Decay)	Penalizes large weights → adds $\lambda \sum w^2$ to loss	More stable, less complex model	Always (default)
Dropout	Randomly turns off neurons during training	Avoid co-adaptations → more robust network	Hidden layers, MLP
Early Stopping	Stop training when validation loss stops improving	Prevents learning noise	Always with val set
Data Augmentation	Generate new realistic observations	Greater variability → reduces overfitting	Vision, NLP, Genomics
Batch Normalization	Normalize activations in each mini-batch	Stabilize training and accelerate convergence	Deep CNN/MLP

Modern Regularization Strategy in Neural Networks

- Objective: to obtain **deep models that are stable, efficient and capable of generalizing** while maintaining the "alive" gradient and balanced training.
- Standard configuration in modern networks:

Level	Typical techniques	Main function
Basic (always present)	**Weight Decay (L2)**, **Batch Normalization**, **Early Stopping**	Stabilize optimization and prevent overfitting
Additional (when needed)	**Dropout**, **Data Augmentation**	Increased robustness and generalization capacity
Output layer	Consistent choice of triggering and loss (e.g. Softmax + CrossEntropy)	Interpretable and task-aligned output

Conclusion – Theoretical part

We have seen:

- Fundamentals of Deep Learning
- Neural networks, loss functions, optimizers
- Training and regularization techniques
- Now let's move on to the practical block: Deep Learning in Python!