

AI & Machine Learning for Genomic Data Science

Master in Genomic Data Science – Università di Pavia

Barbara Tarantino

AA 2025-2026

Programma – Deep Learning in Python Ecosystem

- **Section 1 – Practical Introduction**
- **Section 2 – Tensors and Autograd**
- **Section 3 – Defining a Model in Pytorch**
- **Section 4 – Training a Model in Pytorch**
- **Section 5 – Practical DL workflow in Pytorch**
- **Section 6 – Practical Notebook**

Section 1

Practical introduction

Why Python for AI

- Simple and readable language → ideal for students and researchers.
- Complete scientific and industrial ecosystem:
 - **NumPy, Pandas, Matplotlib** → data analysis and visualization.
 - **Scikit-learn** → classic machine learning models.
 - **PyTorch, TensorFlow**, → deep neural networks.
- Extensive community support: tutorials, documentation, ready-made examples.

Dal Machine Learning al Deep Learning

- **Scikit-learn (ML classico)**
 - High-level API (fit(), predict()).
 - Models: regression, trees, ensembles.
 - Suitable for small/medium scale tabular data.
- **PyTorch (DL)**
 - Direct management of multidimensional tensors.
 - Detailed training loop control.
 - Ability to leverage GPUs for intensive computing.
- Key step: From *manual feature engineering* → to machine learning of complex representations.

Why you need a framework like PyTorch

- Deep neural networks require:
 - Update millions of parameters.
 - Repeated calculations on large matrices.
- Challenges:
 - High computational load.
 - Need for automated procedures.
- PyTorch offers:
 - **Autograd** → automatic calculation of gradients.
 - **GPU support** → parallel acceleration of calculations.
 - **Modular structures (nos. Module)** for complex neural networks.

CPU vs GPU

- **CPU (Central Processing Unit)**
 - Few complex operations in sequence.
 - Optimized for general tasks (software, systems).
- **GPU (Graphics Processing Unit)**
 - Thousands of simple operations in parallel.
 - Born for graphics, today crucial for neural networks.
 - Ideal for applying the same formula to millions of values.
- Practical effect:
 - Training hours/days on CPUs → in minutes/hours on GPUs.

Concept of parallelism

- **Sequential (CPU):** Only one operation at a time.
- **Parallel (GPU):** Thousands of concurrent operations.
- Analogy:
 - CPU = a person who performs exercises in series.
 - GPU = a classroom with a thousand people doing the same exercise together.
- In DL: networks apply the same operations (matrices, derivatives) to huge amounts of numbers → parallelism is essential.

Workflow ML classico vs Deep Learning

- **Classic ML (Scikit-learn)**
 - Tabular data → chosen model (e.g. regression) → fit() → prediction.
 - Compute on CPU, fast for small datasets.
- **Deep Learning (PyTorch)**
 - Complex data (images, sequences, text).
 - Training loop: *forward* → *loss* → *backward* → *update*.
 - GPU-accelerated → intensive computations.
- Key difference: In classic ML, *features* are designed by the user; in DL, the network automatically learns the representations.

Slide 8 – CPU vs GPU numerical example

- Operation: Multiply matrices 1000×1000 .
- **CPU:**
 - Sequential calculation.
 - Time in the order of seconds.
- **GPU:**
 - Parallel calculation on thousands of units.
 - Time in the order of milliseconds.

Difference: Training from days → reduced to hours.

Scikit-learn vs PyTorch (summary)

Aspect	Scikit-learn (ML classico)	PyTorch (Deep Learning)
Type of models	Traditional (regression, trees)	Deep neural networks
API	<code>fit()</code> , <code>predict()</code>	<code>forward()</code> , <code>backward()</code> , training loop
Typical data	Small/medium tables	Images, sequences, text, big data
Calculation	CPU	CPU + GPU (parallelism)
Flexibility	Limited	High
Control	Reduced (black box)	Comprehensive, step-by-step

Towards the Fundamentals of PyTorch

Next step: Understand the fundamental structure of PyTorch, the tensor.

- What is a tensor and how does it differ from arrays and matrices.
- Main operations on tensors.
- Autograd mechanism: automatic calculation of gradients.
- These concepts are the basis for building and training neural networks in Python.

Section 2

Tensors and Autograds

What is a tensor

- Fundamental data structure in PyTorch.
- Generalize mathematical concepts:
 - Scalar (0D) → a single number.
 - Vector (1D) → list of numbers.
 - Matrix (2D) → table of values.
 - Tensor (nD) → multidimensional array (e.g. color images: 3D, image dataset: 4D).
- In DL:
 - Inputs, weights, and outputs → always tensors.

```
pip install torch
import torch
x = torch.tensor([1, 2, 3])
print(x) # 1D Tensor
```

Creating Tensors (Main Functions)

Function	Input	Typical use	Example
<code>torch.tensor(data)</code>	List/array	From existing data	<code>torch.tensor([1,2,3])</code>
<code>torch.zeros(size)</code>	Dimensions	Neutral Initialization	<code>torch.zeros((2,3))</code>
<code>torch.ones(size)</code>	Dimensions	Constant initialization	<code>torch.ones((2,2))</code>
<code>torch.full(size,val)</code>	Dim + Value	Filled tensor	<code>torch.full((2,2), 7)</code>
<code>torch.eye(n)</code>	Entire	Identity matrix	<code>torch.eye(3)</code>
<code>torch.arange(start,end,step)</code>	Extremes + pitch	Regular Sequence	<code>torch.arange(0,10,2)</code>
<code>torch.linspace(start,end,steps)</code>	Extremes + Steps	Equidistant sequence	<code>torch.linspace(0,1,5)</code>
<code>torch.randn(size)</code>	Dimensions	Normal Randoms N(0,1)	<code>torch.randn((3,3))</code>

Fundamental Inputs of Tensors

1. **size (or shape) → the shape of the tensor**

- Indicates the number of dimensions and the number of items per dimension.
- Expressed as a tuple of integers.
- Importance:
 - Each layer of a neural network expects a certain shape.
 - Shape errors are among the most common in DL.
- Examples:

```
torch.zeros((3)) # 3 Element Vector [0, 0, 0]
```

```
torch.ones((2, 3)) # 2x3 array of one
```

```
torch.rand((2, 2, 3)) # 3D Tensor (2 x 2x3 Blocks)
```

Fundamental Inputs of Tensors

2. **dtype (data type) → the type of values contained**

- Specifies the precision and type of the numbers.
- Most used in DL:
 - `torch.float32` → default, a balance between speed and precision.
 - `torch.float64` → higher precision, slower.
 - `torch.int64` → required for classification labels.
- Importance:
 - Consistent calculations → no mismatch.
 - It impacts memory, compatibility and performance.
- Example:

```
torch.ones((2,2), dtype=torch.float32) # decimali  
torch.tensor([1,2,3], dtype=torch.int64) # interi
```

Data type (dtype) handling in tensors

- In Deep Learning, calculations almost always take place at floating points (float32):
 - Weights and inputs take decimal values.
 - Operations such as splitting and derivatives require floating.
 - It is the best compromise between speed and precision.
- Rule of thumb:
 - Inputs and Weights → float32
 - Classification labels → int64
- Attention:
 - `torch.tensor([1,2,3])` → int64 (interi)
 - `torch.tensor([1.,2.,3.])` → float32 (decimali, corretto per DL)

```
a = torch.tensor([1, 2, 3])      # int64 ✗  
b = torch.tensor([1., 2., 3.])    # float32 ✓
```

Always add the . to the numbers → so PyTorch uses float32 directly.

Fundamental Inputs of Tensors

3. **device** → where the tensor is allocated

- "CPU" → standard execution on the processor.
- "cuda" → running on the GPU (if available).
- Importance:
 - GPU speeds up calculations enormously.
 - Model and data must be on the same device, otherwise error.
- Example:

```
x_cpu = torch.ones((2, 3), device="cpu")  
if torch.cuda.is_available():  
    x_gpu = torch.ones((2, 3), device="cuda")  
    print(x_gpu.device)  # cuda:0
```

GPU CPU ↔ Shift

- PyTorch does not automatically move tensors → the user has to do so.
- Use `.to("cuda")` and `.to("cpu")`.

- Practical example:

```
x = torch.ones((2,3)) # Tensor on CPU  
x_gpu = x.to("cuda") # Copied to GPU  
x_back = x_gpu.to("cpu") # Reported on CPU
```

- Typical workflow:

- Training → GPU (speed).
- CPU → analysis/plot (NumPy/Matplotlib compatibility).

Properties of tensors

Useful properties:

- `.shape` → returns the shape (size of the tensor).
- `.ndim` → number of dimensions.
- `.dtype` → type of data contained (float32, int64, ...).
- `.device` → where the tensor (CPU or GPU) is located.
- `.size(dim)` → size of a specific axis.

Essential for controlling the structure, type, and location of data while working in DL.

Properties of tensors

```
x = torch.rand((2, 3), dtype=torch.float32, device="cpu")
```

```
print(x.shape)    # torch. Size([2, 3])
print(x.ndim)     # 2
print(x.dtype)    # torch.float32
print(x.device)   # cpu
print(x.size(0)) # 2 (rows)
print(x.size(1)) # 3 (colonnes)
```

Fundamental operations (1): Indexing and slicing

- They allow you to access specific elements or sections of a tensor.

```
x = torch.tensor([[1, 2, 3],  
                  [4, 5, 6]])
```

```
print(x[0, 1]) # item (row 0, column 1) → 2  
print(x[:, 0]) # prima colonna → [1, 4]  
print(x[1, :]) # second line → [4, 5, 6]
```

Used continuously to access batches, channels, rows, or columns.

Fundamental operations (2): Element-wise

- Apply **element-by-element operations** on tensors of the same shape.

```
a = torch.tensor([1, 2, 3])  
b = torch.tensor([4, 5, 6])
```

```
print(a + b) # somma → [5, 7, 9]  
print(a - b) # subtraction → [-3, -3, -3]  
print(a * b) # multiplication → [4, 10, 18]  
print(a / b) # divisione → [0.25, 0.4, 0.5]
```

They underpin all transformations in neural network layers.

Fundamental operations (3): linear algebra

- Neural networks are based on multiplications of matrices and vectors.

```
A = torch.tensor([[1., 2., 3.],  
                 [4., 5., 6.]])
```

```
B = torch.tensor([[1., 2.],  
                 [3., 4.],  
                 [5., 6.]])
```

```
print(torch.matmul(A, B)) # equivalent: torch.mm(A,B) or A @ B  
# [[22., 28.],  
#  [49., 64.]]
```

Each layer of a network (input → hidden → output) is a multiplication matrix + bias.

Key operations (4): reductions and statistics

- Functions that compress a tensor to simpler values.

```
x = torch.tensor([[1., 2., 3.],  
                  [4., 5., 6.]])
```

```
print(x.sum()) # 21.0 (sum of all elements)  
print(x.mean()) # 3.5 (media)  
print(x.max()) # 6.0 (maximum value)  
print(x.min()) # 1.0 (minimum value)  
print(x.argmax()) # 5 (index of maximum)
```

Useful for calculating metrics, normalizations and cost functions.

Key Operations (5): Broadcasting

- If the dimensions do not match, PyTorch automatically adapts the tensors.

```
a = torch.ones((2, 3))  
b = torch.tensor([1, 2, 3])
```

```
Print(A+B)  
# [[2., 3., 4.],  
#  [2., 3., 4.]]
```

Essential for managing batches of data without manually duplicating data
Tensors.

Tensor vs NumPy array

- Both = multidimensional arrays.
- Very similar syntax: shape, indexing, slicing.
- Easy conversion between the two (`.numpy()`, `torch.from_numpy()`).
- **Key differences:**
- **NumPy**
 - Solo CPU (no GPU).
 - No gradient support.
 - Great for traditional data analysis.
- **PyTorch Tensor**
 - CPU **e** GPU (CUDA).
 - Integrated with Autograd (automatic gradient calculation).
 - Optimized for Deep Learning.

Tensor vs NumPy array

- Comparative example:

```
import numpy as np
a = np.array([1, 2, 3])
b = torch.tensor([1., 2., 3.])

print(type(a))    # <class 'numpy.ndarray'>
print(type(b))    # <class 'torch. Tensor'>
```

In Deep Learning we use PyTorch tensors because they allow training on GPUs and gradient calculation (Autograd).

Why Autograd is needed

- A neural network is a complex function:

$$y = f(x; \theta)$$

- where: x = input, θ = parameters (weights, bias), y = output.
- To train it: we minimize a loss function. $L(y, y_{true})$
- It is necessary to calculate: $\frac{\partial L}{\partial \theta}$, i.e. how the loss varies with respect to the parameters.

Autograd:

- Use the chain rule to calculate gradients.
- Avoid manual calculation (impossible with millions of parameters).

Activate Autograd (requires_grad)

- To tell PyTorch to calculate gradients:
 - use `requires_grad=True` on the tensor.

```
x = torch.tensor(2.0, requires_grad=True)  
y = x**2 + 3 * x # f(x) = x2 + 3x
```

- PyTorch now knows that y depends on x and can compute $\frac{dy}{dx}$

Calculating gradients

- `.backward()` calculates the derivatives of the function with respect to variables with `requires_grad=True`.

```
y.backward()  
print(x.grad)
```

This example shows that Autograd calculates the derivative exactly, then useful for updating the parameters of a network.

Forward e Backward pass

1. Forward pass

1. Input → model (function) → output. f
2. The loss is calculated by comparing output and target.

2. Backward pass

1. Autograd propagates the derivative of the loss with respect to the parameters.
2. Each weight receives its own gradient.

3. Update

1. The optimizer modifies the parameters:

$$\theta_{new} = \theta - \eta \cdot \frac{\partial L}{\partial \theta}$$

1. dove = learning rate. η

This cycle = backpropagation, the heart of Deep Learning.

Example with multiple variables

- PyTorch handles multivariate functions automatically.

```
a = torch.tensor(2.0, requires_grad=True)  
b = torch.tensor(3.0, requires_grad=True)
```

```
z = a * b + b**2      # f(a,b) = ab + b2  
z.backward()
```

```
print(a.grad)    # df/da = b  
print(b.grad)    # df/db = a + 2b
```

Autograd in DL practice

- When training a network:
 - Forward pass → predictions + loss calculation.
 - Backward pass → Autograd calculates weight gradients.
 - Optimizer → updates weights using gradients.
- Without Autograd: it is impossible to train complex models.
- With Autograd: training becomes automatic, efficient, and scalable on GPUs.

Section 3

Defining a Model in PyTorch

The torch.nn module

The `torch.nn` module (`import torch.nn as nn`) provides the building blocks for building neural networks:

- Layers
 - nos. `Linear` → linear transformations (fully connected)
 - nos. `Conv2d` → convolutions (images)
 - nos. `LSTM`, nos. `CRANE` → sequences/text
- Activation functions
 - `nn.ReLU`, `nn.Sigmoid`, `nn.Tanh`
- Cost (loss) functions
 - nos. `MSELoss()` → regression
 - nos. `CrossEntropyLoss()` → classification

These tools are combined to create complex models.

`nos. Module`: The parent class of the models

- Every model in PyTorch inherits from `nos. Module`.
- `nos. Module` automatically manages:
 - recording of parameters (weights, bias),
 - integration with Autograd (gradient calculation),
 - move to CPU/GPU (`.to(device)`),
 - Saving and loading templates (`state_dict`).

Without `nos. Module` you should write everything by hand.

Typical structure of a PyTorch model

A template always contains two parts:

1. `__init__` (Manufacturer)

- Here you define the fixed attributes of the model:
 - layer (es. so-called. Linear),
 - activation functions (e.g. nos. ReLU),
 - other components needed.

2. `forward(self, x)`

- This is where you define the flow of data through layers.
- You don't create new layers here: you use those already declared in `__init__`.

In summary: `__init__` → what the model contains, `forward`, → how the data passes through the model.

Why use super().__init__()

- Each PyTorch model inherits from `nn.Module`.
- In the constructor (`__init__`) the following must always be entered:

```
super().__init__()
```

It is used for:

- register the layers defined as attributes,
- use `model.parameters()` on the parameters,
- integrate with Autograd (gradients),
- allow saving (`state_dict`) and moving to CPU/GPU.

Rule of thumb: in every model that inherits from `nn.Module`,
`super().__init__()` is
obligatory.

Single-layer model

```
import torch
import torch.optim as optim

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(2, 1) # 2 input → 1 output

    def forward(self, x):
        return self.fc(x)
```

- Attribute: `self.fc` = a linear layer.
- Forward pass: Apply directly. $y = Wx + b$

Multi-layered model

- Attributes: two linear layers + activation.
- Forward pass: concatenation of layers → step-by-step data flow.

Flow:

$$x \rightarrow \text{Linear}(2 \rightarrow 4) \rightarrow \text{ReLU} \rightarrow \text{Linear}(4 \rightarrow 1) \rightarrow y$$

Multi-layered model

```
class TwoLayerNet(nn. Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn. Linear(2, 4)    # primo layer: 2 → 4
        self.relu = nos. ReLU() # activation
        self.fc2 = nos. Linear(4, 1) # Second layer: 4 → 1

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

How many units to choose in layers?

- **Input layer** → dimensions fixed by the data (e.g. 2 variables → `in_features=2`).
- **Output layer** → dimensions fixed by the issue:
 - regression → `out_features=1`,
 - binary classification → `out_features=2`,
 - multi-class classification → number of classes.
- **Hidden layer** → free choice, depending on:
 - complexity of the problem (more complex → more units),
 - amount of data (too many neurons without enough data = risk of overfitting),
 - experimentation (trial-and-error, tuning).

Creating an instance

- Class is just the abstract definition.
- To use the template, you need to create an object (instance):

```
model = TwoLayerNet()  
print(model)
```

Output:

```
TwoLayerNet(  
    (fc1): Linear(in_features=2, out_features=4, bias=True)  
    (relude): ReLU()  
    (fc2): Linear(in_features=4, out_features=1, bias=True)  
)
```

Now the `model` has the parameters initialized and is ready to receive input.

Using the model (forward pass)

```
x = torch.tensor([[1.0, 2.0]]) # input with 2 variables  
y_pred = model(x)           # forward pass  
print(y_pred)
```

- `model(x)` calls `forward(x)` automatically.
- The data passes through the defined layers.
- Output = prediction of the model.

The model behaves like a mathematical function learned from the data.

Synthesis

- In `__init__`: Define layers and activations (fixed attributes).
- In `forward`: Describe the flow of data.
- 1 layer → linear, simple model.
- Multiple layers → more expressive model, can capture nonlinear relationships.
- Dimensions of the → input/output layers determined by the data and the problem, hidden chosen by the user.
- Create an instance (`model = TwoLayerNet()`) → ready-to-use model.

Section 4

Training a model in PyTorch

Cost Function

- A neural network approximates a function that produces predictions. $f(x; \theta)\hat{y}$
- To evaluate the goodness of the model, we compare the predictions with the real values. $\hat{y} y$
- The cost function assigns a numeric value that measures the error. $\mathcal{L}(y, \hat{y})$

The goal of training is to minimize the cost function by changing the parameters. θ

Examples of Cost Functions

- Regression (continuous values):
 - nos. `MSELoss()` → mean squared error.
- Binary classification (2 classes):
 - nos. `BCELoss()` or nos. `BCEWithLogitsLoss()`.
- Multi-class classification (multiple classes):
 - nn. `CrossEntropyLoss()`.

```
import torch.nn as nn  
criterion = nn. MSELoss()
```

The choice of loss always depends on the type of problem.

Practical example: calculation of loss

```
import torch
import torch.nn as nn

# Model predictions
y_pred = torch.tensor([2.5, 0.0, 2.1])
# Real Values
y_true = torch.tensor([3.0, -0.5, 2.0])

criterion = nn.MSELoss()
loss = criterion(y_pred, y_true)

print(loss.item()) # 0.1083
```

From model to optimization

- The `torch.nn` module is used to:
 - define the architecture of the model (layers and activations),
 - Specify the cost function.
- However, a model cannot learn without a mechanism that updates the parameters.
- This role is entrusted to `torch.optim` (`import torch.optim as optim`), which implements several numerical optimization algorithms.

What is an optimizer?

- An optimizer is an iterative algorithm that, given the gradients calculated by Autograd, updates the model parameters to reduce the cost function.
- General Formula (Gradient Descent):

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_{\theta} \mathcal{L}(\theta)$$

- θ : Model parameters.
- η : learning rate.
- $\nabla_{\theta} \mathcal{L}$: gradient of the loss with respect to the parameters.

The optimizer controls the speed and stability with which the model converges.

General syntax

- In PyTorch all optimizers are located in the `torch.optim` module.

```
import torch.optim as optim

optimizer = optim.XXX(
    model.parameters(), # parameters to update
    lr=0.01,           # learning rate
    altri_argomenti... # optional, depend
                           by the algorithm
)
```

- `optim.XXX` = name of the optimizer (e.g. SGD, Adam, ...).

General syntax

- Main inputs:
 - `model.parameters()` tells → which parameters to update.
 - LR (Learning Rate) → size of the steps.
 - other hyperparameters → depend on the optimizer (e.g. momentum).
- Output:
 - an optimizer object,
 - which updates the parameters in place when you call `optimizer.step()`.
- Optimizers:
 - `optimizer = optim.SGD(model.parameters(), lr=0.01, ...)`
 - `optimizer = optim.RMSprop(model.parameters(), lr=0.01, ...)`
 - `optimizer = optim.Adam(model.parameters(), lr=0.01, ...)`

Comparative summary

Optimizer	Strengths	Key Parameters	When to use it
SGD	Simple, controllable, theoretically solid	lr, momentum, weight_decay	Simple problems, baseline
Adam	Adaptive, fast, widely used	lr, betas, weight_decay	Default in many practical cases
RMSprop	Suitable for noisy sequences and data	lr, alpha, weight_decay	Non-stationary data, RNN

General structure of a training era

To train a PyTorch model, each iteration (epoch) follows these steps:

- 1.Calculate the predictions (forward pass).
- 2.Calculate the cost function (loss).
- 3.Reset the previous gradients → `optimizer.zero_grad()`.
- 4.Calculate the current gradients → `loss.backward()`.
- 5.Update the parameters → `optimizer.step()`.

These last 3 steps are what make learning possible.

optimizer.zero_grad()

- By default, PyTorch accumulates gradients with each `backward()` call.
- If we don't reset them, the gradients add up across epochs → incorrect results.
- `optimizer.zero_grad()` is used to clean up the gradients before recalculating them.

```
optimizer.zero_grad() # Reset Gradients
```

This is the first statement to be executed in each update cycle.

`loss.backward()`

- Calculate the loss gradients with respect to the model parameters.
- Use Autograd to backpropagate errors.

`loss.backward()`

- After this command:
 - Each model parameter (`model.parameters()`) has its own gradient stored in `.grad`.
 - These gradients are the basis for upgrading.

`optimizer.step()`

- Use the gradients calculated in `loss.backward()`.
- Update the parameters according to the chosen optimizer rule (SGD, Adam, etc.).

`optimizer.step()`

- It doesn't return output → directly changes model parameters during training.

Summary scheme

```
# 1. Reset gradients  
optimizer.zero_grad()
```

```
# 2. Backward pass (gradient calculation)  
loss.backward()
```

```
# 3. Update parameters  
optimizer.step()
```

Always **mandatory order**:

1. zero_grad() → cleaning,
2. backward() → gradient calculation,
3. step() → update.

Training loop in pratica

```
for epoch in range(5):
    y_pred = model(x)                      # 1. forward
    loss = criterion(y_pred, y)              # 2. loss

    optimizer.zero_grad()                   # 3. reset grad
    loss.backward()                         # 4. backward
    optimizer.step()                        # 5. update

    print(f"Epoch {epoch+1}, Loss = {loss.item():.4f}")
```

```
Epoch 1, Loss = 1.2534
Epoch 2, Loss = 0.8421
Epoch 3, Loss = 0.6237
Epoch 4, Loss = 0.5029
Epoch 5, Loss = 0.4213
```

Comparison with Scikit-learn

- **Scikit-learn**
 - `model.fit(X, y)` → easy-to-use, automatic training.
 - Little control over details.
- **PyTorch**
 - You have to write the training loop manually.
 - More code, but more control over:
 - architecture
 - optimizer
 - cost functions,
 - Updating parameters.

PyTorch is designed for research and complex models.

Synthesis

To train a PyTorch model, you need:

- 1. Loss function** → measures error.
- 2. Optimizer** → updates parameters with gradients.
- 3. Training loop** → performs:
 1. forward pass,
 2. loss calculation,
 3. backward pass,
 4. update.

With these three components, the network can learn from data.

Section 5

Practical DL workflow in PyTorch

Introduction to the final part

- So far, we've seen the fundamental building blocks:
 - Tensors, Autograd, nos . Module, loss, optimizers.
- Now let's combine the concepts into a complete end-to-end workflow:
 - Data Preparation → Training → Validation → Rescue.
- This schema is the basis of every Deep Learning project in PyTorch.

Tensore, Dataset e DataLoader

- **torch. Tensor**: A container for numerical data.
- **TensorDataset**: Matches input tensors (X) and target tensors (y).
 - Returns pairs (X[i], y[i]).
- **DataLoader**: Creates batches from the dataset and provides them to the model.
 - During training, the DataLoader provides the model with small groups of examples (*batches*, e.g. 32), on which it calculates the loss, updates the weights and then moves on to the next batch, until an epoch is completed.

Tensore → Dataset (X,y) → DataLoader (batch)

Dataset and DataLoader creation

```
from torch.utils.data import TensorDataset, DataLoader

# Input tensors (floats) and targets (integers)
X_tensor = torch.tensor(X, dtype=torch.float32)      # valori numerici
y_tensor = torch.tensor(y, dtype=torch.long)          # etichette classi

# dataset = coppie (X, y)
dataset = TensorDataset(X_tensor, y_tensor)

# dataloader = batch dal dataset
train_loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Dataset and DataLoader creation

Input:

- `dtype=torch.float32` → dati continui (features).
- `dtype=torch.long` → class labels (0,1,2,...).

Note: `torch.long` in PyTorch = `torch.int64`, we use `torch.long` because some losses (nos. `CrossEntropyLoss`) explicitly require this type.

Training and Test Set Division

- We separate the data into training and testing:

```
from sklearn.model_selection import train_test_split  
  
x_train, x_test, y_train, y_test = train_test_split(  
    X, y, test_size=0.2, random_state=42  
)
```

Training and Test Set Division

Then we create separate DataLoaders:

```
train_dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32),  
                           torch.tensor(y_train, dtype=torch.long))  
  
test_dataset = TensorDataset(torch.tensor(X_test, dtype=torch.float32),  
                           torch.tensor(y_test, dtype=torch.long))  
  
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)  
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

We now have `train_loader` and `test_loader`, where the batches (`X_batch`, `y_batch`) come from.

Training sul Training set

```
for epoch in range(num_epochs):
    for X_batch, y_batch in train_loader:
        y_pred = model(X_batch)           # forward pass
        loss = criterion(y_pred, y_batch)  # calcolo loss

        optimizer.zero_grad() # reset gradients
        loss.backward() # gradient calculation
        optimizer.step() # update parameters
```

- Here you calculate the gradients → are used to update the weights.
- With each batch:
 - the model learns from the data,
 - The weights are changed to reduce the loss.

Validation on the Test Set

- In validation, gradients are not calculated → because the parameters do not have to be updated with respect to training.
- We use the `torch.no_grad()` context:

```
with torch.no_grad(): # disable Autograd in this block
    for X_batch, y_batch in test_loader:
        y_pred = model(X_batch) # forward with fixed weights
        acc = (y_pred.argmax(1) == y_batch).float().mean()
```

- ACC compares the predicted class (index of the maximum value in `y_pred`) with the real class `y_batch`; counts how many are correct, converts them to 0/1 values (`float()`), and calculates the **average = accuracy of the batch**.

What is `torch.no_grad()`?

- In validation, the model only has to predict using the learned weights.
- `torch.no_grad()` is a context → all code inside does not compute gradients.
- It is used to:
 - reduce memory,
 - speed up inference,
 - Avoid accidental weight updates.

Saving and loading the model

- PyTorch saves only the weights (parameters) of the model, contained in the `state_dict`.
- This is more flexible than saving the entire object.

```
# Saving weights
torch.save(model.state_dict(), "model.pth")
```

```
# Loading weights
model = MyModel() # same architecture defined in code
model.load_state_dict(torch.load("model.pth"))
model.eval() # evaluation mode
```

Why save only the weights?

- Lighter and more portable file.
- Environment-agnostic → just have the same model definition.
- It allows you to reuse a trained model without retraining it.

Visual diagram of the complete workflow

Dataset → DataLoader → Model (nos. Module) → Loss → Optimizer

- Training (forward, backward, update)
- Validation/Test (no_grad)
- Rescue (state_dict)

This is the standard pipeline for every PyTorch project.

From workflow to notebook

- We've got you covered:
 - Dataset e DataLoader,
 - Training with gradients,
 - Gradient-free validation,
 - Saving model weights.
- In the practical notebook:
 - Esempio end-to-end (Breast Cancer dataset).
 - Training + validation with loss/accuracy curves.
 - Saving and reloading the model.

From theory to practice [?DeepLearning_Notebook.ipynb](#)

Conclusion Day 3

- Today, we introduced the core principles of Deep Learning:
 - Theoretical foundations: artificial neurons, activation functions, backpropagation, loss functions and optimizers.
 - Training and regularization techniques (dropout, batch norm, early stopping).
 - Practical part in Python with PyTorch: tensors, autograds, model construction, loss and optimizers.
 - End-to-end workflow: Dataset & DataLoader, training, validation, model saving.
 - We have therefore completed the transition from theory to practice of Deep Learning.

Prossimo passo (Day 4): Computer Vision for Medicine.

See you tomorrow!