# Introduction to Neural Networks and Pytorch (M4)

| | |
|---|---|
| Created | @December 16, 2025 11:44 AM |
| Module Code | IBMM04 : Introduction to Neural Networks and Pytorch |

# Module 1: Tensors and Dataset

## Overview of Tensors

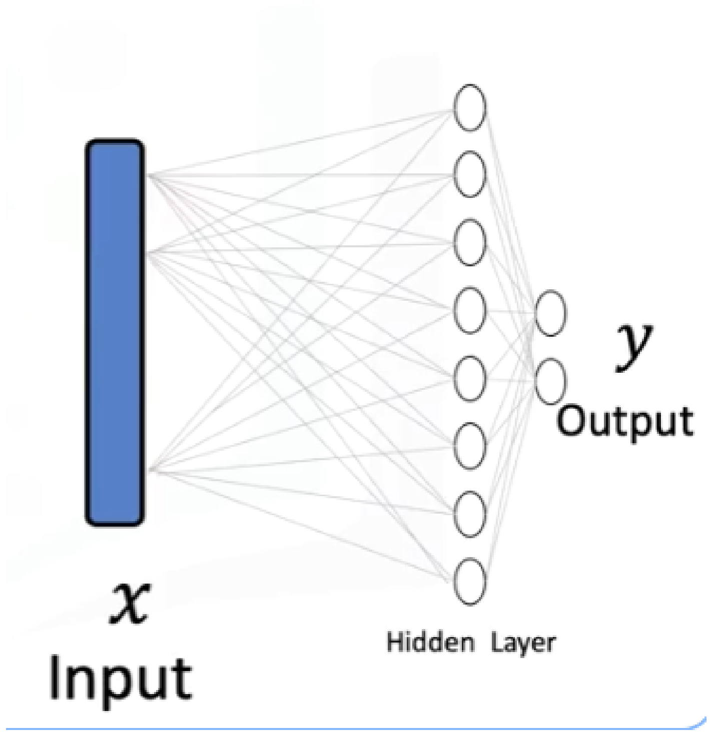
### 1. Neural Networks as Mathematical Functions

Formally, a neural network represents a **parameterized function**:

$$f_\theta : \mathbb{R}^n \to \mathbb{R}^m$$

- $x \in \mathbb{R}^n$: input tensor
- $y \in \mathbb{R}^m$: output tensor
- $\theta$: set of learnable parameters (weights and biases)

The network applies a **composition of linear transformations and nonlinear activation functions** to map inputs to outputs.




## 2. Tensor: Formal Definition

A **PyTorch tensor** is a **multi-dimensional array** that:

1. Stores numerical data
2. Supports efficient mathematical operations
3. Supports **automatic differentiation**

Mathematically, a tensor is an element of:

Mathematically, a tensor is an element of:

$$\mathbb{R}^{d_1 \times d_2 \times \cdots \times d_n}$$

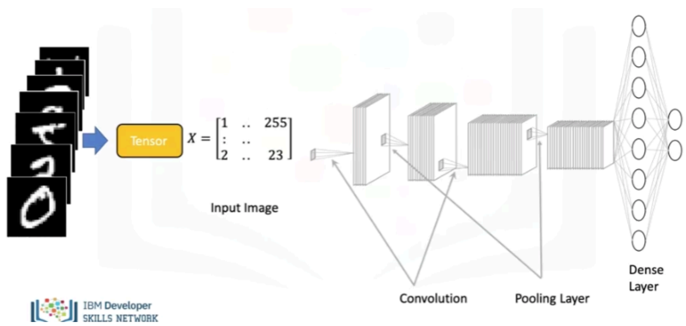
where $n$ is the tensor's **rank (number of dimensions)**.

## 3. Tensor Dimensionality (Rank)

| Rank | Name | Example |
|---|---|---|
| 0 | Scalar | 555 |
| 1 | Vector | [x1,x2,x3] |
| 2 | Matrix | X∈Rm×n |
| 3+ | Higher-order tensor | Images, videos |

Example:

- Grayscale image → H×W
- RGB image → H×W×3H
- Batch of images → N×H×W×C
- 




## 4. Role of Tensors in Neural Networks

In PyTorch, **every component of a neural network is represented as a tensor**:

| Component | Tensor Representation |
|---|---|
| Input data | Tensor |
| Model parameters | Tensor |
| Intermediate activations | Tensor |
| Output | Tensor |
| Gradients | Tensor |

## 5. Linear Transformation Using Tensors

A fully connected layer performs:

y=Wx + b

Where:

Where:

- $x \in \mathbb{R}^n$: input tensor
- $W \in \mathbb{R}^{m \times n}$: weight tensor
- $b \in \mathbb{R}^m$: bias tensor
- $y \in \mathbb{R}^m$: output tensor

This is a **matrix–vector multiplication** followed by vector addition.

---

## 6. Tensor Operations

PyTorch supports:

- Element-wise operations
- Matrix multiplication
- Broadcasting
- Reduction operations (sum, mean)
- Reshaping and slicing

These operations form the computational graph used during training.

---

## 7. Automatic Differentiation (Autograd)

PyTorch uses **reverse-mode automatic differentiation**.

## Key concept:

If a tensor has, then

It means:

> "Track this tensor so we can compute derivatives"

```
requires_grad =True
```

PyTorch:

1. Tracks all operations applied to it
2. Builds a **dynamic computation graph**
3. Computes gradients via backpropagation

Mathematically:

$$\frac{\partial L}{\partial \theta}$$

Where:

- $L$: loss function
- $\theta$: model parameters

---

## 8. Parameters in PyTorch

**What are parameters?**

> Parameters = learnable tensors

Model parameters are tensors wrapped using:

```
torch.nn.Parameter
```

Properties:

- `requires_grad=True` by default
- Automatically registered inside `nn.Module`
- Updated during optimization

Example parameters:

- Weight matrices

- Bias vectors

## 9. Gradients and Training

Gradients answer:

> “How should I change this weight to reduce error?”

Training consists of:

1. **Forward pass** – compute predictions
2. **Loss computation** – measure error
3. **Backward pass** – compute gradients
4. **Parameter update** – optimization step

Gradient update rule:

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$$

Where:

- η: learning rate

## 10. PyTorch Tensors vs NumPy Arrays

| Feature | NumPy | PyTorch |
|---|---|---|
| GPU support | ❌ | ✅ |
| Autograd | ❌ | ✅ |
| Deep learning | ❌ | ✅ |
| Dynamic graph | ❌ | ✅ |

PyTorch tensors can be converted to NumPy arrays **without data copying** when on CPU.

## 11. GPU Acceleration

PyTorch tensors can be placed on:

- CPU
- CUDA-enabled GPU

GPU execution enables:

- Parallel computation
- Faster matrix operations
- Efficient training of deep networks

---

## 12. Dataset Class in PyTorch

The `Dataset` class provides an abstraction for data handling.

It requires implementing:

- `__len__()` → number of samples
- `__getitem__(index)` → return one sample

Benefits:

- Lazy loading
- Memory efficiency
- Easy batching via `DataLoader`
- Shuffling and parallel data loading

---

## 13. Data Pipeline in PyTorch

Typical workflow:

1. Raw data → Dataset
2. Dataset → DataLoader
3. DataLoader → batched tensors
4. Batches → neural network

This pipeline ensures scalable training on large datasets.

# 1-D Tensors in PyTorch

## 1. Tensor: Formal Definition

A **tensor** in PyTorch is a **multi-dimensional numerical array** with:

- A fixed **data type (`dtype`)**
- A fixed **shape**
- Optional **gradient tracking** (`requires_grad`)

Mathematically, a **1-D tensor** is a vector:

$x \in Rn$

## 2. Tensor Rank (Dimensionality)

| Tensor | Rank | Mathematical Meaning |
|---|---|---|
| 0-D | 0 | Scalar |
| 1-D | 1 | Vector |
| 2-D | 2 | Matrix |

- `ndimension()` returns the **rank**
- `size()` returns the **number of elements per dimension**

Example:

$x = [7,4,3,2,6] \Rightarrow rank=1, size = (5)$

## 3. Data Types (`dtype`)

A tensor stores **only one data type**.

Common PyTorch dtypes:

- `torch.float32` (default for floats)
- `torch.float64`

- `torch.int32`
- `torch.int64`
- `torch.uint8` (used in images)

Why dtype matters:

- Memory usage
- Numerical precision
- GPU compatibility

---

## 4. Tensor Creation

A tensor can be created from a Python list:

A tensor can be created from a Python list:

$$x = \mathrm{torch.tensor}([7, 4, 3, 2, 6])$$

You may **explicitly specify dtype**:

$$x = \mathrm{torch.tensor}([1, 2, 3], dtype = torch.int32)$$

Even if input values are floats, the specified dtype **overrides** them.

---

## 5. Type Casting

Type casting converts a tensor to another dtype:

xfloat=x.type(torch.FloatTensor)

xfloat=x.float()

---

## 6. Indexing (Element Access)

Indexing follows **zero-based indexing**:

Indexing follows **zero-based indexing**:

$$x[i] \Rightarrow \text{element at position } i$$

Important:

- `x[i]` is **still a tensor**, not a Python number

To extract a Python scalar:

$$x[i].\text{item}()$$

## 7. Slicing

Slicing follows Python semantics:

$$x[a:b] \Rightarrow \text{elements } a \text{ to } b-1$$

Example:

$$x = [100, 4, 3, 2, 0] \Rightarrow x[1:4] = [4, 3, 2]$$

## 9. Reshaping 1-D → 2-D

Neural networks often require **2-D input tensors**.

Reshape using `view()`:

$$x \in \mathbb{R}^n \Rightarrow x.\text{view}(n, 1)$$

Using `-1` lets PyTorch **infer dimension size**:

$$x.\text{view}(-1, 1)$$

## 10. Tensor ↔ NumPy Conversion

NumPy → Tensor

x_{\text{torch}} = \text{torch.from_numpy}(x_{\text{numpy}})

Tensor → NumPy

xnumpy=x.numpy()

⚠️ **Important**:

- Both share the **same memory**
- Modifying one modifies the other

## 11. Tensor ↔ Python List

Convert tensor to list:

$$x.\text{tolist}()$$

Used when Python-native data is required.

## 12. Vector Operations on 1-D Tensors

### 12.1 Vector Addition

z=u+v

Condition:

- Same shape
- Same dtype

### 12.2 Scalar Multiplication

z=αu

Each element multiplied by scalar.

## 12.3 Hadamard Product (Element-wise)

$$\mathbf{z}_i = \mathbf{u}_i \cdot \mathbf{v}_i$$

Implemented using:

```
u * v
```

## 12.4 Dot Product

$$\mathbf{u} \cdot \mathbf{v} = \sum_i u_i v_i$$

Returns a **scalar tensor**.

---

## 13. Broadcasting

Broadcasting allows operations between tensors of different shapes:

x+c

Scalar `c` is automatically expanded to match tensor shape.

---

## 14. Reduction Operations

Operate over all elements:

| Function | Meaning |
|---|---|
| `mean()` | Average |
| `sum()` | Total |
| `max()` | Maximum |
| `min()` | Minimum |

Result:

- **Scalar tensor**

---

## 15. Universal Functions (Element-wise)

Functions applied to **each element independently**:

$$y_i = f(x_i)$$

Examples:

- `torch.sin()`

- `torch.exp()`
- `torch.log()`

Used heavily in:

- Activation functions
- Feature transformations

---

## 16. `linspace()` Function

Generates evenly spaced values:

$x = linspace(a, b, n)$

Used for:

- Plotting functions
- Signal generation
- Numerical analysis

---

## 17. Plotting with PyTorch

To plot:

1. Convert tensor → NumPy
2. Use Matplotlib

Because Matplotlib **does not accept tensors directly**

---

## 18. Why 1-D Tensors Matter in Deep Learning

1-D tensors represent:

- Feature vectors
- Weight vectors
- Bias terms
- Time-series data
- Embeddings

They are the **fundamental unit** of neural computation.

## Two-Dimensional Tensors

| House | rooms | AGE | .. | Price |
|---|---|---|---|---|
| 1 | 2 | 4 | | 40 |
| 2 | 3 | 10 | | 50 |
| 3 | 4 | 12 | | 23 |
| 4 | 5 | 34 | | 24 |

$$X = \begin{bmatrix} 2 & 3 & .. & 40 \\ 3 & 10 & .. & 50 \\ \boxed{5 & 34 & .. & 24} \end{bmatrix}$$

### 1. Why `view()` / reshaping matters in real neural networks

In theory, tensors are “just arrays”.

In practice, **wrong shape = model breaks**.

Example (why 2D is mandatory)

A linear layer in PyTorch expects:

```
(batch_size, features)
```

If you give:

```
(features,)
```

→ PyTorch **does not know how many samples** you have.

That’s why:

```
x = torch.tensor([1,2,3,4,5])
x = x.view(1, -1)# 1 sample, 5 features
```

**Rule**

- 1D → data point
- 2D → batch of data points

---

## 2. Difference between * and `@` (this causes most bugs)

This is **critical**, so I'll be precise.

### → Hadamard (element-wise)

```
X * Y
```

Used for:

- Feature-wise scaling
- Attention masking
- Gating mechanisms

### `@` or `mm()` → Linear transformation

```
X @ W
```

Used for:

- Dense layers
- Projection layers
- Embeddings
- Transformers

If you use `*` where `@` is required → **wrong math, silent bug**

---

## 3. Broadcasting rules (what PyTorch actually does)

Broadcasting is **not magic**.

## Rule (simplified):

Two dimensions are compatible if:

- They are equal, OR
- One of them is `1`

Example:

```
X.shape = (32,10)# batch of 32 samples
b.shape = (10,)# bias
```

PyTorch treats bias as:

```
(1,10) → (32,10)
```

That's why this works:

```
X + b
```

This is **exactly how bias works in neural networks**.

---

## 4. Why NumPy ↔ Torch memory sharing is dangerous

You already saw they share memory.

Here's why that matters **during training**.

```
x = torch.from_numpy(arr)
arr[0] =999
```

Now:

```
x[0] ==999
```

If:

- `x` is input to model
- you modify NumPy accidentally

→ **training becomes nondeterministic**

**Best practice**

```
torch.tensor(arr)# creates a copy
```

---

## 5. Gradients: why tensors are special

Normal arrays:

```
numbers →output
```

Tensors with autograd:

```
numbers → computation graph →output
```

## Parameter tensor

```
w = torch.tensor([1.0,2.0], requires_grad=True)
```

PyTorch now tracks:

- How `w` is used
- Which operations depend on `w`

After:

```
loss.backward()
```

You get:

```
w.grad
```

This is:

```
∂loss / ∂w
```

That's **training**.

---

## 6. Why tensors must have a single dtype

You might wonder:

> Why can't tensors mix int + float?

Because:

- GPUs require **uniform memory**
- Vectorized operations assume **fixed-size elements**
- Backprop math requires consistency

That's why:

```
torch.tensor([1,2,3.5])
```

→ automatically becomes `float`

---

## 7. 2D tensors = datasets (formal ML view)

A dataset tensor `X` means:

```
X ∈ ℝ^(N × D)
```

Where:

- `N` = number of samples

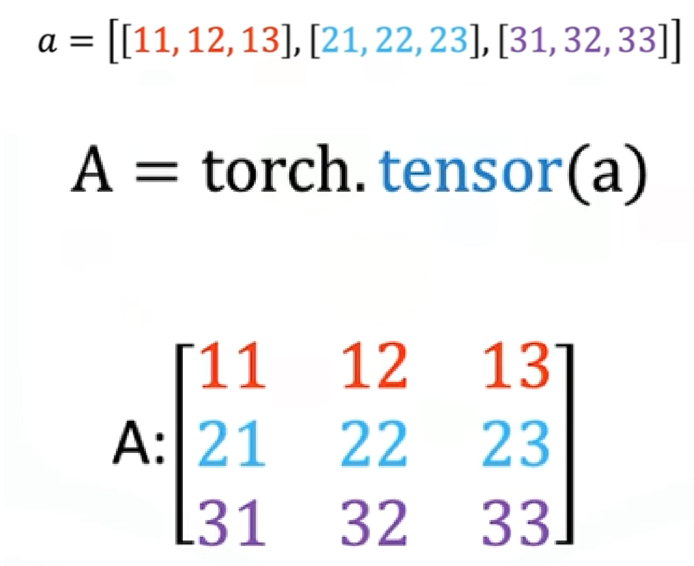
- `D` = number of features

Targets:

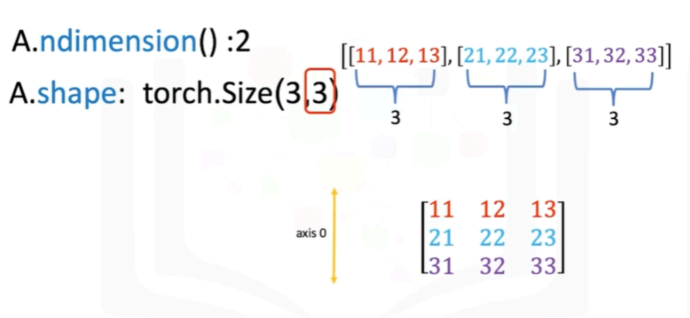
```
y ∈ ℝ^N   (regression)
y ∈ ℕ^N   (classification labels)
```

This is why:

- rows = samples
- columns = features




---



## 8. Matrix multiplication = neurons firing

One neuron mathematically:

```
y = w·x + b
```

Batch version:

```
Y = XW + b
```

Where:

- `X`: (batch, features)
- `W`: (features, neurons)
- `Y`: (batch, neurons)

**Every deep learning model is just stacked matrix multiplications + non-linearities.**

---

## 9. Images → tensors (what actually happens)

Grayscale image:

```
(H, W)
```

Color image:

```
(C,H, W)
```

Batch of images:

```
(N,C,H,W)
```

This is why CNNs expect **4D tensors**, not because of PyTorch — but because of math.

# Differentiation in PyTorch

## 1. Why differentiation exists in deep learning

Neural networks learn by **optimizing parameters** (weights and biases).

To optimize:

- We define a **loss function**
- We compute **gradients**

$$\frac{\partial L}{\partial \theta}$$

- We update parameters using gradient-based methods (SGD, Adam, etc.)

👉 **Differentiation is the backbone of learning**

---

## 2. PyTorch uses Automatic Differentiation (Autograd)

PyTorch does **not** symbolically differentiate equations.

Instead, it uses **automatic differentiation** via a **dynamic computation graph**.

### Key idea:

- Every tensor operation is recorded at runtime
- PyTorch builds a **directed acyclic graph (DAG)**
- Gradients are computed using **reverse-mode differentiation (backpropagation)**

In PyTorch, the **DAG (Directed Acyclic Graph)** is the backbone of the Autograd engine. Every time you perform an operation on a tensor that has `requires_grad=True`, PyTorch builds a graph in the background to track the history of computations.

## 1. How the DAG is Built

When you perform a forward pass, PyTorch creates a graph where:

- **Nodes:** Represent functions (operations like addition, multiplication, or ReLU).
- **Edges:** Represent the flow of data (Tensors).
- **Leaves:** Usually the input tensors or weights you want to optimize.

## 2. The Relationship Between Tensors and Functions

Each tensor involved in a computation has a special attribute called `grad_fn`.

- **Forward Pass:** You move from inputs to the loss.
- **Backward Pass:** You call `.backward()` on the loss. PyTorch then traverses the graph from the output node back to the leaves using the **Chain Rule**.

Here is a simple breakdown of the components:

| Component | Role in the DAG |
|---|---|
| **Leaf Tensors** | Inputs or weights. They don't have a `grad_fn` but receive the final gradients. |
| **Function Nodes** | Represented by `torch.autograd.Function`. These store the recipe for calculating the gradient of an operation. |
| **Output Tensor** | The "Root" of the backward pass (usually the Scalar Loss). |

## 3. A Simple Math Example

If you have y = a *b and z = y + c, the DAG looks like this:

1. **Node 1 (Mul):** Takes a and b, outputs y.
2. **Node 2 (Add):** Takes y and c, outputs z.

When you call `z.backward()`, PyTorch looks at Node 2 first. It knows the derivative of addition is $1$. It then moves to Node 1, where it knows the derivative of multiplication

## 4. Visualizing it in Code

You can actually see these references in PyTorch. If you have a variable `loss`, you can inspect its gradient function:

Python

```
import torch

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

y = a * b
loss = y.exp()

print(loss.grad_fn) # Output: <ExpBackward0 object>
print(loss.grad_fn.next_functions) # Shows the link to the multiplication node
```

---

## 3. `requires_grad=True` (most important flag)

```
x = torch.tensor(2.0, requires_grad=True)
```

> “Track all operations involving this tensor so gradients can be computed later.”

If `requires_grad=False`:

- No graph
- No gradients
- Tensor is treated as a constant

📌 **Only tensors with `requires_grad=True` can receive gradients**

---

## 4. Forward pass: building the computation graph

```
y = x**2
```

Internally PyTorch stores:

- Operation: `power`
- Input tensor: `x`
- Backward rule: derivative of x2x^2×2

Nothing is differentiated yet.

This graph is **dynamic**:

- Built on the fly
- Changes each iteration (very useful for research models)

---

## 5. Backward pass: computing gradients

```
y.backward()
```

What happens internally:

1. PyTorch starts from `y`
2. Applies the **chain rule**
3. Traverses the graph backwards
4. Computes gradients for all relevant tensors

Stored as:

```
x.grad = tensor(4.)
```

---

## 6. Leaf tensors and gradient storage

### Leaf tensor:

- Created directly by the user
- Has `requires_grad=True`
- Stores gradients in `.grad`

Example:

```
x = torch.tensor(2.0, requires_grad=True)# leaf
y = x**2# non-leaf
```

- `x.grad` → stores gradient
- `y.grad` → `None`

📌 **Gradients are stored only for leaf tensors**

This matches neural networks where **parameters are leaves**.

---

## 7. `.grad_fn` and the backward graph

Every non-leaf tensor has:

```
tensor.grad_fn
```

This:

- Points to a **backward operation**
- Knows how to compute gradients

Example:

```
y.grad_fn → PowBackward
```

During `.backward()`:

- PyTorch calls these backward functions
- Propagates gradients upstream

You don't manually handle this — PyTorch does it automatically.

---

## 8. Gradient accumulation (common mistake)

Gradients **accumulate by default**.

```
y.backward()
y.backward()
```

Result:

```
x.grad =8# not 4
```

Because:

grad=grad+new grad

That's why training loops always do:

```
optimizer.zero_grad()
```

📌 Forgetting this causes **wrong learning**

---

## 9. Partial derivatives (multiple variables)

For:

For:

$$f(u, v) = uv + u^2$$

PyTorch computes:

$$\frac{\partial f}{\partial u}, \quad \frac{\partial f}{\partial v}$$

Code:

```
u = torch.tensor(1., requires_grad=True)
v = torch.tensor(2., requires_grad=True)
f = u*v + u**2
```

```
f.backward()
```

Results:

- `u.grad = v + 2u`
- `v.grad = u`

📌 PyTorch automatically applies multivariable chain rule.

---

## 10. Why `backward()` usually needs a scalar

By default:

```
loss.backward()
```

works because **loss is scalar**.

If output is a vector:

```
y = model(x)
```

You must provide:

```
y.backward(torch.ones_like(y))
```

Reason:

- Backprop needs ∂L/∂y
- Scalar loss ⇒ gradient = 1
- Vector output ⇒ must be specified

---

## 11. Connection to neural network training

Training loop:

```
output = model(x)
loss = criterion(output, target)
```

```
loss.backward()
optimizer.step()
```

What PyTorch does:

- Computes gradients for **all parameters**
- Stores them in `param.grad`
- Optimizer updates parameters

This is **exactly the same mechanism** you just learned — just scaled up.

---

## 12. Why this matters

Understanding autograd explains:

- Backpropagation
- Why gradients explode/vanish
- Why `.detach()` works
- Why freezing layers works
- How custom loss functions work

# Simple Dataset

## 1. Why `Dataset` exists (core motivation)

Neural networks **do not train on raw Python lists**.

They expect a **consistent interface** that:

- Knows **how many samples** exist
- Knows **how to fetch one sample**
- Can **apply preprocessing on-the-fly**
- Works with **DataLoader (batching, shuffling, multiprocessing)**

👉 That interface is `torch.utils.data.Dataset`

---

## 2. What a `Dataset` really is

A PyTorch `Dataset` is **just a class** that implements **two methods**:

```
__len__()
__getitem__(index)
```

That's it. Nothing magical.

W**hy these two?**

- `__len__()` → lets PyTorch know dataset size
- `__getitem__()` → tells PyTorch how to fetch *one* sample

This design allows:

- Indexing: `dataset[0]`
- Iteration: `for sample in dataset`
- Batching via `DataLoader`

---

## 3. Structure of a custom Dataset

Internally, a dataset usually stores:

- **Features** → `self.x`
- **Targets** → `self.y`
- **Optional transform** → `self.transform`
- **Dataset length** → `self.length`

```
classToyDataset(Dataset):
def__init__(self, transform=None):
self.x = ...
self.y = ...
self.length = ...
```

```
self.transform = transform
```

This keeps **data storage** and **data processing** cleanly separated.

---

## 4. `__getitem__` = the heart of Dataset

```
def__getitem__(self, index):
    sample = (self.x[index],self.y[index])
ifself.transform:
        sample =self.transform(sample)
return sample
```

Important points:

- Always returns **one sample**
- Usually returns a **tuple** `(input, label)`
- Applies transform **only when accessed**

📌 This means:

> Data is not modified permanently
>
> Transforms are applied **on demand**

---

## 5. Why Dataset behaves like a list

Because:

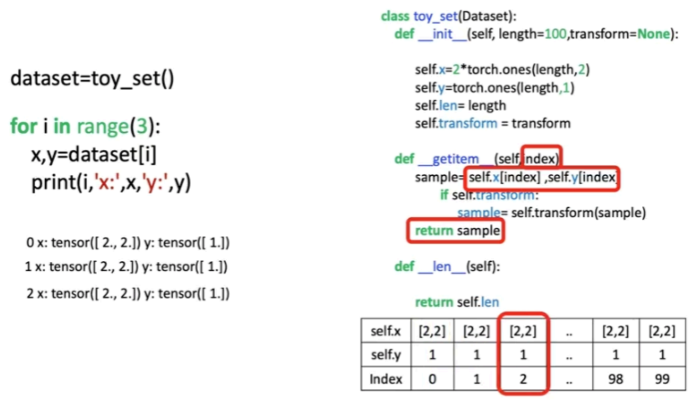
- `dataset[i]` → calls `__getitem__(i)`
- `len(dataset)` → calls `__len__()`

So this works:

```
for x, yin dataset:
    ...
```

Internally:

- Python keeps calling `__getitem__(0), __getitem__(1), ...`




---

## 6. Why transforms are callable classes (not functions)

Transforms are implemented as **callable objects**:

```
classAddMultiply:
def__init__(self, add_x, mul_y):
self.add_x = add_x
self.mul_y = mul_y

def__call__(self, sample):
      x, y = sample
return x +self.add_x, y *self.mul_y
```

Why not simple functions?

Because:

- They can **store parameters**
- They integrate cleanly with `Compose`
- They are reusable and configurable

Callable class = function + memory

---

## 7. Two ways to apply transforms

## ❌ Manual transform (not scalable)

```
sample = dataset[0]
sample = transform(sample)
```

Problems:

- Easy to forget
- Not used by DataLoader
- Breaks pipeline consistency

---

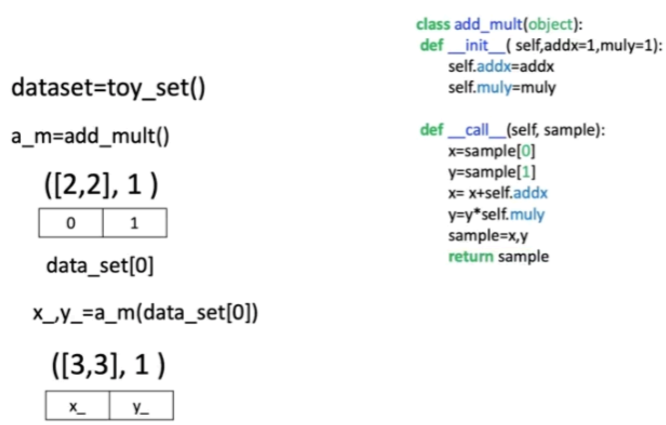
## ✅ Dataset-level transform (correct way)

```
dataset = ToyDataset(transform=transform)
sample = dataset[0]
```

Advantages:

- Always applied
- Automatically used during training
- Works with DataLoader

📌 **This is how PyTorch expects transforms to be used**




---

## 8. Why `Compose` exists

Real pipelines require **multiple transforms**:

Example:

- Normalize
- Scale
- Augment
- Convert dtype

Instead of:

```
x = t3(t2(t1(x)))
```

PyTorch provides:

```
transforms.Compose([t1, t2, t3])
```

---

## 9. What happens when Dataset + Compose work together

Flow for `dataset[i]`:

```
Raw data
  ↓
__getitem__
  ↓
Compose
  ↓
Transform1
  ↓
Transform2
  ↓
Transform3
  ↓
Returned sample
```

Every access is **fresh** and **deterministic**.

---

## 10. Why this design is critical for deep learning

This system enables:

- Large datasets (don't load everything into memory)
- GPU-efficient pipelines
- On-the-fly augmentation
- Clean separation of concerns

| Component | Responsibility |
|---|---|
| Dataset | Fetch raw samples |
| Transform | Modify samples |
| DataLoader | Batch, shuffle, parallelize |
| Model | Learn |

---

# 11. How this connects to real training

Typical training setup:

```
dataset = Dataset(transform=Compose([...]))
loader = DataLoader(dataset, batch_size=32, shuffle=True)

for x, yin loader:
    output = model(x)
    loss = criterion(output, y)
```

Without Dataset + Transforms:

- This loop **breaks**
- Training becomes messy and error-prone

# Dataset

## 1. Why image datasets are different

- Images live on **disk**, not as ready tensors
- Loading all images into RAM is inefficient
- PyTorch uses **lazy loading** → load one image per index

### 2. CSV-based image datasets: why this pattern exists

In real-world datasets:

- Images are stored in folders
- Labels are stored separately (CSV / JSON / XML)

So we store **metadata** (paths + labels) in memory, not images.

Typical CSV structure:

```css
label , filename
5     , img_0001.png
2     , img_0002.png
```

This allows:

- Fast indexing
- Scalable datasets
- Simple shuffling

---

## 2. What the Dataset constructor does

`__init__`:

- Load **CSV / metadata** (filenames + labels)
- Store image directory path
- Store transform object

❌ Does **not** load images

❌ Does **not** convert to tensors

Why?

- Constructor runs **once**
- Loading images here would kill memory and speed

---

## 3. What `__getitem__` does

For an index `i`:

1. Get filename + label from CSV
2. Build image path
3. Open image using **PIL**
4. Apply transform (if any)
5. Return `(image, label)`

This makes datasets scalable and memory-efficient.

Conceptual flow:

```
index → filename →fullpath →open image →transform →return
```

---

## 4. Why PIL → Tensor conversion is delayed

- PIL supports many image formats
- torchvision transforms expect PIL images
- `ToTensor()` converts:
  - Shape → `(C, H, W)`
  - Values → `[0, 1]`
  - Type → `float32`

---

## 5. TorchVision Transforms

Neural networks **do not accept PIL images**.

Transforms handle:

- Cropping / resizing
- Data augmentation
- Tensor conversion
- Normalization

`Compose` chains multiple transforms **in order**.

xample transform pipeline:

```
PILImage
  ↓
Crop /Resize
  ↓
ToTensor
  ↓
Normalized Tensor
```

---

# 6. Channel and batch dimensions

Before `ToTensor()`:

- Image is PIL
- Pixel values are `[0, 255]`

After `ToTensor()`:

- Shape: `(C, H, W)`
- Values: `[0.0, 1.0]`
- Type: `torch.float32`

This matches what neural networks expect.

- Grayscale image → `(1, H, W)`
- RGB image → `(3, H, W)`
- After batching:

```
(batch_size, channels, height, width)
```

## 7. Built-in TorchVision datasets

Datasets like **MNIST / Fashion-MNIST**:

- Already implement Dataset logic
- Support train/test split
- Auto-download
- Accept transforms directly

Key parameters:

- `root` → dataset location
- `train=True/False`

This ensures:

- Proper evaluation
- No accidental mixing
- `download=True`
- `transform=...`
  Applied **every time** a sample is fetched

## 8. End-to-end flow

```
Disk images
  ↓
Dataset (__getitem__)
  ↓
PIL Image
  ↓
Transforms (Compose)
  ↓
Tensor (C,H,W)
```

```
  ↓
DataLoader (batching)
  ↓
Model
```