# Module 1

**Tensors and Datasets**

**Simple Dataset**

## 📌 Differentiation in PyTorch

This section focuses on building a custom dataset class and applying data transformations using PyTorch. It explains how to create iterable dataset objects, apply transformations using callable classes, and chain multiple transformations using composition.

### 🔹 Creating a Dataset Class

A dataset object is created by subclassing the abstract Dataset class provided by PyTorch.

Within the constructor:

* Input features and target values are stored as tensors (x and y), each containing 100 samples. The values are created in the object constructor and assigned to the **self.x** and **self.y** tensors
* The total number of samples is stored in a length attribute.

The dataset class overrides two core methods:

* **\_\_len\_\_**: Returns the number of samples.
* **\_\_getitem\_**\_: Accepts an index and returns a tuple of feature and target tensors corresponding to that index.

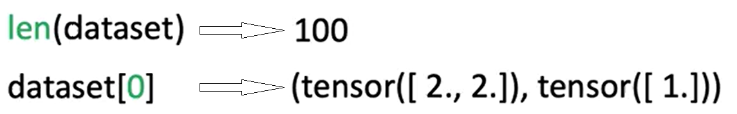
|  |  |
| --- | --- |
|  |  |

🔸 **Accessing Data Samples:**

Individual samples are retrieved using square brackets, which act as a proxy for the \_\_getitem\_\_ method.

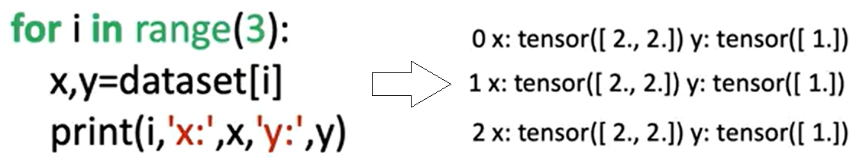
This method returns a tuple:

* The first element corresponds to a feature tensor.
* The second element corresponds to a target tensor.



The dataset behaves like an iterable. It can be accessed using index notation or through iteration in a loop:

* Iterating over the dataset triggers repeated calls to \_\_getitem\_\_, returning one sample per iteration.

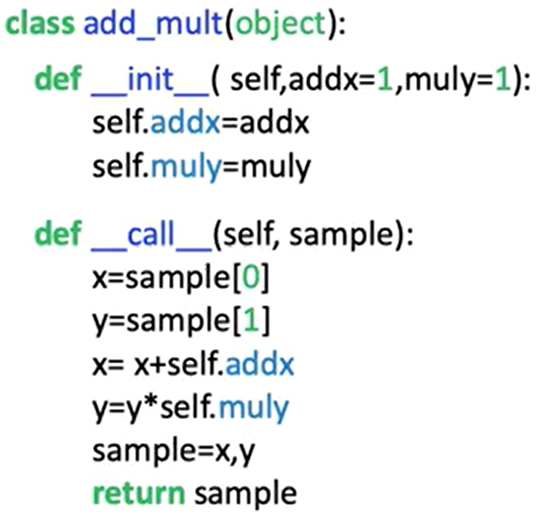


### 🔹 Applying Transforms to a Dataset

Transformations can be applied to samples using **callable classes** instead of standalone functions.

These classes define a **\_\_call\_\_** method, allowing them to behave like functions when passed to the dataset.

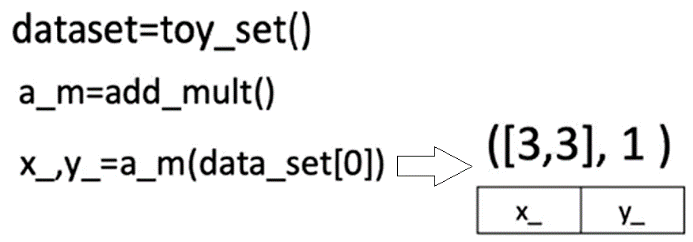
**Custom Transform Class**

* A custom transformation class is defined with two parameters:
  + One to add a constant to the feature tensor.
  + One to multiply the target tensor by a constant.
* When a sample is passed to this transformation object, the transformation is applied and the modified tensors are returned as a tuple.

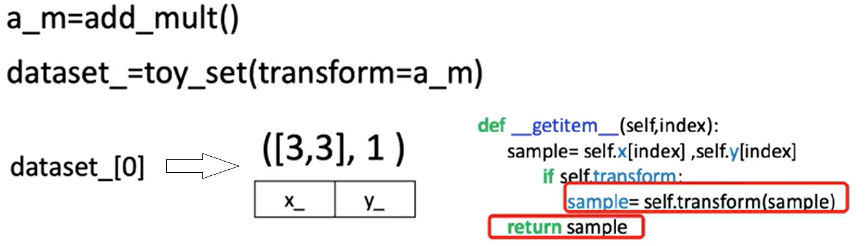
🔸 **Applying Transforms to Dataset Samples:**

There are two methods for applying a transformation:

1. **Manual Application**:
   * The transformation object is created separately.
   * The object is manually applied to a sample retrieved from the dataset.
   * Only the selected sample is transformed.



1. **Automatic Application via Constructor:**
   * The transformation object is passed to the dataset class during initialization.
   * Inside the dataset class, the transform parameter is assigned.
   * During each call to **\_\_getitem\_\_,** the transformation is applied automatically to every sample.
   * This ensures that the transformation is consistently applied across all retrieved data.



### 🔹 Composing Multiple Transforms

PyTorch provides a Compose class for chaining multiple transformations.

A list of transformation objects is passed to the Compose constructor.

When a sample is passed to the composed transform:

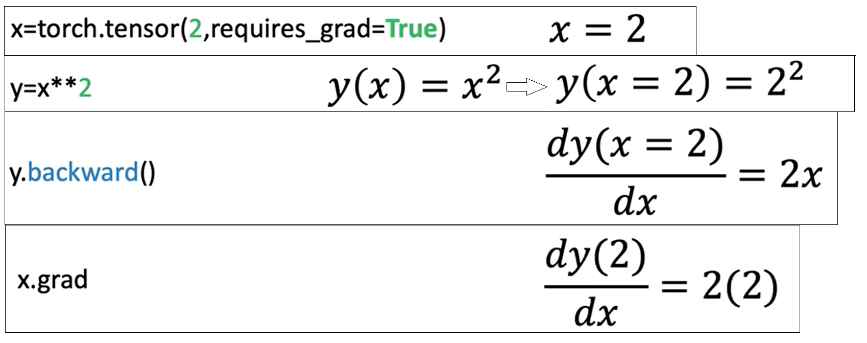
* The first transformation is applied.
* The output is passed to the second transformation.
* The final output is returned as a transformed tuple of tensors.

This compose object can be passed into the dataset class, enabling **automatic application of multiple transformations** during sample retrieval.

Evaluating this derivative at a specific point (e.g., x = 2) gives the **slope of the function** at that point (2 × 2 = 4).

To compute derivatives in PyTorch:

* When creating x (a tensor) a value is specified, functions and derivatives of x are evaluated for the assigned value, in this case 2.
* When a tensor is created with **requires\_grad=True**, PyTorch tracks all operations involving it to allow gradient computation later. It essentially tells PyTorch that the declared value will be used to evaluate functions and derivatives of x using the declared value.
* To differentiate a function defined with a tensor, **.backward()** function is called to trigger backpropagation.
* The result of this differentiation is stored in the **grad** attribute of the original input tensor, reflecting the value of the derivative at that specific input.



### 🔹 The Backward Graph and Tensor Attributes

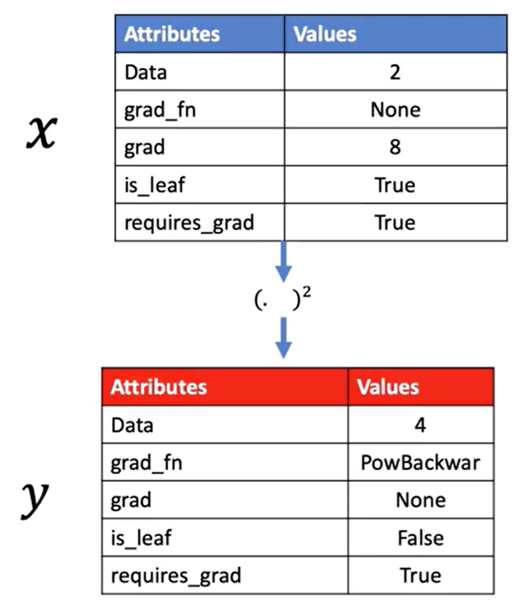
PyTorch supports automatic differentiation by attaching metadata to tensors. **The backward graph is essentially composed of metadata from multiple tensors and ops**, arranged in a way that allows gradient computation.

Basically, the **backward graph** relies on tensor metadata to **compute gradients** correctly.

PyTorch constructs a **backward graph**, where tensors and operations (backward function for example) are nodes. This structure allows tracing back through computations to evaluate derivatives.

Based upon whether a particular tensor is a leaf or not in the graph, pytorch evaluates the derivative of that tensor.

If the leaf attribute for a tensor is set to True, pytorch won’t evaluate its derivative.



Each tensor has important attributes:

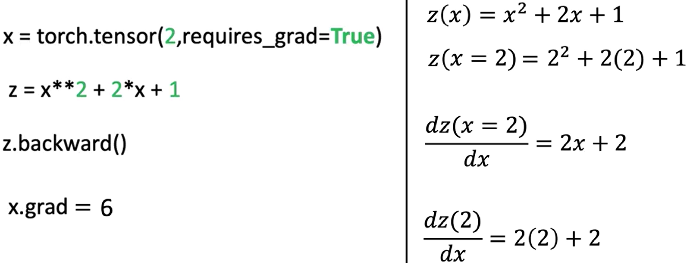
* **Data**: Holds the actual numerical value.
* **Grad**: Stores the computed derivative once calculated.
* **Grad\_fn**: Points to the function used to generate the tensor.
* **Is\_leaf**: Indicates whether the tensor is a leaf node in the graph.
* **Requires\_grad**: Signals that gradients should be tracked for this tensor

ℹ️PyTorch evaluates gradients using this graph, determining how changes in input tensors affect output tensors.

🔸 **Single Variable Differentiation:**

PyTorch allows gradient to be computed automatically:

* Define the tensor x with gradient tracking enabled.
* Define z in terms of x.
* Trigger backpropagation.
* Access the result through the .grad attribute.



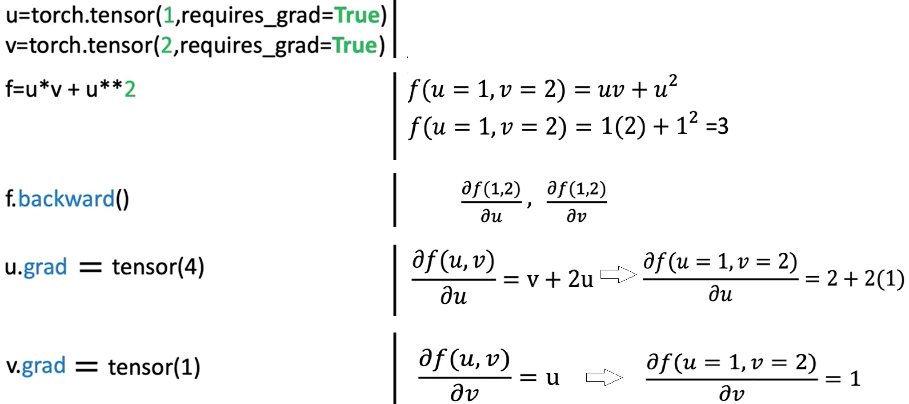
🔸 **Partial Derivatives for Multivariable Functions:**

Partial derivatives measure the change of a function with respect to one input variable, holding others constant.

* Consider a function The partial derivatives are:

|  |  |
| --- | --- |
| respect to u | respect to v |
|  |  |

* PyTorch can compute both partial derivatives by defining both input tensors with requires\_grad=True, constructing the function f, calling the differentiation trigger, and then accessing the gradients for each input separately.



### ✅ Takeaways

✅ PyTorch automates differentiation by building a **computational graph** that tracks how tensors are connected through operations.

✅ Tensors with gradient tracking enabled can be used to compute derivatives using **backward propagation**.

✅ **Single-variable derivatives** and **partial derivatives** are both supported.

✅ Gradients are accessed directly from the input tensors once calculated.

✅ Tensor attributes such as **grad**, **grad\_fn**, and, **is\_leaf** are essential for managing and understanding gradient flows.

✅ Automatic differentiation is critical for training neural networks using optimization techniques like gradient descent.