# Module 1

**Tensors and Datasets**

**Differentiation in PyTorch**

## 📌 Differentiation in PyTorch

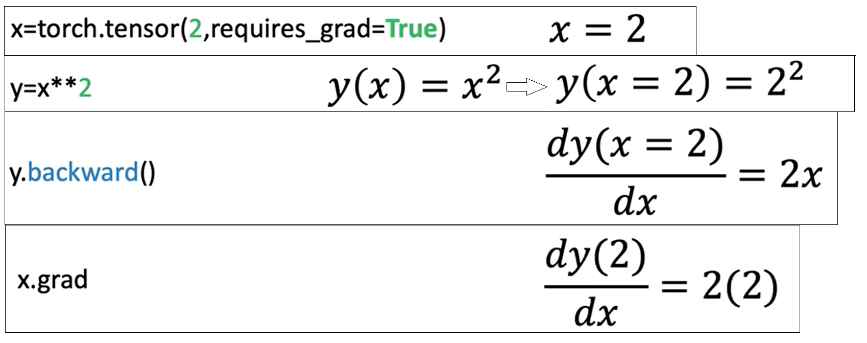
### 🔹 Basic Derivatives

A derivative represents the **rate of change of a function**.

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

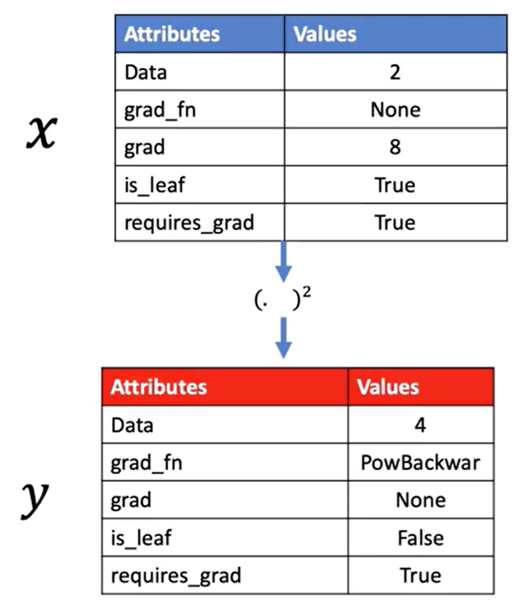
PyTorch supports automatic differentiation by attaching metadata to tensors. **The backward graph is essentially composed of metadata from multiple tensors and ops**, arranged in a way that allows gradient computation.

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:

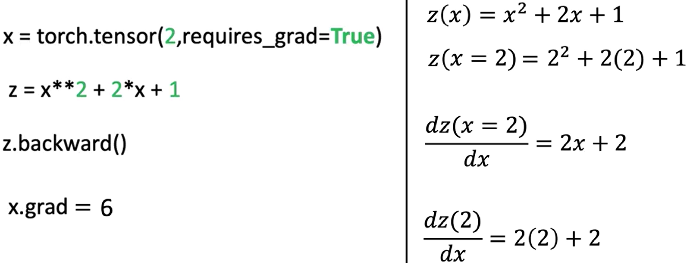
* **Data**: Holds the actual numerical value.
* **Grad**: Stores the computed derivative once calculated.
* **Grad\_fn**: Points to the function used to generate the tensor.
* **Is\_leaf**: Indicates whether the tensor is a leaf node in the graph.
* **Requires\_grad**: Signals that gradients should be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



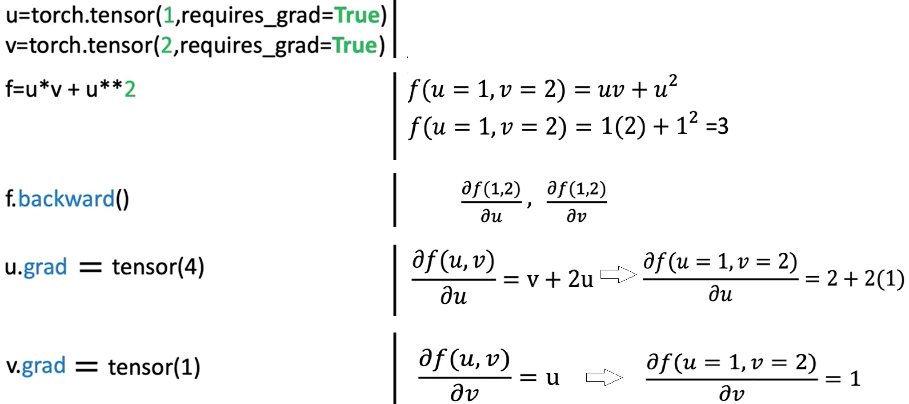
🔸 **Partial Derivatives for Multivariable Functions:**

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.