# Module 1

**Tensors and Datasets**

**M1 – Section 1**

**Tensors 1D & 2D**

## 📌 Overview of Tensors

Tensors are the foundational data structures used to construct and operate neural networks in PyTorch.

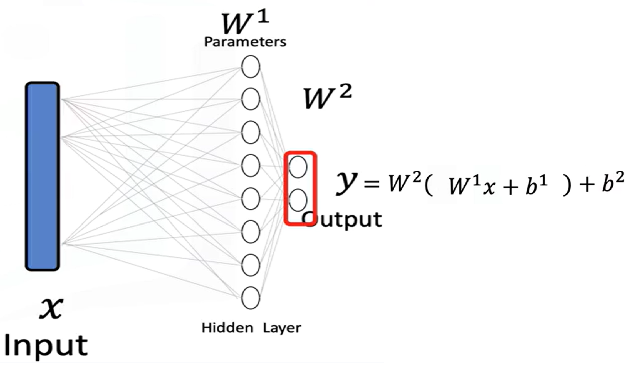
A neural network is fundamentally a mathematical function that accepts one or multiple inputs, processes them, and returns one or more outputs.

In PyTorch, this processing is performed using **tensor operations**, which are generalized versions of familiar mathematical operations like addition and multiplication.

### 🔹 Tensors as Building Blocks

PyTorch tensors serve as the unified representation for **inputs**, **outputs**, and **parameters** within a neural network. These tensors can represent vectors, matrices, or higher-dimensional data structures, depending on the application.

Tensor operations in PyTorch form the computational backbone of how data is manipulated and learned from in neural networks.

PyTorch tensors are a generalized form of **numbers and dimensional arrays** in Python.

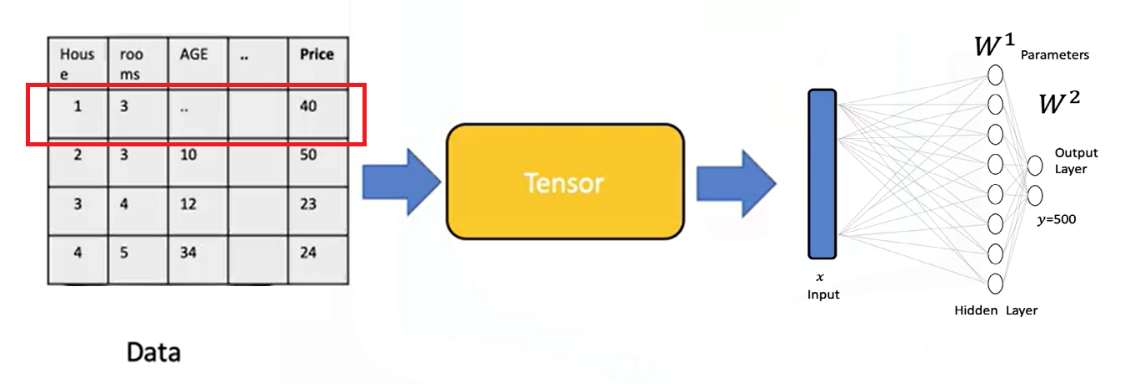
* The **input x** to a neural network is a tensor.
* The **output y** is also a tensor.
* The **parameters** of the model are tensors as well.
* Tensor operations allow the neural network to **transform inputs** into outputs during training and inference.

Neural networks use these tensor operations to apply **mathematical transformations**, often in the form of **vector and matrix operations**. These operations simulate the way real-world data is processed and are used throughout the course as the standard method for feeding data into neural models.

### 🔹 Examples of Tensors in Neural Networks

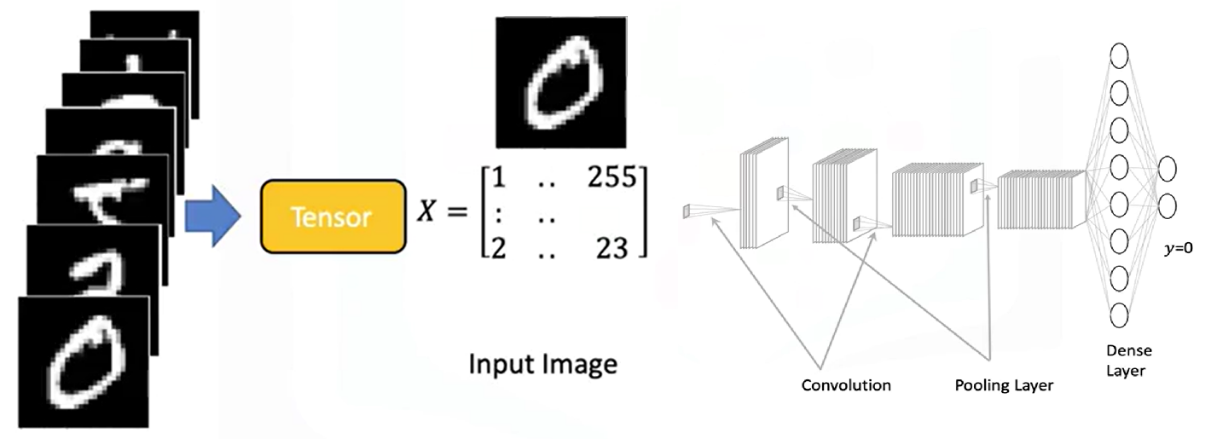
**Databases** can be treated as a series of tensors, where each row represents an input tensor (x) in a neural network.

A tensor is simply just a vector or a rectangular array consisting of numbers.



**Images** can be converted into 2D or 3D PyTorch tensors and used as input for classification tasks.

* + Each tensor of the input is simply a matrix or rectangular array.
  + Images are typically stored as arrays.
  + Neural networks can receive these as tensors and perform classification based on the processed values.
  + For instance, an image can be transformed into a tensor and classified as the digit **zero**.



### 🔹 Tensor Conversion and Compatibility

PyTorch tensors can be easily **converted to NumPy arrays**, and NumPy arrays can also be converted into PyTorch tensors.

This bidirectional conversion enables seamless operation within the **Python ecosystem** and allows integration with many existing Python libraries.

PyTorch also supports **GPU acceleration**, which is crucial for training large neural networks efficiently.

### 🔹 Parameters and Derivatives in PyTorch

**Parameters** in neural networks are specialized tensors that allow for the calculation of **gradients and derivatives**.

These gradients are essential for learning during training.

To enable gradient tracking, PyTorch tensors must be created with **requires\_grad=True**.

This setting allows PyTorch to automatically compute derivatives during backpropagation.

### 🔹 Dataset Class in PyTorch

PyTorch provides a **Dataset class** that simplifies working with large datasets.

Using this class enables efficient data handling, transformation, and loading.

It is especially useful when building neural networks that require batch processing or data augmentation.

### ✅ Takeaways

✅ PyTorch tensors are the core data structures used in building and training neural networks.

✅ Inputs, outputs, and model parameters are all represented as tensors.

✅ Tensor operations in PyTorch generalize familiar mathematical operations and are essential for transforming input data.

✅ Databases and images can be represented as tensors and processed within neural networks.

✅ PyTorch integrates seamlessly with NumPy and supports GPU acceleration for scalable training.

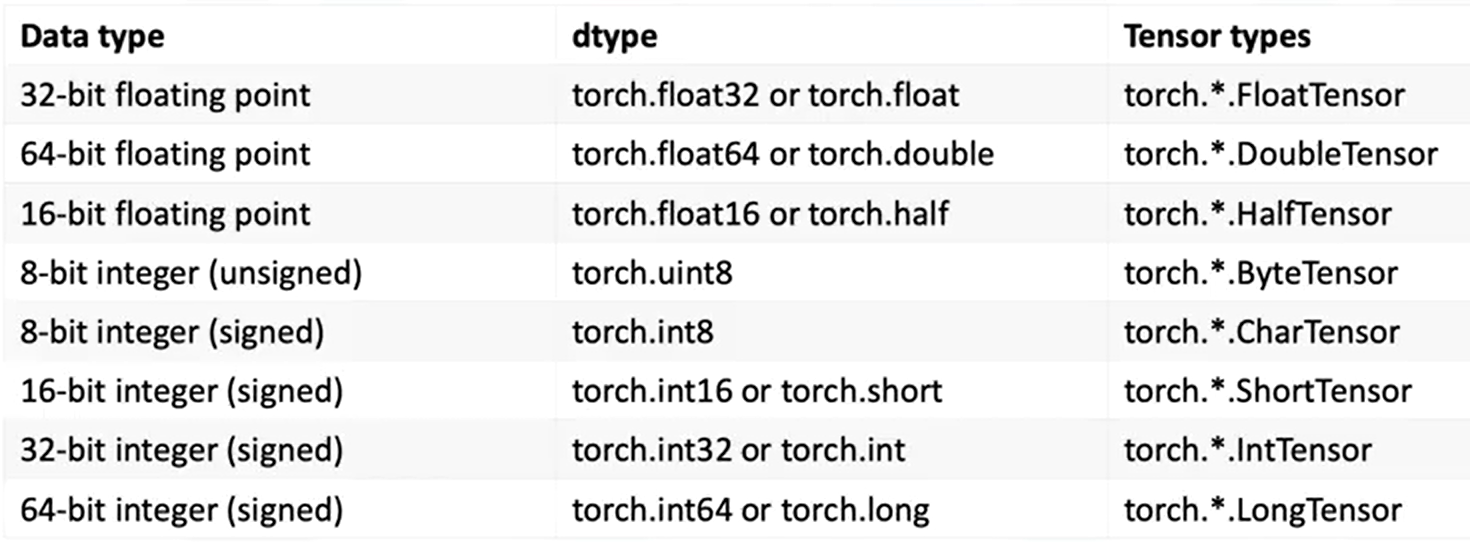
✅ Setting requires\_grad=True enables tensors to compute gradients, allowing for neural network training.

✅ The Dataset class simplifies data management and is essential for working with large-scale training data.

## 📌 Tensors 1D

### 🔹 Understanding 1D Tensors

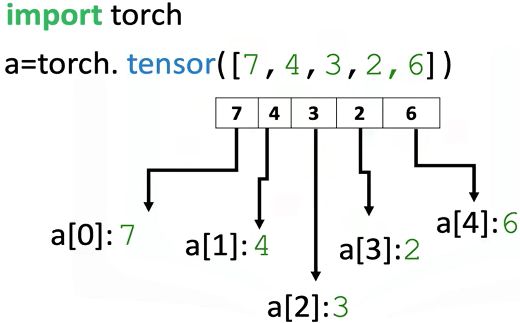
* A **0D tensor** represents a single number.
* A **1D tensor** is an array of numbers and can represent:
* A row in a dataset
* A vector
* A time series
* A tensor contains elements of a **single data type**, there is a variety of different tensor types depending the data type of the elements in the tensor, such as:
* float or double tensors (for real numbers)
* byte tensors (for 8-bit images and unsigned integers)



### 🔹 1D Tensor Operations

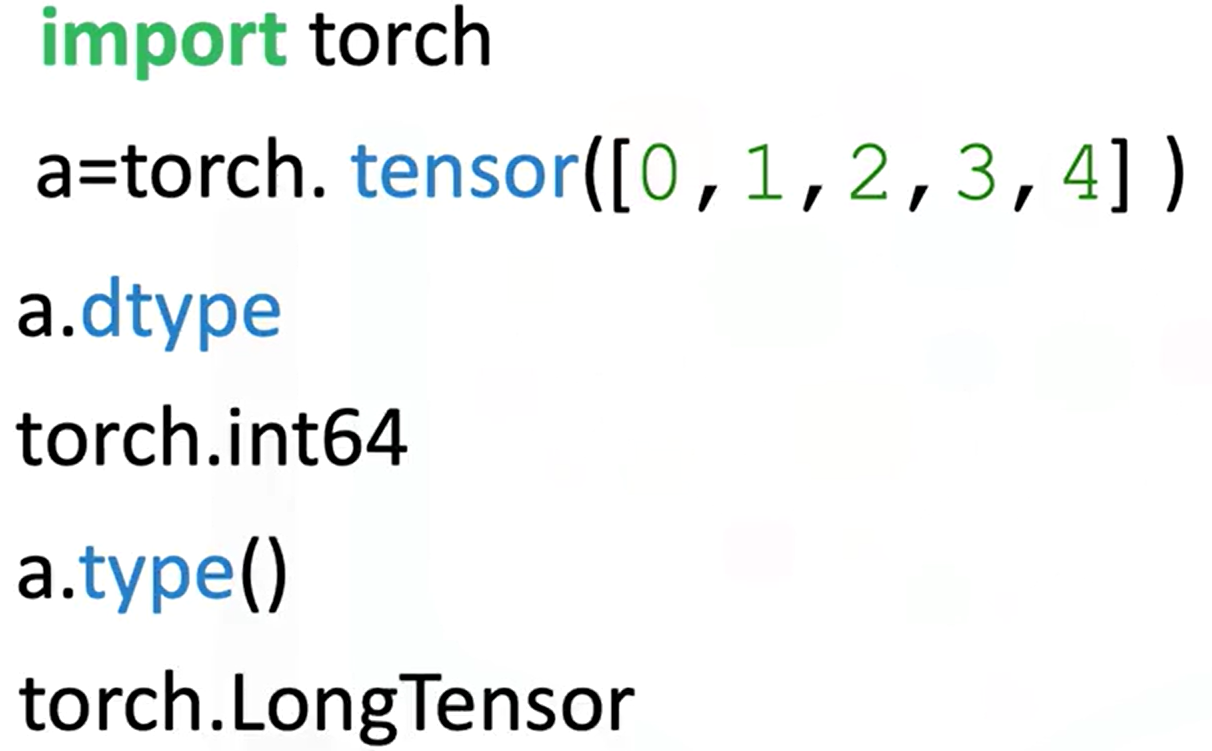
🔸 **Creating a tensor:**

* + Use **torch.tensor()** to convert the list into a tensor.
  + Data can be accessed via index.

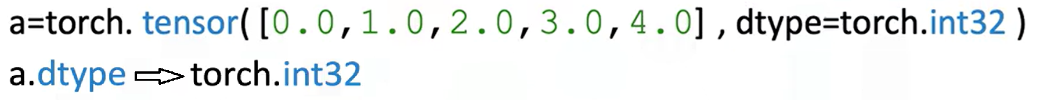


🔸 **Tensor Type and Data Type:**

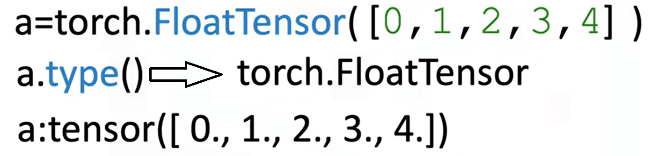
* + Use the **.dtype** attribute to identify the data type stored in a tensor.
  + Use **.type()** to identify the tensor type.



* + Explicitly set the data type using the **dtype** parameter.



* + Using classes like **torch.FloatTensor**.

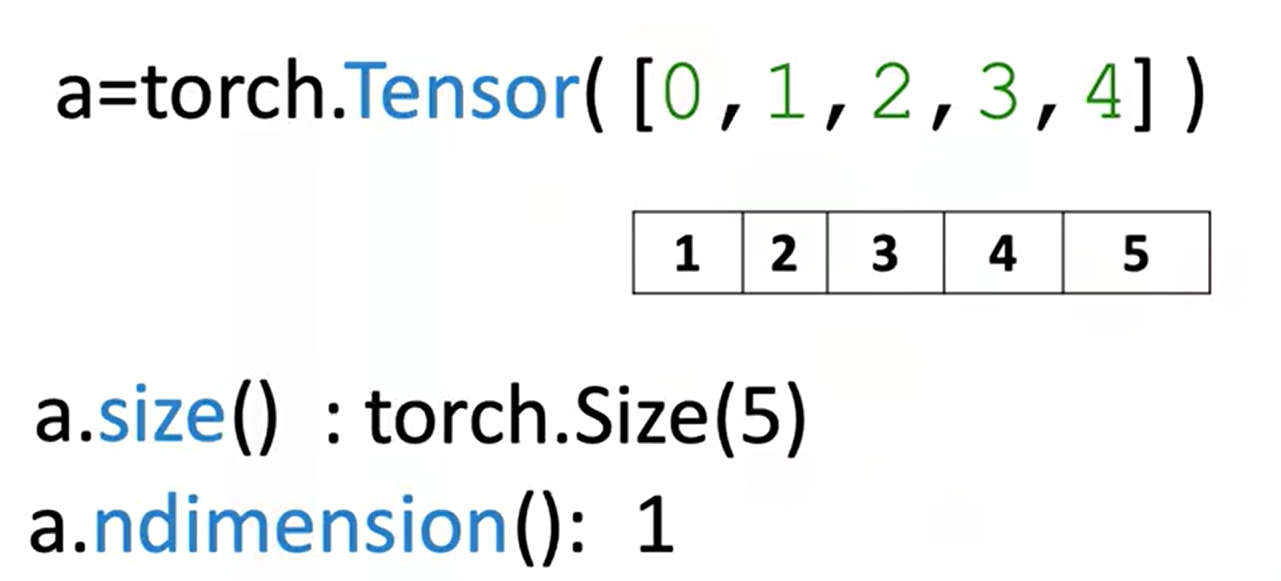


* + Use **.type(torch.FloatTensor)** to convert to a float tensor.



🔸 **Tensor Size and Shape:**

* + Use **.size()** to find the number of elements.
  + Use **.ndimension()** to find the number of dimensions (tensor rank).

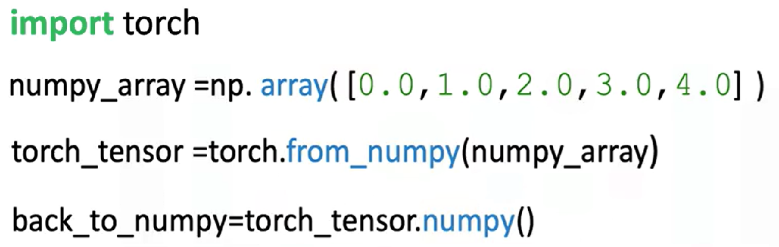


* + Convert a 1D tensor to 2D using **.view(number\_rows, number\_cols)**:
* **view(5, 1)** turns a 1D tensor with 5 elements into a 2D column tensor.
* Use **view(-1,1)** to let PyTorch infer dimensions.



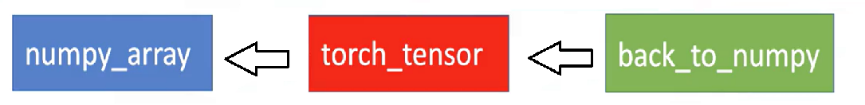
🔸 **Tensor Conversion with NumPy and Pandas**

* + Convert a NumPy array to a tensor with **torch.from\_numpy()**
  + Convert a tensor to a NumPy array using **.numpy()**



**⚠️ Memory sharing:**

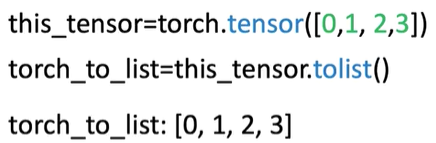
Modifying the original NumPy array affects the PyTorch tensor and vice versa.



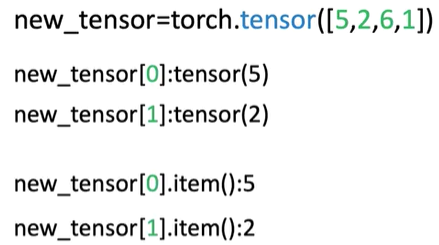
* + Convert Pandas series to tensor:
* Use **.values** to get the NumPy array
* Then apply **torch.from\_numpy()**

****

* + Convert tensor to list:
* Use **.tolist()** to get a Python list

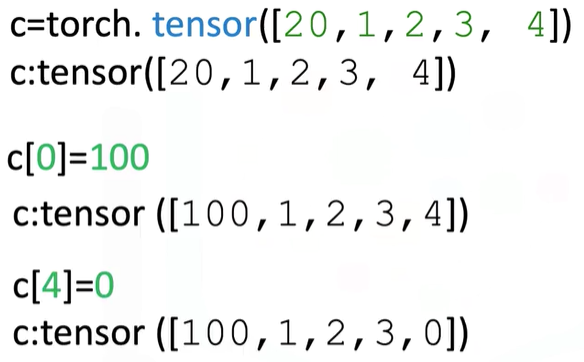


* + Convert tensor element to number:
* Use **.item()** to extract a Python number from a single-element tensor

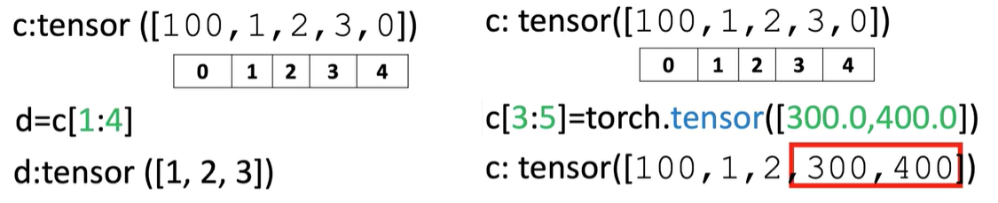


🔸 **Indexing and Slicing:**

* + Access tensor elements with index (e.g., a[0])
  + Assign new values to specific elements (e.g., a[0] = 100)



* + Slice a tensor like a list: a[1:3]
  + Assign values to slices (e.g., a[1:3] = [1, 2])

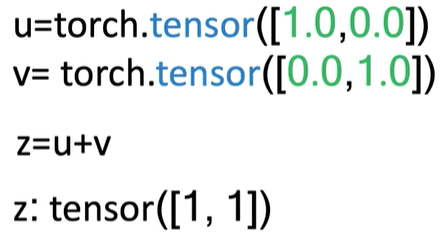


🔸 **Basic Tensor Operations:**

These operations are essential for building neural networks and understanding how tensors interact mathematically:

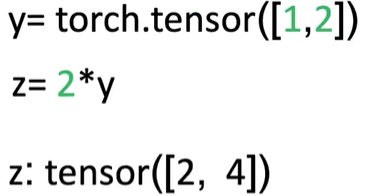
* + Vector Addition:

Combine two tensors element-wise.



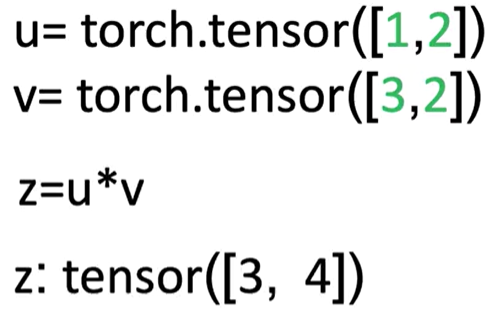
* + Scalar Multiplication:

Multiply each element of a tensor by a scalar.



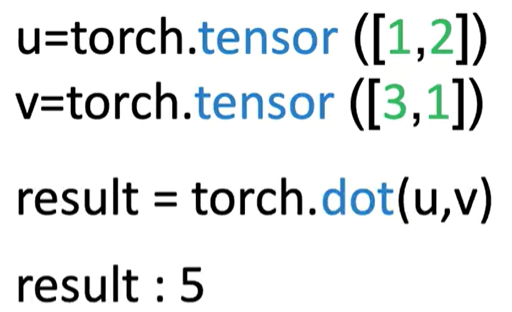
* + Hadamard Product (Element-wise Multiplication)

Multiply corresponding elements of two tensors.



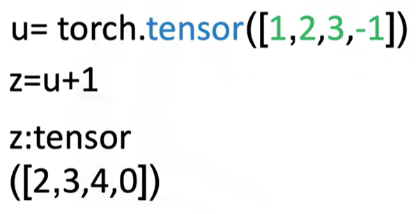
* + Dot Product

Produces a single number that measures similarity between two vectors.



* + Broadcasting

Adding a scalar to a tensor adds it to each element.

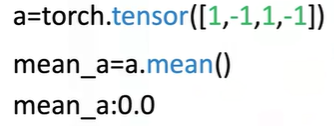


🔸 **Universal Functions:**

Apply operations across all elements:

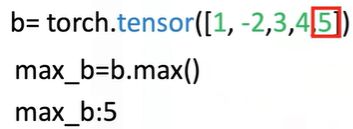
* + **a.mean()**:

Computes the average

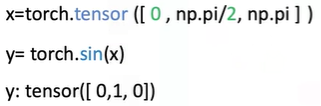


* + **b.max()**:

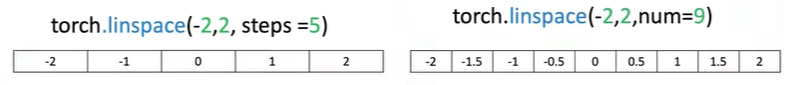
Returns the maximum value



* + Use functions like **torch.sin()** to apply to every element of a tensor

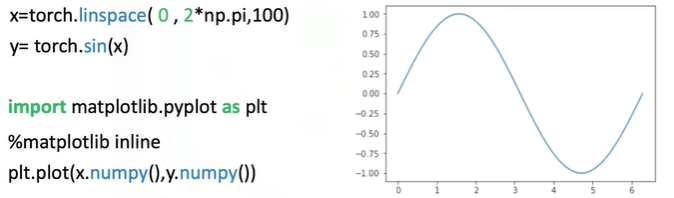


* + Use **torch.linspace(start, end, steps)** to generate evenly spaced values.



🔸 **Plotting with Tensors:**

* + Use **matplotlib.pyplot**
  + Use **%matplotlib inline** for inline notebook rendering
  + Convert tensors to NumPy before plotting: **.numpy()**



### ✅ Takeaways

✅ **1D tensors** are core structures for data representation in PyTorch.

✅ Tensors can be **easily created, indexed, sliced, and reshaped** using intuitive syntax.

✅ PyTorch supports **type casting**, **NumPy/Pandas conversion**, and **interoperability** with Python tools.

✅ Tensor operations include **vector arithmetic**, **dot product**, **broadcasting**, and **universal functions** like mean and max.

✅ PyTorch allows mathematical functions (like sine) to be applied element-wise, enabling visualization and numerical analysis.

✅ Tools like linespace and matplotlib can be combined with PyTorch tensors for **function plotting and visualization**.

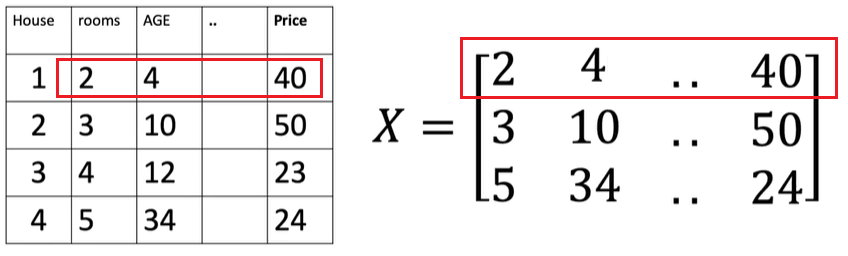
## 📌 Two-Dimensional Tensors

### 🔹 Understanding 2D Tensors

A 2D tensor is a container that holds numerical values of the same type and is typically visualized as a matrix.

In real-world applications, 2D tensors can represent:

* **Tabular data**, where each row is a sample (e.g., a house), and each column is a feature (e.g., number of rooms, age, price).



* **Grayscale images**, where pixel intensities range from 0 (black) to 255 (white), forming a 2D grid.



Tensors can be extended beyond two dimensions:

* **3D tensors** are used to represent color images, where each channel (red, green, blue) has its own 2D matrix of intensity values.



* Higher-dimensional tensors (e.g., 4D) are also used in deep learning.

### 🔹 Creating 2D Tensors

A 2D tensor can be created from a nested list, where each inner list represents a row.

The structure is interpreted as a rectangular matrix:

* The outer dimension represents rows.
* The inner dimension represents columns (eg: each element on a list, in this case since we have three elements in each list we have three columns).

Important tensor attributes:

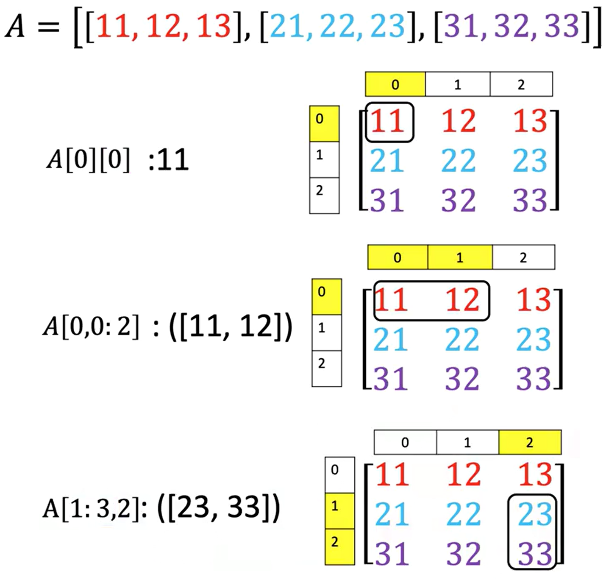
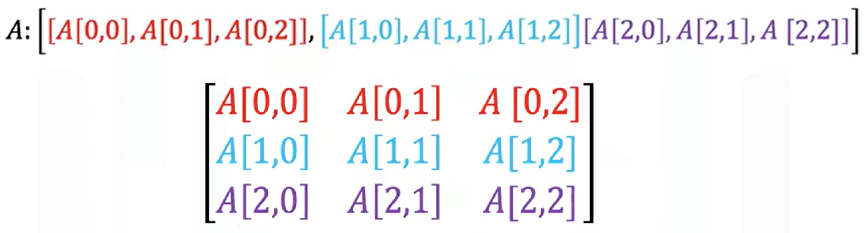
* **Number of dimensions (rank)** can be queried to confirm the tensor structure.
* **Shape** returns the count of rows and columns.
* **Size** can be used interchangeably to obtain shape.
* **Number of elements** can be calculated by multiplying rows and columns or using a built-in method.

### 🔹 Indexing and Slicing

Indexing allows extraction of individual values, partial rows, or partial columns from the tensor for further computation or inspection.

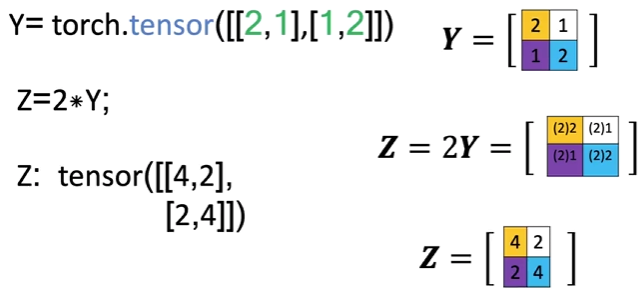
It’s performed using two indices:

* The first index corresponds to the **row**.
* The second index corresponds to the **column**.



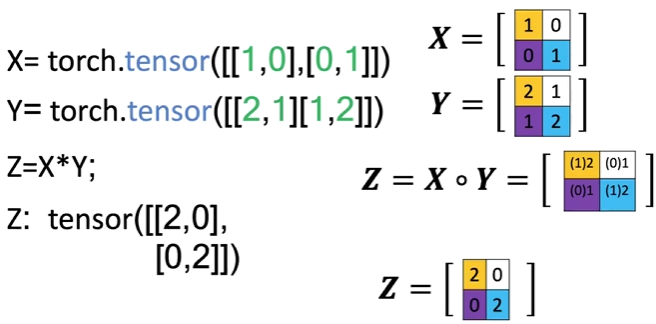
### 🔹 Basic Operations on 2D Tensors

🔸 **Addition:**

* + Two tensors of the same shape can be added together.
  + This performs element-wise addition, similar to matrix addition in linear algebra.
  + Each element in the result is the sum of the corresponding elements in the input tensors.

**🔸 Scalar Multiplication:**

* + Multiplying a 2D tensor by a scalar scales each individual element.
  + The resulting tensor is the same shape, but each value is multiplied by the scalar.

🔸 **Hadamard Product (Element-wise Multiplication):**

* + Multiplies corresponding elements of two tensors of the same shape.
  + Produces a new tensor where each value is the product of the matching elements from the inputs.

🔸 **Matrix Multiplication:**

* + Follows standard linear algebra rules:

The number of columns in matrix A must match the number of rows in matrix B.

* + For each element in the resulting matrix:

Compute the dot product between a row from matrix A and a column from matrix B.

* + The result is a new matrix with a shape defined by the row count of matrix A and the column count of matrix B.
  + Matrix multiplication yields a meaningful transformation of input features, often used in neural network layers.

### ✅ Takeaways

✅ 2D tensors are commonly used to represent both **structured data** (like spreadsheets or tables) and **images** (grayscale and multi-channel).

✅ Tensors can be **indexed, sliced, and reshaped** to access and manipulate specific data points or submatrices.

✅ Arithmetic operations like **addition**, **scaling**, **element-wise multiplication**, and **matrix multiplication** are supported natively and follow familiar linear algebra principles.

✅ 2D tensors provide the foundation for **layered neural network computations**, especially in the early stages of data processing and feature transformation.

✅ The structure and operations on tensors mirror real-world mathematical concepts, making them an intuitive and powerful abstraction for machine learning.

# M. 1 – Section 2

**Differentiation in PyTorch**

## 📌 Differentiation in PyTorch

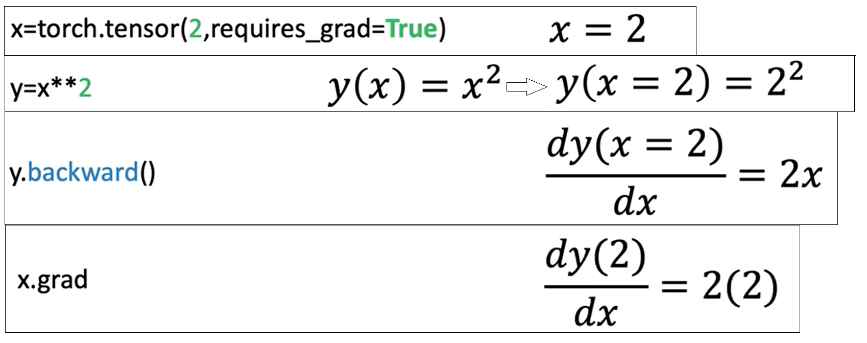
### 🔹 Basic Derivatives

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:

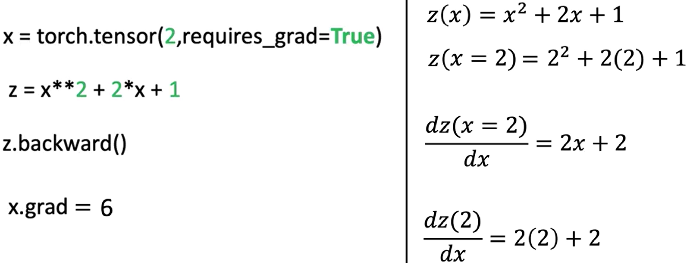
* **Data**: Holds the actual numerical value.
* **Grad**: Stores the computed derivative once calculated.
* **Grad\_fn**: Points to the function used to generate the tensor.
* **Is\_leaf**: Indicates whether the tensor is a leaf node in the graph.
* **Requires\_grad**: Signals that gradients should be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



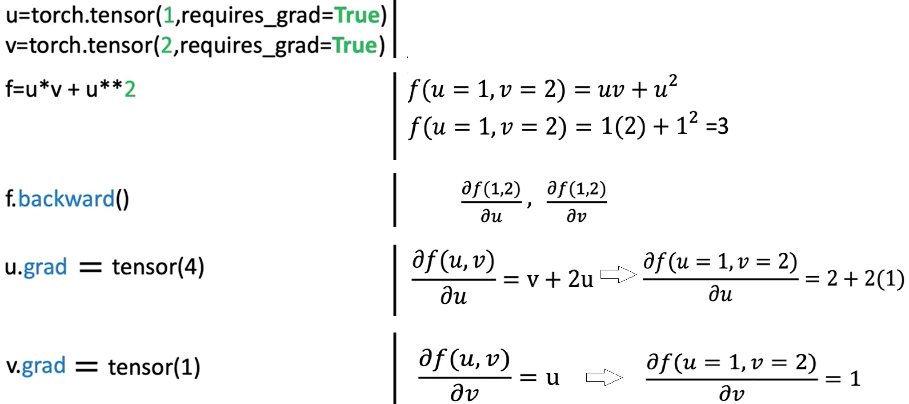
🔸 **Partial Derivatives for Multivariable Functions:**

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.

# M. 1 – Section 3

**Dataset**

## 📌 Simple Dataset

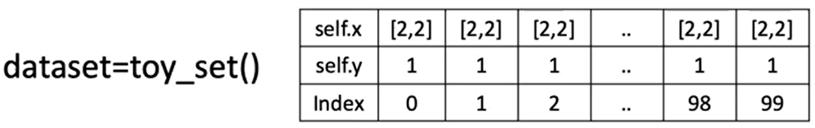
### 🔹 Creating a Dataset Class

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:

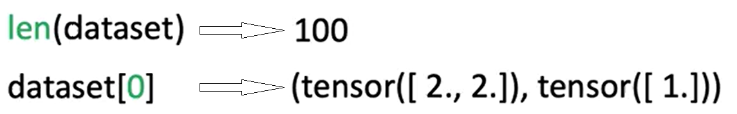
* **\_\_len\_\_**: Returns the number of samples.
* **\_\_getitem\_**\_: Accepts an index and returns a tuple of feature and target tensors corresponding to that index.

🔸 **Accessing Data Samples:**

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

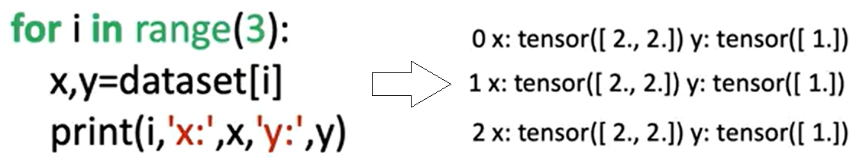
This method returns a tuple:

* 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.

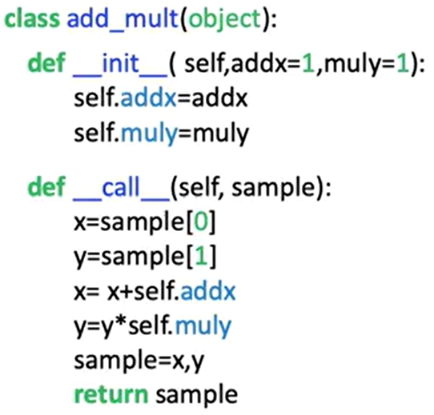


### 🔹 Applying Transforms to a Dataset

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.

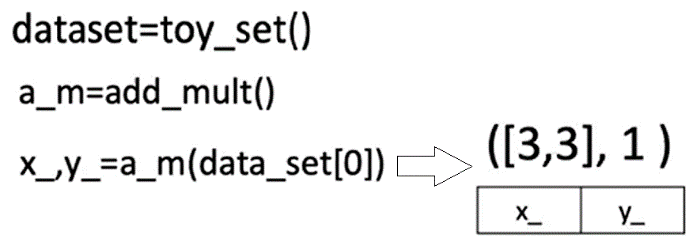
**Custom Transform Class**

* 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.

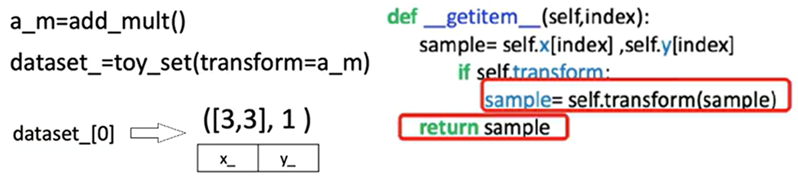
🔸 **Applying Transforms to Dataset Samples:**

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

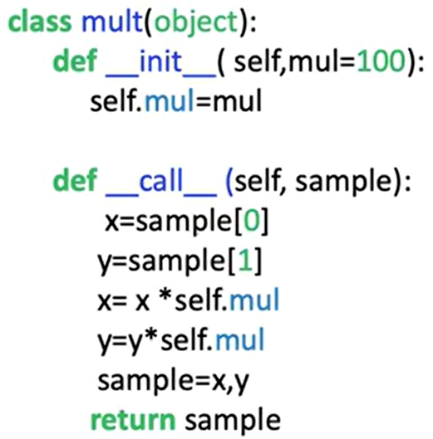
PyTorch provides a Compose class for chaining multiple transformations.

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.

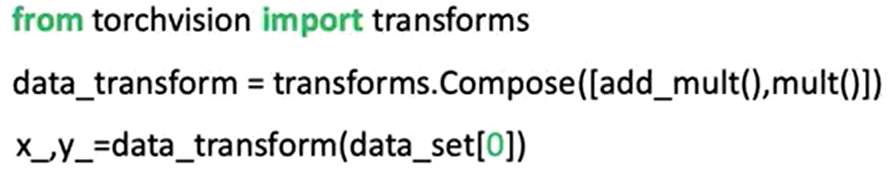
This compose object can be passed into the dataset class, enabling **automatic application of multiple transformations** during sample retrieval.

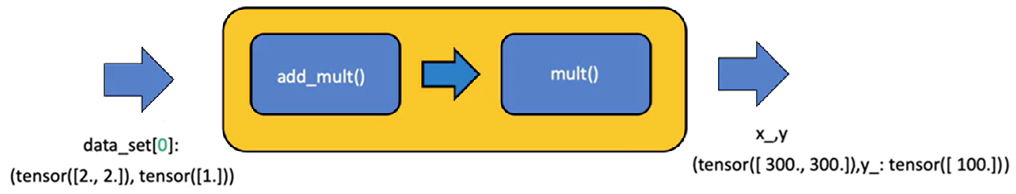


Let's say we would like to apply another transform, the class “**mult**” will multiply all the elements of a tensor by the value mul.

1. **Manual Application**:

In the constructor, we place a list. The first element of the list is the constructor for the first transform, the second element of the list is the constructor for the second transform.





1. **Automatic Application via Constructor:**

The compose object can be applied directly in the dataset constructor, each time a sample is retrieved, the original tensor is passed to the compose object (the first transform is applied, then the second transform is applied).



### ✅ Takeaways

✅Custom dataset objects can be built by subclassing PyTorch’s Dataset class and implementing the length and indexing methods.

✅Data stored in tensors can be accessed, indexed, and iterated over in a structured and repeatable way.

✅Transformations can be implemented as callable classes for better modularity and reuse.

✅Applying transformations during dataset construction enables efficient preprocessing at the data loading stage.

✅Multiple transformations can be composed using PyTorch's Compose utility, allowing sequential data processing in a clean and scalable manner.

## 📌 Image Dataset

This section explains how to build a custom dataset class for image data using PyTorch, how to preprocess image inputs using TorchVision transforms, and how to work with TorchVision's built-in datasets.

### 🔹 Constructing an Image Dataset

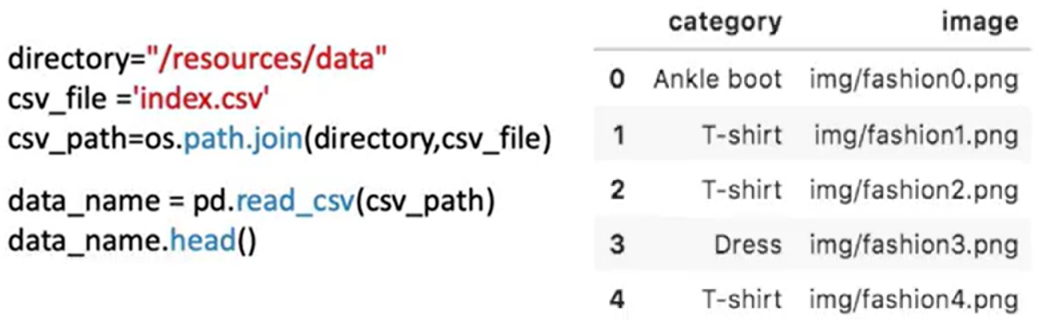
To construct an image dataset, the process begins by importing libraries from PyTorch, Pandas, and TorchVision.



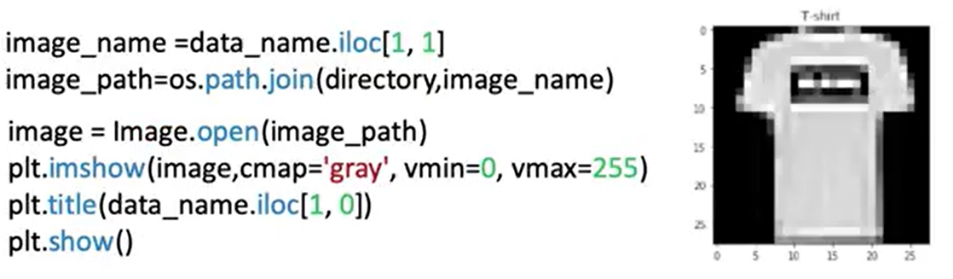
The dataset is built using Zalando’s **Fashion-MNIST** training set, which contains: 60,000 grayscale images, with 28 × 28 pixels resolutions, and 10 distinct classes representing types of clothing.

The dataset is provided in the form of folder of image files (a CSV file mapping each image file to a class label):

* The first column contains the clothing label (class).
* The second column image file name (the image file path is constructed by combining the base directory with the image file name).



Images in the dataset can be loaded using **Image.open(path)** and stored in a variable.

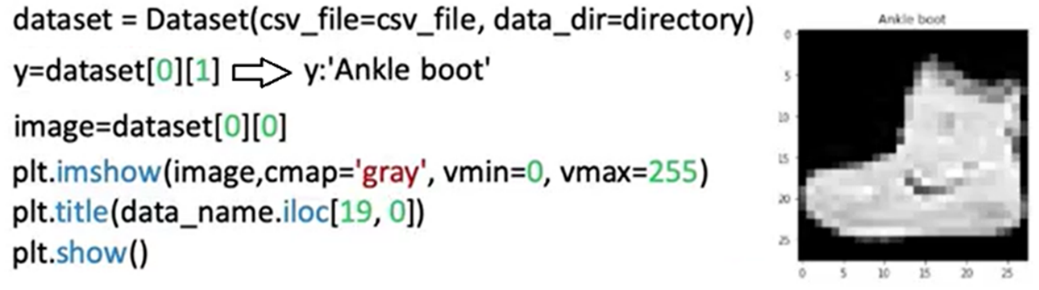


### 🔹 Building the Custom Dataset Class

The image dataset class follows the same structure as a PyTorch Dataset subclass:

* In the **constructor**:
  + The CSV file is loaded.
  + The image names and labels are stored as a DataFrame attribute (self.data\_names).
* The **\_\_getitem\_\_** method is responsible for:
  + Receiving an index.
  + Retrieving the image name and label from the DataFrame.
  + Building the full image path.
  + Loading the image using the path.
  + Assigning the class label to y.
  + Returning a tuple of (image, y).

⚠️ This approach avoids loading all images into memory at once, making it scalable to large datasets.



### 🔹 TorchVision Image Transforms

TorchVision includes a powerful module of image transforms used during data preprocessing. Transforms are applied to modify images before passing them to a neural network.



🔸 **Composing Transforms:**

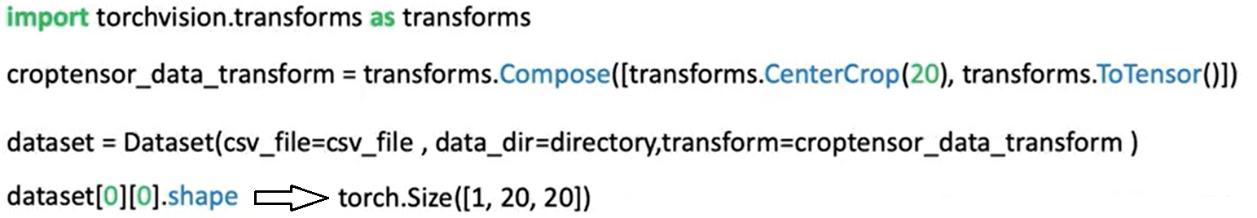
Multiple transforms can be combined into a sequence using **transforms.Compose**.

A Compose object accepts a list of transforms, when a sample is passed through the Compose object:

* + The first transform is applied.
  + The result is passed to the next transform.
  + The final transformed image is returned.

The composed transform is passed into the dataset’s constructor and applied automatically when samples are retrieved via **\_\_getitem\_\_**.

ℹ️ After transformation, image tensors have an extra dimension representing the batch or channel axis, required for model compatibility.



### ✅ Takeaways

✅Image datasets can be built using a CSV file mapping image names to class labels.

✅Data should be loaded **on demand** in the \_\_getitem\_\_ method to conserve memory and enable scalability.

✅TorchVision transforms such as cropping and tensor conversion are essential for preparing image inputs for model training.

✅The Compose utility allows multiple preprocessing steps to be applied sequentially and efficiently.