# Module 1

**Tensors and Datasets**

**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.

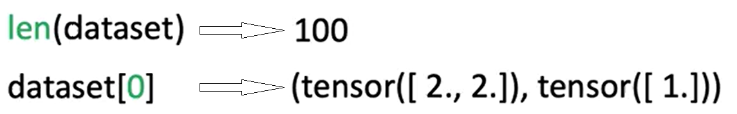
|  |  |
| --- | --- |
|  |  |

🔸 **Accessing Data Samples:**

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

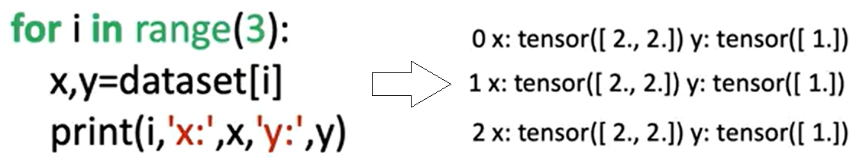
This method returns a tuple:

* 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.

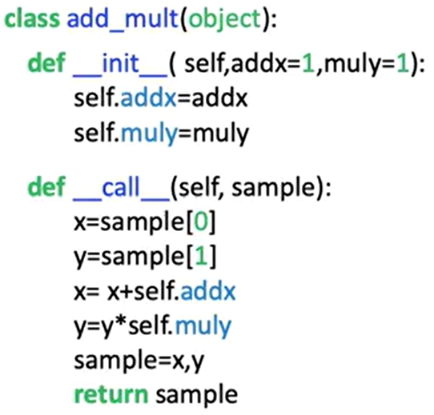


### 🔹 Applying Transforms to a Dataset

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.

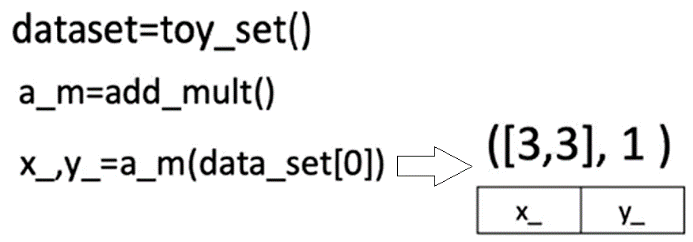
**Custom Transform Class**

* 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.

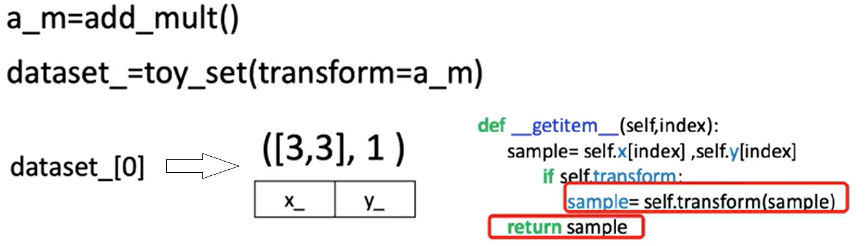
🔸 **Applying Transforms to Dataset Samples:**

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.

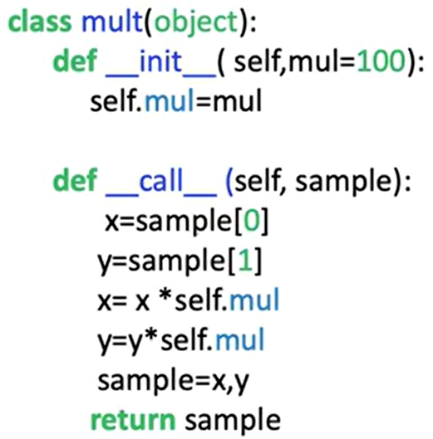
A list of transformation objects is passed to the Compose constructor.

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.

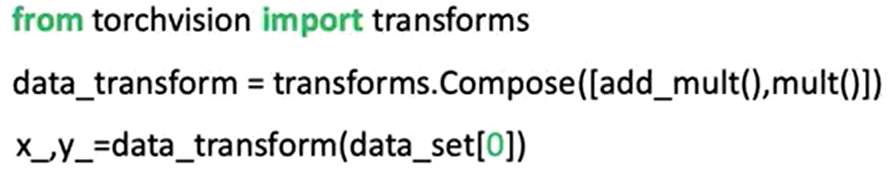
This compose object can be passed into the dataset class, enabling **automatic application of multiple transformations** during sample retrieval.

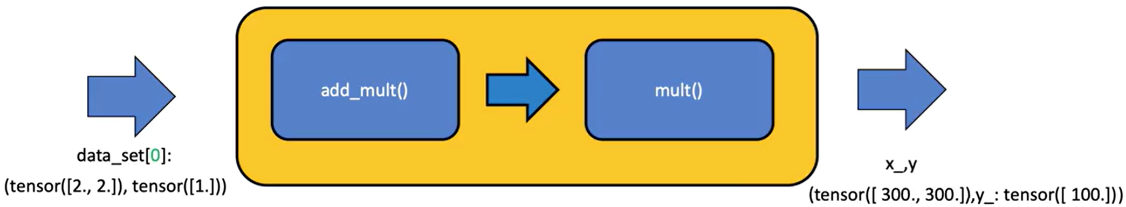
Let's say we would like to apply another transform, the class “**mult**” will multiply all the elements of a tensor by the value mul.



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

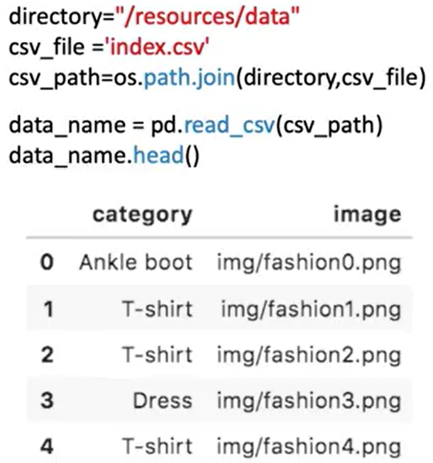
To construct an image dataset, the process begins by importing libraries from PyTorch, Pandas, and TorchVision.



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:**

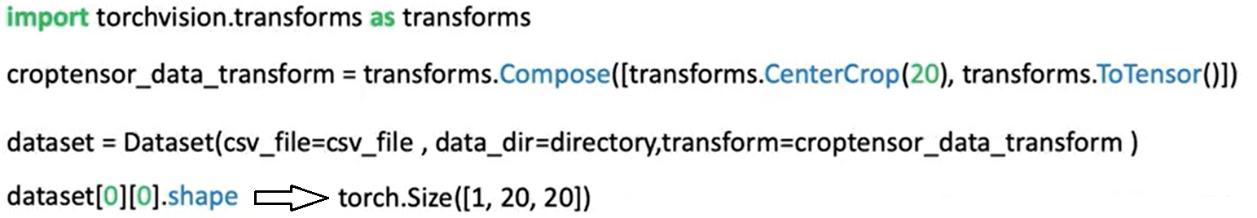
Multiple transforms can be combined into a sequence using **transforms.Compose**.

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.