Data preprocessing

Datasets and Dataloaders

MAC 2014-2020 Cooperación Territorial
Interreg
Fondo Europeo de Desarrollo Regional
EUROPEAN UNION
UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
T M E D C
Tecnología Médica para el Desarrollo Sostenible

1
Deep Learning workflow

1.1 Example of PyTorch workflow

What we are going to cover:

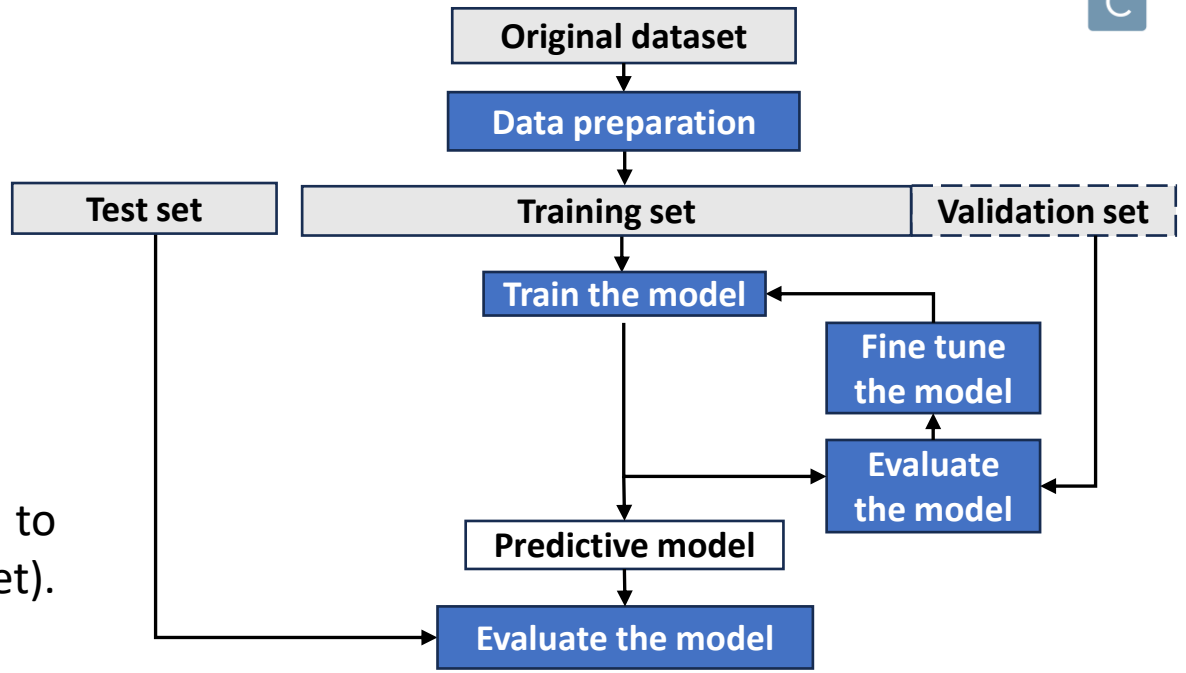
In this bootcamp, we will explore a standard PyTorch workflow, which can be adapted as necessary while following the main outline of steps. Steps 1-3 will be improved through experimentation until we get the desired results in step 4.

1. Getting and preparing data	2. Building or picking a pretrained model	3. Training	4. Evaluation	5. Deployment
▪ Collect the (labeled) data. ▪ Split them into a training /testing set. ▪ Apply transforms.	2 options: ▪ Train a model from scratch. ▪ Transfer learning using a pretrained model.	▪ Pick a loss function and optimizer. ▪ Build a training loop. ▪ Fit the model to the training set.	Calculate performance metrics on the testing set such as accuracy, recall and precision to evaluate the model.	Embeb the model wherever you need (desktop apps, enterprise systems…) and use it to make predictions.

1.2 Data preprocessing: Basic concepts

In this module we will explore the first phase

1. Getting and preparing data



A. Data splitting

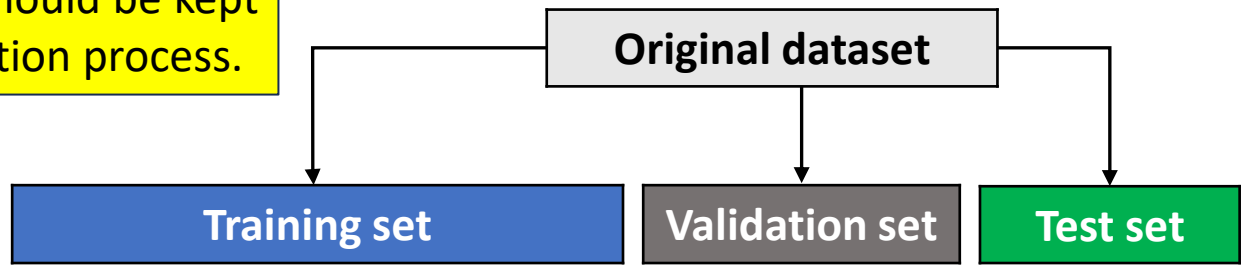
One the of most important steps in a deep learning project is to create a training and test set (and, when required, a validation set). Each split of the dataset serves a specific purpose:

Split	Training set	Validation set	Test set
Purpose	Used to train the machine learning model by learning patterns and relationships in the data.	Used to fine-tune and optimize the model during the training process. It helps to select the best-performing version and prevent overfitting.	Used to evaluate the final performance and generalization ability of the trained model on unseen data.
Percentage of total dataset (Tipically)	~60-80%	~10-20%	~10-20%
How often is it used?	Always	Often but not always	Always

1.2 Data preprocessing: Basic concepts

Very important!: The **training, testing, and validation datasets** should be kept **independent** and **not mixed** to ensure the integrity of the evaluation process.

Evaluating a model using training data can lead to higher metrics that do not accurately reflect its real-world performance, leading to misleading information about the model's true capabilities.



In many machine learning libraries, there are existing functions or modules to split data into training, validation, and testing sets. Here are some examples (in the links provided you will find more information about the parameters and outputs of each function):

- **PyTorch**
  `torch.utils.data.random_split`**:** Randomly split a dataset into non-overlapping new datasets of given lengths.

- **Scikit-learn (sklearn):**
  `sklearn.model_selection.train_test_split`**:** This function splits the data into training and testing sets. You can use it multiple times to further split the training set into training and validation sets.

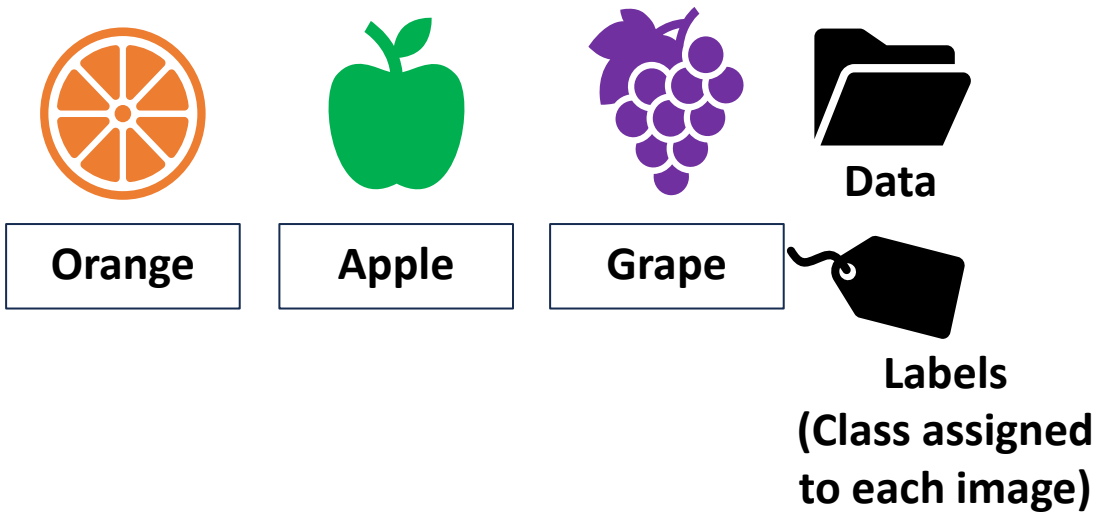
1.2 Data preprocessing: Basic concepts

B. Supervised learning: Labeled data

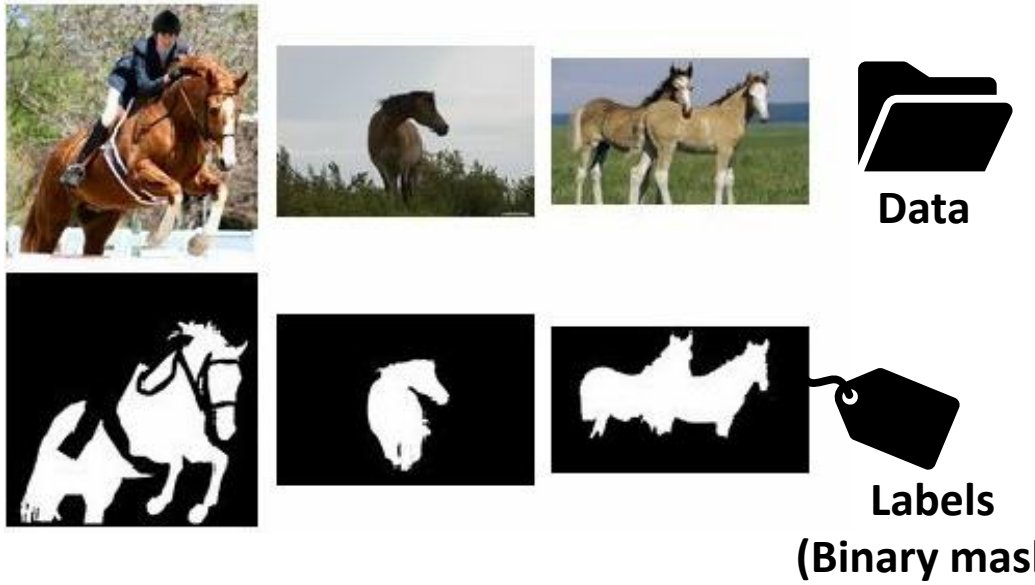
We will focus on **supervised learning**, where algorithms learn patterns and relationships in data using labeled examples. In this method, the **dataset** includes **input samples** with corresponding **target labels** (the desired outputs). The model learns from this labeled data to make predictions on new, unseen data.

Examples of input data and labels for a classification and segmentation problem are shown below:

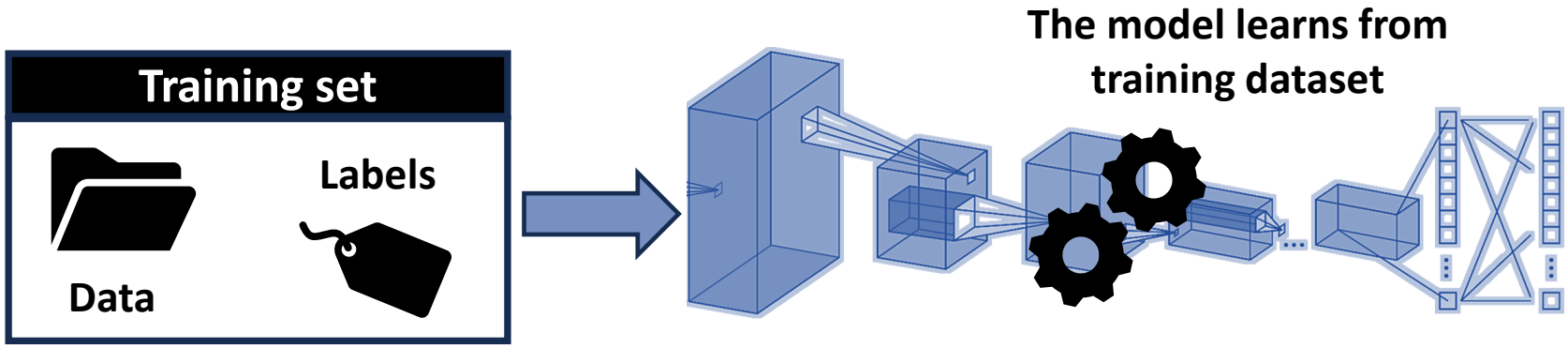
Multiclass classification



Semantic segmentation



1.2 Data preprocessing: Basic concepts



Overfitting occurs when a model becomes too specialized to the training data (learns its peculiarities) and performs poorly on new, unseen data. In this scenario, the model fails to generalize the knowledge and instead merely memorizes the training dataset.

Later on, we will explore various approaches to address overfitting. In terms of data preprocessing, there are primarily two options available to mitigate this effect:

- **Get more data:** Increasing the amount of data provides the model with a greater chance to discern generalizable patterns that can be applied to new examples

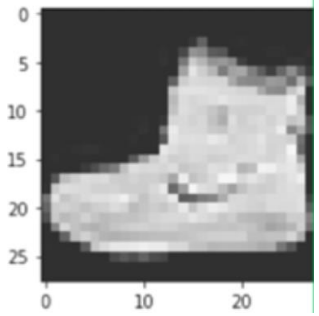
- **Use data augmentation**

1.2 Data preprocessing: Basic concepts

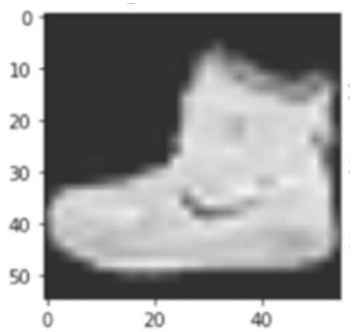
C. Data augmentation

Data augmentation is a technique widely used to generate artificial alterations on the original samples to **diversify** and **increase** the **training data**, especially when we have a small training set. The use of data augmentation is common when working with images, applying typical transformations such as rotations, scaling, brightness and contrast modifications, horizontal and vertical flipping or noise addition, among others. That way, we increase the diversity of your training dataset without collecting more data.

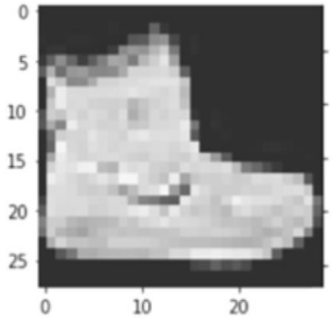
Original image



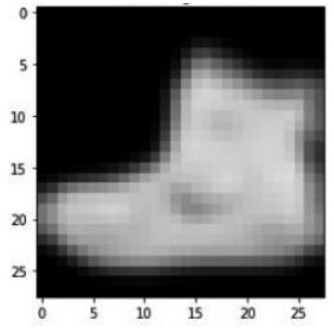
Data augmentation



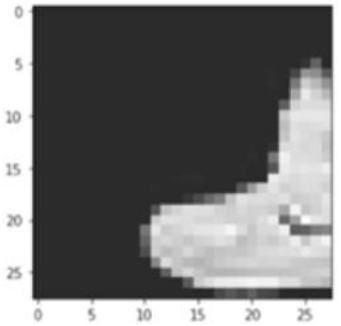
Scaling (50x50)



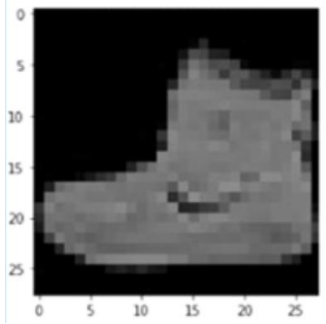
Horizontal flip



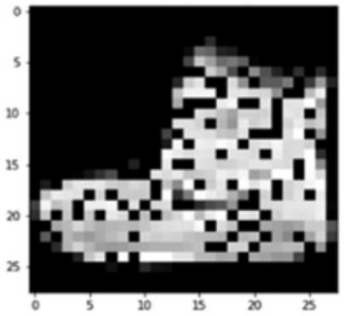
Gaussian blurring



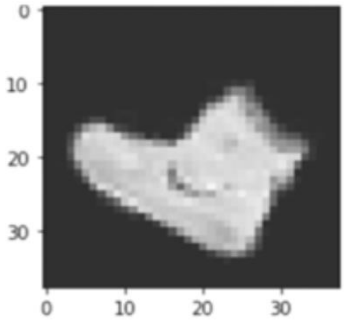
Pixel translation



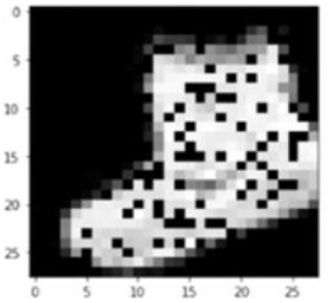
Contrast modification
Pixels multiplied by 0.5



Pixel dropout



Rotation by 30 degrees



Rotation + pixel dropout

2

PyTorch Datasets and Dataloaders

2.1 Pytorch for computer vision

PyTorch offers a range of pre-built datasets and functions tailored to specific problem domains.

PyTorch Libraries	torchvision	torchtext	torchaudio
Description	Package consisting of popular datasets, model architectures, and common image transformations for computer vision.	This library facilitates natural language processing (NLP) tasks. It simplifies the process of loading, processing, and iterating over text data.	It is focused on audio processing and analysis. It provides datasets, transforms, and utilities for working with audio data.

Essential computer vision modules in PyTorch for data preprocessing:

PyTorch module	Description
Pre-Built datasets: **`torchvision.datasets`**	Here you'll find many example computer vision datasets, providing access to the images and corresponding labels. It also contains a series of base classes for building custom datasets.
Data Transforms: **`torchvision.transforms`**	It provides a wide range of common image transformations and data augmentation techniques.
Custom Datasets: **`torch.utils.data.Dataset`**	Base dataset class for PyTorch which stores the samples and their corresponding labels. PyTorch enables the creation of custom datasets by subclassing the Dataset class.
Data Loaders: **`torch.utils.data.DataLoader`**	This class is used to efficiently load and iterate over the dataset. It allows for parallel data loading, shuffling, and batching.

torchvision

* **Note:** The **torch.utils.data.Dataset** and **torch.utils.data.DataLoader** classes aren't only for computer vision in PyTorch. They are capable of dealing with many different types of data.

2.2 Common Deep Learning datasets

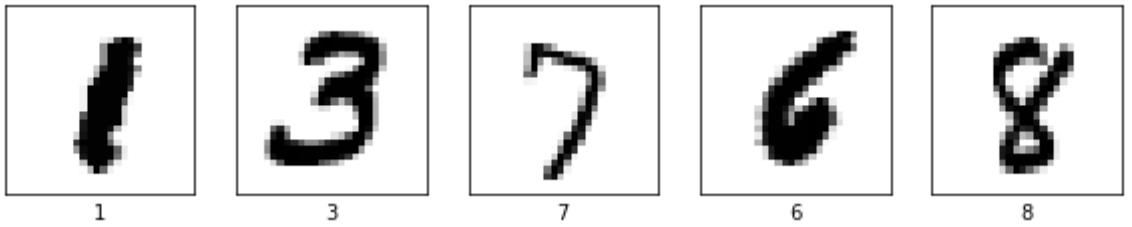
Frequently, in our work, we encounter widely-used datasets such as MNIST or CIFAR, which come pre-labeled and are widely recognized. These datasets are well-structured and require minimal preprocessing. Now, let's explore some examples of datasets available within the PyTorch library.

Additional information about torchvision datasets in [this link]

Image classification

- `torchvision.datasets.MNIST()`

  Dataset consisting of grayscale images of handwritten digits (0-9). It has over 60,000 training images and 10,000 test images.



- `torchvision.datasets.FashionMNIST()`

  This dataset includes clothing items like T-shirts, trousers, bags, etc. The number of training and testing samples is 60,000 and 10,000 respectively.

- `torchvision.datasets.CIFAR10()`
  `torchvision.datasets.CIFAR100()`

  The CIFAR dataset comprises colored images and offers two versions: CIFAR10 and CIFAR100, with 10 and 100 classes respectively. In both versions, there is a total of 60,000 images available.

2.2 Common Deep Learning datasets

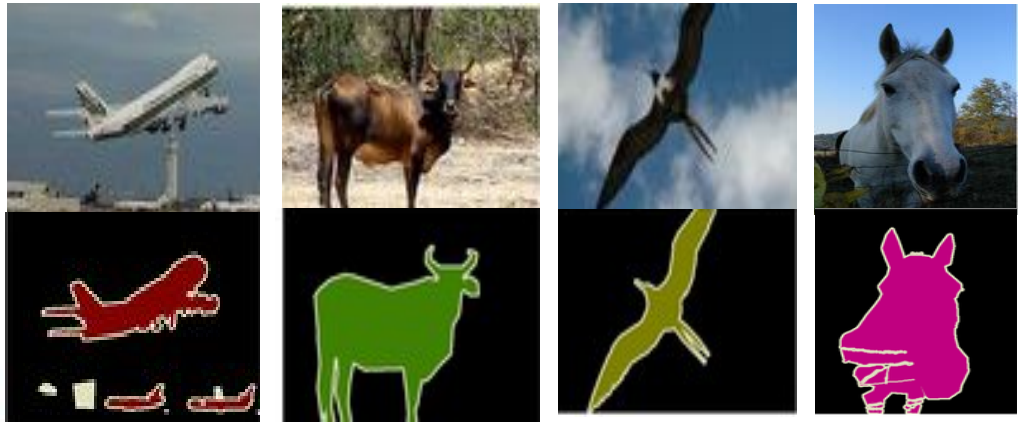
- `torchvision.datasets.ImageNet()`

  It consists of over 1.2 million images spread across 10,000 classes. Usually, this dataset is loaded on a high-end hardware system as a CPU alone cannot handle datasets this big in size.

Image segmentation and detection

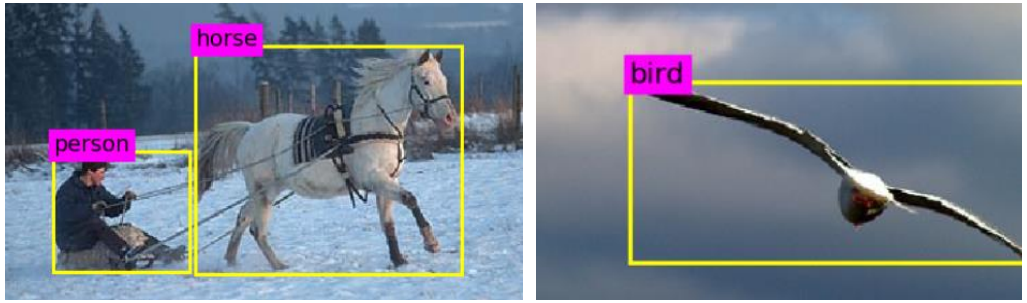
- `torchvision.datasets.VOCSegmentation()`

  This dataset is widely used for semantic segmentation tasks in computer vision. It consists of images from the PASCAL VOC challenge and provides pixel-level annotations for object classes within the images. The dataset contains several thousand images for training and evaluation.



- `torchvision.datasets.VOCDetection()`

  VOCDetection dataset is also part of the PASCAL VOC challenge and is primarily used for object detection tasks. It includes images annotated with bounding boxes around objects of interest and provides labels for various object categories.



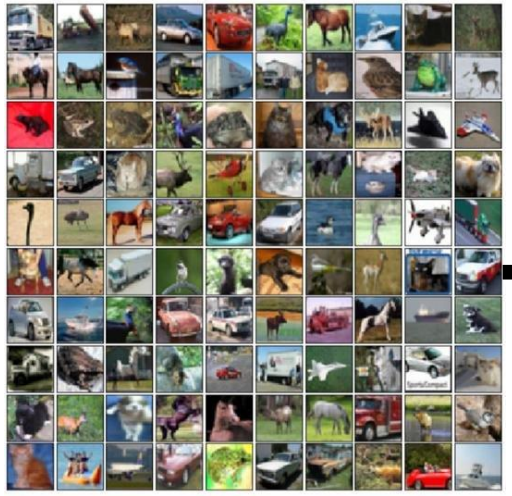
2.3 Dataloader

The torch.utils.data.DataLoader class is used in PyTorch to load and iterate over datasets during training or inference. When creating a DataLoader, we highlight four of its parameters:

- **dataset**: Dataset from which to load the data.
- **batch_size**: The number of samples in each batch (default: 1).
- **shuffle**: Boolean value indicating whether to shuffle the data order between epochs (default: False).
- **num_workers**: The number of subprocesses to use for data loading. It allows parallel data loading for faster processing (default: 0).

A batch refers to a group or subset of samples from a dataset that are processed together as a single unit during training or inference, rather than working with individual samples separately. Powers of 2 are commonly used for batch sizes: 16, 32, 64, 128…

`torchvision.datasets.CIFAR10`



[torch.utils.data.DataLoader]

```
# Create a DataLoader to iterate over the dataset
dataloader = DataLoader(dataset = cifar_dataset,
                        batch_size = 32,
                        shuffle = True,
                        num_workers = 1)
```

2.4 Example

```python
from torchvision import transforms
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader

# Define the transformation to preprocess the data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load the CIFAR10 dataset
cifar_dataset = CIFAR10(root ='./data',
                        train = True,
                        download = True,
                        transform = transform)

# Create a DataLoader to iterate over the dataset
dataloader = DataLoader(dataset = cifar_dataset,
                        batch_size = 32,
                        shuffle = True,
                        num_workers = 1)
```

Load a pre-built dataset

```python
# Load the CIFAR10 dataset
cifar_dataset = CIFAR10(root ='./data',
                        train = True,
                        download = True,
                        transform = transform)
```

Create a DataLoader

```python
# Create a DataLoader to iterate over the dataset
dataloader = DataLoader(dataset = cifar_dataset,
                        batch_size = 32,
                        shuffle = True,
                        num_workers = 1)
```

3
Custom Datasets

3.1 Work with your own data

We will work with 2 datasets that can be downloaded in the following links:

- Multiclass-image-dataset-airplane-car-ship
- Multi-class weather dataset

By considering the distinct structure of each dataset, we can explore effective strategies tailored to specific cases. It is worth noting that the means of transport dataset is already divided into pre-existing training and test sets.

When working with our own data, we have two primary options for handling datasets:

A. Use the **ImageFolder class**: It provides simplicity and convenience by automatically handling data loading and organization. It can be used when the dataset follows a specific directory structure.

B. Create a **custom dataset**: This approach offers more flexibility and customization options. It allows working with diverse data formats and structures.

```
Transport_Dataset/
    train/
        airplanes/
            airplane1.jpg
            airplane2.jpg
            ...
        cars/
            cars1.jpg
            cars2.jpg
            ...
        ships/
            ships1.jpg
            ships2.jpg
            ...
    test/
        airplanes/
            airplane1.jpg
            airplane2.jpg
            ...
        cars /
            cars1.jpg
            cars2.jpg
            ...
        ships/
            ships1.jpg
            ships2.jpg
            ...
```

```
Weather_Dataset /
        Sunrise/
            sunrise1.jpg
            sunrise2.jpg
            ...
        Rain/
            rain1.jpg
            rain2.jpg
            ...
        Shine/
            shine1.jpg
            shine2.jpg
            ...
        Cloudy/
            cloudy1.jpg
            cloudy2.jpg

            ...
```

3.2 ImageFolder class

ImageFolder is a generic data loader class in torchvision that facilitates loading our own image dataset. It is particularly useful for image datasets that follow a **specific directory structure**, where the data is organized into separate subdirectories, with **each subdirectory representing a distinct class** (as in the examples shown in the previous slide). In such cases, the ImageFolder class simplifies the process of loading and managing the dataset.

For example, to work with the “Transport_Dataset” we can use the following code:

```
from torchvision import datasets
from pathlib import Path

# Setup path to data folder
image_path = Path("../datasets/Transport_Dataset") # Path to Transport_Dataset folder

# Setup train and testing paths
train_dir = image_path / "train"
test_dir = image_path / "test"

# Use ImageFolder to create dataset(s)
train_data = datasets.ImageFolder(root = train_dir, # folder of images
                                  transform = data_transform, # transforms to perform on data (images)
                                  target_transform = None) # transforms to perform on labels (if necessary)

test_data = datasets.ImageFolder(root = test_dir,
                                 transform = data_transform)
```

3.3 Custom dataset

Working with custom datasets in PyTorch involves **creating a subclass of the torch.utils.data.Dataset class** and overriding some of its methods to provide access to our own data. This allows us to define how our dataset is loaded, preprocessed, and accessed during training or evaluation.

A custom Dataset class must implement at least three functions:

```python
import torch
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, ...):
        # Load and preprocess your data here

    def __len__(self):
        # Return the total number of samples in the dataset

    def __getitem__(self, index):
        # Retrieve and return a specific sample and its corresponding label/target
```

The **__init__** function is executed once when instantiating the Dataset object. It initializes the directory containing the images, the annotations file and transforms.

The **__len__** function returns the size of the dataset (the total number of samples).

The **__getitem__** function loads and returns a sample from the dataset at the given index.

An example showcasing the implementation of a custom dataset can be found in the provided notebooks. For further information and guidance on custom datasets, you can refer to the PyTorch Documentation

3.4 Code examples

Let’s code!

Work with pre-built datasets and create a custom dataset:

In the following notebooks, we will dive into the functionality of PyTorch's Dataset and DataLoader classes. We will learn how to load, process and visualize samples and labels, while also exploring how transformations and data augmentation techniques can modify images. To illustrate these concepts, we will showcase the usage of both a built-in PyTorch dataset and a personalized dataset created from our own images.