Dataset e Dataloader

Dataset e Dataloader

Premessa

In PyTorch il caricamento e la trasformazione dati è resa astratta e semplificata da due moduli principali:

Dataset

Utilizzato per il caricamento di dataset «standard» e dataset propri.

Dataset per:

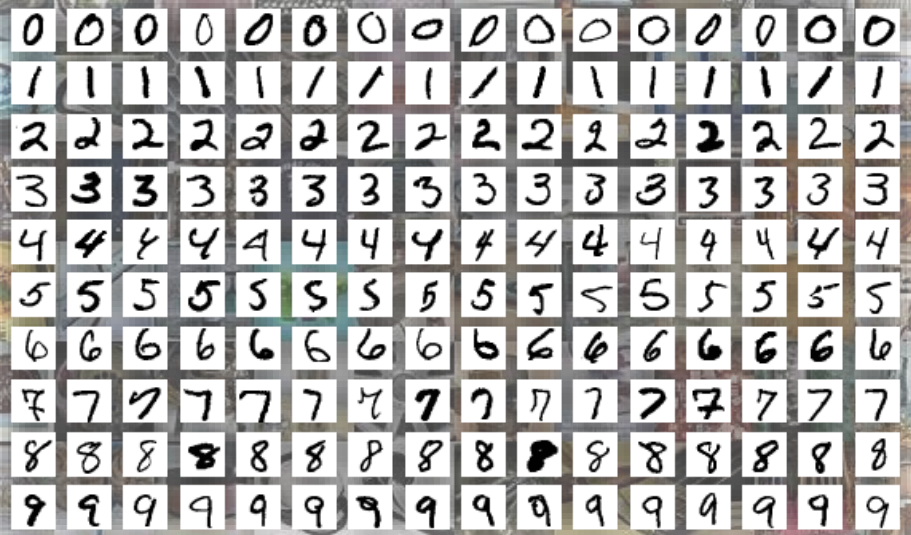
- *Immagini* : *torchvision*
- *Testo* : *torchtext*
- *Audio* : *torchaudio*

Dataloader

Per semplificare l'accesso diretto ai campioni o a parte dei campioni, un pezzo per volta.

Dataset e Dataloader

Esempi di dataset immagini

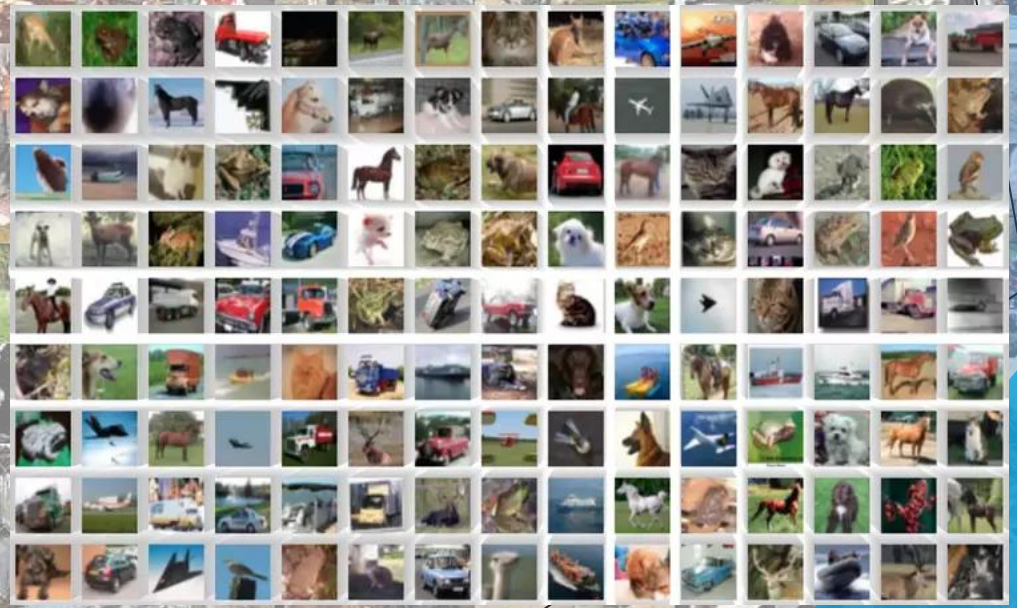


MNIST, FashionMNIST...

```
CLASS torchvision.datasets.FashionMNIST(root: str, train: bool = True,
      transform: Union[Callable, NoneType] = None, target_transform:
      Union[Callable, NoneType] = None, download: bool = False) → None
```

```
CLASS torchvision.datasets.CIFAR10(root: str, train: bool = True,
      transform: Union[Callable, NoneType] = None, target_transform:
      Union[Callable, NoneType] = None, download: bool = False) → None
```

CIFAR10, CIFAR100...



Dataset e Dataloader

Dataset custom

Creare un Dataset custom in PyTorch significa gestire manualmente la fase di accesso, recupero e, se necessario, manipolazione dei dati.

Si fa principalmente quando il dataset è costituito da dati a cui è difficile accedere direttamente o si vuole un controllo particolare su questi ultimi.

Si definisce custom Dataset un oggetto dalle seguenti caratteristiche:

1. È una classe Python.
2. Deriva dal modulo Dataset.
3. Implementa un metodo *__init__*.
4. Implementa un metodo *__len__*.
5. Implementa un metodo *__getitem__*.

Dataset e Dataloader

Dataset custom

Un Dataset custom è una classe Python che deriva dal modulo Dataset. Facendo questo si potrà sfruttare la possibilità di utilizzare il modulo Dataloader e i sistemi automatici di trasformazione e manipolazione dati.

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Dataset e Dataloader

Dataset custom

Implementa un metodo *__init__* nel quale si memorizzano tutti i parametri utili all'accesso ai dati, alla loro validazione e manipolazione.

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Dataset e Dataloader

Dataset custom

Implementa un metodo *__len__* utilizzato, quando necessario, per indicare la dimensione totale del dataset; il numero di campioni.

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Dataset e Dataloader

Dataset custom

Implementa un metodo *__getitem__*, fondamentale, nel quale viene inserita tutta la logica di accesso, lettura ed eventualmente manipolazione dei dati al fine poi di restituire i campioni, le etichette…

```python
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Dataset e Dataloader

Il Dataloader

Ottenuto un dataset, custom e meno, si può sfruttare il Dataloader per ottenere un metodo iterativo di accesso ai dati.

Nella sua versione più semplice, il Dataloader richiede semplicemente di conoscere il Dataset da cui estrarre dati.

Se necessario, è però possibile settare una delle molteplici opzioni, fra cui:

- La quantità di dati per iterazione, ***batch_size***.
- L'eventuale mescolamento dei dati, ***shuffle***.
- …

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Dataset e Dataloader

Trasformazioni dei dati

Con Dataset e Dataloader è possibile sfruttare una gran quantità di operazioni di modifica dedicate ai dati ed applicate ad essi in fase di accesso e caricamento.

Le operazioni vanno semplicemente indicate e fornite al modulo Dataset dedicato all'accesso dati.

```
transforms = torch.nn.Sequential(
    transforms.CenterCrop(10),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
)
```

Il modulo PyTorch dedicato alle trasformazioni è, nel caso delle immagini, *torchvision.transforms* . Di seguito un riferimento alle trasformazioni possibili:

- Rif: *Illustration of transforms*

Proviamo?