Published in MLearning.ai

Eugenia Anello Follow

Jan 27, 2022 · 6 min read · ✦ · Listen

Manipulating Pytorch Datasets

How to work with Dataloaders and Datasets for Deep Learning

Photo by Valdemaras D. on Unsplash

The post is the second in a series of guides to build deep learning models with Pytorch. Below, there is the full series:

Part 1: Pytorch Tutorial for Beginners

Part 2: Manipulating Pytorch Datasets (this post)

Part 3: Understand Tensor Dimensions in DL models

Part 4: CNN & Feature visualizations

Part 5: Hyperparameter tuning with Optuna

Part 6: K Fold Cross Validation

Part 7: Convolutional Autoencoder

Part 8: Denoising Autoencoder

Part 9: Variational Autoencoder

The goal of the series is to make Pytorch more intuitive and accessible as possible through examples of implementations. There are many tutorials on the Internet to use Pytorch to build many types of challenging models, but it can also be confusing at the same time because there are always slight differences when you pass from one tutorial to another. In this series, I want to start from the simplest topics to the more advanced ones.

The purpose of this post is to help you in getting familiar with the Pytorch Datasets. Before going deeper with models based on neural networks, you need to understand the general concepts and ideas of two main classes that provide data: **Dataset** and **Dataloader**. You are wondering what is difference between the Dataset and the Dataloader. The **Dataset** is used to store the samples and the corresponding targets. If you already worked with Pandas, it's very similar to a standard Pandas Dataframe as structure.

Why should we also need the Dataloader? The **Dataloader** is important to get easy access to the Dataset and split the data into **batches**, where the batches are the number of samples passed in one iteration to the Machine Learning model. The more the dataset is huge, the more time is needed to train the model. Dividing the dataset into batches provides a way to speed up the training speed of the model.

In this post, I provide an overview to work with these two classes. Later, I show how to perform four common operations to manipulate your dataset:

1. **Filter class from Pytorch Dataset**
2. **Concatenate Pytorch Datasets**
3. **Convert Pandas Dataframe into Pytorch Dataset**

4. Random Sampling from Pytorch Dataset

Import libraries and dataset

There are two important libraries you should keep attention at:

- `torch.utils.data` contains two main classes to store the data: `Dataset` and `Dataloader`
- `torchvision` provides pre-loaded datasets, like MNIST and Fashion MNIST

```
import torch
from torch.utils.data import Dataset,DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

Let's now import the Fashion MNIST [1], a popular dataset that contains Zalando's article images. Each sample is constituted by an image and the corresponding label that belongs to one of the ten classes.

```
1  dataset_path = 'datasets'
2  batch_size=256
3  train_dataset = datasets.FashionMNIST(dataset_path, transform=ToTensor(), train=True, download=True
4  test_dataset  = datasets.FashionMNIST(dataset_path, transform=ToTensor(), train=False, download=Tru
```

.py hosted with ❤ by GitHub view raw

To check out how many samples the training and test datasets contain, we need the well-known function `len` :

```
print(len(train_dataset)) #60000
print(len(test_dataset)) #10000
```

We can also get access separately to the images and the labels:

```
train_dataset.data
# tensor([[[0, 0, 0,  ..., 0, 0, 0],...[0, 0, 0,  ..., 0, 0, 0]]], #
dtype=torch.uint8)

train_dataset.targets
# tensor([9, 0, 0,  ..., 3, 0, 5])
```

This output means that the first image corresponds to class 9, the successive two images to class 0 and so on [2]. We can easily verify it by displaying the first image of the training dataset:

```
labels_map = {0: "T-Shirt",1: "Trouser",2: "Pullover",3: "Dress",4: "Coat",5: "Sandal",
              6: "Shirt",7: "Sneaker",8: "Bag",9: "Ankle Boot"}

img, label = train_dataset[0]
plt.title('{}:{}'.format(label,labels_map[label]))
plt.axis("off")
plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

.py hosted with ❤ by GitHub view raw

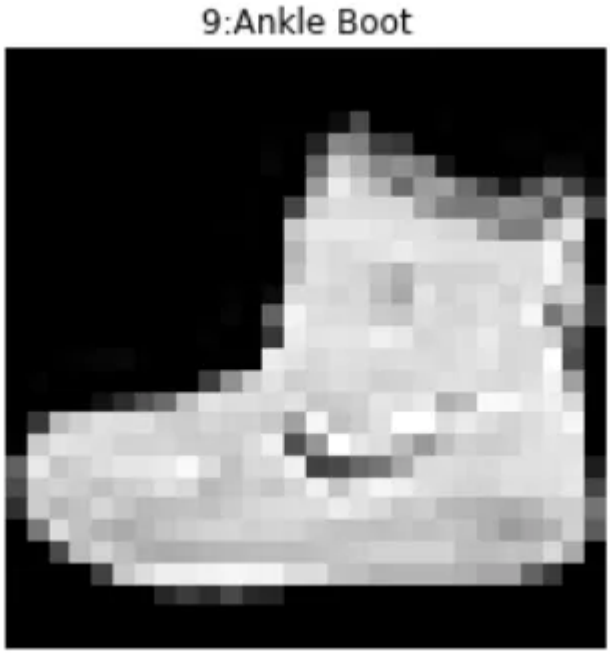

Illustration by Author

Now, we can create a new data loader, based on the training dataset, with a batch size equal 256:

```
train_loader = DataLoader(dataset=train_dataset, batch_size=256,
shuffle=True)
```

We can iterate through the DataLoader using the `iter` and `next` functions:

```
train_features, train_labels = next(iter(train_loader))
print(len(train_labels)) #256
```

We selected a batch containing 256 images, each one belonging to different classes.

1. Filter class from Pytorch Dataset

Let's suppose that we only want the images of T-shirts from the training dataset. We'll need the following lines of code:

```
from torch.utils.data import Subset

train_idx = np.where((train_dataset.targets==0))[0]
train_subset = Subset(train_dataset, train_idx)
train_loader_subset = DataLoader(train_subset, shuffle=True, batch_size=batch_size)
```

.py hosted with ❤ by GitHub view raw

We have employed:

- `np.where` to yield a NumPy array containing the indexes where there are only images of t-shirts in the training dataset.
- `Subset` to select a subset of the dataset, based on the indices passed.

Let's display some images contained in train_subset:

```
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):

    sample_idx = torch.randint(len(train_subset), size=(1,)).item()
    img, label = train_subset[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title('{}:{}'.format(label,labels_map[label]))
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")

plt.show()
```

.py hosted with ❤ by GitHub view raw

Illustration by Author

There is a rich variety of t-shirts, don't you think?

2. Concatenate Pytorch datasets

Let's merge the training dataset and test dataset into a unique dataset:

```
from torch.utils.data import ConcatDataset

dataset = ConcatDataset([train_dataset,test_dataset])
data_loader = DataLoader(dataset=dataset, batch_size=batch_size, shuffle=True)
len(dataset) #70000
len(data_loader) #274
```

.py hosted with ❤ by GitHub view raw

The class `ConcatDataset` takes in a list of multiple datasets and returns a concatenation of these ones. In this way, we obtain a dataset of 7000 samples and a dataloader with 274 batches.

3. Convert Pandas dataframe into pytorch dataset

To train a neural network implemented with Pytorch, we need a Pytorch dataset. The issue is that Pytorch doesn't provide all the datasets of the world and we need to make some efforts to convert the dataframes into the Pytorch tensor. For example, let's import the Boston housing prices dataset from sklearn [3]:

```
from sklearn.datasets import load_boston
import pandas as pd

boston  = load_boston()
df = pd.DataFrame(data=boston.data,columns=boston.feature_names)
df['target'] = boston.target
df.head()
```

.py hosted with ❤ by GitHub view raw

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	target
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Illustration by Author

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

target = torch.tensor(df['target'].values)
features = torch.tensor(df.drop('target', axis = 1).values)
train = TensorDataset(features, target)
train_loader = DataLoader(train, batch_size=2, shuffle=True)
```

.py hosted with ❤ by GitHub view raw

Let's iterate again through the dataloader to visualize what we have obtained:

```python
train_features, train_labels = next(iter(train_loader))
print(train_features)
print(train_labels)
```

```
tensor([[3.4660e-02, 3.5000e+01, 6.0600e+00, 0.0000e+00, 4.3790e-01, 6.0310e+00,
         2.3300e+01, 6.6407e+00, 1.0000e+00, 3.0400e+02, 1.6900e+01, 3.6225e+02,
         7.8300e+00],
        [5.6440e-02, 4.0000e+01, 6.4100e+00, 1.0000e+00, 4.4700e-01, 6.7580e+00,
         3.2900e+01, 4.0776e+00, 4.0000e+00, 2.5400e+02, 1.7600e+01, 3.9690e+02,
         3.5300e+00]], dtype=torch.float64)
tensor([19.4000, 32.4000], dtype=torch.float64)
```

Illustration by Author

We can notice that a batch containing two samples is printed. Unluckily `TensorDataset` can have some limitations when the dataset's structure is more complex.

An alternative and more efficient method is to create a class, which inherits the attributes and the methods of the class `Dataset` . Especially, we can override the methods of the class `Dataset` , depending on the dataset you are working with:

- `__getitem__` method is used for accessing ith sample of the dataset.
- `__len__` method returns the dataset's number of sample.

```python
class Boston_Dataset(Dataset):

    def __init__(self,df):
        self.df = df

    def __getitem__(self,idx):
        self.data = torch.from_numpy(self.df.drop(['target'],axis=1).values)
        self.targets = torch.from_numpy(self.df['target'].values)
        return self.data[idx],self.targets[idx].item()

    def __len__(self):
        return len(self.targets)
```

.py hosted with ❤️ by GitHub view raw

Then, we create a new object belonging to the class `Boston_Dataset` , which will be passed to the `data_loader` function:

```
b_data = Boston_Dataset(df=df)
b_loader = DataLoader(dataset=b_data,batch_size=4,shuffle=True)

for x,y in b_loader:
    print(y)
```

```
tensor([24.0000, 21.6000, 34.7000, 33.4000], dtype=torch.float64)
tensor([36.2000, 28.7000, 22.9000, 27.1000], dtype=torch.float64)
tensor([16.5000, 18.9000, 15.0000, 18.9000], dtype=torch.float64)
tensor([21.7000, 20.4000, 18.2000, 19.9000], dtype=torch.float64)
tensor([23.1000, 17.5000, 20.2000, 18.2000], dtype=torch.float64)
tensor([13.6000, 19.6000, 15.2000, 14.5000], dtype=torch.float64)
tensor([15.6000, 13.9000, 16.6000, 14.8000], dtype=torch.float64)
```

4. Random Sampling from the Dataset

Sometimes it can happen that you want to select random samples from the dataset. Theoretically, there is already the class `RandomSampler` that does it automatically, but it works only with replacement. For this reason, I will need the following trick to select the samples **WITHOUT** replacement.

If you instead prefer to draw the sample WITH replacement, you can directly use the class `RandomSampler`

Final thoughts:

That's it! Now you should be able to work with Pytorch datasets with less effort. It can be confusing at first, but after some practice, you'll manage all the difficulties. Thanks for reading. Have a nice day!

References:

[1]https://pytorch.org/vision/main/generated/torchvision.datasets.FashionMNIST.html

[2] https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

[3] https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_boston.html