

[home](#) [blog](#) [search](#)

Getting Started with PyTorch: Understanding Dataset and DataLoader

January 2, 2025

The best source for PyTorch tutorials is actually the PyTorch website itself. In fact, most of the tutorial I am going to give below is inspired by the quick start tutorial on the PyTorch website: [PyTorch Quick Start Tutorial](#). That said, I hope the comments I am going to make and the explanations I am going to provide will help you understand the fundamentals of PyTorch deeply and will enable you to appreciate the framework the same way I do.

One thing I've always loved about PyTorch is its intuitive design - it feels like writing regular Python code while giving you access to powerful deep learning capabilities. Unlike some other frameworks that hide the mechanics behind abstractions, PyTorch lets you peek under the hood and understand what's really happening.

Alright, here we go...

What is Needed to Train a PyTorch Model?

To train a model in PyTorch, you primarily need three things:

1. **Data:** Methods to load your data and make it available for PyTorch to consume. This includes loading the data from disk or memory, performing any necessary preprocessing (e.g., converting NumPy arrays to PyTorch tensors), batching, shuffling, etc.
2. **Model:** A class that describes your model structure and forward pass. PyTorch makes it convenient by requiring you to define only the forward pass while relying on its built-in gradient tracking for the backward pass. However, if you need to implement a custom function, you'll need to define both forward and backward pass logic for that function. Most of the time, the functionalities you need are already implemented in PyTorch.
3. **Training Loop:** This is where everything is stitched together. The training loop involves loading data, using it to train the model, calculating the loss function, propagating back the gradients, and updating the model parameters. PyTorch's flexibility allows you to implement the training loop yourself, instead of relying on higher-level abstractions like `model.fit`. This hands-on approach deepens your understanding of deep learning.

This particular blog post focuses on the **Data** aspect.

Note: In upcoming blog posts, we will dive deep into the remaining components - exploring Model architecture and implementation, followed by putting it all together in the Training Loop.

Before we begin, make sure to install the required packages: `torch`, `torchvision`, and `pandas`. You can install these using `pip`, Python's package installer.

Data

Two crucial classes you should be familiar with in PyTorch are `Dataset` and `DataLoader`:

- The `Dataset` class defines your dataset and how to fetch a single row of it.
- The `DataLoader` class wraps the `Dataset` class and handles batching, shuffling, and utilizing Python's `multiprocessing` to speed up data retrieval.

From PyTorch documentation:

"`Dataset` stores the samples and their corresponding labels, and `DataLoader` wraps an iterable around the `Dataset` to enable easy access to the samples."

PyTorch also provides a list of built-in datasets to get you started.

Dataset Class

Toy Dataset: FashionMNIST

```
from torchvision import datasets
from torchvision.transforms import ToTensor

# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

The above code downloads the FashionMNIST dataset and splits it into training and test data based on the `train` parameter. The downloaded data is in the

PIL.Image.Image data type, but the `transform=ToTensor()` parameter converts it into PyTorch tensors.

FashionMNIST is a simple dataset where each row in the training data represents a single-channel image with 28×28 pixels, making the feature vector size (1×28×28). The dataset contains 10 classes labeled from 0 to 9 (e.g., Coat, Sandal, etc.).

Inspecting the Dataset

We can examine some properties of the dataset:

```
print(training_data)
```

Output:

```
Dataset FashionMNIST
  Number of datapoints: 60000
  Root location: data
  Split: Train
  StandardTransform
  Transform: ToTensor()
```

You can inspect the first data point:

```
print(len(training_data[0])) # 2
```

This indicates each data point is a tuple of features and label. Let's examine these components:

```
# Features
print(type(training_data[0][0])) # torch.Tensor

# Label
print(type(training_data[0][1])) # int
```

```
# Shape of features
print(training_data[0][0].shape) # torch.Size([1, 28, 28])
```

In the above example, we used PyTorch's built-in `torchvision` datasets for simplicity. You can also create your own `Dataset` class, which is particularly useful for custom datasets, such as those in a Pandas DataFrame.

Lazy Loading

You might wonder if the FashionMNIST dataset loads all the data into memory. The answer is: it does not. Instead, it downloads the images to a directory and performs lazy loading—only loading the image into memory when you access it (e.g., `training_data[0]`). This is crucial for working with large datasets, as it prevents memory exhaustion.

This is one of PyTorch's strengths - it's designed to handle datasets that are too large to fit in memory. Imagine working with a dataset of high-resolution images or videos that could easily be hundreds of gigabytes. With lazy loading, you can work with these datasets on even modest hardware.

The `DataLoader` class handles batching and fetching data on demand from the `Dataset`. We'll discuss this more in later sections.

Your Own Pandas Dataset

Below is an implementation of a custom `Dataset` class to load data from a Pandas DataFrame:

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

class DataFrameDataset(Dataset):
    def __init__(self, dataframe, feature_names, label_name):
        """
        Initializes the dataset.
```

```

    Args:
        dataframe (pd.DataFrame): The dataframe containing features a
        feature_names (list of str): List of column names to use as f
        label_name (str): Name of the column to use as the label.
    """
    self.dataframe = dataframe
    self.feature_names = feature_names
    self.label_name = label_name

def __len__(self):
    """
    Returns the length of the dataset.
    """
    return len(self.dataframe)

def __getitem__(self, idx):
    """
    Fetches the row at the given index.

    Args:
        idx (int): Index of the row to fetch.

    Returns:
        tuple: (features, label), where features is a tensor of input
        label is a tensor containing the output.
    """
    row = self.dataframe.iloc[idx]
    features = torch.tensor(row[self.feature_names].values, dtype=torch.float32)
    label = torch.tensor(row[self.label_name], dtype=torch.float32)
    return features, label

```

Custom Dataset Details

Creating a custom Dataset class involves:

1. Inheriting from the `torch.utils.data.Dataset` base class
2. An `__init__` function to load your data or point to its location
3. A `__len__` function to return the dataset size

4. A `__getitem__` function to fetch one row from your dataset

The example above accepts a Pandas DataFrame, a list of feature column names, and the label column name. This dataset is fully loaded into memory (unlike FashionMNIST), making it suitable for small-to-medium datasets. For larger datasets, you may need to implement strategies like lazy loading, which is beyond the scope of this post but might be covered in future blogs.

Example: Creating and Using the Dataset

```
import pandas as pd

# Example DataFrame
data = {
    "feature1": [1.0, 2.0, 3.0],
    "feature2": [4.0, 5.0, 6.0],
    "label": [0, 1, 0]
}
df = pd.DataFrame(data)
print(df)
```

Output:

	feature1	feature2	label
0	1	4	0
1	2	5	1
2	3	6	0

```
# Define features and label
feature_names = ["feature1", "feature2"]
```

```
label_name = "label"

# Create the dataset
dataset = DataFrameDataset(df, feature_names, label_name)

# Access elements
for i in range(len(dataset)):
    features, label = dataset[i]
    print(f"Features: {features}, Label: {label}")
```

Output:

```
Features: tensor([1., 4.]), Label: 0.0
Features: tensor([2., 5.]), Label: 1.0
Features: tensor([3., 6.]), Label: 0.0
```

In this example, we build a toy Pandas DataFrame and use the custom `Dataset` class to interact with it as a PyTorch dataset.

DataLoader Class

The `DataLoader` class is simply a wrapper around the `Dataset` class. More specifically, it wraps an iterable around the `Dataset` to enable easy access to the samples. Let's dive into some examples. Back to the FashionMNIST `Dataset` we declared above.

```
from torch.utils.data import DataLoader

batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
```



```
print(f"Shape of y: {y.shape} {y.dtype}")  
break
```

Output:

```
Shape of X [N, C, H, W]: torch.Size([64, 1, 28, 28])  
Shape of y: torch.Size([64]) torch.int64
```

Since we know each row in the training data corresponds to one single-channel image of dimensions (1×28×28) and a single `int` label, it readily follows that for a batch of 64 images, we will have a Tensor of shape (64, 1, 28, 28) for the input features and a tensor of shape (64,) for the output labels. Notice the `shuffle=True` parameter in the `train_dataloader` which is meant to shuffle the data in the batch. Shuffling training data is a common practice meant to ensure the model generalizes well by exposing it to varied data order during training.

The `DataLoader` class is an iterable around the `FashionMNIST Dataset`. We can loop over it as we do with any iterable, but we cannot access specific indices, which is again expected for iterable objects. In fact, if you attempt to do so, you'll get (`TypeError: 'DataLoader' object is not subscriptable`). Since the `Dataset` object has an internal `__getitem__` method, the `DataLoader` will interface natively with this method to fetch a full `batch_size` worth of data. This will become apparent when we talk about the **Training Loop** in a follow-up blog post.

The `DataLoader` class can also interface with custom datasets like the `Pandas DataFrame`-based dataset described earlier, handling batching, shuffling, and other optimizations in the same way.

Conclusion

In this post, we covered the fundamentals of data handling in PyTorch, focusing on two essential classes:

- The `Dataset` class for defining and accessing your data
- The `DataLoader` class for efficient batching and iteration

Understanding these data handling components is often overlooked in favor of jumping straight to model building, but proper data management is the foundation of any successful deep learning project. The flexibility PyTorch offers through these classes allows you to work with any data type and format, from simple CSV files to complex multi-modal datasets.

One of the most powerful aspects of PyTorch's data pipeline is how it scales from simple prototypes to production systems. The same `Dataset` and `DataLoader` abstractions work whether you're experimenting on a small dataset locally or processing terabytes of data across multiple GPUs.


We explored both built-in datasets like FashionMNIST and implementing custom datasets for Pandas DataFrames. Stay tuned for the next posts in this series where we'll explore:

- Part 2: Building Models in PyTorch
- Part 3: Implementing the Training Loop

References


- [PyTorch Quick Start Tutorial](#)


 [RSS](#)

 [source code](#)

 [github](#)

 [linkedin](#)

 [resume](#)

 [google scholar](#)

© 2025 MIT Licensed