PyTorch Tutorial

08. Dataset and DataLoader

pytorch 中的工具

目标：拿出一个 mini-Batch

结合随机梯度下降和梯度下降，均衡 性能和训练时间

Revision: Manual data feed

```
xy = np.loadtxt( 'diabetes.csv.gz' , delimiter= ',' , dtype=np.float32)
x_data = torch.from_numpy(xy[:,:-1])
y_data = torch.from_numpy(xy[:, [-1]])

......

for epoch in range(100):
    # 1. Forward
    y_pred = model(x_data)
    loss = criterion(y_pred, y_data)
    print(epoch, loss.item())
    # 2. Backward
    optimizer.zero_grad()
    loss.backward()
    # 3. Update
    optimizer.step()
```

Use all of the data

Terminology: Epoch, Batch-Size, Iterations

```
# Training cycle
for epoch in range(training_epochs):
    # Loop over all batches
    for i in range(total_batch):
```

训练次数

每次循环执行一个 mini-Batch

对每个 Batch 迭代一次训练中的

一次用到所有样本的训练

Definition: Epoch

One forward pass and one backward pass of **all the training examples**.

Definition: Batch-Size

The **number of training examples** in one forward backward pass.

每次训练的样本数量

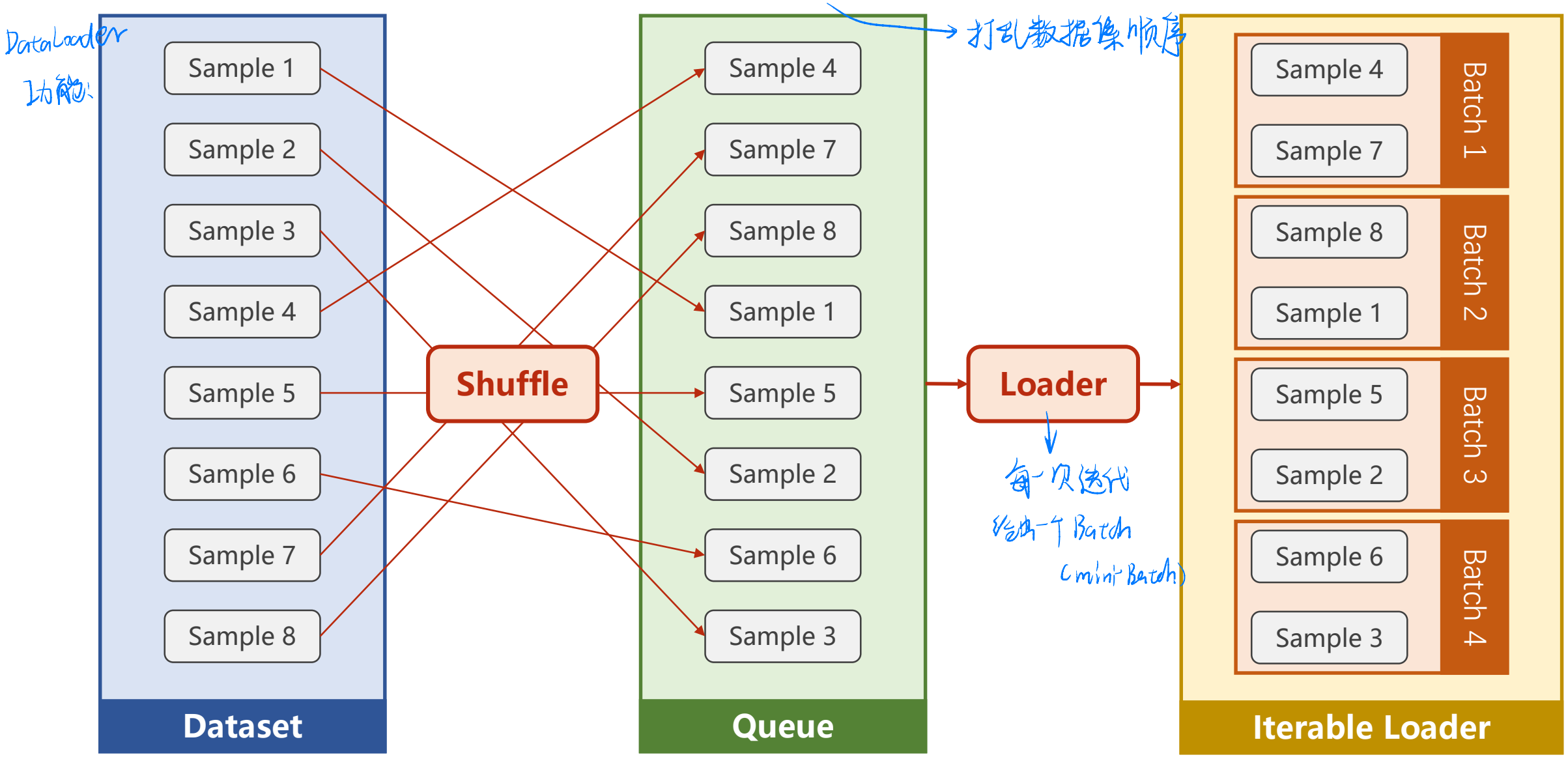
即 1 个 mini-Batch 中的样本量

Definition: Iteration

Number of passes, each pass using [**batch size**] number of examples.

1 个 Batch 中分了多少个 mini-Batch

DataLoader: batch_size=2, shuffle=True



How to define your Dataset

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

class DiabetesDataset(Dataset):
    def __init__(self):
        pass

    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

Dataset is an **abstract** class. We can define our class inherited from this class.

How to define your Dataset

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

class DiabetesDataset(Dataset):
    def __init__(self):
        pass

    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

DataLoader is a class to help us loading data in PyTorch.

How to define your Dataset

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

class DiabetesDataset(Dataset):
    def __init__(self):
        pass

    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

DiabetesDataset is inherited from abstract class **Dataset**.

How to define your Dataset

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

class DiabetesDataset(Dataset):
    def __init__(self):
        pass

    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

The expression, **dataset[index]**, will call this magic function.

__init__ 构造数据类的2种方法

① 将数据集一次性传入 __init__，使用时用下标选择即可

若数据集过大

② 只是作一些读写工作，(数据在文件中)

如，定义一个列表保存各个文件名；对标签 { 如果复杂：放入文件中，列表到列表 / 如果简单：直接加载

在 __getitem__ 中

选择到各种的文件

How to define your Dataset

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

class DiabetesDataset(Dataset):
    def __init__(self):
        pass

    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

This magic function returns length of dataset.

How to define your Dataset

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

class DiabetesDataset(Dataset):
    def __init__(self):
        pass

    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

This magic function returns length of dataset.

How to define your Dataset

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

class DiabetesDataset(Dataset):
    def __init__(self):
        pass

    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

Construct DiabetesDataset object.

How to define your Dataset

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

class DiabetesDataset(Dataset):
    def __init__(self):
        pass

    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

dataset = DiabetesDataset()
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
```

(mini-batch 大小)

shuffle 打乱, 一般为 Ture

num_workers 指在读数据时并行几个进程

Initialize loader with batch-size, shuffle, process number.

Extra: *num_workers* in Windows

```
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
......
for epoch in range(100):
    for i, data in enumerate(train_loader, 0):
        ......
```

So we have to **wrap** the code with an if-clause to protect the code from executing multiple times.

The implementation of multiprocessing is different on Windows, which uses **spawn** instead of **fork**. (Linux)

So left code will cause:

RuntimeError:

An attempt has been made to start a new process before the current process has finished its bootstrapping phase.

This probably means that you are not using fork to start your child processes and you have forgotten to use the proper idiom in the main module:

```
if __name__ == '__main__':
    freeze_support()
    ...
```

The "freeze_support()" line can be omitted if the program is not going to be frozen to produce an executable.

Extra: *num_workers* in Windows

```
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)
......
if __name__ == '__main__':
    for epoch in range(100):
        for i, data in enumerate(train_loader, 0):
            # 1. Prepare data
```

要保证代码封装起来

So we have to **wrap** the code with an if-clause to protect the code from executing multiple times.

Example: Diabetes Dataset

例：

```
class DiabetesDataset(Dataset):
    def __init__(self, filepath):
        xy = np.loadtxt(filepath, delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, :-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len

dataset = DiabetesDataset('diabetes.csv.gz')
train_loader = DataLoader(dataset=dataset, batch_size=32, shuffle=True, num_workers=2)
```

xy.shape=(行数,列数)，是个元组，xy.shape[0]即为行数=样本数

返回一个元组

Example: Using DataLoader

循环的变化：

```
for epoch in range(100):
    for i, data in enumerate(train_loader, 0):
        # 1. Prepare data
        inputs, labels = data
        # 2. Forward
        y_pred = model(inputs)
        loss = criterion(y_pred, labels)
        print(epoch, i, loss.item())
        # 3. Backward
        optimizer.zero_grad()
        loss.backward()
        # 4. Update
        optimizer.step()
```

为了获得迭代次数，放到这里

即数据集使用的正确类别

预测函数

梯度清零

Dataset每次拿出一个样本 x[i]，y[i]

而Data_loader，每次根据Batch大小将x，y变为矩阵 [ ], [ ]，并自动转换为Tensor

X Y

Classifying Diabetes

```python
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader

class DiabetesDataset(Dataset):
    def __init__(self, filepath):
        xy = np.loadtxt(filepath, delimiter=',', dtype=np.float32)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, :-1])
        self.y_data = torch.from_numpy(xy[:, [-1]])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len

dataset = DiabetesDataset('diabetes.csv.gz')
train_loader = DataLoader(dataset=dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=2)

class Model(torch.nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.linear1 = torch.nn.Linear(8, 6)
        self.linear2 = torch.nn.Linear(6, 4)
        self.linear3 = torch.nn.Linear(4, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.linear1(x))
        x = self.sigmoid(self.linear2(x))
        x = self.sigmoid(self.linear3(x))
        return x

model = Model()

criterion = torch.nn.BCELoss(size_average=True)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for epoch in range(100):
    for i, data in enumerate(train_loader, 0):
        # 1. Prepare data
        inputs, labels = data
        # 2. Forward
        y_pred = model(inputs)
        loss = criterion(y_pred, labels)
        print(epoch, i, loss.item())
        # 3. Backward
        optimizer.zero_grad()
        loss.backward()
        # 4. Update
        optimizer.step()
```

完整代码

仍是分成四部分

1. **Prepare dataset**
   Dataset and Dataloader
2. **Design model using Class**
   inherit from nn.Module
3. **Construct loss and optimizer**
   using PyTorch API
4. **Training cycle**
   forward, backward, update

The following dataset loaders are available

- MNIST
- Fashion-MNIST
- EMNIST
- COCO
- LSUN
- ImageFolder
- DatasetFolder
- Imagenet-12
- CIFAR
- STL10
- PhotoTour

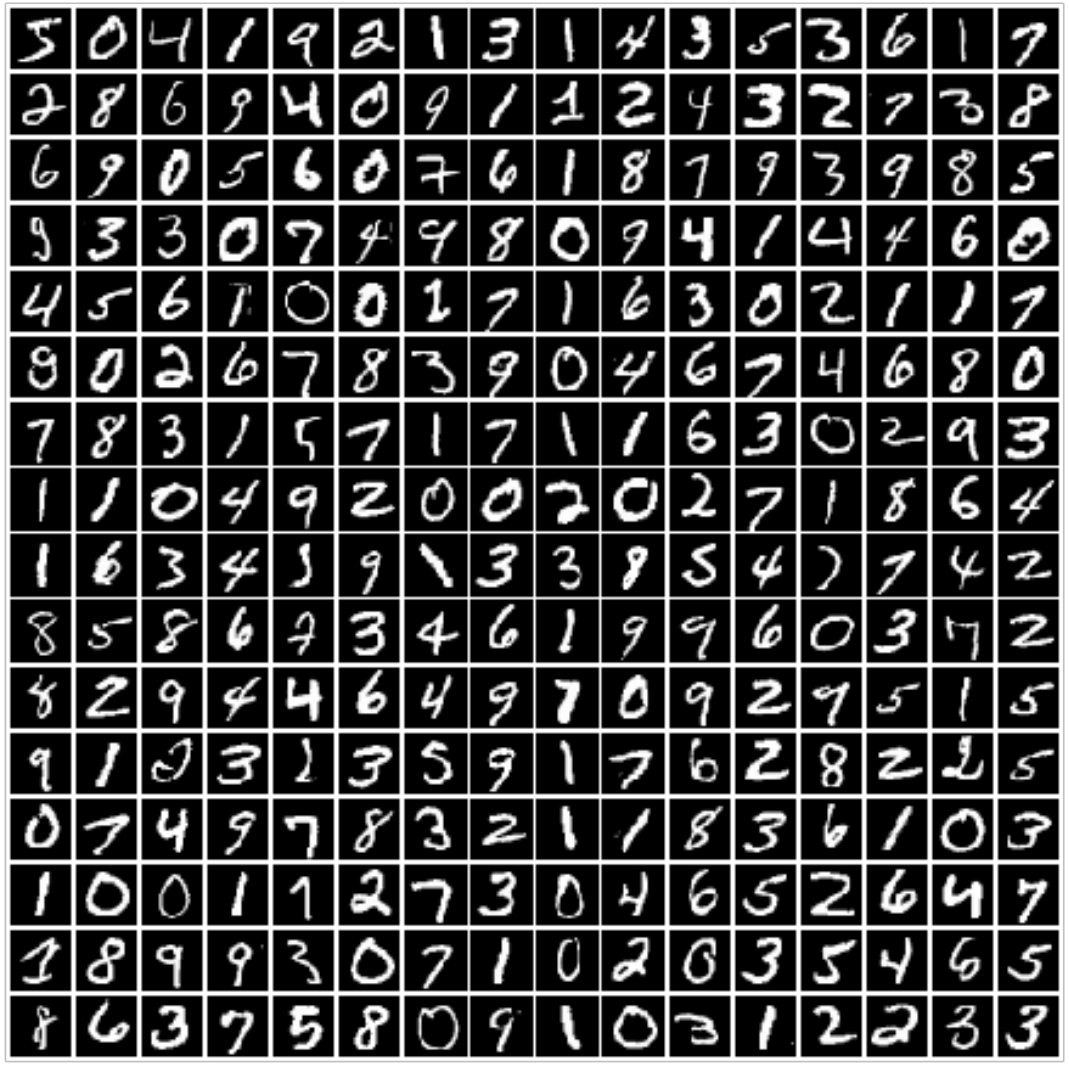
torchvision.datasets

All datasets are subclasses of `torch.utils.data.Dataset` i.e, they have `__getitem__` and `__len__` methods implemented. Hence, they can all be passed to a `torch.utils.data.DataLoader` which can load multiple samples parallelly using `torch.multiprocessing` workers. For example:

```
imagenet_data = torchvision.datasets.ImageFolder('path/to/imagenet_root/')
data_loader = torch.utils.data.DataLoader(imagenet_data,
                                          batch_size=4,
                                          shuffle=True,
                                          num_workers=args.nThreads)
```

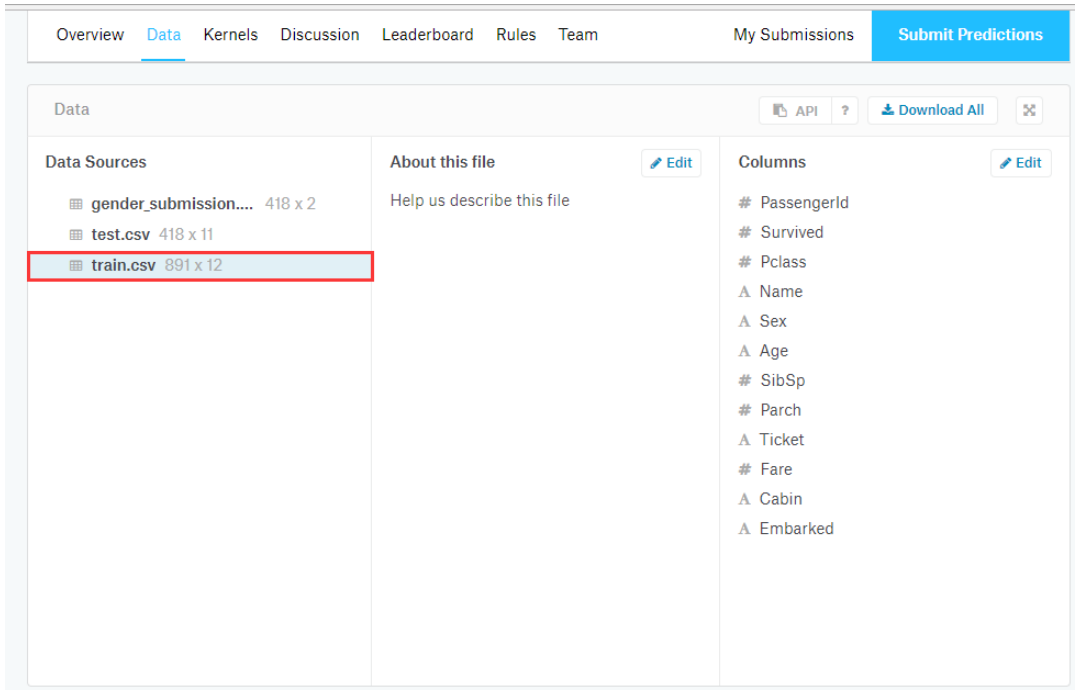
Example: MINST Dataset

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

train_dataset = datasets.MNIST(root='../dataset/mnist',
                               train=True,
                               transform= transforms.ToTensor(),
                               download=True)
test_dataset = datasets.MNIST(root='../dataset/mnist',
                              train=False,
                              transform= transforms.ToTensor(),
                              download=True)

train_loader = DataLoader(dataset=train_dataset,
                          batch_size=32,
                          shuffle=True)
test_loader = DataLoader(dataset=test_dataset,
                         batch_size=32,
                         shuffle=False)

for batch_idx, (inputs, target) in enumerate(train_loader):
    ......
```



Exercise 8-1

- Build DataLoader for
  - Titanic dataset: https://www.kaggle.com/c/titanic/data
- Build a classifier using the DataLoader



PyTorch Tutorial

08. Dataset and DataLoader