深度学习数据集的创建与读取

数据 （计算机术语）

数据(data)是事实或观察的结果，是对客观事物的逻辑归纳，是用于表示客观事物的未经加工的的原始素材。

数据可以是连续的值，比如声音、图像，称为模拟数据。也可以是离散的，如符号、文字，称为数字数据。

在计算机系统中，数据以二进制信息单元 0,1 的形式表示。

数据 （汉语词语）

数据就是数值，也就是我们通过观察、实验或计算得出的结果。数据有很多种，最简单的就是数字。数据也可以是文字、图像、声音等。数据可以用于科学研究、设计、查证等。

数据的形式在深度学习中是什么样的呢？数据在深度学习中是以多维数组的形式存储的，而数据代表的就是样本的数字信息特征，一般来说，对于单一样本的特征，可以像影响房价的因素那样是一维向量数组，也可以是图像特征那样是二维数组，还可以是后面介绍的语言文字那样是一维词向量数组，在训练的时候，通常会批量读取训练，所以数据集会在原有的维度基础上加一个 batch_size 的维度。

一直在强调，深度学习中最重要的因素之一是数据集，而刚刚入门深度学习的时候也经常会在数据集上无从下手。一般来说，针对数据集，我们需要明确 2 个要素，以什么样的方式存储在哪里，怎么读取用于训练。

数据存储与数据划分

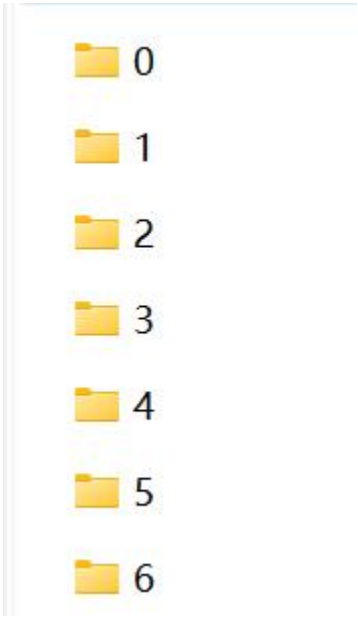
将数据按照一定的格式存储起来形成的集合就是数据集。

深度学习中，接下来我们将结合 pytorch 的实例详细介绍一下数据集相关知识。

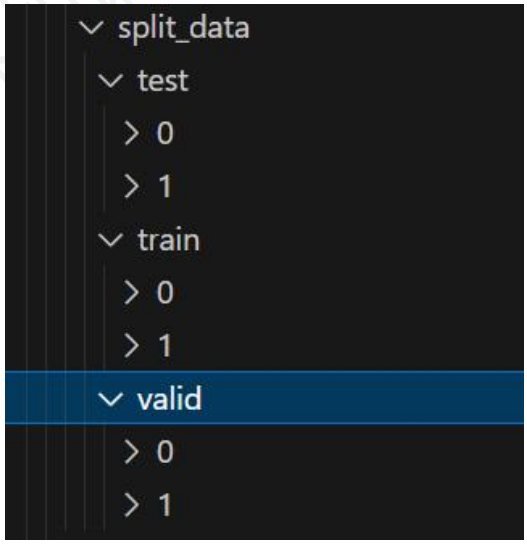
除非使用已经封装好的数据集如 MNIST 等可以直接调用 api 接口导入数据，我们在网上下载也好，自己收集标注也好，都需要将图片，音频，视频，语言文字等模拟数据存储在电脑或服务器特定的路径位置下。

另一点值得注意的是下载和收集的数据要符合特定的格式（尤其不同的模型可能要求数据的格式是不同的，比如 MASK-RCNN 和 YOLO），这样才能方便代码读取，这里暂时不做过多的展开介绍，后面进行补充，这篇文章重点还是介绍数据的读取过程。

数据集的两个重要因素一是原始数据特征，比如图片，文字等，二是标签，比如类别或者物体分割时的位置等，在存储时要分清他们的位置，便于读取。 物体分割时比如 MASK-RCNN 和 YOLO，需要使用标注工具 label 等手动获取标签；对于分类任务，一般可以将不同类别的图片划分到不同的子文件夹里，并命名为类别或数字，读取标签的时候直接读取子文件夹名称即可。如下图是表情分类任务的数据集，以不同的数字代表表情类别。



数据集一般会划分为训练，测试和验证三个子集，网上下载好的数据一般是划分好的，若没有划分好或者是自己收集制作的，可以自己划分。最快速上手的方式就是手动分割，不过代码具有泛用性，数据划分主要是路径的处理和文件的遍历，下面以具体例子说明，最开始是 data 文件夹下有 2 个类别的子文件夹 0 和 1，



现在我们想要将左图中的原始数据切割成右图中的分 train，test，valid 的数据集，关键点就是遍历左边的数据，然后设置比例分为三份，并按比例将其复制到右边。这里边涉及几个重要的文件处理函数，一个是 os.makedirs("path")，在检测到路径无文件夹存在时调用创建文件夹；一个是 os.listdir(path),会将路径下的文件或文件夹名称储存到列表里；一个是 os.walk(root_dir)，是深度遍历的方式遍历自根文件夹下的所有子文件夹及文件，返回的是三元组（root，subdirs,filenames），

每一层遍历：

root 保存的就是当前遍历的文件夹的绝对路径；

subdirs 保存当前文件夹下的所有子文件夹的名称（仅一层，孙子文件夹不包括）

filenames 保存当前文件夹下的所有文件的名称

举个例子，
-dir1
-1.jpg
-subdir1
--2.jpg
--3.jpg
-subdir2
--4.jpg
--5.jpg
os.walk() 之后遍历的结果就是
第一次遍历输出三元组
dir1 [subdir1，subdir2]，1.jpg
第二次遍历输出三元组
dir/subdir1，[]，[2.jpg，3.jpg]
第三次遍历输出三元组
dir/subdir2，[]，[4.jpg，5.jpg]
而且需要注意的是，这三个输出是在同一个 for path，dirnames，filenames in os.walk(root)下递归执行的

检测路径是否存在，若不存在，则创建此路径。

```
def makedir(new_dir):
    if not os.path.exists(new_dir):
        os.makedirs(new_dir)
```

设置路径，将它们组合在一起。相对于 Python 文件所在位置的相对路径。

```
dataset_dir = os.path.join("..", "..", "data", "RMB_data")
```

深度遍历，获取根文件夹和子文件夹下的文件，并对其按照设置的比例存储，以遍历第一个子文件夹为例说明一下：

```
#进入深度遍历
for root, dirs, files in os.walk(dataset_dir):
#开始遍历第一个子文件夹
    for sub_dir in dirs:
#将此子文件夹下的文件名存储到列表里
        imgs = os.listdir(os.path.join(root, sub_dir))
        imgs = list(filter(lambda x: x.endswith('.jpg'), imgs))
```

```
            random.shuffle(imgs)
            img_count = len(imgs)
#设置划分比例
            train_point = int(img_count * train_pct)
            valid_point = int(img_count * (train_pct + valid_pct))
#开始按照划分比例切分图片文件
            for i in range(img_count):
                if i < train_point:
#根据归属于不同的比例填入或创建目标路径
                    out_dir = os.path.join(train_dir, sub_dir)
                elif i < valid_point:
                    out_dir = os.path.join(valid_dir, sub_dir)
                else:
                    out_dir = os.path.join(test_dir, sub_dir)
#创建目标路径
                makedir(out_dir)
#将文件从源文件复制到目标文件位置
                target_path = os.path.join(out_dir, imgs[i])
                src_path = os.path.join(dataset_dir, sub_dir, imgs[i])
                shutil.copy(src_path, target_path)
```

shutil.copy()Python 中的方法用于将源文件的内容复制到目标文件或目录。它还会保留文件的权限模式，但不会保留文件的其他元数据(例如文件的创建和修改时间)。源必须代表文件，但目标可以是文件或目录。如果目标是目录，则文件将使用源中的基本文件名复制到目标中。另外，目的地必须是可写的。如果目标是文件并且已经存在，则将其替换为源文件，否则将创建一个新文件。

数据的变换

在图像数据集的训练中，经常会利用 torchvision.transforms.Compose()将对图像预处理的操作以列表[,,]的形式组合在一起，然后赋值给到 data_transform，并作为后续创建 dataset 时的参数，最常用的分为两类：

一是图像本身的变换，如随机裁剪和翻转，图像变换的时候要注意维度保持一致。

（1）transforms.CenterCrop(size)

从图片中心截取 size 大小的图片。

（2）transforms.RandomCrop(size,padding,padding_mode)

随机裁剪区域。

（3）transforms.RandomResizedCrop(size,scale,ratio)

随机大小，随机长宽比的裁剪。

（1）transforms.RandomHorizationalFlip(p)

依据概率 p 水平翻转。
（2）transforms.RandomVerticalFlip(p)
依据概率 p 垂直翻转。
（3）transforms.RandomRotation(degrees,resample,expand)
二是将“图像”转化为“tensor”，我们知道，任何数据（包括图像音频等）在计算机上的计算都是要转化成数字信息的，所以我们需要将图像数据转化成 pytorch 中数字信息以及存储格式 tensor。
transforms.Totensor（）
将图像数据转换为 tensor。
transforms.Normalize(mean,std,inplace)
逐通道的标准化，每个通道先求出平均值和标准差，然后标准化。Inplace 表示是否原地操作。
下面代码为常用的数据变换格式，还有很多其它变换方法，可以在需要使用的时候查询。

```
train_transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])
```

数据集转换 dataset

获得并划分处理好的格式数据之后，需要利用代码将其转变为模型训练可用的数据集，所谓数据集，其实就是一个负责处理索引(index)到样本(sample)映射的一个类(class)。

数据集的三个核心参数便是从格式数据传递的存储路径，给每一个数据加的索引，以及数据集的大小（数据集总数）。

dataset(继承于 Dataset)是可以我们自己用代码实现的一个类，这个类中主要包括__init__()，__getitem__()，__len__()这三个函数：

1）__init__()：构造函数，该函数里面会定义存放数据的路径，一般由人给定输入。

2）__getitem__()：这个函数把传入的索引 Index 和数据（标签）对应起来，返回数据（标签）；

3）__len__()：这个函数获取数据集中样本的总个数。

Pytorch 提供两种数据集： *Map 式数据集* Iterable 式数据集

目前深度学习将数据转换成训练可用的数据集有三种形式，

PyTorch 集成的数据集接口

实际上，PyTorch 提供了很多常用数据集的接口，如果使用这些数据集的话，可以直接使用对应的包加载，会方便很多，比如：

- torchvision.datasets 就提供了很多视觉方向的数据集：https://pytorch.org/docs/stable/torchvision/datasets.html?highlight=torchvision datasets
- torchtext 则提供了很多文本处理方向的数据集
- torchaudio 提供了很多音频处理方向的数据集

当然 PyTorch 也可以配合其他包来获得数据以及对数据进行处理，比如：

- 对于视觉方面，配合 Pillow、OpenCV 等
- 对于音频处理方面，配合 scipy、librosa 等
- 对于文本处理方面，配合 Cython、NLTK、SpaCy 等

实际例子来说，比如 MNIST,CIFAR10 等，这时可以直接利用一句代码读取出来：

在 PyTorch 中 CIFAR10 是一个写好的 Dataset，我们使用时只需以下代码：

```
data = datasets.CIFAR10("./data/", transform=transform, train=True, download=True)
```

此时，代码会检查当前路径下“./data”是否存在 cifar10 数据集，若不存在，download 为 true 时则启动下载，transform 后面会介绍。

ImageFolder 等 API

二是网络上一些公开的数据集，这时可以直接下载下来，然后利用定好的方法将其转化为模型训练的数据集，比如我们下载一些图片数据，放到一个文件夹下，这时可以使用 ImageFolder 这个方便的 API。

```
FaceDataset = datasets.ImageFolder('./data', transform=img_transform)
```

这个是应用比较广泛的方法，因为很多时候我们会将图片放到特定的文件夹路径下。这里需要注意标签是利用 api 自动读取的文件夹的名称并返回列表的。

自定义一个数据集

三则是自定义一个数据集，**torch.utils.data.Dataset** 是一个表示数据集的抽象类。任何自定义的数据集都需要继承这个类并覆写相关方法。这个其实是更深入的了解一下 dataset 数据集的本质。

Map 式数据集

一个 Map 式的数据集必须要重写 **getitem**(self, index),**len**(self) 两个内建方法，用来表示从索引到样本的映射（Map）.

Map 式的数据集 dataset，举个例子，当使用 dataset[idx]命令时，可以在你的硬盘中读取你的数据集中第 idx 张图片以及其标签（如果有的话）;len(dataset)则会返回这个数据集的容量。

自定义类大致是这样的：

```
class CustomDataset(data.Dataset):#需要继承 data.Dataset
    def __init__(self):
        # 可以初始化文件路径或者数据转换格式等其它处理信息
    def __getitem__(self, index):
        # 根据输入的索引参数读取一个数据和标签，可以将数据进行处理后返回数据和标签
    def __len__(self):
        # 返回数据集的大小
```

例子-1： 图片文件储存在“./data/faces/”文件夹下，图片的名字并不是从 1 开始，而是从 final_train_tag_dict.txt 这个文件保存的字典中读取，label 信息也是用这个文件中读取。

```
from torch.utils import data
import numpy as np
from PIL import Image
class face_dataset(data.Dataset):
    def __init__(self,label_dict):
        self.file_path = './data/faces/'
        self.label_dict=label_dict
    def __getitem__(self,index):
        label = list(self.label_dict.values())[index-1]
        img_id = list(self.label_dict.keys())[index-1]
        img_path = self.file_path+str(img_id)+".jpg"
        img = np.array(Image.open(img_path))
```

```
        return img,label

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

Iterable 式数据集

一个 Iterable（迭代）式数据集是抽象类 **data.IterableDataset** 的子类，并且覆写了 **iter** 方法成为一个迭代器。这种数据集主要用于数据大小未知，或者以流的形式的输入，本地文件不固定的情况，需要以迭代的方式来获取样本索引。

小结

在实际的应用中，可以根据具体的任务或调用 API，或自己重写数据集类，一般会在创建数据集的步骤中将数据变换列表 transforms 作为参数写进去，直接对数据进行变换处理。

数据样本加载 DataLoader

创建完成数据集 dataset 后，还不能直接用于网络训练，下一步需要构建迭代器 DataLoader 作为网络训练时读取数据的方式，而数据集 dataset 是 DataLoader 实例化的一个参数。一般格式为：

```
train_dataloader=DataLoader(train_dataset,batch_size=16,shuffle=True)
```

Dataloader 的一些常用参数

Dataloader 的一些重要的参数如下，除了第一个 dataset 参数外，一般会设置 batch_size，shuffle，其他均视情况可选：

- **dataset**（第一个参数，必须的参数）：一个 Dataset 的实例，即前面三种方式创建传入的数据集（或者其他可迭代对象）
- **batch_size**：整数值，每个 batch 的样本数量，即 batch 大小，默认为 1
- **shuffle**：bool 值，如果设置为 True，则在每个 epoch 开始的时候，会对数据集的数据进行重新排序，默认 False
- sampler：传入一个自定义的 Sampler 实例，定义从数据集中取样本的策略，Sampler 每次返回一个索引，默认为 None
- batch_sampler：也是传入一个自定义的 Sampler 实例，但是与 sampler 参数不同的是，它接收的 Sampler 是一次返回一个 batch 的索引，默认为 None
- num_workers：整数值，定义有几个进程来处理数据。0 意味着所有的数据都会被加载进主进程，默认 0

- collate_fn：传入一个函数，它的作用是将一个 batch 的样本打包成一个大的 tensor，tensor 的第一维就是这些样本，如果没有特殊需求可以保持默认即可（后边会详细介绍）
- pin_memory：bool 值，如果为 True，那么将加载的数据拷贝到 CUDA 中的固定内存中。
- drop_last：bool 值，如果为 True，则对最后的一个 batch 来说，如果不足 batch_size 个样本了就舍弃，如果为 False，也会继续正常执行，只是最后的一个 batch 可能会小一点（剩多少算多少），默认 False
- timeout：如果是正数，表明等待从加载一个 batch 等待的时间，若超出设定的时间还没有加载完，就放弃这个 batch，如果是 0，表示不设置限制时间。默认为 0

Dataloader 参数之间的互斥

值得注意的是，Dataloader 的参数之间存在互斥的情况，这些初步了解即可。主要针对自己定义的采样器：

- sampler：如果自行指定了 sampler 参数，则 shuffle 必须保持默认值，即 False
- batch_sampler：如果自行指定了 batch_sampler 参数，则 batch_size、shuffle、sampler、drop_last 都必须保持默认值
  如果没有指定自己是采样器，那么默认的情况下（即 sampler 和 batch_sampler 均为 None 的情况下），Dataloader 的采样策略是如何的呢：
- sampler：shuffle = True：sampler 采用 RandomSampler，即随机采样

  shuffle = Flase：sampler 采用 SequentialSampler，即按照顺序采样

- batch_sampler：采用 BatchSampler，即根据 batch_size 进行 batch 采样

Sampler 类是一个很抽象的父类，其主要用于设置从一个序列中返回样本的规则，即采样的规则。

Dataloader 其实还有一个比较重要的参数是 collate_fn，它接收一个 callable 对象，比如一个函数，它的作用是将每次迭代出来的数据打包成 batch。

举个例子，如果我们在 Dataloader 中设置了 batch_size 为 8，实际上，从 Dataloader 所读取的数据集 Dataset 中取出数据时得到的是单独的数据，比如我们的例子中，每次采样得到一个 tuple：(image, label)，因此 collate_fn 的作用就有了，它负责包装 batch，即每从数据集中抽出 8 个这样的 tuple，它负责把 8 个 (image, label)包装成一个 list: [images, labels]，这个 list 有两个元素，每一个是一个 tensor，比如第一个元素，实际上是一个 8×size(image) 的 tensor，即给原来的数据增加了一维，也就是最前边的 batch 的维度，labels 也同理。

有时候我们可能会需要实现自己的包装逻辑，所以需要自定义一个函数来完成定制化的如上的内容，只要将该函数名传递给 collate_fn 参数即可。

DataLoader 读取数据样本的流程

综上，概括一下 DataLoader 读取数据样本的流程：

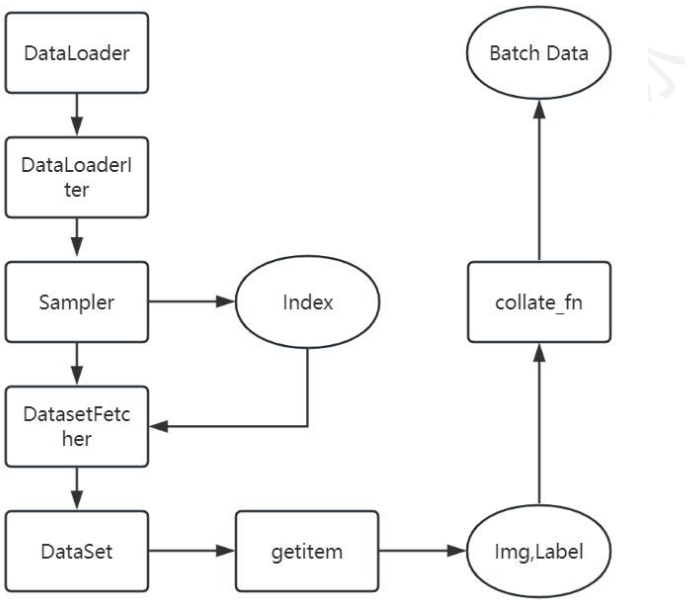
DataLoader 首先创建一个迭代器 DataloaderIter，

然后这个迭代器会去访问 Sampler 得到读取数据的索引值；

根据这个索引值去访问创建好的数据集 DataSet；

利用 DataSet 的 getitem 函数读取对应索引值的样本数据；

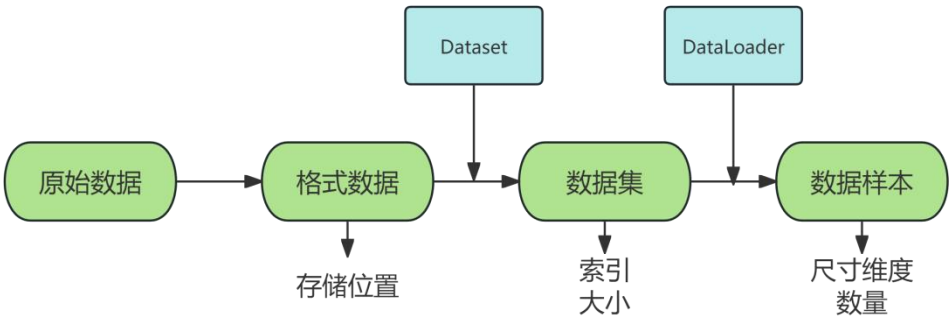
然后利用 collate_fn 将独立的样本数据打包成 batch_size 大小的 Batch Data。



深度学习数据集的创建与读取流程

而整个 PyTorch 中深度学习数据读取的流程是这样的：

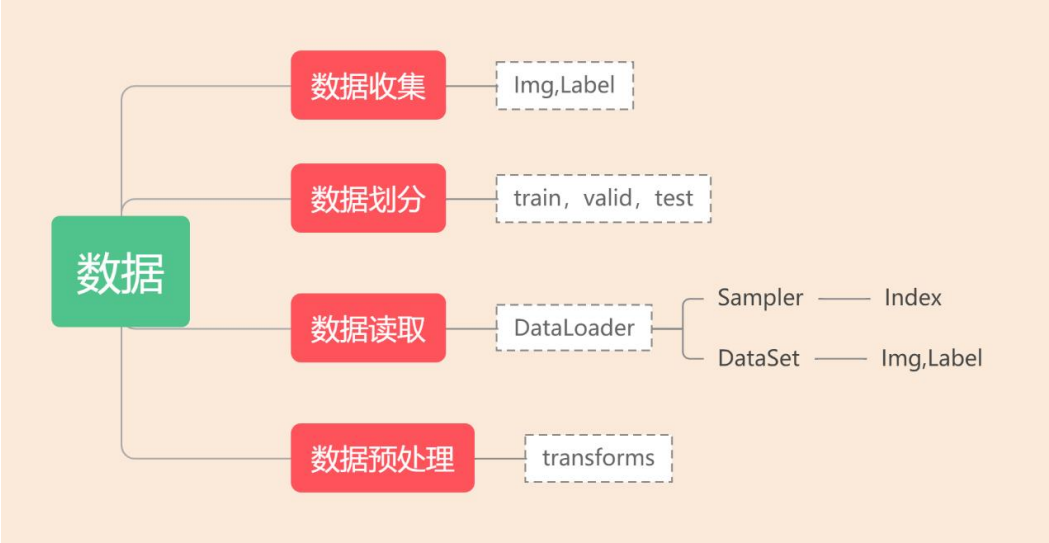
1. 创建 Dateset
2. Dataset 传递给 DataLoader
3. DataLoader 迭代产生训练数据提供给模型



对应的一般都会有这三部分代码

```
#  创建 Dateset(可以自定义)
    dataset = face_dataset # Dataset 部分自定义过的 face_dataset
# Dataset 传递给 DataLoader
    dataloader =
torch.utils.data.DataLoader(dataset,batch_size=64,shuffle=False,num_workers=8)
# DataLoader 迭代产生训练数据提供给模型
    for i in range(epoch):
        for index,(img,label) in enumerate(dataloader):
            pass
```

到这里应该就 PyTorch 的数据集和数据传递机制应该就比较清晰明了了。Dataset 负责建立索引到样本的映射，DataLoader 负责以特定的方式从数据集中迭代的产生一个个 batch 的样本集合。在 enumerate 过程中实际上是 dataloader 按照其参数 sampler 规定的策略调用了其 dataset 的 getitem 方法。其中，还会涉及数据的变化形式。



数据处理

一般来说 PyTorch 中深度学习训练的流程是这样的：

1.收集数据

2.数据预处理

3.将处理好的数据创建 Dateset

4. Dataset 传递给 DataLoader

5. DataLoader 迭代产生训练数据提供给模型

对应的一般都会有这三部分代码

数据收集 (Data Collection)

这是第一步，旨在获取原始数据。数据可以来自多种渠道：

公共数据集 (Public Datasets)：如 MNIST（手写数字）、CIFAR-10/100（物体图像）、ImageNet、COCO（目标检测）等。Nlp 中，如用于情感分析的 IMDB 影评、SST；用于问答的 SQuAD；用于翻译的 WMT 系列；用于语言建模的 WikiText 等。

网络爬虫 (Web Scraping/Crawling)：从互联网上爬取所需的图片等数据。从新闻网站、社交媒体、论坛等抓取文本。

人工标注 (Manual Annotation)：通过公司内部或众包平台（如 Amazon Mechanical Turk）对数据进行标注。

业务数据 (Business Data)：公司业务中搜索查询、产品评论、客服对话记录等。

传感器数据 (Sensor Data)：来自摄像头、麦克风、IoT 设备等。

文档与书籍：公司内部文档、电子书等。

关键点：确保数据的**多样性**和**质量**，并注意版权和隐私问题。

cv 数据预处理与转化 (Data Preprocessing & Transformation)

原始数据通常不能直接输入模型，需要进行一系列转化。这些操作通常使用 transforms 模块来组合完成。

常见转化操作包括：

尺寸调整 (Resize)：将所有图像调整为统一尺寸（如 224x224），以满足模型输入要求。

数据类型转换 (ToTensor)：将 PIL 图像或 NumPy 数组转换为 PyTorch 张量 (Tensor)，并将像素值从 [0, 255] 缩放到 [0.0, 1.0]。

标准化 (Normalization)：对张量进行**减均值、除以标准差**的操作。这使得数据分布以 0 为中心，标准差为 1，可以**加速模型收敛，提高训练稳定性**。常用 ImageNet 的均值 [0.485, 0.456, 0.406] 和标准差 [0.229, 0.224, 0.225]，或者在自己数据集上计算。

数据增强 (Data Augmentation)：**仅用于训练集**，通过对训练数据进行随机变换（如随机裁剪、水平翻转、颜色抖动等）来人工增加数据量和多样性，是防止过拟合、提升模型泛化能力的有效手段。

代码实例：

```
# 定义训练和测试时的数据转化组合# 训练转化：包含数据增强
train_transforms = transforms.Compose([
    transforms.RandomCrop(32, padding=4),          # 随机裁剪（数据增强）
    transforms.RandomHorizontalFlip(),             # 随机水平翻转（数据增强）
    transforms.ToTensor(),                         # 转为 Tensor 并缩放到[0,1]
    transforms.Normalize(                          # 标准化
        mean=[0.4914, 0.4822, 0.4465],             # CIFAR-10 数据集的 RGB 均值
        std=[0.2023, 0.1994, 0.2010]               # CIFAR-10 数据集的 RGB 标准差
```

```
    )])
# 测试转化：不包含数据增强，只需最基本的预处理
test_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.4914, 0.4822, 0.4465],
        std=[0.2023, 0.1994, 0.2010]
    )])
```

3.图像预处理-transforms

3.1 图像标准化

transforms.Normalize(mean,std,inplace)

逐通道的标准化，每个通道先求出平均值和标准差，然后标准化。Inplace 表示是否原地操作。

3.2 图像裁剪

```
train_transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])
```

（1）transforms.CenterCrop(size)

从图片中心截取 size 大小的图片。

（2）transforms.RandomCrop(size,padding,padding_mode)

随机裁剪区域。

（3）transforms.RandomResizedCrop(size,scale,ratio)

随机大小，随机长宽比的裁剪。

3.3 图像旋转

（1）transforms.RandomHorizationalFlip(p)

依据概率 p 水平翻转。

（2）transforms.RandomVerticalFlip(p)

依据概率 p 垂直翻转。

（3）transforms.RandomRotation(degrees,resample,expand)

transforms 方法

Transforms Methods

一、裁剪

1. transforms.CenterCrop

2. transforms.RandomCrop

3. transforms.RandomResizedCrop

4. transforms.FiveCrop

5. transforms.TenCrop

二、翻转和旋转

1. transforms.RandomHorizontalFlip

2. transforms.RandomVerticalFlip

3. transforms.RandomRotation

三、图像变换

• 1. transforms.Pad

• 2. transforms.ColorJitter
• 3. transforms.Grayscale
• 4. transforms.RandomGrayscale
• 5. transforms.RandomAffine
• 6. transforms.LinearTransformation
• 7. transforms.RandomErasing
• 8. transforms.Lambda
• 9. transforms.Resize
• 10. transforms.Totensor
• 11. transforms.Normalize

四、transforms 的操作

• 1. transforms.RandomChoice
• 2. transforms.RandomApply
• 3. transforms.RandomOrder

```
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),

    # 1 CenterCrop
    # transforms.CenterCrop(512),       # 512

    # 2 RandomCrop
    # transforms.RandomCrop(224, padding=16),
    # transforms.RandomCrop(224, padding=(16, 64)),
    # transforms.RandomCrop(224, padding=16, fill=(255, 0, 0)),
    # transforms.RandomCrop(512, pad_if_needed=True),     # pad_if_needed=True
    # transforms.RandomCrop(224, padding=64, padding_mode='edge'),
    # transforms.RandomCrop(224, padding=64, padding_mode='reflect'),
    # transforms.RandomCrop(1024, padding=1024, padding_mode='symmetric'),

    # 3 RandomResizedCrop
    # transforms.RandomResizedCrop(size=224, scale=(0.5, 0.5)),

    # 4 FiveCrop
    # transforms.FiveCrop(112),
    # transforms.Lambda(lambda crops: torch.stack([(transforms.ToTensor()(crop)) for crop in crops])),
```

```
    # 5 TenCrop
    # transforms.TenCrop(112, vertical_flip=False),
    # transforms.Lambda(lambda crops: torch.stack([(transforms.ToTensor()(crop)) for crop in crops])),

    # 1 Horizontal Flip
    # transforms.RandomHorizontalFlip(p=1),

    # 2 Vertical Flip
    # transforms.RandomVerticalFlip(p=0.5),

    # 3 RandomRotation
    # transforms.RandomRotation(90),
    # transforms.RandomRotation((90), expand=True),
    # transforms.RandomRotation(30, center=(0, 0)),
    # transforms.RandomRotation(30, center=(0, 0), expand=True),     # expand only for center rotation

    transforms.ToTensor(),
    transforms.Normalize(norm_mean, norm_std),
])
```

nlp 数据预处理与转化 (Data Preprocessing & Transformation)

这是 NLP 与 CV 最不同的地方。其核心步骤是构建一个 **“词汇表 (Vocabulary)”**，并建立 **文本 -> 数值索引** 的映射。

常见文本预处理操作：

分词 (Tokenization):将句子分割成更小的单元（词、子词或字符）。

构建词汇表 (Building Vocabulary):

创建一个字典，将每个唯一的词（Token）映射到一个唯一的整数索引（index）。

通常会加入一些特殊 Token，如 <UNK>（未知词）、<PAD>（填充符）、<SOS>（序列开始）、<EOS>（序列结束）。

数值化/索引化 (Numericalization/Indexing):

使用构建好的词汇表，将分词后的句子转换成整数索引列表。比如['This', 'movie', 'is', 'great'] -> [12, 345, 7, 29]

填充/截断 (Padding/Truncating):

一个批次内的文本序列必须是等长的，但原始句子长度各不相同。

填充 (Padding): 在短序列的末尾添加 <PAD> 对应的索引，直到达到指定长度。

截断 (Truncating): 将长序列末尾超出指定长度的部分切除。

```
from collections import Counter
#  假设的原始数据
raw_data = [
    ("This movie is great", 1),
    ("I hate this film", 0),
    ("What a fantastic story", 1),
    ("Terrible acting and boring plot", 0),
    # ...  更多数据
    ]
# ---  构建词汇表  ---

class Vocabulary:
    def __init__(self,frequency_threshold):
        self.index2word={0:'PAD',1:'UNK'}
        self.word2index={'PAD':0,'UNK':1}
        self.frequency_threshold=frequency_threshold

    def __len__(self):
        return len(self.index2word)

    def build_vocabulary(self,sentence_list):
        frequencies=Counter()
        idx=2     #  从 2 开始，因为 0 和 1 已经被特殊 Token 占用
        # 1.  统计所有词的出现频率
        for sentence in sentence_list:
            for word in sentence.split(): #  这里用简单空格分词
                frequencies[word]+=1
        # 2.  将频率超过阈值的词加入词汇表
        for word,freq in frequencies.items():
```

```
                if freq>self.frequency_threshold:
                    self.word2index[word]=idx
                    self.index2word[idx]=word
                    idx+=1

    # 将文本转化为数值索引列表
    def numericalize_text(self,text):
        token_text=text.split()
        numericalize_text=[self.word2index[token] if token in self.word2index
                                    else self.word2index['UNK']
                                    for token in token_text]
        return numericalize_text
```

数据划分

数据划分主要是路径的处理。

```
def makedir(new_dir):
    if not os.path.exists(new_dir):
        os.makedirs(new_dir)
```

检测路径是否存在，若不存在，则创建此路径。

```
dataset_dir = os.path.join("..", "..", "data", "RMB_data")
```

设置路径，将它们组合在一起。相对于 Python 文件所在位置的相对路径。

```
    for root, dirs, files in os.walk(dataset_dir):
        for sub_dir in dirs:
            imgs = os.listdir(os.path.join(root, sub_dir))
            imgs = list(filter(lambda x: x.endswith('.jpg'), imgs))
            random.shuffle(imgs)
            img_count = len(imgs)

            train_point = int(img_count * train_pct)
            valid_point = int(img_count * (train_pct + valid_pct))

            for i in range(img_count):
                if i < train_point:
                    out_dir = os.path.join(train_dir, sub_dir)
                elif i < valid_point:
                    out_dir = os.path.join(valid_dir, sub_dir)
                else:
                    out_dir = os.path.join(test_dir, sub_dir)
```

```
            makedir(out_dir)

            target_path = os.path.join(out_dir, imgs[i])
            src_path = os.path.join(dataset_dir, sub_dir, imgs[i])

            shutil.copy(src_path, target_path)
```

```
os.walk
```

每一层遍历：

root 保存的就是当前遍历的文件夹的绝对路径；

dirs 保存当前文件夹下的所有子文件夹的名称（仅一层，孙子文件夹不包括）

files 保存当前文件夹下的所有文件的名称

其次，发现它的遍历文件方式，在图的遍历方式中，那可不就是深度遍历嘛！！

（1） os.listdir() 方法用于返回指定的文件夹包含的文件或文件夹的名字的列表。

shutil.copy()Python 中的方法用于将源文件的内容复制到目标文件或目录。它还会保留文件的权限模式，但不会保留文件的其他元数据(例如文件的创建和修改时间)。源必须代表文件，但目标可以是文件或目录。如果目标是目录，则文件将使用源中的基本文件名复制到目标中。另外，目的地必须是可写的。如果目标是文件并且已经存在，则将其替换为源文件，否则将创建一个新文件。

创建 Dateset

Dataset 的本质

Dataset 的本质是一个继承自 PyTorch 中 torch.utils.data.Dataset **类的抽象接口（Abstract Class）**。是一个包装了数据和对应标签的类，封装了两个特定方法__getitem__(self, index)和__len__(self)，自定义类的时候也需要重写这两个方法。

所谓数据集，其实就是一个负责处理索引(index)到样本(sample)映射的一个类(class)。

内置数据集

（1）CIFAR10 是 CV 训练中经常使用到的一个数据集，在 PyTorch 中 CIFAR10 是一个写好的 Dataset，我们使用时只需以下代码：

data = datasets.CIFAR10("./data/", transform=transform, train=True, download=True)

datasets.CIFAR10 就是一个 Datasets 子类，data 是这个类的一个实例。

这里我们指定数据存储的根目录 './data'，如果目录下没有数据，会自动下载# train=True 下载训练集，train=False 下载测试集

（2）用自己在一个文件夹中的数据作为数据集时可以使用 ImageFolder 这个方便的 API。

FaceDataset = datasets.ImageFolder('./data', transform=img_transform)

自定义创建一个 Dataset？

```
class CustomDataset(Dataset):#需要继 Dataset
    def __init__(self):
        # 1.初始化文件路径等信息
    def __getitem__(self, index):
        # 1. 从文件里读取一个数据和标签，两者可以分别实现
        # 2. 对数据进行预处理.
        # 3. 返回数据和标签的元组

    def __len__(self):
        return length 数据长度可以通过各种方式计算
```

一个自定义的 Dataset 类通常包含以下部分：

属性 (Attributes)：

data: 存储样本数据（如图像路径、图像数据矩阵等）。

labels 或 targets: 存储样本对应的标签。

transform: 存储预处理和数据增强的变换组合。

root_dir: 存储数据所在的根目录路径。

...以及其他任何存储数据信息所需的变量。

核心方法 (Core Methods)：

__init__(self, ...): **构造函数**。在创建数据集对象时调用，用于初始化上述属性。在这里，你通常会读取数据清单（如一个包含图片路径和标签的 CSV 文件），或者直接加载所有数据到内存（如果数据集很小）。

__getitem__(self, index): **最重要的方法**。它定义了如何通过索引（一个整数）来获取一个数据样本（包括数据和标签）。在这个方法里，你会进行读取数据（如从硬盘读图片）、应用 transform、返回 Tensor 格式的数据和标签等操作。

__len__(self): 返回数据集的大小，即样本的总数量。

为什么可以通过索引（如 dataset[i]）来获取样本？

这得益于 Python 的 **特殊方法（Dunder/Magic Methods）** 机制。可以参考迭代器的 iter 和 next。

当你写下 dataset[i] 时，Python 解释器实际上会调用你实现的 dataset.__getitem__(i) 方法。__getitem__ 方法定义了索引操作 [] 的行为。

同样地，当你使用 len(dataset) 时，Python 会调用 dataset.__len__() 方法。

PyTorch 的 DataLoader 在背后就是通过调用这两个方法来高效地、并行地获取数据批次和确定数据总量的。

cv-自定义猫狗分类数据集

场景：假设我们有一个猫狗分类数据集，文件结构如下：

```
data/custom_cats_and_dogs/
├── cat.0.jpg
├── cat.1.jpg
├── ...
├── dog.0.jpg
├── dog.1.jpg
└── ...
```

```
import os
import torch
from torch.utils.data import Dataset
from PIL import Image # 用于读取图片
class CatsAndDogsDataset(Dataset):

    def __init__(self, root_dir):
        """
        参数:
            root_dir (string): 数据集的根目录（例如 'data/custom_cats_and_dogs'）。
```

```
        """
        self.root_dir = root_dir
        # 1. 收集所有图片的路径
        self.image_paths = []
        for file_name in os.listdir(root_dir):
            if file_name.endswith('.jpg'): # 假设都是 jpg 文件
                self.image_paths.append(os.path.join(root_dir, file_name))

        # 2. 根据文件名创建标签：猫为 0，狗为 1
        self.labels = []
        for path in self.image_paths:
            file_name = os.path.basename(path) # 例如 'cat.0.jpg'
            if file_name.startswith('cat'):
                self.labels.append(0)
            elif file_name.startswith('dog'):
                self.labels.append(1)
            else:
                # 如果遇到既不是猫也不是狗的文件，可以跳过或抛出异常
                # 这里我们选择跳过，但在构建 image_paths 列表时就应该过滤掉
                pass

    def __len__(self):
        """返回数据集中的总样本数"""
        return len(self.image_paths)

    def __getitem__(self, index):
        """
        获取一个样本
        参数:
            index (int): 索引
        返回:
            tuple: (image, label) 图像数据和其对应的标签。
        """
        # 1. 根据索引从列表中找到图片路径
        img_path = self.image_paths[index]

        # 2. 读取图片
        # PIL.Image.open 默认是懒加载的，直到我们操作图片数据时才会真正读入文件
        image = Image.open(img_path).convert('RGB') # 确保图片是 RGB 三通道

        # 3. 获取对应的标签
```

```
        label = self.labels[index]


        # 4. 返回 (图像, 标签) 元组
        return image, label

# 1. 实例化数据集对象
custom_dataset = CatsAndDogsDataset(
    root_dir='./data/custom_cats_and_dogs')
# 2. 检查数据集长度
print(f'数据集总共有 {len(custom_dataset)} 个样本。')
# 3. 通过索引直接访问样本
single_image, single_label = custom_dataset[42] # 获取第 43 个样本 print(f‘样本 43 的图像形状: {single_image.shape}，标签: {single_label}’)
```

nlp-电影评论数据集

自然语言处理（NLP）中序列数据的处理流程与图像处理在核心思想（Dataset + DataLoader）上是一致的，但具体的预处理和转化步骤有显著差异，因为它处理的是**离散的符号（文字）** 而非**连续的像素值**。

我们以一个简单的电影评论情感分析（二分类：正面/负面）为例，展示如何自定义一个 NLP Dataset。

假设：我们的数据格式是一个列表，列表中的每个元素是一个元组 (评论文本, 标签)，其中标签 0 代表负面，1 代表正面。

```
# 利用前面的预处理步骤构建词汇表
vocab=Vocabulary(frequency_threshold=1)
all_sentences=[text for text,label in raw_data]
vocab.build_vocabulary(all_sentences)
print(len(vocab))
print(vocab.word2index) # 查看词到索引的映射
# --- 第二步：自定义 NLP Dataset ---

class TextDataset:
    def __init__(self,raw_data,vocab,max_len):
        self.raw_data=raw_data
        self.vocab=vocab
        self.max_len=max_len

    def __len__(self):
```

```
        return len(self.raw_data)

    def __getitem__(self, index):
        text,label=raw_data[index]
        numericalized_text=vocab.numericalize_text(text)
        length=len(numericalized_text)
        if length<self.max_len:
            # 填充
            padded=
numericalized_text+[vocab.word2index['PAD']*(self.max_len-length)]
        else:
            # 截断
            padded=numericalized_text[:self.max_len]
            length=self.max_len
        return torch.tensor(padded),torch.tensor(label,dtype=torch.float32),length
# 实例化数据集
dataset = TextDataset(raw_data, vocab, max_len=10)
```

现代 NLP 的简化流程（使用 Hugging Face Transformers）

在现代 NLP 中，我们通常使用诸如 Hugging Face 的 Transformers 库，它将这些复杂步骤（分词、数值化、填充）全部封装在了一个 Tokenizer 中，极大地简化了工作流。

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from torch.utils.data import Dataset, DataLoader
# 1. 加载预训练模型的 tokenizer
model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
# 2. 自定义 Dataset 变得非常简单
class HFTextDataset(Dataset):
    def __init__(self, raw_data, tokenizer, max_len=128):
        self.raw_data = raw_data
        self.tokenizer = tokenizer
        self.max_len = max_len

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

    def __getitem__(self, index):
        text, label = self.raw_data[index]
```

```
        # Tokenizer 完成所有繁重工作：分词、加特殊 Token([CLS], [SEP])、数值化、填充/截断
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            padding='max_length', # 动态填充通常在 collate_fn 中做，这里也可以‘max_length’
            truncation=True,
            return_attention_mask=True,
            return_tensors='pt', # 返回 PyTorch 张量
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)
        }

hf_dataset = HFTextDataset(raw_data, tokenizer)
hf_dataloader = DataLoader(hf_dataset, batch_size=2, shuffle=True)
for batch in hf_dataloader:
    input_ids = batch['input_ids']
    attention_mask = batch['attention_mask']
    labels = batch['labels']
    print(f"Input IDs shape: {input_ids.shape}")
    # 可以直接将 batch 输入给 Hugging Face 的模型
    # outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
    break
```

总结：NLP 与 CV 数据处理的对比

步骤	计算机视觉 (CV)	自然语言处理 (NLP)
原始数据	图像 (像素矩阵)	文本 (字符串)
预处理	Resize, Crop, Normalize	**分词 (Tokenization)**
数值化	像素值天然是数字	**构建词汇表，将词映射为索引**
统一长度	调整图像尺寸	**填充 (Padding) 或截断 (Truncating)**
核心工	torchvision.transforms	Tokenizer **(自定义或 Hugging Face)**

步骤	计算机视觉 (CV)	自然语言处理 (NLP)
具		
额外信息	-	序列实际长度 (for RNN), Attention Mask (for Transformer)

NLP 数据处理的精髓在于通过 **Tokenizer** 和 **Vocabulary** 搭建起一座连接**人类语言**和**模型数字世界**的桥梁。现代库（如 Hugging Face Transformers）已经将这座桥建造得非常完善，让我们可以更专注于模型和任务本身。

映射 Map 式和迭代式 iterable 数据集对比

Pytorch 提供两种数据集： *Map 式数据集* Iterable 式数据集

Map-style（映射式）和 Iterable-style（迭代式）数据集是 PyTorch 中 Dataset 类的两种基本范式，它们适用于不同的场景，有着显著的区别和一定的联系。

共同接口：它们都继承自 torch.utils.data.Dataset 基类（或 IterableDataset，它是 Dataset 的子类），因此都可以被传递给 torch.utils.data.DataLoader。

共同目标：它们的最终目的都是**向模型提供数据**，DataLoader 会负责从它们那里获取数据并组合成批次。

可转换性：理论上，一个支持随机访问的数据源可以被实现为任一种风格。但根据数据源的特性选择正确的风格至关重要。

核心概念与区别

特性	Map-Style 数据集	Iterable-Style 数据集
核心思想	将数据集看作一个**映射（Map）**，即一个从**索引（index）**到**样本（sample）**的键值对结构。	将数据集看作一个**数据流（Stream）**，即一个可迭代的、可能无限长的序列。
核心方法	必须实现 __getitem__(index) 和 __len__()。	必须实现 __iter__()，返回一个迭代器。**不要求实现** __len__()。
访问方式	**随机访问 (Random Access)**。可以通过任意索引直接获取任何一个样本。	**顺序访问 (Sequential Access)**。只能像遍历链表一样，一个一个地获取下一个样本。
数据源	适合存储在内存或支持随机访问的介质上的数据（如单个文件、内存中的列表、图像文件集）。	适合**流式数据、大规模数据集**（无法全部加载到内存）、**数据库查询结果、实时生成的数据**。

特性	Map-Style 数据集	Iterable-Style 数据集
多进程加载	与 DataLoader 的 num_workers > 0 完美配合。每个工作进程通过索引独立读取数据，高效且随机顺序准确。	在多进程下 (num_workers > 0) 需要特别小心，因为每个工作进程都会独立地迭代整个数据集，可能导致数据重复。需要手动分片（worker_init_fn）。
长度已知	**是**。数据集大小是固定且预先知道的。	**通常未知**。数据可能来自一个无限长的生成器，或者无法预先计算总大小。

代码实例对比

让我们通过一个具体的例子来对比：我们有一个非常大的文本文件 huge_data.txt，每行是一个样本（例如，一句话）。

目标：创建一个 Dataset 来读取这个文件中的样本。

实例 1：Map-Style 实现 (不推荐的做法)

```
from torch.utils.data import Dataset, DataLoader
class MapStyleTextDataset(Dataset):
    def __init__(self, file_path):
        self.file_path = file_path
        # ！！！警告：在初始化时将所有数据读入内存 ！！！
        with open(file_path, 'r', encoding='utf-8') as f:
            # 将每一行都存储在一个列表中
            self.samples = [line.strip() for line in f]

    def __len__(self):
        # 长度就是列表的长度
        return len(self.samples)

    def __getitem__(self, index):
        # 通过索引从内存中的列表随机访问
        return self.samples[index]
```

缺点： __init__ 中的 self.samples = ... 会将整个大文件加载到内存中。如果文件有 100GB，你的程序就会立刻占用 100GB 内存，这通常是不可行的。

适用性：仅当数据集很小，可以完全放入内存时，这种 Map-Style 实现才是高效且方便的。这也是为什么 MNIST、CIFAR-10 等标准数据集通常这样实现。

实例 2：Iterable-Style 实现 (推荐用于大规模文件)

```
from torch.utils.data import IterableDataset, DataLoader
```

```
import torch
class IterableStyleTextDataset(IterableDataset):
    def __init__(self, file_path):
        self.file_path = file_path
        # 这里不预先加载数据，只记录文件路径

    def __iter__(self):
        # 创建一个迭代器
        worker_info = torch.utils.data.get_worker_info()

        # 如果是在单进程模式下 (num_workers=0)，worker_info 是 None
        if worker_info is None:
            # 单进程：直接打开文件从头开始读
            file_iter = open(self.file_path, 'r', encoding='utf-8')
        else:
            # 多进程：需要将文件分片给各个工作进程
            # 假设文件可以被简单地按行划分
            total_workers = worker_info.num_workers
            worker_id = worker_info.id
            # 这是一个简化的分片逻辑，实际可能需要更复杂的逻辑来避免偏斜
            file_iter = open(self.file_path, 'r', encoding='utf-8')
            # 当前 worker 跳过不属于它的行
            for i, line in enumerate(file_iter):
                if i % total_workers == worker_id:
                    yield line.strip()
            file_iter.close()
            return

        # 迭代文件对象，逐行 yield
        for line in file_iter:
            yield line.strip()
        file_iter.close()
```

优点：内存友好。在任何时候，只有当前的一行（或一个批次）数据在内存中。它可以处理远超内存大小的数据集。

挑战：多进程处理：需要手动实现分片逻辑（get_worker_info），确保每个子进程读取数据的不同部分，而不是重复读取全部数据。这是最复杂的一部分。

打乱顺序 (Shuffling)：无法像 Map-Style 那样简单地实现完美的全局随机打乱。常见的做法是使用一个**大型的缓冲池（Shuffle Buffer）**：

顺序读取大量样本到缓冲池，然后从池中随机抽取，抽一个再补充一个。

未知长度：len(iterable_dataset) 会抛出错误，因为你可能无法或不想先扫描一遍整个文件来计数。

实例 3：结合两者优势（使用 __getitem__ 但延迟加载）

对于大型文件，还有一种混合思路：**在 __getitem__ 中执行磁盘读取**。

```
class EfficientMapStyleTextDataset(Dataset):
    def __init__(self, file_path):
        self.file_path = file_path
        # 预先计算并存储每个样本的偏移量（文件指针位置），而不是样本本身
        self.offsets = []
        with open(file_path, 'r', encoding='utf-8') as f:
            offset = 0
            for line in f:
                self.offsets.append(offset)
                offset = f.tell() # 获取当前读取位置

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

    def __getitem__(self, index):
        # 根据存储的偏移量，直接跳到文件的那个位置读取一行
        with open(self.file_path, 'r', encoding='utf-8') as f:
            f.seek(self.offsets[index])
            line = f.readline()
        return line.strip()
```

优点：保持了 Map-Style 的所有优势（随机访问、完美 shuffle、多进程安全），同时避免了将整个数据集加载到内存中。内存中只存储了偏移量列表，通常比原始数据小几个数量级。

缺点：每次 __getitem__ 都要进行**一次磁盘 I/O**。如果磁盘慢（如机械硬盘），并且 num_workers 设得很高，大量的随机寻道可能会成为性能瓶颈。使用 SSD 会好很多。

总结与如何选择

场景	推荐风格	理由

场景	推荐风格	理由
数据集小，能放入内存	**Map-Style**	实现简单，性能最佳，支持完美随机打乱。
数据集巨大，无法放入内存	**Iterable-Style**	唯一选择。内存效率最高。需处理多进程分片和打乱逻辑。
数据集大，但样本在支持随机访问的介质上（如数据库、索引文件）	**Map-Style (延迟加载)**	如上例 3。既能处理大数据，又保留了随机访问的便利性。需要确保底层数据源支持高效随机读取。
数据是实时生成的（如强化学习环境交互）	**Iterable-Style**	数据本质上是无限的、流式的，无法用索引映射。
数据来自网络流（如 Kafka, RabbitMQ）	**Iterable-Style**	典型的数据流场景，只能顺序消费。

简单决策树：

你的数据能轻松放进内存吗？ -> **能：用 Map-Style**。

不能放进内存，但数据在文件中，且你很关心训练时的随机性？ -> **尝试用 Map-Style + 偏移量的方式**。

数据太大，或者根本就是一个无限流/数据库查询？ -> **用 Iterable-Style，并仔细处理多工作者配置**。

数据加载器 (DataLoader)：

DataLoader 是连接 **数据集（Dataset）** 和 **模型（Model）** 的关键桥梁，负责管理如何从数据集中抽取样本并将其组合成模型训练所需的批次（Batch）。

DataLoader **是一个高效的数据加载器，它是一个迭代器（Iterator）。** 你可以将任何一个 Dataset 对象（无论是 Map-style 还是 Iterable-style）传递给它，它会自动帮你：

从 Dataset 中读取数据。

将多个单个样本组合成一个批次（Batching）。

（可选）打乱数据顺序（Shuffling）。

（可选）使用多进程并行加载数据（Multiprocessing）。

它的核心作用是**隐藏数据加载的复杂性，并为训练循环提供一个干净、高效的批量数据流**。

功能	作用描述	为什么重要
批处理 (Batching)	将 n 个样本组合成一个大的张量。例如，将 32 张图片组合成 [32, 3, 224, 224] 的张量。	利用 GPU 的并行计算能力，同时处理多个样本，极大提升训练效率。
打乱顺序 (Shuffling)	在每个 epoch 开始时，重新排列数据的顺序。	防止模型学习到与样本顺序无关的偏差，有助于模型更好地泛化。**仅用于训练集**。
多进程加载 (Multiprocessing)	使用多个子进程（Workers）来预加载数据。	将数据加载（特别是耗时的 CPU 操作，如图像解码、增强）与 GPU 计算**重叠**起来，避免 GPU 等待数据，从而最大限度地提高 GPU 利用率。这是解决 I/O 瓶颈的关键。
自动内存固定 (Pin Memory)	将数据直接加载到页锁定内存（Pinned Memory）中。	enables faster data transfer from CPU to GPU when using CUDA GPUs. 这是 CUDA 的一个优化技巧。

基本使用方法

使用前面创建的 dataset 实例化对象传递给 DataLoader。

```
from torch.utils.data import DataLoader
# 超参数
BATCH_SIZE = 64
NUM_WORKERS = 4   # 用于数据加载的子进程数
# 创建 Training DataLoader
# 训练集需要打乱顺序，并且通常使用多进程
train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,          # 关键参数：打乱顺序
    num_workers=NUM_WORKERS, # 关键参数：多进程数
    pin_memory=True        # 关键参数：锁页内存，GPU 训练时建议开启)
# 创建 Test/Validation DataLoader
# 测试集不需要打乱顺序，num_workers 可以少一些或不使用
test_loader = DataLoader(
    dataset=test_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,         # 测试集不打乱！
    num_workers=2,
    pin_memory=True)
```

DataLoader 是一个可迭代对象，可以直接在 for 循环中使用。

```
# 核心：迭代 DataLoader
for i in range(epoch):
    for batch_idx, (data, target) in enumerate(train_loader):
        Pass
        注意这里迭代的# 每个 batch 包含一个批次的 (data, target)
```

核心参数详解

参数	说明	常用值
dataset	要加载的数据集对象，Dataset 类的实例。-	
batch_size	超参数，每个批次包含的样本数量。	32, 64, 128, 256 (取决于 GPU 内存)
shuffle	是否在每个 epoch 打乱数据顺序。	True (训练集), False (测试/验证集)
num_workers	用于数据加载的子进程数量。	0 (主进程加载), 2, 4, 8 (根据 CPU 核心数调整)
pin_memory	如果为 True，数据加载器会将张量复制到 CUDA 的锁页内存中。	True (使用 GPU 时), False (使用 CPU 时)
drop_last	如果数据集大小不能被 batch_size 整除，是否丢弃最后一个不完整的批次。	False (默认), True (避免最后一个小批次影响训练稳定性)
collate_fn	**高级功能:** 如何将多个样本组合成一个批次的函数。默认为 torch.stack。	用于处理可变长度序列（如文本、音频）的填充（Padding）。

一个 collate_fn 的简单示例

假设我们处理的是不定长的文本序列，我们需要自定义一个函数来对批次内的序列进行填充。

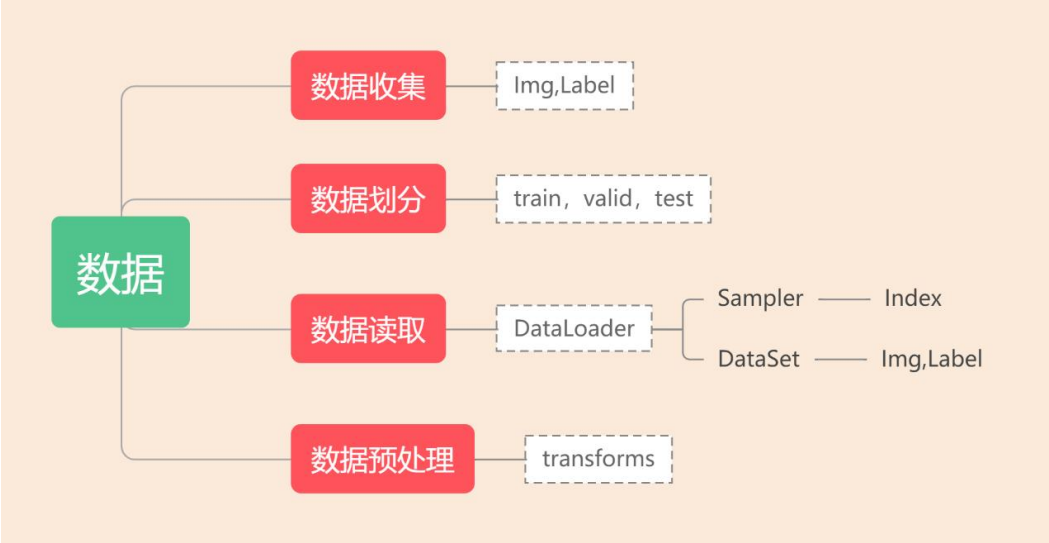
```
from torch.nn.utils.rnn import pad_sequence
def custom_collate_fn(batch):
    # batch 是一个列表，每个元素是 (data, target) 元组，来自 dataset[i]
    # 例如：batch = [(src1, tgt1), (src2, tgt2), ..., (srcN, tgtN)]
    # 将 batch 中的源句子和目标句子分开
    sources, targets = zip(*batch)
    # 对源句子进行填充
    # 假设 sources 是整数索引列表
```

```
        sources_padded = pad_sequence([torch.tensor(s) for s in sources], batch_first=True, padding_value=0)
        # 目标标签可能不需要填充，直接堆叠
        targets = torch.tensor(targets)
        return sources_padded, targets
# 使用自定义的 collate_fn
text_loader = DataLoader(
        dataset=some_text_dataset,
        batch_size=32,
        shuffle=True,
        collate_fn=custom_collate_fn # 使用自定义的批处理函数)
```

总结

到这里应该就 PyTorch 的数据集和数据传递机制应该就比较清晰明了了

Dataset 负责建立索引到样本的映射，DataLoader 负责以特定的方式从数据集中迭代的产生一个个 batch 的样本集合。在 enumerate 过程中实际上是 dataloader 按照其参数 sampler 规定的策略调用了其 dataset 的 getitem 方法。其中，还会涉及数据的变化形式。



参考资料

https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

https://gladdduck.github.io/2022/11/09/笔记-Pytorch 数据集加载/
深度之眼 pytorch 学习课程