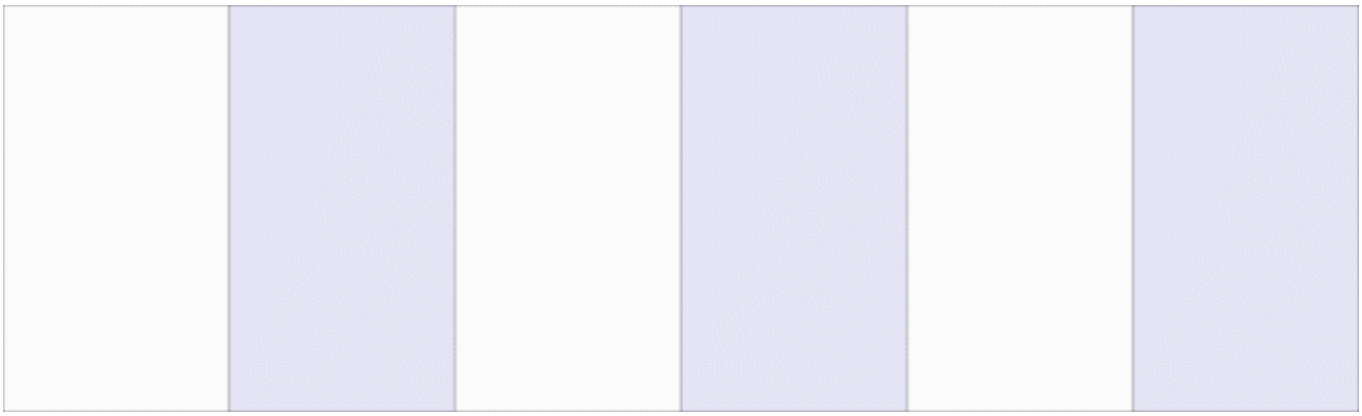
实践教程｜源码级理解Pytorch中的Dataset和DataLoader

CV开发者都爱看的 极市平台 2023-08-13 22:01:12 发表于广东 手机阅读



作者丨梁云1991

来源丨算法美食屋

编辑丨极市平台

极市导读

本文30分钟带你达到对Pytorch中的Dataset和DataLoader的源码级理解，并提供构建数据管道的3种常用方式的范例，扫除你构建数据管道的一切障碍。 >>加入极市CV技术交流群，走在计算机视觉的最前沿

朋友，你还在为构建Pytorch中的数据管道而烦扰吗？你是否有遇到过一些复杂的数据集需要设计自定义collate_fn却不知如何下手的情况？你是否有遇到过数据管道加载速度过慢成为训练性能瓶颈却不知道如何优化的情况？

本篇文章就是你需要的，30分钟带你达到对Pytorch中的Dataset和DataLoader的源码级理解，并提供构建数据管道的3种常用方式的范例，扫除你构建数据管道的一切障碍。

0，Dataset和DataLoader功能简介

Pytorch通常使用Dataset和DataLoader这两个工具类来构建数据管道。

Dataset定义了数据集的内容，它相当于一个类似列表的数据结构，具有确定的长度，能够用索引获取数据集中的元素。

而DataLoader定义了按batch加载数据集的方法，它是一个实现了 `__iter__` 方法的可迭代对象，每次迭代输出一个batch的数据。

DataLoader能够控制batch的大小，batch中元素的采样方法，以及将batch结果整理成模型所需输入形式的方法，并且能够使用多进程读取数据。

在绝大部分情况下，用户只需实现Dataset的 `__len__` 方法和 `__getitem__` 方法，就可以轻松构建自己的数据集，并用默认数据管道进行加载。

对于一些复杂的数据集，用户可能还要自己设计 DataLoader中的 collate_fn方法以便将获取的一个批次的数据整理成模型需要的输入形式。

一，深入理解Dataset和DataLoader原理

1，获取一个batch数据的步骤

让我们考虑一下从一个数据集中获取一个batch的数据需要哪些步骤。

(假定数据集的特征和标签分别表示为张量 `X` 和 `Y` ，数据集可以表示为 `(X,Y)` ，假定batch大小为 `m`)

1，首先我们要确定数据集的长度 `n` 。结果类似： `n = 1000` 。

2，然后我们从 `0` 到 `n-1` 的范围中抽样出 `m` 个数(batch大小)。假定 `m=4` ，拿到的结果是一个列表，类似： `indices = [1,4,8,9]`

3，接着我们从数据集中去取这 `m` 个数对应下标的元素。拿到的结果是一个元组列表，类似： `samples = [(X[1],Y[1]),(X[4],Y[4]),(X[8],Y[8]),(X[9],Y[9])]`

4，最后我们将结果整理成两个张量作为输出。

拿到的结果是两个张量，类似 `batch = (features,labels)` ，

其中 `features = torch.stack([X[1],X[4],X[8],X[9]])`

`labels = torch.stack([Y[1],Y[4],Y[8],Y[9]])`

2，Dataset和DataLoader的功能分工

上述第1个步骤确定数据集的长度是由 Dataset的 `__len__` 方法实现的。

第2个步骤从 0 到 n-1 的范围中抽样出 m 个数的方法是由 DataLoader的 sampler 和 batch_sampler 参数指定的。

sampler 参数指定单个元素抽样方法，一般无需用户设置，程序默认在DataLoader的参数 shuffle=True 时采用随机抽样， shuffle=False 时采用顺序抽样。

batch_sampler 参数将多个抽样的元素整理成一个列表，一般无需用户设置，默认方法在DataLoader的参数 drop_last=True 时会丢弃数据集最后一个长度不能被batch大小整除的批次，在 drop_last=False 时保留最后一个批次。

第3个步骤的核心逻辑根据下标取数据集中的元素 是由 Dataset的 __getitem__ 方法实现的。

第4个步骤的逻辑由DataLoader的参数 collate_fn 指定。一般情况下也无需用户设置。

Dataset和DataLoader的一般使用方式如下：

```
import torch
from torch.utils.data import TensorDataset,Dataset,DataLoader
from torch.utils.data import RandomSampler,BatchSampler


ds = TensorDataset(torch.randn(1000,3),
                   torch.randint(low=0,high=2,size=(1000,)).float())
dl = DataLoader(ds,batch_size=4,drop_last = False)
features,labels = next(iter(dl))
print("features = ",features )
print("labels = ",labels )
```

将DataLoader内部调用方式步骤拆解如下：

```
# step1: 确定数据集长度 (Dataset的 __len__ 方法实现)
ds = TensorDataset(torch.randn(1000,3),
                   torch.randint(low=0,high=2,size=(1000,)).float())
print("n = ", len(ds)) # len(ds)等价于 ds.__len__()

# step2: 确定抽样indices (DataLoader中的 Sampler和BatchSampler实现)
sampler = RandomSampler(data_source = ds)
batch_sampler = BatchSampler(sampler = sampler,
                             batch_size = 4, drop_last = False)
for idxs in batch_sampler:
```

```
        indices = idxs
        break
    print("indices = ",indices)

    # step3: 取出一批样本batch (Dataset的 __getitem__ 方法实现)
    batch = [ds[i] for i in  indices]  #  ds[i] 等价于 ds.__getitem__(i)
    print("batch = ", batch)

    # step4: 整理成features和labels (DataLoader 的 collate_fn 方法实现)
    def collate_fn(batch):
        features = torch.stack([sample[0] for sample in batch])
        labels = torch.stack([sample[1] for sample in batch])
        return features,labels

    features,labels = collate_fn(batch)
    print("features = ",features)
    print("labels = ",labels)
```

3，Dataset和DataLoader的核心源码

以下是 Dataset和 DataLoader的核心源码，省略了为了提升性能而引入的诸如多进程读取数据相关的代码。

```
import torch
class Dataset(object):
    def __init__(self):
        pass

    def __len__(self):
        raise NotImplementedError

    def __getitem__(self,index):
        raise NotImplementedError


class DataLoader(object):
    def __init__(self,dataset,batch_size,collate_fn = None,shuffle = True,drop_last = Fals
        self.dataset = dataset
        self.sampler =torch.utils.data.RandomSampler if shuffle else \
           torch.utils.data.SequentialSampler
        self.batch_sampler = torch.utils.data.BatchSampler
        self.sample_iter = self.batch_sampler(
            self.sampler(self.dataset),
            batch_size = batch_size,drop_last = drop_last)
        self.collate_fn = collate_fn if collate_fn is not None else \
            torch.utils.data._utils.collate.default_collate
```

```
    def __next__(self):
        indices = next(iter(self.sample_iter))
        batch = self.collate_fn([self.dataset[i] for i in indices])
        return batch

    def __iter__(self):
        return self
```

我们来测试一番

```
class ToyDataset(Dataset):
    def __init__(self,X,Y):
        self.X = X
        self.Y = Y
    def __len__(self):
        return len(self.X)
    def __getitem__(self,index):
        return self.X[index],self.Y[index]

X,Y = torch.randn(1000,3),torch.randint(low=0,high=2,size=(1000,)).float()
ds = ToyDataset(X,Y)

dl = DataLoader(ds,batch_size=4,drop_last = False)
features,labels = next(iter(dl))
print("features = ",features )
print("labels = ",labels )
```

完美, 和预期一致!

二，使用Dataset创建数据集

Dataset创建数据集常用的方法有：

- 使用 torch.utils.data.TensorDataset 根据Tensor创建数据集(numpy的array，Pandas的DataFrame需要先转换成Tensor)。
- 使用 torchvision.datasets.ImageFolder 根据图片目录创建图片数据集。
- 继承 torch.utils.data.Dataset 创建自定义数据集。

此外，还可以通过

- torch.utils.data.random_split 将一个数据集分割成多份，常用于分割训练集，验证集和测试集。
- 调用Dataset的加法运算符(+)将多个数据集合并成一个数据集。

1，根据Tensor创建数据集

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


# 根据Tensor创建数据集

from sklearn import datasets
iris = datasets.load_iris()
ds_iris = TensorDataset(torch.tensor(iris.data),torch.tensor(iris.target))

# 分割成训练集和预测集
n_train = int(len(ds_iris)*0.8)
n_val = len(ds_iris) - n_train
ds_train,ds_val = random_split(ds_iris,[n_train,n_val])

print(type(ds_iris))
print(type(ds_train))


# 使用DataLoader加载数据集
dl_train,dl_val = DataLoader(ds_train,batch_size = 8),DataLoader(ds_val,batch_size = 8)

for features,labels in dl_train:
    print(features,labels)
    break


# 演示加法运算符（`+`）的合并作用

ds_data = ds_train + ds_val

print('len(ds_train) = ',len(ds_train))
print('len(ds_valid) = ',len(ds_val))
print('len(ds_train+ds_valid) = ',len(ds_data))

print(type(ds_data))
```

2，根据图片目录创建图片数据集

```
import numpy as np
import torch
from torch.utils.data import DataLoader
from torchvision import transforms,datasets


#演示一些常用的图片增强操作


from PIL import Image
img = Image.open('./data/cat.jpeg')
img


# 随机数值翻转
transforms.RandomVerticalFlip()(img)


#随机旋转
transforms.RandomRotation(45)(img)


# 定义图片增强操作

transform_train = transforms.Compose([
   transforms.RandomHorizontalFlip(), #随机水平翻转
   transforms.RandomVerticalFlip(), #随机垂直翻转
   transforms.RandomRotation(45),  #随机在45度角度内旋转
   transforms.ToTensor() #转换成张量
  ]
)

transform_valid = transforms.Compose([
    transforms.ToTensor()
  ]
)


# 根据图片目录创建数据集

def transform_label(x):
    return torch.tensor([x]).float()

ds_train = datasets.ImageFolder("./eat_pytorch_datasets/cifar2/train/",
            transform = transform_train,target_transform= transform_label)
```

```
ds_val = datasets.ImageFolder("./eat_pytorch_datasets/cifar2/test/",
                              transform = transform_valid,
                              target_transform= transform_label)


print(ds_train.class_to_idx)

# 使用DataLoader加载数据集

dl_train = DataLoader(ds_train,batch_size = 50,shuffle = True)
dl_val = DataLoader(ds_val,batch_size = 50,shuffle = True)


for features,labels in dl_train:
    print(features.shape)
    print(labels.shape)
    break
```

3，创建自定义数据集

下面我们通过另外一种方式，即继承 torch.utils.data.Dataset 创建自定义数据集的方式来对 cifar2构建 数据管道。

```
from pathlib import Path
from PIL import Image

class Cifar2Dataset(Dataset):
    def __init__(self,imgs_dir,img_transform):
        self.files = list(Path(imgs_dir).rglob("*.jpg"))
        self.transform = img_transform

    def __len__(self,):
        return len(self.files)

    def __getitem__(self,i):
        file_i = str(self.files[i])
        img = Image.open(file_i)
        tensor = self.transform(img)
        label = torch.tensor([1.0]) if  "1_automobile" in file_i else torch.tensor([0.0])
        return tensor,label


train_dir = "./eat_pytorch_datasets/cifar2/train/"
test_dir = "./eat_pytorch_datasets/cifar2/test/"
```

```
# 定义图片增强
transform_train = transforms.Compose([
   transforms.RandomHorizontalFlip(), #随机水平翻转
   transforms.RandomVerticalFlip(), #随机垂直翻转
   transforms.RandomRotation(45),  #随机在45度角度内旋转
   transforms.ToTensor() #转换成张量
  ]
)

transform_val = transforms.Compose([
    transforms.ToTensor()
  ]
)


ds_train = Cifar2Dataset(train_dir,transform_train)
ds_val = Cifar2Dataset(test_dir,transform_val)


dl_train = DataLoader(ds_train,batch_size = 50,shuffle = True)
dl_val = DataLoader(ds_val,batch_size = 50,shuffle = True)


for features,labels in dl_train:
    print(features.shape)
    print(labels.shape)
    break
```

三，使用DataLoader加载数据集

DataLoader能够控制batch的大小，batch中元素的采样方法，以及将batch结果整理成模型所需输入形式的方法，并且能够使用多进程读取数据。

DataLoader的函数签名如下。

```
DataLoader(
    dataset,
    batch_size=1,
    shuffle=False,
    sampler=None,
    batch_sampler=None,
    num_workers=0,
    collate_fn=None,
    pin_memory=False,
    drop_last=False,
    timeout=0,
```

```
    worker_init_fn=None,
    multiprocessing_context=None,
)
```

一般情况下，我们仅仅会配置 dataset, batch_size, shuffle, num_workers,pin_memory, drop_last这六个参数，

有时候对于一些复杂结构的数据集，还需要自定义collate_fn函数，其他参数一般使用默认值即可。

DataLoader除了可以加载我们前面讲的 torch.utils.data.Dataset 外，还能够加载另外一种数据集 torch.utils.data.IterableDataset。

和Dataset数据集相当于一种列表结构不同，IterableDataset相当于一种迭代器结构。它更加复杂，一般较少使用。

- dataset：数据集
- batch_size: 批次大小
- shuffle: 是否乱序
- sampler: 样本采样函数，一般无需设置。
- batch_sampler: 批次采样函数，一般无需设置。
- num_workers: 使用多进程读取数据，设置的进程数。
- collate_fn: 整理一个批次数据的函数。
- pin_memory: 是否设置为锁业内存。默认为False，锁业内存不会使用虚拟内存(硬盘)，从锁业内存拷贝到GPU上速度会更快。
- drop_last: 是否丢弃最后一个样本数量不足batch_size批次数据。
- timeout: 加载一个数据批次的最长等待时间，一般无需设置。
- worker_init_fn: 每个worker中dataset的初始化函数，常用于 IterableDataset。一般不使用。

```
#构建输入数据管道
ds = TensorDataset(torch.arange(1,50))
dl = DataLoader(ds,
                batch_size = 10,
                shuffle= True,
                num_workers=2,
                drop_last = True)
#迭代数据
```

```
for batch, in dl:
    print(batch)
```

```
tensor([43, 44, 21, 36,  9,  5, 28, 16, 20, 14])
tensor([23, 49, 35, 38,  2, 34, 45, 18, 15, 40])
tensor([26,  6, 27, 39,  8,  4, 24, 19, 32, 17])
tensor([ 1, 29, 11, 47, 12, 22, 48, 42, 10,  7])
```

点击阅读原文进入CV社区

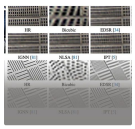
收获更多技术干货

阅读原文

喜欢此内容的人还喜欢

ICCV 2023｜南开程明明团队提出适用于SR任务的新颖注意力机制（已开源）

极市平台



实践教程｜使用 OpenCV 进行特征提取（颜色、形状和纹理）

极市平台

YOLOv5帮助母猪产仔？南京农业大学研发母猪产仔检测模型并部署到Jetson Nano开发板

极市平台