PyTorch的Dataset与DataLoader详细使用教程

2022.07.19

- 学习如何构建训练数据集以及如何将多个训练样本构建成Mini-Batch

1. 总述

- DataSet是针对单个样本而言的，如何从磁盘中将数据映射为X和Y的形式，即提取到数据特征部分和标签部分
- DataLoder是针对多个样本而言的，获取单个样本之后，可以组合成Mini-Batch，或者将数据保存在GPU中

2. 使用

2.1 DataSet简单示例（从已有模块中导入数据）

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

2.2 如何构建自己的DataSet，以及构建Mini-Batch（一般化）

2.2.1 获取单个样本的示例

- 一个自定义的DataSet必须要实现这三个方法：__ init __(参数初始化)， __ len __ （长度）， __ __ getitem __ (从磁盘中读取数据，然后根据索引能够返回对应的样本)

```
import os
import pandas as pd
from torchvision.io import read_image
```

```
class CustomImageDataset(Dataset):
  def __init__(self, annotations_file, img_dir, transform=None, target_transform=None): # annotations_file：标注文件，可能是一个csv，img_dir：自己的数据保存在哪里，transform：对数据特征或标签做变换（自己定义transform函数）
    self.img_labels = pd.read_csv(annotations_file)
    self.img_dir = img_dir
    self.transform = transform
    self.target_transform = target_transform

  def __len__(self):
    return len(self.img_labels) # 返回数据长度

  def __getitem__(self, idx): # idx表示索引
    img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
    image = read_image(img_path)
  label = self.img_labels.iloc[idx, 1]
    if self.transform:
     image = self.transform(image) # 对原始数据进行处理
  if self.target_transform:
      label = self.target_transform(label) # 对label进行处理
  return image, label # 返回数据、标签
```

对transform和target_transform函数说明：当我们从磁盘中的DataSet读取数据和特征之后，可能还无法直接喂入模型之中，比如图片，需要对图片的大小、通道或像素值有一定的约束，所以一般会传入一个自定义的transform和target_transform函数，然后再get_item方法中调用，返回可以喂入模型中的数据和特征

2.2 为什么要组成Mini-Batch

DataSet检索数据集的特征，并一次标记单个样本。在训练模型时，我们通常希望在“minibatches”中传递样本，在每个epoch重新shuffle数据以减少模型过度拟合，并使用Python的多处理来加速数据检索

2.2 DataLoder类源码参数说明

```
def __init__(self, dataset: Dataset[T_co], batch_size: Optional[int] = 1,
             shuffle: Optional[bool] = None, sampler: Union[Sampler, Iterable, None] = None,
             batch_sampler: Union[Sampler[Sequence], Iterable[Sequence], None] = None,
             num_workers: int = 0, collate_fn: Optional[_collate_fn_t] = None,
             pin_memory: bool = False, drop_last: bool = False,
             timeout: float = 0, worker_init_fn: Optional[_worker_init_fn_t] = None,
             multiprocessing_context=None, generator=None,
             *, prefetch_factor: int = 2,
             persistent_workers: bool = False,
             pin_memory_device: str = ""):
    torch._C._log_api_usage_once("python.data_loader")
```

dataset：创建DataSet实例化对象，再传入

batch_size：一个batch中有多少个样本

shuffle：在每个epoch之后是否要打乱数据

sampler：如何对数据进行采样，可以默认也可以自己实现

num_workers：默认为0，即只用主进程加载数据

pin_memory：可以把数据保存在GPU中，就不需要每次都保存

drop_last：是否把最后一个batch丢掉

collate：对所采样的数据再处理，输入是一个batch，输出也是batch，不过可能经过了处理，如padding，甚至更可能的操作，这些操作都放在collate_fn中

2.4 DataLoder示例

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

```
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

我们已将该DataSet加载到 DataLoader 中，并可以根据需要循环访问该数据集。下面的每个迭代都会返回一批train_features和train_labels（分别包含 batch_size=64 个特征和标签）。由于我们指定了shuffle=True，因此在循环访问所有批次后，数据将被随机排列（为了更细粒度地控制数据加载顺序，请查看 Samplers）