基于逻辑回归的手写数字识别 —— Kaggle实现

- 数据获取
- 数据处理
- 模型构建
- 训练过程

汇报人：程征

数据集获取

```python
import torchvision
import torchvision.transforms as transforms  # 处理数据模块

mnist = torchvision.datasets.MNIST(root=''  # 若本地已下载好手写数字数据集，可以直接在此输入本地路径地址
                                   ,train=True  # 是否下载训练集，若为False，将下载测试集数据
                                   ,download=True  # 是否自动下载
                                   ,transform=transforms.ToTensor()  # 简单来说就是把图像数据转换成tensor
                                  )
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to MNIST/raw/train-images-idx3-ubyte.gz
9913344/? [00:00<00:00, 41430091.74it/s]
Extracting MNIST/raw/train-images-idx3-ubyte.gz to MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to MNIST/raw/train-labels-idx1-ubyte.gz
29696/? [00:00<00:00, 986488.61it/s]
Extracting MNIST/raw/train-labels-idx1-ubyte.gz to MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to MNIST/raw/t10k-images-idx3-ubyte.gz
1649664/? [00:00<00:00, 25508543.09it/s]
Extracting MNIST/raw/t10k-images-idx3-ubyte.gz to MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to MNIST/raw/t10k-labels-idx1-ubyte.gz
5120/? [00:00<00:00, 191239.32it/s]
Extracting MNIST/raw/t10k-labels-idx1-ubyte.gz to MNIST/raw
```

```python
mnist  # 数据集说明
```

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

```python
# 查看数据集数量
len(mnist)
```

```
60000
```

数据集获取

探索性数据分析

```
mnist.data[0]
```

```
tensor([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
           0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,  18,
          18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170, 253,
         253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253, 253,
         253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253, 253,
```

```
#查看特征张量形状
mnist.data.shape

# 查看特征张量类型
mnist.data.dtype
```

```
torch.Size([60000, 28, 28])

torch.uint8
```

十进制数据类型，范围在0-255之间

```
# 查看标签
mnist.targets

# 查看标签张量类型
mnist.targets.dtype

# 查看标签类别
mnist.targets.unique()
```

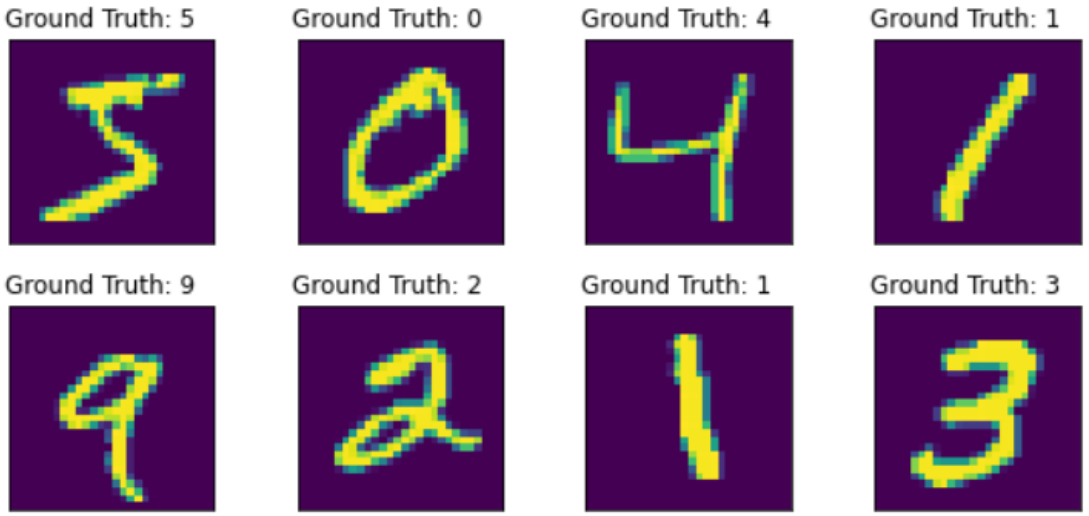
```
tensor([5, 0, 4,  ..., 5, 6, 8])

torch.int64

tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

此时，我们可以看到数据集张量形状为[60000,28,28]，其中60000表示数据集的个数，28和28分别代表图片像素的高度和宽度，然而神经网络无法识别此shape的张量，需要进行调整。另外，关于图像样本真实数据形状应为[sample_size, H -height, W -width, C -color]，C表述颜色通道数（红绿蓝），此处C为1，因此省略了也不会改变数据的结构分布。

数据集获取

探索性数据分析

```
# 查看图像
fig = plt.figure(figsize=(8,4))  # 设置画布大小
for i in range(0,8):
    plt.subplot(2,4,i+1)  # 设置子图 2行4列第i个
    plt.tight_layout()  # 自动调整布局
    # 绘制图像，由于imshow()方法仅支持ndarray格式的数据，所以要进行数据类型转换
    plt.imshow(mnist[i][0].view((28, 28)).numpy())
    plt.title("Ground Truth: {}   ".format(mnist.targets[i])) # 设置标题
    plt.xticks([])  # 设置x轴刻度不显示
    plt.yticks([])  # 设置y轴刻度不显示
plt.show()
```




```
# torch.view() 用于改变tensor的形状
mnist[0][0].shape
# 方式一
mnist[0][0].view(28,28).shape
# 方式二
mnist[0][0].view(-1,28).shape
```

```
torch.Size([1, 28, 28])
```

```
torch.Size([28, 28])
```

```
torch.Size([28, 28])
```

```
# .numpy() 将tensor转换为ndarray数据类型
type(mnist[0][0].view(28,28))
type(mnist[0][0].view(28,28).numpy())
```

```
torch.Tensor
```

```
numpy.ndarray
```

数据处理 —— 构建MnistDataset类

为了方便Pytorch进行数据调用，我们需要在Dataset下面创建一个用于表示Mnist数据集的子类。在创建Dataset的子类过程中，必须要重写getitem方法和len方法，其中getitem方法返回输入索引后对应的特征和标签，而len方法则返回数据集的总数据个数。当然，在必须要进行的init初始化过程中，我们也可输入可代表数据集基本属性的相关内容，包括数据集的特征、标签、大小等。

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

class MnistDataset(Dataset):
    def __init__(self, data):
        self.features = data.data  # features属性返回数据集特征
        self.labels = data.targets  # labels属性返回数据集标签
        self.lens = len(data.data)  # lens属性返回数据集大小
    def __getitem__(self, index):
        # 调用该方法时需要输入index数值，方法最终返回index对应的特征和标签
        return (self.features[index,:], self.labels[index])
    def __len__(self):
        # 调用该方法不需要输入额外参数，方法最终返回数据集大小
        return self.lens
```

```python
mnist_data = MnistDataset(mnist)  # 类的实例化
mnist_data.features.shape  # 查看特征的形状
mnist_data.labels  # 查看标签
mnist_data.lens  # 查看数据集大小
```

```
torch.Size([60000, 28, 28])

tensor([5, 0, 4,  ..., 5, 6, 8])

60000
```

数据处理 —— 划分训练集测试集

```python
from torch.utils.data import random_split  # 数据集切分函数

# 确定训练集、测试集大小，此处以8：2划分训练集和测试集
num_train = int(mnist_data.lens * 0.8)
num_test = mnist_data.lens - num_train
num_train, num_test  # (48000, 12000)

mnist_train, mnist_test = random_split(mnist_data, [num_train, num_test])
```

此时切分的结果是一个映射式的对象，包含dataset和indices两个属性，其中dataset属性用于查看原数据集对象，indices属性用于查看切分后数据集的每一条数据的索引号。

```python
mnist_train.dataset
mnist_data
mnist_train.dataset == mnist_data
```

```
<__main__.MnistDataset at 0x7fa0256ff050>

<__main__.MnistDataset at 0x7fa0256ff050>

True
```

```python
mnist_train.indices
```

```
[7107,
 20904,
 43486,
 21104,
 39950,
 55275,
 11249,
 10313,
 11300,
 13605,
 19578,
```

这里我们可以发现mnist_train.dataset和mnist_data指向同一个地址，说明它们之间存在是映射关系，而不是重新创建了一个对象，这样做的好处在于当我们在处理大规模数据时，通过映射的方法进行数据集的划分可大幅减少内存空间的占用，提高内存利用效率。

数据处理 —— 创建用于处理批数据的方法

```python
def collate_fn(examples):
    '''数据整理函数,对一个批次的数据进行处理'''
    # 将特征转换成32位浮点型数据
    inputs = torch.stack([ex[0].float() for ex in examples])
    # 将特征转换成64位整型数据
    # 后面进行模型迭代时，若使用多分类交叉熵损失函数，则要求输入标签数据必须为torch.long类型数据
    targets = torch.tensor([ex[1] for ex in examples],dtype=torch.long)
    return inputs, targets
```

```python
# torch.stack() 堆叠函数，将参与堆叠的对象放在一个更高维度的张量里，参与堆叠的张量形状必须完全相同
a = torch.tensor([1,2,3], dtype=torch.float)
b = torch.tensor([4,5,6], dtype=torch.float)

a
b
torch.stack([a,b])
torch.stack([a,b]).shape
```

```
tensor([1., 2., 3.])

tensor([4., 5., 6.])

tensor([[1., 2., 3.],
        [4., 5., 6.]])
torch.Size([2, 3])
```

这一步在此数据集上非必须，此处是为了演示统一对一个批次的数据进行处理的方法。

数据处理 —— 封装数据

```python
from torch.utils.data import DataLoader  # 装载数据，使其可迭代

train_loader = DataLoader(mnist_train,  # 要进行封装的数据
                          batch_size=128,  # 批次大小，一次输入到神经网络中的数据量
                          shuffle=True,  # 是否打乱数据，一般训练集上进行数据打乱操作，测试集上则不需要
                          collate_fn=collate_fn  # 数据处理函数，对一个批次的数据进行处理
                         )
test_loader = DataLoader(mnist_test, batch_size=128, shuffle=False, collate_fn=collate_fn)

len(train_loader), len(test_loader)
```

```
(375, 94)
```

```python
train_loader.dataset
mnist_train
train_loader.dataset == mnist_train
```

这里其实也只是进行了数据对象的映射

```
<torch.utils.data.dataset.Subset at 0x7fa025568790>

<torch.utils.data.dataset.Subset at 0x7fa025568790>

True
```

数据处理 —— 注意事项

有很多博客在介绍PyTorch深度学习建模过程中的数据集划分时，会使用scikit-learn中的train_test_split函数。该函数是可以非常便捷的完成数据集切分，但这种做法只能用于单机运行的数据，并且切分之后还要调用Dataset、DataLoader模块进行数据封装和加载，切分过程看似简单，但其实会额外占用非常多的存储空间和计算资源，当进行超大规模数据训练时，所造成的影响会非常明显（当然，也有可能由于数据规模过大，本地无法运行）。

因此，为了更好的适应深度学习真实应用场景，在使用包括数据切分等常用函数时，函数使用优先级是：Pytorch原生函数和类 > 依据张量及其常用方法手动创建的函数 > Scikit-Learn函数

模型构建

线性层的创建 nn.Linear

[20]:
```
import torch
from torch import nn

torch.manual_seed(10)

# 创建一个线性层
linear = nn.Linear(in_features=2, out_features=1)  #输入特征为2，输出特征为1
linear.weight  # w 随机生成
linear.bias  # b 偏置项
```

[20]: `<torch._C.Generator at 0x7f5bddd8e410>`

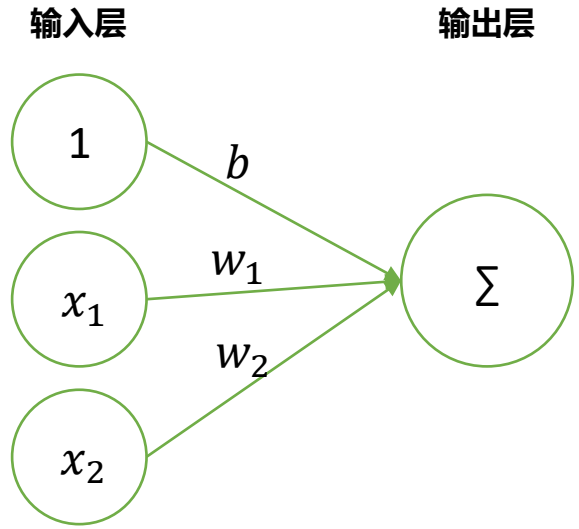
[20]:
```
Parameter containing:
tensor([[-0.0593, -0.0242]], requires_grad=True)
```

[20]:
```
Parameter containing:
tensor([-0.2652], requires_grad=True)
```

[21]:
```
#这里应注意，pytorch计算是按列计算的，此过程由pytorch内部自动转置完成
linear.weight.shape
```

[21]: `torch.Size([1, 2])`

创建一个输入特征为2，输出特征为1的线性层相当于构建了一个$\hat{z} = w_1x_1 + w_2x_2 + b$映射关系



模型构建

线性层的创建 nn.Linear

当我们的输入数据为

x_1	x_2
0	0
1	0
0	1
1	1

其计算过程便为：

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + b = \hat{z}$$

维度变换过程：$(4, 2) \times (2, 1) \longrightarrow (4, 1)$

Pytorch实现

[30]:
```
# 创建一个输入特征为2，输出特征为1的线性层
linear = nn.Linear(in_features=2, out_features=1)
x = torch.tensor([[0,0],[1,0],[0,1],[1,1]], dtype=torch.float)
x

output = linear(x)
output
```

[30]:
```
tensor([[0., 0.],
        [1., 0.],
        [0., 1.],
        [1., 1.]])
```

[30]:
```
tensor([[-0.1247],
        [ 0.0380],
        [-0.5292],
        [-0.3666]], grad_fn=<AddmmBackward0>)
```

模型构建

逻辑回归

```python
class LogisticRegression(nn.Module):
    def __init__(self, in_features=300, num_class=10):
        # 查找个这个类的父类，并调用父类的初始化方法
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(in_features, num_class)  # 隐含层的第一层
        self.activate = torch.sigmoid  # sigmoid激活函数，实现非线性变换

    def forward(self, inputs):
        inputs = inputs.view(-1, 28*28)  # 维度变换，转换成Pytorch可以处理的数据类型
        linear_out = self.linear(inputs)
        outputs = self.activate(linear_out)
        return outputs
```

Sigmoid函数：

$$\sigma = \frac{1}{1 + e^{-z}}$$

训练过程

定义超参数

```
# 输入特征个数 .numel() 统计张量元素个数,这里将后两维作为特征数目，即图片像素的高度和宽度
in_features = mnist.data[0].numel()
mnist.data.shape
mnist.data[0].shape
mnist.data[0].numel()
```

[39]: torch.Size([60000, 28, 28])

[39]: torch.Size([28, 28])

[39]: 784

[40]:
```
# 输出特征个数,即标签类别数
num_class = len(mnist.targets.unique())
num_class
```

[40]: 10

[46]:
```
batch_size = 128  # 批次大小
learning_rate = 0.001  # 学习率，可以简单理解为模型迭代的速率，详细了解查阅梯度下降原理即可
num_epochs = 10  # 训练数据被使用次数，一个epoch表示优化算法将全部数据都是用了一次
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')  # 当GPU可用时，使用GPU；否则使用CPU
```

训练过程

将超参数以字典形式存放

```
#将超参数以字典形式存放
config = {'in_features':mnist.data[0].numel(),  # 输入特征个数
          'num_class':len(mnist.targets.unique()), # 输出特征个数
          'batch_size':128,
          'learning_rate':0.001,
          'num_epochs':10,
          'device': torch.device('cuda' if torch.cuda.is_available() else 'cpu')
         }
```

模型实例化、定义损失函数及优化方法

```
from torch import optim

# 模型实例化
model = LogisticRegression(in_features=config['in_features'], num_class=config['num_class'])
# 将模型加载至GPU
model.to(config['device'])
# 损失函数
creterion = nn.CrossEntropyLoss()
# 优化方法
optimizer = optim.SGD(model.parameters(), lr=config['learning_rate'])  # 小批次随机梯度下降
# 查看是否切换至GPU
config['device']
```

```
[15]: LogisticRegression(
        (linear): Linear(in_features=784, out_features=10, bias=True)
      )
[15]: device(type='cuda')
```

训练过程

开始迭代

```python
# 开始训练
for epoch in range(config['num_epochs']):
    total_loss = 0
    total_acc = 0
    for batch in tqdm(train_loader, desc=f'Training Epoch {epoch+1}'):
        # 将数据加载至GPU
        inputs, targets = [x.to(config['device']) for x in batch]
        # 将特征带入到模型
        probs = model(inputs)
        # 计算损失
        loss = creterion(probs, targets)
        optimizer.zero_grad()  # 梯度清零
        loss.backward()  # 反向传播
        optimizer.step()  # 更新参数
        acc = (probs.argmax(dim=1) == targets).sum().item()  # item()用于在只包含一个元素的tensor中提取值
        total_acc += acc  # 最终得到整个epoch的准确率
        total_loss += loss.item()  # 最终得到整个epoch的损失

    # 打印的是整个eopch上的样本损失的平均值以及准确率
    print(f'\tTrain Loss:{total_loss/len(train_loader):.4f}\tTrain Accuracy:{total_acc/len(mnist_train):.4f}')
```

```
Training Epoch 1: 100% 375/375 [00:00<00:00, 422.32it/s]
        Train Loss:2.0011       Train Accuracy:0.4233
Training Epoch 2: 100% 375/375 [00:00<00:00, 420.71it/s]
        Train Loss:1.7812       Train Accuracy:0.6390
Training Epoch 3: 100% 375/375 [00:00<00:00, 430.68it/s]
        Train Loss:1.7159       Train Accuracy:0.7079
Training Epoch 4: 100% 375/375 [00:00<00:00, 416.55it/s]
```

训练过程

模型测试

[38]:

```python
def evaluate(model, data_loader, device):
    test_acc = 0
    model.eval()  # 切换到测试模式
    with torch.no_grad():  # 不计算梯度
        for batch in data_loader:
            inputs, targets = [x.to(device) for x in batch]
            probs = model(inputs)
            test_acc += (probs.argmax(dim=1) == targets).sum().item()
    model.train()  # 切换到训练模式
    return test_acc
```

训练过程

完整训练过程

一次导入所有需要用的模块

```
import torch
from torch import nn, optim
import torchvision
import torchvision.transforms as transforms  # 处理数据模块
from torch.utils.data import random_split  # 数据集切分
from torch.utils.data import Dataset  # 用于数据整理
from torch.utils.data import DataLoader  # 装载数据，使其可迭代

import matplotlib.pyplot as plt
from tqdm.auto import tqdm #以进度条的方式显示迭代的速度

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = 'all'
```

训练过程

完整训练过程 （数据 + 模型 + 训练算法）

```python
def train(config):
    # 数据整理
    mnist_data = MnistDataset(mnist)
    # 确定训练集、测试集大小，此处以8：2划分训练集和测试集
    num_train = int(mnist_data.lens * 0.8)
    num_test = mnist_data.lens - num_train
    # 数据集划分
    mnist_train, mnist_test = random_split(mnist_data, [num_train, num_test])
    # 装载数据
    train_loader = DataLoader(mnist_train, batch_size=config['batch_size'], shuffle=True, collate_fn=collate_fn)
    test_loader = DataLoader(mnist_test, batch_size=config['batch_size'], shuffle=False, collate_fn=collate_fn)

    # 模型实例化
    model = LogisticRegression(in_features=config['in_features'], num_class=config['num_class'])
    # 将实例化后的模型加载至GPU
    model.to(config['device'])

    # 损失函数
    creterion = nn.CrossEntropyLoss()
    # 优化方法
    optimizer = optim.SGD(model.parameters(), lr=config['learning_rate'])
```

训练过程

完整训练过程 （数据 + 模型 + 训练算法）

```
# 开始训练
for epoch in range(config['num_epochs']):
    total_loss = 0
    total_acc = 0
    for batch in tqdm(train_loader, desc=f'Training Epoch {epoch+1}'):
        inputs, targets = [x.to(config['device']) for x in batch]
        probs = model(inputs)
        loss = creterion(probs, targets)
        optimizer.zero_grad()  # 梯度清零
        loss.backward()  # 反向传播
        optimizer.step()  # 更新参数
        acc = (probs.argmax(dim=1) == targets).sum().item()  # item()用于在只包含一个元素的tensor中提取值
        total_acc += acc  # 最终得到整个epoch的准确率
        total_loss += loss.item()  # 最终得到整个epoch的损失

    # 打印的是整个eopch上的样本损失的平均值以及准确率
    print(f'\tTrain Loss:{total_loss/len(train_loader):.4f}\tTrain Accuracy:{total_acc/len(mnist_train):.4f}')

    # 在测试集上测试
    test_acc = evaluate(model, test_loader, config['device'])
    print(f'\tTest Accuracy:{test_acc/len(mnist_test):.4f}')
```

训练过程

完整训练过程 （数据 + 模型 + 训练算法）

```python
if __name__ == '__main__':
    #设置随机数种子
    torch.manual_seed(100)
    #导入数据
    mnist = torchvision.datasets.MNIST(root='', train=True, download=True, transform=transforms.ToTensor())
    #将超参数以字典形式存放
    config = {'in_features':mnist.data[0].numel(),
              'num_class':len(mnist.targets.unique()),
              'batch_size':128,
              'learning_rate':0.001,
              'num_epochs':10,
              'device': torch.device('cuda' if torch.cuda.is_available() else 'cpu')
             }
    train(config)
```

[13]: <torch._C.Generator at 0x7f5485573c10>

Training Epoch 1: 100% 375/375 [00:00<00:00, 414.06it/s]

Train Loss:1.9397 Train Accuracy:0.4356
Test Accuracy:0.5978

Training Epoch 2: 100% 375/375 [00:00<00:00, 418.49it/s]

Train Loss:1.7721 Train Accuracy:0.6471
Test Accuracy:0.6691

Training Epoch 3: 100% 375/375 [00:01<00:00, 400.73it/s]

Train Loss:1.7439 Train Accuracy:0.6828
Test Accuracy:0.6875

学习资料：https://www.bilibili.com/video/BV1Gg411u7Lr?spm_id_from=333.999.0.0&vd_source=3ddfe9036e33daef6be41f0d67c9199d Pytorch基础快速入门