实验一：猫狗识别

实验目标

学会如何使用 PyTorch 进行图像识别。本实验训练LeNet神经网络对猫狗数据进行识别。

实验环境

- 语言环境：Python3.8
- 深度学习环境：
    - torch==1.10.0+cu113
    - torchvision==0.11.1+cu113

实验步骤

1、数据集下载

数据链接：https://pan.baidu.com/s/1YREL1omT9YJrp9B1PBPTfQ （提取码：ionw）

鼓励除了上述的实验数据外，测试更多的数据任务。

2、导入pytorch深度学习库

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda, Compose
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
import numpy as np
```

3、数据读取与预处理

为了解决数据不足的问题，共测试了以下几种数据增强方式，但发现并非所有的增强操作都会产生正面影响。

- 不进行数据增强：79.2%
- 随机旋转：80.8%
- 随机旋转+高斯模糊模糊：83.3%
- 随机垂直翻转：73.3%

通过选择注释中的代码，可以测试不同方式的数据增强。

```
train_datadir = './1-cat-dog/train/'
test_datadir  = './1-cat-dog/val/'

train_transforms = transforms.Compose([
    transforms.Resize([224, 224]),  # 将输入图片resize成统一尺寸
    # transforms.RandomRotation(degrees=(-10, 10)),  #随机旋转，-10到10度之间随机选
    # transforms.RandomHorizontalFlip(p=0.5),  #随机水平翻转 选择一个概率概率
    # transforms.RandomVerticalFlip(p=0.5),  #随机垂直翻转
    # transforms.RandomPerspective(distortion_scale=0.6, p=1.0), #随机视角
    # transforms.GaussianBlur(kernel_size=(5, 9), sigma=(0.1, 5)), #随机选择的高斯模糊模糊图像
    transforms.ToTensor(),           # 将PIL Image或numpy.ndarray转换为tensor，并归一化到[0,1]之间
    transforms.Normalize(            # 标准化处理-->转换为标准正太分布（高斯分布），使模型更容易收敛
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225])  # 其中 mean=[0.485,0.456,0.406]与std=[0.229,0.224,0.225] 从数据集中随机抽样计算得到的。
])

test_transforms = transforms.Compose([
    transforms.Resize([224, 224]),  # 将输入图片resize成统一尺寸
    transforms.ToTensor(),           # 将PIL Image或numpy.ndarray转换为tensor，并归一化到[0,1]之间
    transforms.Normalize(            # 标准化处理-->转换为标准正太分布（高斯分布），使模型更容易收敛
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225])  # 其中 mean=[0.485,0.456,0.406]与std=[0.229,0.224,0.225] 从数据集中随机抽样计算得到的。
])
```

```
train_data =
datasets.ImageFolder(train_datadir,transform=train_transforms)

test_data  =
datasets.ImageFolder(test_datadir,transform=test_transforms)

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=4,
                                           shuffle=True,
                                           num_workers=1)
test_loader  = torch.utils.data.DataLoader(test_data,
                                           batch_size=4,
                                           shuffle=True,
                                           num_workers=1)
```

数据增强补充：

`transforms.Compose` 的信息：https://pytorch-cn.readthedocs.io/zh/latest/torchvision/torchvision-transform/

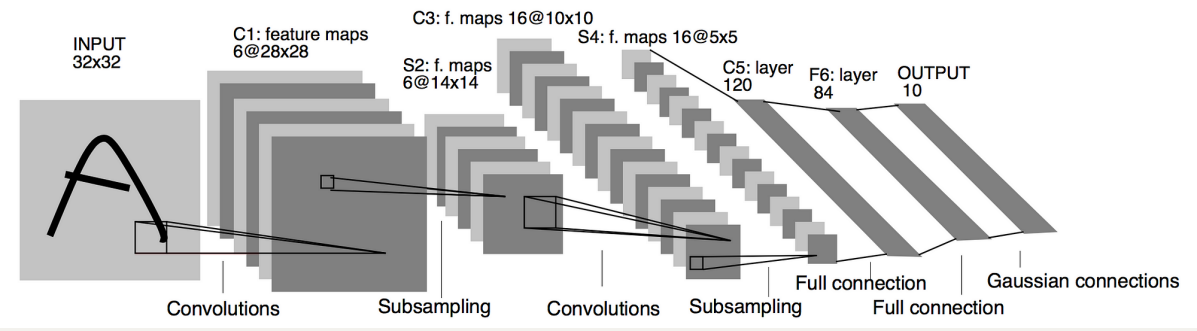
数据增强方面的拓展：https://pytorch.org/vision/stable/auto_examples/plot_transforms.html#sphx-glr-auto-examples-plot-transforms-py

对应的API：https://pytorch.org/vision/stable/transforms.html

鼓励大家使用不同的数据增强方法测试效果。

```
for X, y in test_loader:
    print("Shape of X [N, C, H, W]: ", X.shape)
    print("Shape of y: ", y.shape, y.dtype)
    break
```

4、定义模型



LeNet是一种经典的卷积神经网络（CNN）架构，主要由以下几个关键组件构成：

1. **卷积层**：使用多个卷积核提取特征。例如，第一层通常采用6个5x5的卷积核，生成6个28x28的特征图。卷积操作通过局部连接和共享权重有效地捕捉局部特征。
2. **激活函数**：通常使用Sigmoid或Tanh作为激活函数，增加网络的非线性特性。
3. **池化层**：引入平均池化（或最大池化）操作来降低特征图的维度，减少计算复杂度，同时提高模型的鲁棒性。池化层通常跟随卷积层，减少特征图的空间大小（例如，从28x28缩小到14x14）。
4. **全连接层**：在网络的后面部分，展平后的特征图被连接到多个全连接层，如C5层（120个神经元）和F6层（84个神经元），用于综合提取的特征并进行最终分类。
5. **输出层**：采用softmax函数生成每个类别的概率分布，通常用于多类分类任务。

LeNet通过组合卷积、激活、池化和全连接操作，能够有效提取图像中的层次特征，使其在手写数字识别等任务中表现优越。尽管相对简单，LeNet的设计理念为后续更复杂的CNN架构（如AlexNet、VGG和ResNet）奠定了基础。

以下是一种简单的LeNet网络的实现，鼓励大家修改模型，提升训练效果。

```
import torch.nn.functional as F

# 找到可以用于训练的 GPU
device = "cuda" if torch.cuda.is_available() else "cpu"
print("Using {} device".format(device))

# 定义模型
class LeNet(nn.Module):
    # 一般在__init__中定义网络需要的操作算子，比如卷积、全连接算子等等
    def __init__(self):
        super(LeNet, self).__init__()
        # Conv2d的第一个参数是输入的channel数量，第二个是输出的channel数
量，第三个是kernel size
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # 由于上一层有16个channel输出，每个feature map大小为5*5，所以全连
接层的输入是16*5*5
        self.fc1 = nn.Linear(16*53*53, 120)
        self.fc2 = nn.Linear(120, 84)
        # 最终有10类，所以最后一个全连接层输出数量是10
        self.fc3 = nn.Linear(84, 2)
        self.pool = nn.MaxPool2d(2, 2)
    # forward这个函数定义了前向传播的运算，只需要像写普通的python算数运算那样
就可以了
```

```
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        # 下面这步把二维特征图变为一维，这样全连接层才能处理
        x = x.view(-1, 16*53*53)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = LeNet().to(device)
print(model)
```

```
LeNet(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=44944, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=2, bias=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

5、损失函数与优化器

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

6、定义训练函数

在单个训练循环中，模型对训练数据集进行预测（分批提供给它），并反向传播预测误差从而调整模型的参数。

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # 计算预测误差
        pred = model(X)
        loss = loss_fn(pred, y)
```

```
        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

7、定义测试函数

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) ==
y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg
loss: {test_loss:>8f} \n")
```

8、训练

```
epochs = 20
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_loader, model, loss_fn, optimizer)
    test(test_loader, model, loss_fn)
print("Done!")
```

```
Epoch 1
-------------------------------
loss: 0.697082  [    0/  480]
loss: 0.686452  [  400/  480]
Test Error:
 Accuracy: 50.8%, Avg loss: 0.692428

Epoch 2
-------------------------------
loss: 0.696046  [    0/  480]
loss: 0.674288  [  400/  480]
Test Error:
 Accuracy: 50.0%, Avg loss: 0.690799

Epoch 3
-------------------------------
loss: 0.682432  [    0/  480]
loss: 0.677850  [  400/  480]
Test Error:
 Accuracy: 58.3%, Avg loss: 0.686088

Epoch 4
-------------------------------
loss: 0.707287  [    0/  480]
loss: 0.681919  [  400/  480]
...
Test Error:
 Accuracy: 73.3%, Avg loss: 0.498096
```

拓展任务

在原始代码中，训练准确率仅有73.3%，鼓励大家通过修改模型结构参数、使用不同的数据增强方法等等提升识别任务的准确率。

引用

🔥本文 ***GitHub*** https://github.com/kzbkzb/Python-AI 已收录

- 作者：K同学啊

- 来自专栏：《深度学习100例》-PyTorch版本