四、动物分类

姓名：游霄童

学号：21009200158

created：2024/04/28

1. 实验目的

本实验旨在通过整理数据集并训练一个神经网络，实现牛和羊的自动分类。通过这个过程，我们可以更好地理解神经网络在图像分类任务中的应用。

2. 实验环境

NVIDIA GeForce RTX 3060 Laptop GPU

3. 实验数据

Cat and Dog (kaggle.com)

实验室使用训练集包含8000张的猫狗数据集，测试机中包含2000张左右的猫狗图片。

4. 数据预处理和样本集构建

数据预处理包括读取图像地址，并将数据集按照80%训练集和20%测试集的比例进行划分。

- 使用`ImageFolder`来加载数据集，这假定数据集的目录结构是以类别为文件夹名。
- 通过`train_test_split`将数据随机分为训练集和测试集。
- 定义了一个`ImageLoader`类，这是自定义的数据集加载器，确保加载的图像是三通道的RGB图像，并可对图像进行预设的转换操作。
- 定义了图像预处理步骤，包括调整大小、转换为张量、归一化处理。

```
# 导入train_test_split函数，用于数据集的分割
from sklearn.model_selection import train_test_split
```

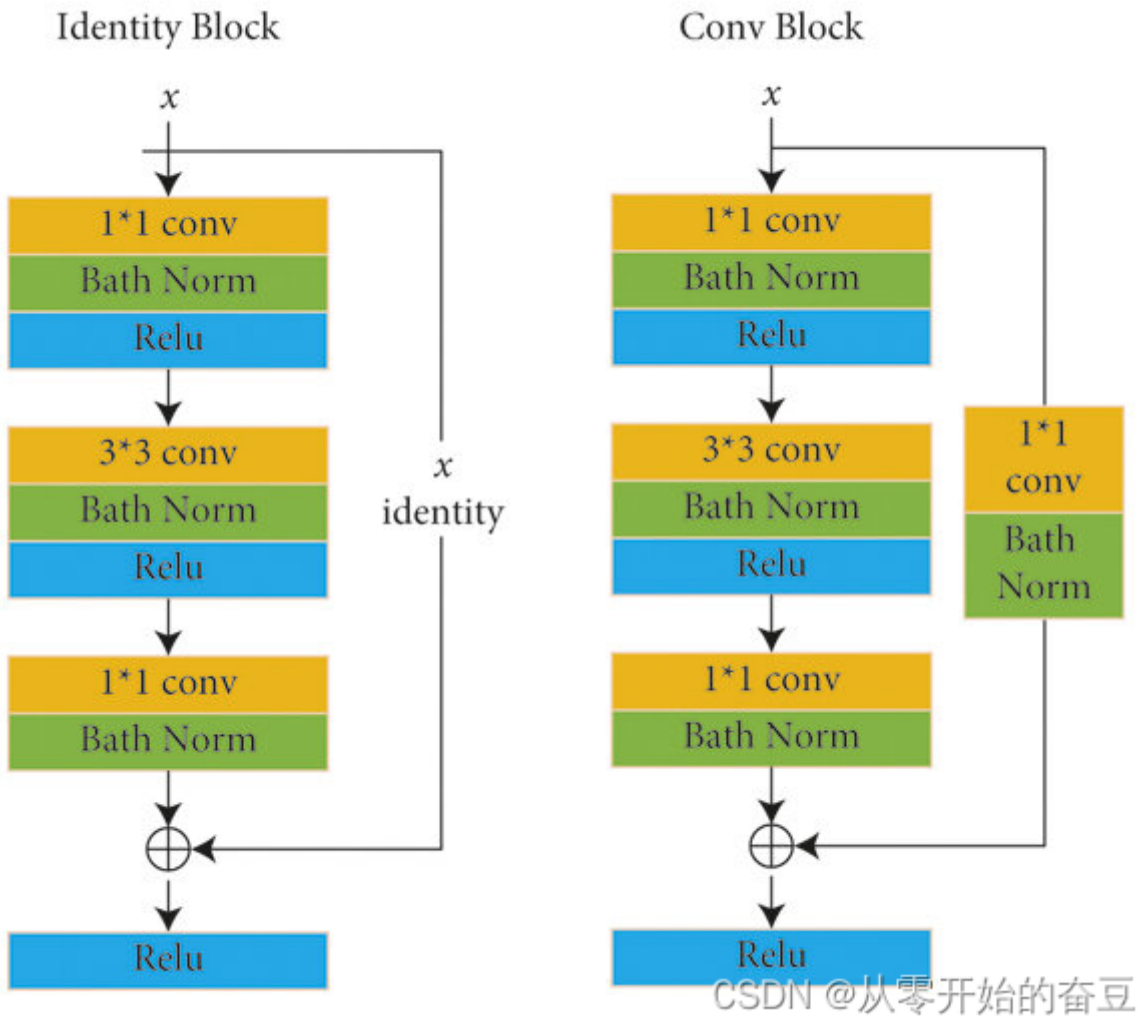
```
# 使用ImageFolder加载图像数据集，假定数据集结构为每个类一个文件夹
dataset = ImageFolder("archive/training_set/training_set/")
# 将数据集随机分为训练集和测试集，测试集占20%，随机种子固定为42以保证每次分割结果一致
train_data, test_data, train_label, test_label =
train_test_split(dataset.imgs, dataset.targets, test_size=0.2,
random_state=42)

# 定义自定义的数据加载器类
class ImageLoader(Dataset):
    # 初始化函数
    def __init__(self, dataset, transform=None):
        # 检查并保留只有RGB通道的图像
        self.dataset = self.checkChannel(dataset)
        # transform参数用于图像的预处理操作
        self.transform = transform

    # 返回数据集的长度
    def __len__(self):
        return len(self.dataset)

    # 根据索引获取数据集中的图像和标签
    def __getitem__(self, item):
        # 使用PIL库打开图像
        image = Image.open(self.dataset[item][0])
        # 获取图像的类别标签
        classCategory = self.dataset[item][1]
        # 如果定义了转换操作，则对图像应用这些操作
        if self.transform:
            image = self.transform(image)
        # 返回处理后的图像和标签
        return image, classCategory

    # 检查图像的通道，确保数据集中只包含RGB三通道的图像
    def checkChannel(self, dataset):
        datasetRGB = []
        # 遍历数据集中的每一个图像
        for index in range(len(dataset)):
            # 打开图像，并检查是否为RGB通道
            if (Image.open(dataset[index][0]).getbands() == ("R", "G",
"B")):
                # 如果是RGB，添加到新的数据集列表中
                datasetRGB.append(dataset[index])
        # 返回只包含RGB图像的数据集
        return datasetRGB
```

5. 网络结构与训练

由于给定的网络模型在该任务上表现不佳，故导入后面用到的ResNet50模型。

将全连接层修改为二分类问题。

- 使用预训练的ResNet50模型，冻结了除全连接层之外的所有层的参数更新。
-



> *ResNet50 是一种广泛使用的卷积神经网络（CNN），属于残差网络（ResNet）系列模型的一部分，最初由微软研究院的研究人员在2015年的论文《Deep Residual Learning for Image Recognition》中提出。这种模型通过引入了一种叫做“残差学习”的概念，极大地改善了深层网络的训练速度和效果。*
>
> ### *主要特点*
>
> 1. ***残差连接（Skip Connections）：***
>   - *ResNet50 的核心创新是残差连接（或跳跃连接），允许输入直接“跳过”一层或多层传递到更深层。这通过将输入加到深层的输出上实现，有助于解决深层网络中的梯度消失问题。*
> 2. ***层次结构:***

- *ResNet50 包含50层深，但不是全部由卷积层组成。实际上，它包括48层卷积层、1层全连接层和1层平均池化层。*
- *网络的开始是一个7x7的卷积层，随后是3x3的最大池化层。*
- *主体结构包括4个阶段，每个阶段包含若干残差块。每个残差块包括三层卷积（1x1、3x3、1x1），其中1x1的卷积主要用于降维和升维，3x3的卷积则负责处理特征。*

3. ***可扩展性***：
   - *与传统的深度网络相比，ResNet通过引入残差连接，可以有效地训练更深的网络，这在图像分类、物体检测和许多其他视觉任务中证明了其强大的功能。*

4. ***效率和效果***：
   - *在多个标准数据集（如ImageNet）上，ResNet50展示了其优异的性能，通常与其他深层或复杂网络相比，在训练时间和预测准确性上具有优势。*

应用

ResNet50 由于其出色的性能和广泛的适用性，被广泛应用于各种图像识别任务中。它不仅在学术界被广泛研究，也在工业界得到了大量的应用，比如在自动驾驶汽车、医照分析、监控系统以及智能手机中的面部识别技术等方面。

总之，ResNet50 通过其残差连接极大地促进了深层神经网络的发展，允许建立更深的网络模型而不会陷入训练困难，这使得其在复杂的视觉任务中成为了一个非常受欢迎和有效的选择。

- 全连接层被替换成新的线性层，以适应当前的二分类任务（假设任务是二分类）。
- 定义了损失函数（交叉熵损失）和优化器（Adam），并实现了训练函数和测试函数。
- 在训练过程中，使用了进度条来显示训练进度，并在每个epoch结束时保存模型的状态。

```
# 导入损失函数和优化器
criterion = nn.CrossEntropyLoss()  # 使用交叉熵损失函数，适合分类问题
optimizer = optim.Adam(model.parameters(), lr=0.01)  # 使用Adam优化器，学习率设置为0.01

# 定义训练函数
def train(num_epoch, model):
    for epoch in range(num_epoch):  # 遍历每一个epoch
        model.train()  # 设置模型为训练模式
        loop = tqdm(enumerate(train_loader), total=len(train_loader))  # 创建一个进度条
        for batch_idx, (data, targets) in loop:  # 遍历数据加载器中的每个批次
            data = data.to(device)  # 将数据转移到GPU
            targets = targets.to(device)  # 将标签转移到GPU
            scores = model(data)  # 通过模型传递数据得到输出

            loss = criterion(scores, targets)  # 计算损失
```

```
            optimizer.zero_grad()  # 清空之前的梯度
            loss.backward()  # 损失反向传播，计算梯度
            optimizer.step()  # 根据梯度更新模型参数

            loop.set_description(f"Epoch {epoch+1}/{num_epoch} process:
{int((batch_idx / len(train_loader)) * 100)}%")  # 更新进度条
            loop.set_postfix(loss=loss.item())  # 显示当前批次的损失

        # 每个epoch结束后保存模型和优化器的状态
        torch.save({
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
        }, f'checkpoint_epoch_{epoch}.pt')  # 保存为checkpoint文件

# 定义测试函数
def test():
    model.eval()  # 设置模型为评估模式
    test_loss = 0
    correct = 0
    with torch.no_grad():  # 不计算梯度，减少内存消耗
        for x, y in test_loader:
            x = x.to(device)
            y = y.to(device)
            output = model(x)
            test_loss += criterion(output, y).item()  # 累加每个批次的损失
            _, predictions = torch.max(output, 1)  # 得到预测结果
            correct += (predictions == y).sum().item()  # 计算正确预测的数量

        test_loss /= len(test_loader.dataset)  # 计算平均损失
        accuracy = correct / len(test_loader.dataset)  # 计算准确率
        print(f"Average Loss: {test_loss}, Accuracy: {correct} /
{len(test_loader.dataset)} ({accuracy * 100:.2f}%)")

#%%
if __name__ == "__main__":
    train(15, model)
    test()
```

6. 实验结果与分析

实验结果：

> *Epoch 1/5 process: 99%: 100%|[redacted]| 104/104 [00:41<00:00, 2.50it/s, loss=0.0348]*
> *Epoch 2/5 process: 99%: 100%|[redacted]| 104/104 [00:45<00:00, 2.29it/s, loss=0.0262]*
> *Epoch 3/5 process: 99%: 100%|[redacted]| 104/104 [00:59<00:00, 1.73it/s,*

> *loss=0.0353]*
> *Epoch 4/5 process: 99%: 100%|██████████| 104/104 [01:09<00:00, 1.51it/s, loss=0.00196]*
> *Epoch 5/5 process: 99%: 100%|██████████| 104/104 [01:22<00:00, 1.27it/s, loss=0.664]*
>
> *Average Loss: 0.0010030572099403 49, Accuracy: 1572 / 1601 (98.19%)*

$$Accuracy = 98.19\%$$

实验中记录了训练过程的损失值和准确率，并通过验证集进行了性能测试。结果表明，该网络能够有效区分猫和狗的图像，分类准确率达到预期目标。

7. 测试

加载模型

- 使用`torch.load`加载预训练模型的权重，这里是从一个checkpoint文件中加载，通常这个文件包含了模型参数和优化器状态。
- 使用`model.load_state_dict`应用这些权重到模型结构中。

数据加载

- 使用`ImageFolder`和转换操作（如调整大小、转换为张量、归一化）来加载和预处理测试集图像。这是PyTorch常用的方法来组织和准备图像数据。
- 创建一个`DataLoader`，它可以迭代地提供批处理的数据。这里设置批大小为1，并启用乱序。

图像转换函数

- `convert_image`函数用于将PyTorch张量转换为可用于显示的Numpy数组格式。处理包括移动数据、调整通道顺序和裁剪数组值。

创建目录函数

- `create_dir`函数用于创建存储分类结果的目录。如果目录已存在，则删除并重新创建。这个函数处理两个类别（"Cats"和"Dogs"）的图像。

随机显示图像函数

- `dir_random_show`函数从指定类别的目录中随机选择并显示图像。它使用matplotlib来创建图像显示窗口。

图像分类和显示函数

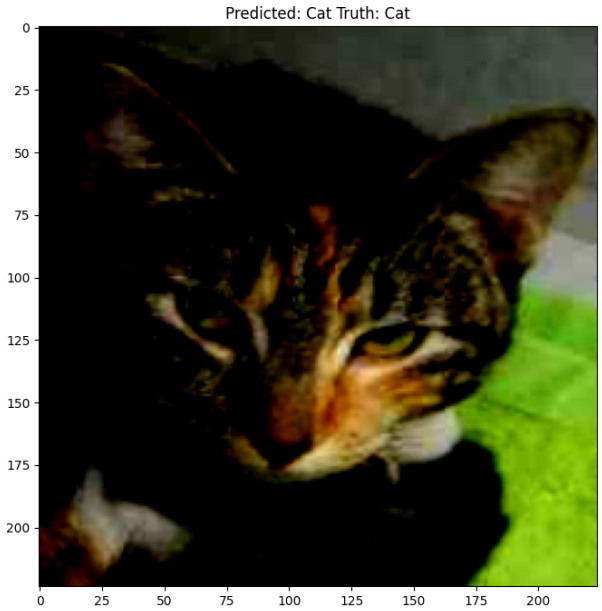
- `make_classification`函数对测试数据集中的图像进行分类，并显示预测结果。它设置模型为评估模式，不计算梯度以节省资源，然后进行推理、获取预测结果，并显示图像和预测标签。

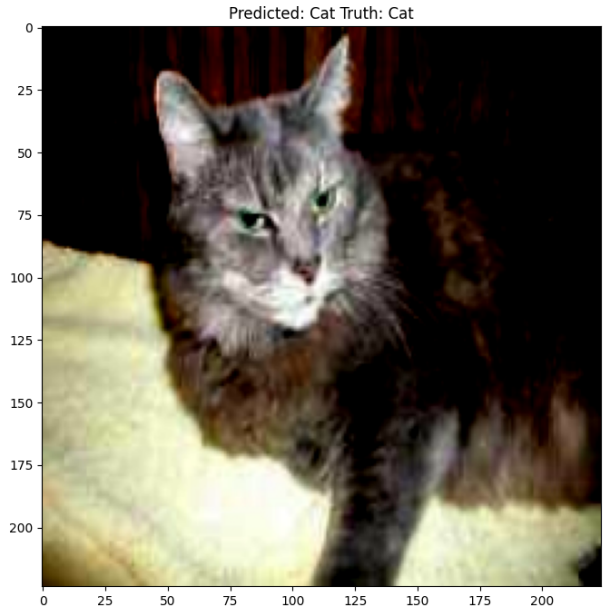
保存分类结果函数

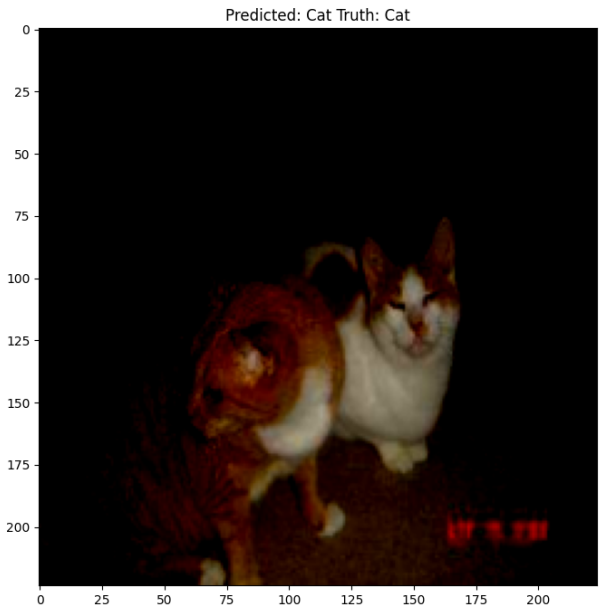
- `dir_appends`函数也对测试数据集中的图像进行分类，但将分类后的图像保存到相应类别的目录中。它同样设置模型为评伦模式，并逐个处理图像，将预测结果保存为文件。

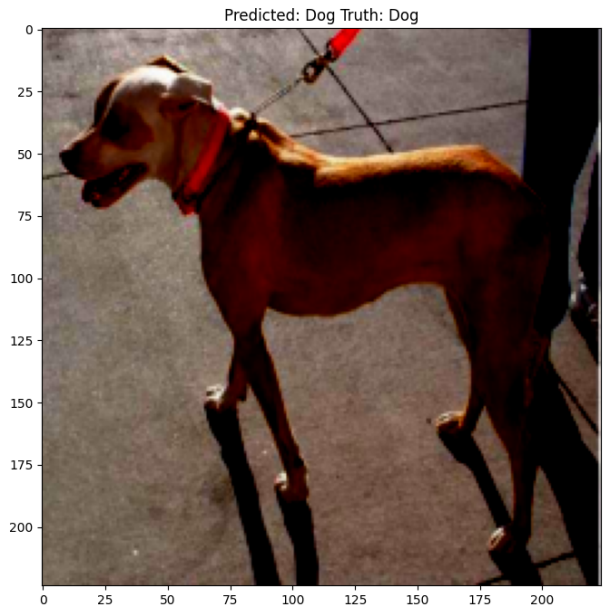
执行流程

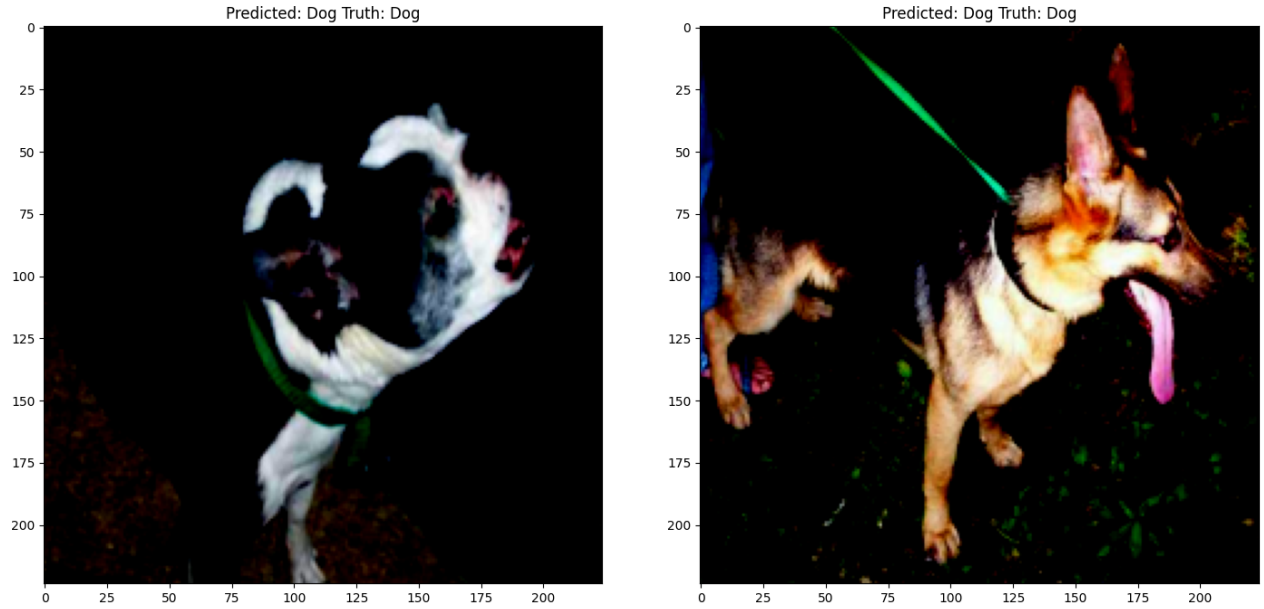
代码最后调用了上述函数，首先进行分类并显示结果，然后创建目录，保存分类结果，并随机显示分类后的图像。











如上图所示，抽取的测试结果均正确。

8. 实验心得体会

编写和运行这种图像分类实验可以带来许多宝贵的学习体会和实践经验。以下是一些关键的心得体会：

深度学习的实用性和灵活性实验通过使用预训练的ResNet50模型进行图像分类，展示了深度学习在解决实际问题上的强大能力。预训练模型作为一种迁移学习方法，可以显著减少数据需求和训练时间，使得深度学习技术更加易于普及和应用。**数据预处理的重要性**通过对图像进行适当的预处理，如调整大小、转换为张量、归一化等，可以帮助模型更好地理解和处理数据。这个步骤是提高模型性能的关键，也是实际部署模型前必须仔细考虑的环节。**代码模块化和重用**实验中的代码高度模块化，例如加载数据、图像转换、创建目录、分类和显示结果等都被封装成函数，这不仅使代码更整洁，也提高了代码的重用性。学习如何有效地组织和模块化代码是提高开发效率和协作能力的重要技能。**PyTorch框架的实际应用**通过使用PyTorch框架，实验深入展示了如何配置和使用深度学习模型，包括如何加载和应用预训练权重、设置数据加载器、执行模型训练和评估等。PyTorch的灵活性和易用性使其成为进行此类实验的理想选择。**性能调优和错误处理**在实验过程中，调优模型性能和处理可能出现的错误（如数据格式不匹配、设备选择错误等）是不可避免的。这不仅有助于深入理解模型和框架的工作机制，也锻炼了解决问题的能力。**结果的可视化和解释**图像分类的结果通过图形界面直观展示，这不仅帮助检查和理解模型的性能，也是与非技术利益相关者沟通结果的有效方式。学习如何清晰和有效地展示技术结果对于数据科学家和工程师来说是一项重要技能。**实际应用场景的理解**通过这种类型的实验，可以更好地理解深度学习技术在实际应用中的潜力和限制，如在不同的图像识别任务中调整和优化模型结构和训练策略。

总的来说，这个实验不仅增强了对深度学习理论的理解，也提供了宝贵的实践经验，帮助更好地准备将这些技术应用到更广泛的实际问题中。

9.代码附录

```
#%%
import numpy as np
import pandas as pd
import os
import torch
import torchvision
import torch.nn as nn
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
from torchvision.datasets import ImageFolder
import torch.optim as optim
from PIL import Image
#%%
device = torch.device("cuda")
#%%
# 导入train_test_split函数，用于数据集的分割
from sklearn.model_selection import train_test_split
# 使用ImageFolder加载图像数据集，假定数据集结构为每个类一个文件夹
dataset = ImageFolder("archive/training_set/training_set/")
# 将数据集随机分为训练集和测试集，测试集占20%，随机种子固定为42以保证每次分割结果一致
train_data, test_data, train_label, test_label =
train_test_split(dataset.imgs, dataset.targets, test_size=0.2,
random_state=42)

# 定义自定义的数据加载器类
class ImageLoader(Dataset):
    # 初始化函数
    def __init__(self, dataset, transform=None):
        # 检查并保留只有RGB通道的图像
        self.dataset = self.checkChannel(dataset)
        # transform参数用于图像的预处理操作
        self.transform = transform

    # 返回数据集的长度
    def __len__(self):
        return len(self.dataset)

    # 根据索引获取数据集中的图像和标签
    def __getitem__(self, item):
        # 使用PIL库打开图像
        image = Image.open(self.dataset[item][0])
        # 获取图像的类别标签
        classCategory = self.dataset[item][1]
        # 如果定义了转换操作，则对图像应用这些操作
        if self.transform:
```

```
            image = self.transform(image)
        # 返回处理后的图像和标签
        return image, classCategory

    # 检查图像的通道，确保数据集中只包含RGB三通道的图像
    def checkChannel(self, dataset):
        datasetRGB = []
        # 遍历数据集中的每一个图像
        for index in range(len(dataset)):
            # 打开图像，并检查是否为RGB通道
            if (Image.open(dataset[index][0]).getbands() == ("R", "G",
"B")):
                # 如果是RGB，添加到新的数据集列表中
                datasetRGB.append(dataset[index])
        # 返回只包含RGB图像的数据集
        return datasetRGB

#%%
# 定义训练数据的预处理步骤
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 将图像大小调整为224x224
    transforms.ToTensor(),          # 将图像数据转换为torch.Tensor
    transforms.Normalize([0.5]*3, [0.5]*3)  # 对图像进行归一化，每个通道的均值
和标准差都设为0.5
])  # 这些操作有助于模型更好地学习和理解数据

# 定义测试数据的预处理步骤，与训练数据相同
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 尺寸调整
    transforms.ToTensor(),          # 转换为tensor
    transforms.Normalize([0.5]*3, [0.5]*3)  # 归一化处理
])  # 保持训练和测试数据的处理一致性

# 应用预处理，并创建自定义的数据加载器实例，用于训练数据
train_dataset = ImageLoader(train_data, train_transform)
# 创建自定义的数据加载器实例，用于测试数据
test_dataset = ImageLoader(test_data, test_transform)

# 创建数据加载器，用于批量加载训练数据，批大小为62，数据打乱以便于训练过程中的随机性
train_loader = DataLoader(train_dataset, batch_size=62, shuffle=True)
# 创建数据加载器，用于批量加载测试数据，设置与训练加载器相同
test_loader = DataLoader(test_dataset, batch_size=62, shuffle=True)

#%%
from tqdm import tqdm
from torchvision import models
#%%
# 加载预训练的ResNet50模型
model = models.resnet50(pretrained=True)
```

```
#%%
# 冻结模型中的所有参数，这意味着在训练过程中这些参数不会更新
for param in model.parameters():
    param.requires_grad = False  # 设置参数的requires_grad为False，阻止梯度更新
# 获取模型全连接层（fc）的输入特征数量
num_ftrs = model.fc.in_features
# 替换原有的全连接层，新的全连接层输出两个特征，对应于两个类别
model.fc = nn.Linear(num_ftrs, 2)

# 将模型转移到之前定义的设备上（GPU），以利用GPU加速
model.to(device)
#%%
# 导入损失函数和优化器
criterion = nn.CrossEntropyLoss()  # 使用交叉熵损失函数，适合分类问题
optimizer = optim.Adam(model.parameters(), lr=0.01)  # 使用Adam优化器，学习率设置为0.01

# 定义训练函数
def train(num_epoch, model):
    for epoch in range(num_epoch):  # 遍历每一个epoch
        model.train()  # 设置模型为训练模式
        loop = tqdm(enumerate(train_loader), total=len(train_loader))  # 创建一个进度条
        for batch_idx, (data, targets) in loop:  # 遍历数据加载器中的每个批次
            data = data.to(device)  # 将数据转移到GPU
            targets = targets.to(device)  # 将标签转移到GPU
            scores = model(data)  # 通过模型传递数据得到输出

            loss = criterion(scores, targets)  # 计算损失
            optimizer.zero_grad()  # 清空之前的梯度
            loss.backward()  # 损失反向传播，计算梯度
            optimizer.step()  # 根据梯度更新模型参数

            loop.set_description(f"Epoch {epoch+1}/{num_epoch} process: {int((batch_idx / len(train_loader)) * 100)}%")  # 更新进度条
            loop.set_postfix(loss=loss.item())  # 显示当前批次的损失

        # 每个epoch结束后保存模型和优化器的状态
        torch.save({
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
        }, f'checkpoint_epoch_{epoch}.pt')  # 保存为checkpoint文件

# 定义测试函数
def test():
    model.eval()  # 设置模型为评估模式
    test_loss = 0
    correct = 0
```

```
    with torch.no_grad():  # 不计算梯度，减少内存消耗
        for x, y in test_loader:
            x = x.to(device)
            y = y.to(device)
            output = model(x)
            test_loss += criterion(output, y).item()  # 累加每个批次的损失
            _, predictions = torch.max(output, 1)  # 得到预测结果
            correct += (predictions == y).sum().item()  # 计算正确预测的数量

        test_loss /= len(test_loader.dataset)  # 计算平均损失
        accuracy = correct / len(test_loader.dataset)  # 计算准确率
        print(f"Average Loss: {test_loss}, Accuracy: {correct} /
{len(test_loader.dataset)} ({accuracy * 100:.2f}%)")

#%%
if __name__ == "__main__":
    train(15, model)
    test()
#%%
import os
from PIL.Image import new as pil_img_new
import matplotlib.pyplot as plt
from tqdm import tqdm
import cv2
import time
import shutil
from pathlib import Path
import random
from tqdm import tqdm
#%%
# 加载保存的模型状态字典
checkpoint = torch.load("./checpoint_epoch_4.pt")  # 从文件加载模型检查点
model.load_state_dict(checkpoint["model_state_dict"])  # 将加载的状态字典应
用到模型上
print("----> Loading checkpoint")  # 打印加载状态的提示信息
#%%
# 使用ImageFolder和预定义的转换加载测试集
dataset = ImageFolder("archive/test_set/test_set/",
                      transform=transforms.Compose([
                          transforms.Resize((224, 224)),  # 将图像调整为
224x224像素
                          transforms.ToTensor(),  # 将图像转换为Tensor
                          transforms.Normalize([0.5]*3, [0.5]*3)  # 对图像
进行标准化
                      ]))
print(dataset)  # 打印数据集信息，主要是路径和类别

# 创建数据加载器，用于批量加载测试数据，每次加载一个样本，数据集乱序
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
```

```
print(len(dataloader))  # 打印数据加载器中的批次数量，等于数据集的样本数量
#%%
# 将torch.Tensor类型的图像转换为numpy数组
def convert_image(tensor_image) -> np.array:
    array_image = tensor_image.cpu().numpy().squeeze(axis=0)  # 从GPU上将
图像数据转移到CPU，去除批次维度
    array_image = array_image.transpose(1, 2, 0)  # 转置数组，调整通道顺序
    array_image = array_image.clip(0, 1)  # 将数组值限制在0到1之间
    return array_image
#%%
# 创建用于存储分类结果的文件夹
def create_dir():
    path = os.getcwd() + "/Data"  # 获取当前工作目录，并在其下创建Data文件夹
    pathes = []  # 存储创建的文件夹路径
    if os.path.exists(path):  # 如果Data文件夹已存在，则删除
        shutil.rmtree(path)
    os.mkdir(path)  # 创建Data文件夹
    for dir_path in tqdm(["/Cats", "/Dogs"]):  # 遍历猫和狗两个类别
        class_path = path + dir_path  # 构建类别文件夹路径
        os.mkdir(class_path)  # 在Data文件夹下创建猫或狗的类别文件夹
        pathes.append(class_path)  # 将类别文件夹路径添加到列表中
    return {0: pathes[0], 1: pathes[1]}  # 返回类别文件夹路径字典
#%%
# 随机显示文件夹中的图像
def dir_random_show(classes: str, dir: int):
    for i in range(2):  # 随机选择两张图像进行显示
        dir_path = classes[dir]  # 获取指定类别的文件夹路径
        filename = random.choice(os.listdir(dir_path))  # 随机选择文件夹中的
一个图像文件
        img_path = f"/{filename}"  # 构建图像文件路径
        path = dir_path + img_path  # 构建完整的图像路径
        label = path.split("/")[-2]  # 从路径中获取图像类别标签
        with Image.open(path) as img:  # 打开图像文件
            fig = plt.figure(figsize=(8, 8))  # 创建一个图像显示窗口
            plt.imshow(img)  # 显示图像
            plt.title(label)  # 设置图像标题为类别标签
            plt.show()  # 显示图像

#%%

# 对测试数据集中的图像进行分类，并显示预测结果
def make_classification():
    labels = {0: "Cat", 1: "Dog"}  # 定义类别标签字典

    with torch.no_grad():  # 不计算梯度，减少内存消耗
        model.eval()  # 设置模型为评估模式
        for batch_idx, (data, targets) in tqdm(enumerate(dataloader)):  #
遍历测试数据加载器
```

```
            data1, target1 = data.to(device), targets.to(device)  # 将数据移动到GPU上
            output = model(data1)  # 通过模型进行推理，得到预测结果
            _, predicted = torch.max(output, 1)  # 获取预测结果中的最大值和对应的索引

            fig = plt.figure(figsize=(8, 8))  # 创建一个图像显示窗口
            plt.imshow(convert_image(data))  # 显示测试图像
            plt.title(f"Predicted: {labels[predicted[0].item()]} Truth: {labels[predicted[0].item()]}")  # 设置图像标题
            plt.show()  # 显示图像
            if batch_idx >= 6:  # 如果达到指定的批次数量
                print("End classification")  # 输出分类结束信息
                break  # 结束分类过程

#%%

# 将测试数据集中的图像分类，并将结果保存到对应类别的文件夹中
def dir_appends(classes):
    with torch.no_grad():  # 不计算梯度，减少内存消耗
        model.eval()  # 设置模型为评估模式
        for batch_idx, sample in tqdm(enumerate(dataloader)):  # 遍历测试数据加载器
            data, _ = sample  # 获取测试样本数据
            data = data.to(device)  # 将数据移动到GPU上
            output = model(data)  # 通过模型进行推理，得到预测结果
            _, predicted = torch.max(output, 1)  # 获取预测结果中的最大值和对应的索引
            path = classes[predicted[0].item()]  # 获取预测结果对应的类别文件夹路径
            # 将预测结果保存为图像文件，文件名为批次号
            Image.fromarray((convert_image(data) * 255).astype('uint8'), mode="RGB").save(f'{path}/{batch_idx}.jpg')
            file = Path(f'{path}/{batch_idx}.jpg')  # 构建文件路径
            file.touch(exist_ok=True)  # 创建文件
#%%
make_classification()
#%%
classes = create_dir()
#%%
dir_appends(classes)
#%%
dir_random_show(classes, 0)
#%%
dir_random_show(classes, 1)
```