成　绩	
评阅人	

复 旦 大 学

本 科 生 课 程 论 文

论文题目：　CIFAR-10 图像分类实验报告

修读课程：　人工智能的编程基础（AIE310006.02）

选课学期：　2025 春学期

选课学生：　徐晨欣（24300130064）

完成日期：　2025.6.20

目录

1 项目背景及要求

本项目来源于 Kaggle 网站所举办的 CIFAR-10 - Object Recognition in Images 竞赛。

CIFAR-10 是一个经典的计算机视觉数据集，广泛应用于机器学习领域的图像分类任务。CIFAR-10 是一个 8000 万张微小图像数据集的子集，由 60000 张 32*32 的彩色图像组成，其中包含 10 个对象类（飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船、卡车），每个类中有 6000 张图像，每个类别完全互斥。

在本项目中，要求：

实现基于 PyTorch 的卷积神经网络（CNN）模型完成图像分类

对于测试集中的每张图像，预测给定的 ID 标签。

在 Kaggle 官网上提交项目代码并进行评估，根据分类准确性（正确预测的标签的百分比）评估提交内容。

2 实验环境

系统版本：Windows 11 24H2 26100.4061

PyThon 版本：PyThon 3.12

包管理工具：Anaconda Navigator

IDE / Editor：PyCharm Community Edition 2024.3.4

训练过程使用 Kaggle 网站 GPU P100 加速

3 实验原理

3.1 卷积神经网络架构

卷积层：局部提取特征

池化层：采样保留关键特征

全连接层：整合全局特征进行分类

3.2 核心算法

反向传播：通过链式法则计算卷积核梯度

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial f} \cdot \mathrm{X}$$

损失函数：交叉熵损失

$$L = -\sum_{C=1}^{10} y_c \log p_c$$

优化器：SGD

$$v_{t+1} = \gamma v_t + \eta \nabla L(W_t)$$

$$W_{t+1} = W_t - v_{t+1}$$

3.3 评估指标

准确率：整体分类准确率

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

精确率：正类预测可靠性

$$Precision = \frac{TP}{TP + FP}$$

召回率：正类样本覆盖率

$$Recall = \frac{TP}{TP + FN}$$

4 实验设计

4.1 数据准备

在实验准备阶段，下载了 Kaggle 竞赛中提供的数据集 train 和 test，但实际训练过程中，发现 Kaggle 提供的 train 数据集没有事先将图像分为 10 类，需要手动分类，所以实际实验过程中，使用了 PyTorch 自带的 CIFAR-10 数据集进行训练。

由于 test 数据集中的图片格式与所需图片格式不相符，考虑到兼容性，对测试集数据进行预处理。

```
#数据预处理
    transform = transforms.Compose([
        transforms.Resize((32,32)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
……

# 加载并预处理图像
            image = Image.open(img_path).convert('RGB')
```

图 1 数据预处理代码

4.2 模型构建

模型中设置两个卷积层，使最后输出尺寸为 8*8。

模型使用 ReLU 函数作为激活函数

由于在最初训练中，模型过拟合严重，因此，添加了 Dropout（结构性正则化），防止过拟合。

```python
#定义卷积神经网络模型
class CIFAR10Model(Module):
    def __init__(self):
        super().__init__()
        #第一个卷积层的输入通道为3(RGB)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        #池化层
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        #结构性正则化
        self.dropout = nn.Dropout(p=0.5)
        #全连接层特征参数
        self.fc1 = nn.Linear(in_features=64*8*8, out_features=128)
        self.fc2 = nn.Linear(in_features=128, out_features=10)

    def forward(self,x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool(x)#输出尺寸16*16

        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool(x)#输出尺寸8*8

        x = x.view(-1, 64*8*8)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

图 2 模型构建函数

4.3 模型训练

表 1 模型训练参数

参数	设置值
批量大小	64
学习率 lr	0.01
训练轮次 epoch	50
优化器	SGD
损失函数	交叉熵 CrossEntropyLoss

由于训练数据集较大，设置 50 个 epoch，使用 SGD 优化器和交叉熵损失函数，为防止过拟合，使用了 L2 正则化。

```
        #训练阶段
        model.train()
        train_loss = 0.0
        correct_train = 0
        total_train = 0

        for idx, (trn_x, trn_y) in enumerate(trn_loader):
            trn_x = trn_x.to(device)
            trn_y = trn_y.to(device)

            sgd.zero_grad()#梯度清零
            trn_y_pred = model(trn_x)
            loss = loss_fn(trn_y_pred, trn_y)
            loss.backward()
            sgd.step()
            # 计算训练准确率
            _, predicted = torch.max(trn_y_pred.data, 1)
            total_train += trn_y.size(0)
            correct_train += (predicted == trn_y).sum().item()

            train_loss += loss.item()

        # 计算平均训练损失和准确率
        avg_train_loss = train_loss / len(trn_loader)
        train_acc = 100 * correct_train / total_train
```

图 3 模型训练函数

```
#模型验证
        model.eval()
        test_loss = 0.0
        correct_test = 0
        total_test = 0

        with torch.no_grad():  # 禁用梯度计算
            for idx, (tst_x, tst_y) in enumerate(tst_loader):
                tst_x = tst_x.to(device)
                tst_y = tst_y.to(device)

                tst_y_pred = model(tst_x)
                loss = loss_fn(tst_y_pred, tst_y)
                test_loss += loss.item()

                # 计算验证准确率
                _, predicted = torch.max(tst_y_pred.data, 1)
                total_test += tst_y.size(0)
                correct_test += (predicted == tst_y).sum().item()

            # 计算平均验证损失和准确率
        avg_test_loss = test_loss / len(tst_loader)
        test_acc = 100 * correct_test / total_test
```

图 4 模型评估函数

4.4 模型测试

最开始测试模型时，使用以下结构处理测试集：

```
for filename in test_files:
    image = Image.open(img_path).convert('RGB')
    image = transform(image).unsqueeze(0).to(device)
    with torch.no_grad():
        output = model(image)
```

图 5 最初的测试集数据处理结构

逻辑简单，但是每张图片单独打开，转换，送入 GPU 处理且没有使用 batch，每张图片重新打开 model（），速度很慢，很长时间都没有处理完成。

因此在修改代码时，使用 ImageFolder 和 Dataset 来构建测试集，用 DataLoader（batch_size=64）将图片批量送入模型，批量预测后在一次性写入结果。

```python
class TestDataset(Dataset):
    def __init__(self, img_dir, transform):
        self.img_dir = img_dir
        self.transform = transform
        self.filenames = sorted([f for f in os.listdir(img_dir) if f.endswith('.png')],
                                key=lambda x: int(os.path.splitext(x)[0]))

    def __len__(self):
        return len(self.filenames)

    def __getitem__(self, idx):
        fname = self.filenames[idx]
        img_path = os.path.join(self.img_dir, fname)
        image = Image.open(img_path).convert('RGB')
        image = self.transform(image)
        return image, fname

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

model = CIFAR10Model().to(device)
model.load_state_dict(torch.load("best_model.pth"))
model.eval()

test_dir = "./test"
dataset = TestDataset(test_dir, transform)
loader = DataLoader(dataset, batch_size=64, shuffle=False)

predictions = []
with torch.no_grad():
    for images, filenames in loader:
        images = images.to(device)
        outputs = model(images)
        _, preds = torch.max(outputs, 1)
        for fname, pred in zip(filenames, preds):
            class_name = class_names[pred.item()]
            predictions.append((fname, class_name))

# 排序并写入
predictions.sort(key=lambda x: int(os.path.splitext(x[0])[0]))
with open("predictions.csv", "w") as f:
    f.write("id,label\n")
    for idx, (_, label) in enumerate(predictions):
        f.write(f"{idx+1},{label}\n")

print("Saved predictions.csv")
```

图 6 修改后的测试函数代码

5 实验结果与分析

5.1 模型训练及评估结果

在 50 个 epoch 的训练过程中，模型的损失值逐步减小，准确率逐渐上升，最终损失值为 0.5685，准确率达到 79.98%。说明模型正常拟合。

在评估过程中，模型损失值和准确率有所起伏。

5.2 分析

```
start training ...
Epoch [1/50] | Train Loss: 2.0861 | Train Acc: 24.97% | Test Loss: 1.8121 | Test Acc: 36.58%
Epoch [2/50] | Train Loss: 1.7703 | Train Acc: 36.38% | Test Loss: 1.6186 | Test Acc: 41.97%
Epoch [3/50] | Train Loss: 1.6180 | Train Acc: 41.76% | Test Loss: 1.4833 | Test Acc: 46.64%
Epoch [4/50] | Train Loss: 1.5167 | Train Acc: 45.27% | Test Loss: 1.4564 | Test Acc: 48.14%
Epoch [5/50] | Train Loss: 1.4474 | Train Acc: 47.58% | Test Loss: 1.3396 | Test Acc: 51.28%
Epoch [6/50] | Train Loss: 1.3897 | Train Acc: 49.64% | Test Loss: 1.2899 | Test Acc: 54.07%
Epoch [7/50] | Train Loss: 1.3433 | Train Acc: 51.61% | Test Loss: 1.2512 | Test Acc: 55.30%
Epoch [8/50] | Train Loss: 1.3017 | Train Acc: 53.53% | Test Loss: 1.2319 | Test Acc: 55.85%
Epoch [9/50] | Train Loss: 1.2612 | Train Acc: 54.96% | Test Loss: 1.1931 | Test Acc: 57.65%
Epoch [10/50] | Train Loss: 1.2263 | Train Acc: 56.28% | Test Loss: 1.2233 | Test Acc: 56.99%
Epoch [11/50] | Train Loss: 1.1936 | Train Acc: 57.63% | Test Loss: 1.1140 | Test Acc: 61.06%
Epoch [12/50] | Train Loss: 1.1637 | Train Acc: 58.76% | Test Loss: 1.1201 | Test Acc: 60.22%
Epoch [13/50] | Train Loss: 1.1278 | Train Acc: 60.17% | Test Loss: 1.1189 | Test Acc: 59.87%
Epoch [14/50] | Train Loss: 1.1027 | Train Acc: 61.01% | Test Loss: 1.0993 | Test Acc: 60.52%
Epoch [15/50] | Train Loss: 1.0716 | Train Acc: 61.97% | Test Loss: 1.1526 | Test Acc: 59.11%
Epoch [16/50] | Train Loss: 1.0472 | Train Acc: 63.18% | Test Loss: 0.9958 | Test Acc: 64.63%
Epoch [17/50] | Train Loss: 1.0230 | Train Acc: 63.97% | Test Loss: 1.0811 | Test Acc: 61.60%
Epoch [18/50] | Train Loss: 1.0045 | Train Acc: 64.67% | Test Loss: 0.9776 | Test Acc: 65.67%
Epoch [19/50] | Train Loss: 0.9797 | Train Acc: 65.63% | Test Loss: 0.9835 | Test Acc: 66.09%
Epoch [20/50] | Train Loss: 0.9603 | Train Acc: 66.32% | Test Loss: 0.9315 | Test Acc: 67.25%
Epoch [21/50] | Train Loss: 0.9390 | Train Acc: 67.01% | Test Loss: 0.9865 | Test Acc: 64.56%
Epoch [22/50] | Train Loss: 0.9260 | Train Acc: 67.72% | Test Loss: 0.9365 | Test Acc: 67.09%
Epoch [23/50] | Train Loss: 0.9067 | Train Acc: 68.13% | Test Loss: 0.9053 | Test Acc: 68.12%
Epoch [24/50] | Train Loss: 0.8873 | Train Acc: 68.79% | Test Loss: 0.9058 | Test Acc: 68.29%
Epoch [25/50] | Train Loss: 0.8711 | Train Acc: 69.35% | Test Loss: 0.9229 | Test Acc: 67.71%
Epoch [26/50] | Train Loss: 0.8558 | Train Acc: 69.90% | Test Loss: 0.9248 | Test Acc: 67.66%
Epoch [27/50] | Train Loss: 0.8411 | Train Acc: 70.46% | Test Loss: 0.8925 | Test Acc: 68.81%
Epoch [28/50] | Train Loss: 0.8278 | Train Acc: 71.03% | Test Loss: 0.8722 | Test Acc: 69.33%
Epoch [29/50] | Train Loss: 0.8158 | Train Acc: 71.31% | Test Loss: 0.8892 | Test Acc: 69.14%
Epoch [30/50] | Train Loss: 0.7983 | Train Acc: 71.99% | Test Loss: 0.9162 | Test Acc: 68.36%
Epoch [31/50] | Train Loss: 0.7807 | Train Acc: 72.69% | Test Loss: 0.8523 | Test Acc: 70.58%
Epoch [32/50] | Train Loss: 0.7722 | Train Acc: 73.06% | Test Loss: 0.8850 | Test Acc: 68.93%
Epoch [33/50] | Train Loss: 0.7585 | Train Acc: 73.49% | Test Loss: 0.8371 | Test Acc: 71.07%
Epoch [34/50] | Train Loss: 0.7510 | Train Acc: 73.68% | Test Loss: 0.9789 | Test Acc: 66.54%
Epoch [35/50] | Train Loss: 0.7387 | Train Acc: 73.95% | Test Loss: 0.8373 | Test Acc: 70.66%
Epoch [36/50] | Train Loss: 0.7169 | Train Acc: 74.76% | Test Loss: 0.8156 | Test Acc: 71.44%
Epoch [37/50] | Train Loss: 0.7145 | Train Acc: 74.93% | Test Loss: 0.8285 | Test Acc: 71.04%
Epoch [38/50] | Train Loss: 0.6968 | Train Acc: 75.59% | Test Loss: 0.8226 | Test Acc: 71.57%
Epoch [39/50] | Train Loss: 0.6861 | Train Acc: 76.09% | Test Loss: 0.8208 | Test Acc: 71.69%
Epoch [40/50] | Train Loss: 0.6774 | Train Acc: 76.30% | Test Loss: 0.8274 | Test Acc: 71.46%
Epoch [41/50] | Train Loss: 0.6660 | Train Acc: 76.52% | Test Loss: 0.8274 | Test Acc: 71.85%
Epoch [42/50] | Train Loss: 0.6485 | Train Acc: 76.86% | Test Loss: 0.8442 | Test Acc: 70.57%
Epoch [43/50] | Train Loss: 0.6430 | Train Acc: 77.37% | Test Loss: 0.8555 | Test Acc: 70.39%
Epoch [44/50] | Train Loss: 0.6292 | Train Acc: 77.75% | Test Loss: 0.8480 | Test Acc: 71.01%
Epoch [45/50] | Train Loss: 0.6189 | Train Acc: 78.12% | Test Loss: 0.8207 | Test Acc: 71.98%
Epoch [46/50] | Train Loss: 0.6089 | Train Acc: 78.58% | Test Loss: 0.8148 | Test Acc: 72.33%
Epoch [47/50] | Train Loss: 0.6007 | Train Acc: 78.81% | Test Loss: 0.8468 | Test Acc: 71.45%
Epoch [48/50] | Train Loss: 0.5881 | Train Acc: 79.23% | Test Loss: 0.8292 | Test Acc: 71.85%
Epoch [49/50] | Train Loss: 0.5780 | Train Acc: 79.62% | Test Loss: 0.8164 | Test Acc: 72.47%
```

图 7 模型训练与评估结果

训练集是模型所了解的数据，模型可以调整来适配训练集数据，在训练的过程中，模型记住了数据，因此能够做到损失稳定下降，准确率逐步上升。

但是，测试集并未参与模型训练过程，模型处理测试集数据时只能依靠模型自身的泛化能力。模型泛化能力不足时，模型能够记住训练集，但无法处理测试集中差异较大的样本，并且训练集中数据的复杂程度不同，训练集中可能难以识别。

损失值与准确率的波动说明模型在“记住数据”与“学会一般规律”之间浮动。

5.3 测试集结果

模型能够正确将结果写入“predictions.csv”文件中，且格式符合要求。

（由于测试集数据较多，报告中仅展示“predictions.csv”文件中的部分结果）

在 kaggle 平台提交的测试结果如下

id	label
1	deer
2	airplane
3	automobile
4	ship
5	bird
6	cat
7	airplane
8	bird
9	bird
10	dog

图 9 模型写入“predictions.csv”文件中的部分内容

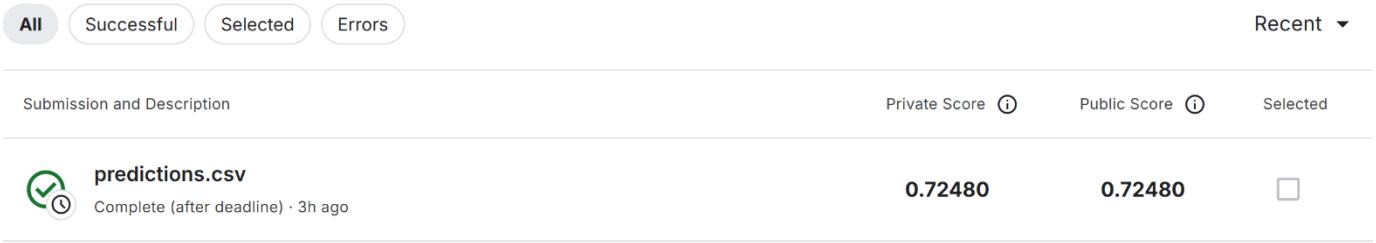

图 8 Kaggle 网站测评结果

这一项目以准确率作为评分结果，我的提交内容评分为 0.72480，与保存的模型准确率 72.47%基本保持一致。

6 实验反思

epoch 较少：通常情况下，设置 epoch 为 50 到 100 之间比较合理，虽然本次实验中设置 epoch 为 50，也能相对较好地达到一定的准确率。由于硬件条件有限，没能尝试更高的 epoch，如果设置 epoch 为 100，可能能达到更好的训练结果。

数据过拟合情况：在最开始尝试的过程中，没有使用结构性正则化和 L2 正则化，导致数据过拟合严重。但是在使用这两个改进措施之后，仍然有过拟合的现象存在。在查阅相关资料后，我认为，还可以加入数据增强，利用 RandomCrop、RandomHorizontalFlip 等函数对原始图像数据做出一系列随机变换，创造更多训练样本，提高模型泛化能力。

batch_size 设置：在本次实验中，设置 batch_size 为 64，迭代次数多但占用内存较小，梯度噪声较大，泛化性可能更好，所需 epoch 数也更多，准确率较高的同时损失曲线波动性更大。若设置为 128，迭代次数减少，收敛时间短，梯度噪声小，稳定性更高能获得更平滑的损失曲线，但准确率吧可能略低。

综上，如果有机会的话，应该尝试不同的 epoch，learning rate，batch_size 组合，获得更好的训练结果。

7 附录

完整代码

model.py

```
import torch
from torch.nn import Module
from torch import nn
import torch.nn.functional as F

#定义卷积神经网络模型
class CIFAR10Model(Module):
    def __init__(self):
        super().__init__()
        #第一个卷积层的输入通道为3(RGB)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        #池化层
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        #结构性正则化
        self.dropout = nn.Dropout(p=0.5)
        #全连接层特征参数
        self.fc1 = nn.Linear(in_features=64*8*8, out_features=128)
        self.fc2 = nn.Linear(in_features=128, out_features=10)

    def forward(self,x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool(x)#输出尺寸16*16

        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool(x)#输出尺寸8*8

        x = x.view(-1, 64*8*8)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

train.py

```
import torch
import torchvision
from torch.nn import CrossEntropyLoss
from torch.utils.data import DataLoader
from torchvision import transforms
from torch.optim import SGD
from model import CIFAR10Model

if __name__ == '__main__':
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    #数据预处理
    #图像转化为张量，并归一化到[-1,1]
    transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5,))])

    #加载数据集
    trn_dataset = torchvision.datasets.CIFAR10(root='./data/train', download=True,transform=transform)
    tst_dataset = torchvision.datasets.CIFAR10(root='./data/train', train=False, download=True,transform=transform)

    #数据加载器
    batch_size = 64
    trn_loader = DataLoader(trn_dataset, batch_size=batch_size, shuffle=True)
    tst_loader = DataLoader(tst_dataset, batch_size=batch_size, shuffle=False)

    #设置损失函数和优化器
    model = CIFAR10Model().to(device) #将模型移到GPU
    sgd = SGD(model.parameters(), lr=0.01, weight_decay=1e-3)
    loss_fn = CrossEntropyLoss()

    #开始训练
    print('start training ...')

    total_epoch = 50
    prev_acc = 0.0
    save_path = "best_model.pth"
    for current_epoch in range(total_epoch):
        #训练阶段
        model.train()
        train_loss = 0.0
        correct_train = 0
        total_train = 0

        for idx, (trn_x, trn_y) in enumerate(trn_loader):
            trn_x = trn_x.to(device)
            trn_y = trn_y.to(device)

            sgd.zero_grad()#梯度清零
            trn_y_pred = model(trn_x)
```

```
            loss = loss_fn(trn_y_pred, trn_y)
            loss.backward()
            sgd.step()
            # 计算训练准确率
            _, predicted = torch.max(trn_y_pred.data, 1)
            total_train += trn_y.size(0)
            correct_train += (predicted == trn_y).sum().item()

            train_loss += loss.item()

        # 计算平均训练损失和准确率
        avg_train_loss = train_loss / len(trn_loader)
        train_acc = 100 * correct_train / total_train
```

```
        #模型验证
        model.eval()
        test_loss = 0.0
        correct_test = 0
        total_test = 0

        with torch.no_grad():  # 禁用梯度计算
            for idx, (tst_x, tst_y) in enumerate(tst_loader):
                tst_x = tst_x.to(device)
                tst_y = tst_y.to(device)

                tst_y_pred = model(tst_x)
                loss = loss_fn(tst_y_pred, tst_y)
                test_loss += loss.item()

                # 计算验证准确率
                _, predicted = torch.max(tst_y_pred.data, 1)
                total_test += tst_y.size(0)
                correct_test += (predicted == tst_y).sum().item()

            # 计算平均验证损失和准确率
        avg_test_loss = test_loss / len(tst_loader)
        test_acc = 100 * correct_test / total_test

    # 打印每个epoch的结果
    print(f'Epoch [{current_epoch + 1}/{total_epoch}] | '
          f'Train Loss: {avg_train_loss:.4f} | Train Acc: {train_acc:.2f}% | '
          f'Test Loss: {avg_test_loss:.4f} | Test Acc: {test_acc:.2f}%')

    # 保存最佳模型
    if test_acc > prev_acc:
        prev_acc = test_acc
        torch.save(model.state_dict(), save_path)

    print(f'Model saved with test accuracy: {test_acc:.2f}%')
```

test.py

```
import os
from PIL import Image
from torch.utils.data import Dataset, DataLoader
import torch
from torchvision import transforms

from model import CIFAR10Model

class TestDataset(Dataset):
    def __init__(self, img_dir, transform):
        self.img_dir = img_dir
        self.transform = transform
        self.filenames = sorted([f for f in os.listdir(img_dir) if f.endswith('.png')],
                                key=lambda x: int(os.path.splitext(x)[0]))

    def __len__(self):
        return len(self.filenames)

    def __getitem__(self, idx):
        fname = self.filenames[idx]
        img_path = os.path.join(self.img_dir, fname)
        image = Image.open(img_path).convert('RGB')
        image = self.transform(image)
        return image, fname

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']
```

```
model = CIFAR10Model().to(device)
model.load_state_dict(torch.load("best_model.pth"))
model.eval()

test_dir = "./test"
dataset = TestDataset(test_dir, transform)
loader = DataLoader(dataset, batch_size=64, shuffle=False)

predictions = []
with torch.no_grad():
    for images, filenames in loader:
        images = images.to(device)
        outputs = model(images)
        _, preds = torch.max(outputs, 1)
        for fname, pred in zip(filenames, preds):
            class_name = class_names[pred.item()]
            predictions.append((fname, class_name))

# 排序并写入
predictions.sort(key=lambda x: int(os.path.splitext(x[0])[0]))
with open("predictions.csv", "w") as f:
    f.write("id,label\n")
    for idx, (_, label) in enumerate(predictions):
        f.write(f"{idx+1},{label}\n")

print("Saved predictions.csv")
```

个人感想

这一次的课程作业应该是我第一次完成一个完整的项目，在完成 project 的过程中，我收获了很多，也碰到了之前课堂学习中没有碰到过的情况。当我第一次完成模型训练评估时，看到显示 100%准确率，我很意外，发现是训练集集混入测试数据后，我又哭笑不得。

回看整个过程，写代码似乎并不成为最大的障碍，最大的困难似乎是加载庞大的数据集、等待了几个小时的训练过程却最终收获一片红色的报错、尝试使用 Kaggle 网站上的 GPU 却发现并不如想象中的快速……

在这门课最开始的时候，我始终觉得很难，上课听不懂，课后的作业没有 AI 帮助根本无法完成，在和同学聊天时候，我说“这门课是我本学期最硬且最听不懂的课”。在完成 project 的过程中，我依旧觉得很困难，依旧是需要 AI 帮助，但是我好像获得了很多，算法的黑箱好像向我露出了一个小小的缝隙。

谢谢老师和助教一学期以来的教学和帮助。