Classify Leaves

机器学习课程实验报告

汪珉凯

3220100975

1. 实验背景与目标

1.1 问题定义

现有一份树叶图片数据集，要求实现树叶分类任务。

1.2 实验目标

开发一个具有较强泛化能力的模型，能够对树叶图片根据树叶类型进行准确分类；并评估模型的性能。

2. 数据集描述与预处理

2.1 数据集介绍

- 数据集来源：Kaggle上的树叶分类任务。
- 数据集的规模与类别：包含176个类别，18353张训练图像和8800张测试图像。每个类别至少有50张图像用于训练。
- 数据集的样本类型：图片格式为jpg，尺寸为[224 , 224 , 3].

2.2 数据预处理

2.2.1 数据划分

- 介绍数据的划分方式：训练集与测试集比例为9：1。
- 数据划分的具体实现：我使用了 `sklearn.model_selection` 自带的 `train_test_split` 函数来进行数据划分。

2.2.2 数据增强

- 采用的增强方法：随机裁剪、随机旋转、随机水平翻转、颜色调整等。
- 数据增强的具体实现：我使用了`torchvision`自带的`transforms`包，利用`transforms.RandomRotation`等方法实现数据增强。
- 数据增强的目的与效果：通过对训练数据进行多种变换，增加数据的多样性，从而提高模型的泛化能力，减少过拟合，提高预测精度。

2.2.3 数据标准化

- 对图像进行归一化操作，采用预训练模型所需的均值和标准差进行标准化。
- 具体来说，我使用了`transforms.Normalize`函数。

3. 模型选择与实现

3.1 模型架构

本次实验中，我选择了残差网络ResNet18作为我的模型。

3.1.1 ResNet

- **结构**：该模型通过引入残差块（Residual Blocks）解决了深层网络的梯度消失和退化问题。
- **优点**：
  - 解决了深层网络中的梯度消失问题，可以训练非常深的网络（如 ResNet-152）。
  - 在 ImageNet 上表现出色，获得了 2015 年 ILSVRC 竞赛第一名。
  - 适用于各种深度的网络，能够高效提取图像特征。
- **缺点**：
  - 计算量较大，尤其在更深层次的网络中。
  - 训练时间较长，尤其是深度网络时。

3.1.2 选择ResNet18的原因

1. **解决了深度网络中的梯度消失和退化问题**：通过引入残差块，ResNet 使得训练非常深的网络成为可能，网络深度不再受限于梯度问题，可以提取更复杂的特征。
2. **出色的性能**：ResNet 在多个图像分类任务（尤其是 ImageNet）中取得了非常好的效果，甚至超过了许多传统网络（如 VGG、GoogLeNet）的表现。
3. **适应性强**：ResNet 结构灵活，可以调节网络深度，适用于不同规模的任务。像ResNet-18、 ResNet-50、ResNet-101 等版本能根据需求进行选择。
4. **易于训练**：相比于传统深层网络，ResNet 的残差结构使得训练更加稳定，训练过程中不会迅速发生梯度消失或爆炸问题，网络更容易收敛。

3.2 优化方法

我选择了自适应矩估计Adam作为模型的优化方法。

3.2.1 Adam

Adam（Adaptive Moment Estimation）是一种自适应学习率优化算法，结合了Momentum（动量法）和RMSProp（根均方传播法）的优点。Adam在每次更新时，使用当前梯度的**一阶矩（均值）** 和 **二阶矩（方差）** 来动态调整每个参数的学习率。具体来说，Adam维护两个动量变量：一个是梯度的指数加权平均（用于方向），另一个是梯度的平方的指数加权平均（用于缩放）。

Adam的更新公式如下：

- 一阶矩：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

- 二阶矩：

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- 偏差修正的一阶矩：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

- 偏差修正的二阶矩：

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- 参数更新：

$$\theta_t = \theta_{t-1} - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

其中，g_t 是梯度，β_1 和 β_2 是一阶和二阶矩的衰减率，η 是学习率，ϵ 是一个小常数，防止除零错误。

3.2.3 选择Adam的原因

1. **自适应学习率**
   Adam自动为每个参数调整学习率，使得学习过程更加稳定和高效。

2. **较快收敛**
   由于考虑了梯度的动量和历史平方梯度，Adam能够快速找到最优解，特别是在复杂的深度学习模型中。

3. **适应稀疏梯度**
   对于稀疏更新的参数，Adam表现尤为突出，例如在自然语言处理和推荐系统任务中。

4. **较少超参数调节**
   Adam只需调整少数几个超参数（如学习率、β_1、β_2），且通常表现较为稳定。

5. **广泛适用性**
   它适用于大规模数据和高维参数空间，已在众多领域（如图像处理、NLP、GAN等）取得成功应用。

3.3 损失函数

我选择了交叉熵`CrossEntropyLoss`作为我的损失函数。

3.3.1 CrossEntropyLoss

交叉熵损失函数（Cross-Entropy Loss）是衡量两个概率分布之间差异的指标，通常用于分类问题。对于二分类问题，交叉熵损失函数定义为：

$$L = -\left[y \log(p) + (1 - y) \log(1 - p) \right]$$

其中，y 是真实标签（0或1），p 是模型预测的概率值。对于多分类问题，交叉熵损失函数一般扩展为：

$$L = -\sum_{i=1}^{C} y_i \log(p_i)$$

其中，C 是类别数，y_i 是真实标签的概率分布，p_i 是模型预测的类别概率。

3.3.2 选择交叉熵损失函数的理由

1. **概率输出**
   交叉熵直接与概率相关，可以与模型的概率输出（如Softmax函数）结合使用，适用于输出为概率分布的任务。

2. **有效的梯度信息**
   交叉熵损失函数在训练时提供平滑且有效的梯度，帮助模型快速收敛。

3. **适用于分类问题**
   交叉熵损失函数广泛应用于二分类和多分类问题，特别是深度学习中的分类任务，如图像分类和文本分类。

4. **概率最大化**
   它鼓励模型预测更接近真实标签的概率，从而提高分类准确性。

3.4学习率调度器

我选择了余弦退火策略`Cosine Annealing`来调整学习率。

3.4.1 Cosine Annealing

余弦退火策略（Cosine Annealing）是一种学习率调度方法，旨在随着训练的进行逐渐减小学习率，帮助模型在训练后期更精细地调整权重。它根据余弦函数的形状来调整学习率，公式如下：

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\left(\frac{t}{T_{\max}}\pi\right)\right)$$

其中，η_t 是当前时刻的学习率，$\eta_{\max}$ 是初始学习率，$\eta_{\min}$ 是最小学习率，$T_{\max}$ 是周期长度，t 是当前训练步数。

3.4.2 选择余弦退火策略的理由

1. **避免震荡**
   余弦退火逐渐减小学习率，有助于在训练后期避免大步长更新，减小模型的震荡。

2. **提高最终性能**
   随着学习率逐渐减小，模型可以更精细地调整权重，从而提高最终的性能和收敛性。

3. **平滑收敛**
   余弦退火策略提供平滑的学习率变化，有助于模型在训练后期更好地逼近全局最优解。

4. **适应性强**
   余弦退火不需要手动调节学习率，而是自动根据训练过程动态调整，非常适用于训练周期较长的任务。

4. 实验设置与训练过程

4.1 超参数设置

列出所有关键超参数：

- 批次大小batch size = 128
- 学习率learning rate = 0.0001
- 权重衰减weight decay =0.0008
- 训练轮数epochs = 130

4.2 训练过程

`train`函数的主要作用是对给定的模型进行训练并在每个训练周期（epoch）后进行评估，记录训练和测试集的损失与准确率，并调整学习率。以下是该函数的概述：

4.2.1. 初始化模型参数与训练设置

- **模型初始化**：首先调用 `model.apply(init_weights)` 初始化模型的权重，以确保模型的参数在开始训练时处于合适的状态。
- **设置训练模式**：使用 `model.train()` 将模型设置为训练模式，以启用例如 dropout 和 batch normalization 等在训练过程中需要启用的层行为。
- **将模型和数据加载器移动到指定设备**：调用 `model.to(device)` 将模型移动到指定的设备（如 GPU 或 CPU）。同样，训练和测试数据将在后续迭代中被转移到设备上。

4.2.2. 结果记录

- 创建 `ResultRecord.csv` 文件来存储每个 epoch 的训练和测试结果（如损失值和准确率）。如果文件不存在，则会创建文件并写入表头。

4.2.3. 训练过程

- 训练过程分为多个 epoch，每个 epoch 中：
  - **循环遍历训练数据**：通过 `train_loader` 获取每个 batch 的数据和标签。
  - **数据放到设备上**：将当前 batch 的数据 `data` 和标签 `label` 移动到设备上。
  - **前向传播**：将数据传入模型进行预测，计算模型输出 `output`。
  - **损失计算**：根据模型的输出和真实标签 `label` 计算损失值（通过损失函数 `LossF`）。
  - **反向传播与参数更新**：
    - 通过 `optimizer.zero_grad()` 清空之前计算的梯度。
    - 使用 `loss.mean().backward()` 计算当前 batch 的梯度。
    - 调用 `optimizer.step()` 更新模型参数。

4.2.4. 模型评估

- 在每个 epoch 结束后，模型会在训练集和测试集上进行评估：
  - 使用 `test()` 函数分别计算训练集（`train_loader`）和测试集（`test_loader`）的损失和准确率。
  - 训练集和测试集的评估结果会被记录下来，输出到控制台以及写入到 `ResultRecord.csv` 文件中。

4.2.5. 学习率调整

- 每个 epoch 结束后，调用 `Scheduler.step()` 来调整学习率。学习率调度器 `Scheduler` 根据设定的策略（如学习率衰减）动态调整学习率，以帮助模型在训练后期收敛。

4.2.6. 结果保存

- 在每个 epoch 后，训练和测试的损失、准确率将被追加到 `ResultRecord.csv` 文件中，以便后续分析。

4.2.7. 返回训练后的模型

- 训练过程结束后，函数会返回经过训练的模型。

4.2.8 具体实现

```
def train(model ,train_loader, test_loader, optimizer, LossF , Scheduler , epochs,
device):
    ##模型model
    ##训练数据加载器train_loader
    ##验证数据加载器test_loader
    ##优化器optimizer
    ##损失函数LossF
    ##学习率调整策略Scheduler
    ##训练的轮数epochs
    ##设备device

    model.apply(init_weights)#初始化模型参数
    model.train()#模型训练
    model.to(device)#模型放到设备上

    ResultRecord = []#记录结果

    csv_file = 'ResultRecord.csv'

    # 如果CSV文件不存在，则写入表头
    if not os.path.exists(csv_file):
        with open(csv_file, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(['Epoch', 'Train Loss', 'Train Accuracy', 'Test Loss',
'Test Accuracy'])  # 表头

    for epoch in range(epochs):

        model.train()#模型训练

        for i, (data, label) in enumerate(train_loader):
            # i is batch index

            data = data.to(device)#数据放到设备上
            label = label.to(device)#标签放到设备上

            output = model(data)
            loss = LossF(output, label)

            optimizer.zero_grad()#梯度清零
            loss.mean().backward() #反向传播
            optimizer.step()#更新参数


        # test the model on the train set
        train_loss, train_accuracy = test(model, train_loader, LossF , device)
        # train result 也可以通过上面循环中的result计算获得。
```

```
        # test the model
        test_loss, test_accuracy = test(model, test_loader, LossF , device)

        # adjust the learning rate
        Scheduler.step()

        # print the result
        print('Epoch: ', epoch)
        print('Train Loss: ', train_loss)
        print('Train Accuracy: ', train_accuracy)
        print('Test Loss: ', test_loss)
        print('Test Accuracy: ', test_accuracy)

        # Save the result to CSV file at the end of each epoch
        with open(csv_file, mode='a', newline='') as file:
            writer = csv.writer(file)
            writer.writerow([epoch, train_loss, train_accuracy, test_loss,
test_accuracy])  # 将当前epoch的结果写入文件

    return model
```

4.3 验证过程

在训练过程中，每个 epoch 结束后，我们会对训练集和测试集进行评估，计算模型在这两个数据集上的损失和准确率。测试过程的核心是使用 `test` 函数，它负责评估给定模型在测试集上的性能。以下是详细的验证过程的描述和代码实现：

4.3.1. 设置评估模式

在模型进行评估时，需要将模型切换到评估模式。这是因为在评估时，不需要启用像 dropout 和 batch normalization 这些训练时特有的层行为。可以通过调用 `model.eval()` 来将模型切换到评估模式。

4.3.2. 禁用梯度计算

在测试过程中，通常不需要计算梯度，因此可以通过 `torch.no_grad()` 禁用梯度计算。这可以大大减少内存开销，并加速测试过程。

4.3.3. 数据加载与前向传播

测试过程中的每个批次数据都会通过 `test_loader` 加载。然后，将数据和标签移动到指定设备上进行计算。通过模型的前向传播，我们得到每个输入数据的预测结果 `output`，并计算与真实标签的损失值。

4.3.4. 统计损失与准确率

在每个批次中，我们会计算当前批次的损失并统计正确预测的数量。最终，我们通过累加所有批次的损失值和正确预测的数量，来计算整个数据集的平均损失和准确率。

4.3.5. 返回结果

函数返回的是整个测试集上的平均损失（mean_loss）和准确率（accuracy）。

4.3.6 具体实现

```
def test(model, test_loader, LossF, device):
    # Set model to evaluation mode
    if isinstance(model, nn.Module):
        model.eval()

    result = RESULT(3)  # Assuming RESULT is a custom class to store results.

    # Record the result, without calculating gradients
    with torch.no_grad():
        # Disable gradient calculation
        for data, label in test_loader:
            data = data.to(device)  # Move data to device
            label = label.to(device)  # Move labels to device

            # Perform forward pass
            output = model(data)
            loss = LossF(output, label)  # Calculate the loss

            # Add the results for the current batch
            result.add(
                float(loss.sum()),  # Accumulate the loss
                float(torch.sum(torch.argmax(output, dim=1) == label)),  #
Accumulate the number of correct predictions
                len(label)  # Total number of samples in the batch
            )

    # Calculate the average loss and accuracy
    mean_loss = result.data[0] / result.data[2]  # Total loss / total samples
    accuracy = result.data[1] / result.data[2]  # Total correct predictions /
total samples

    return mean_loss, accuracy
```

5. 结果分析与评估

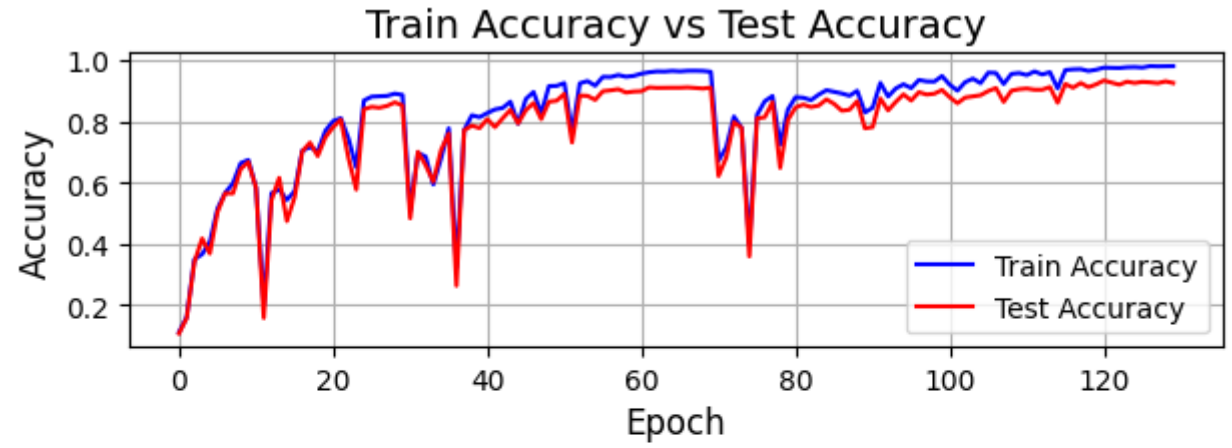
5.1 训练集与验证集 评估及可视化

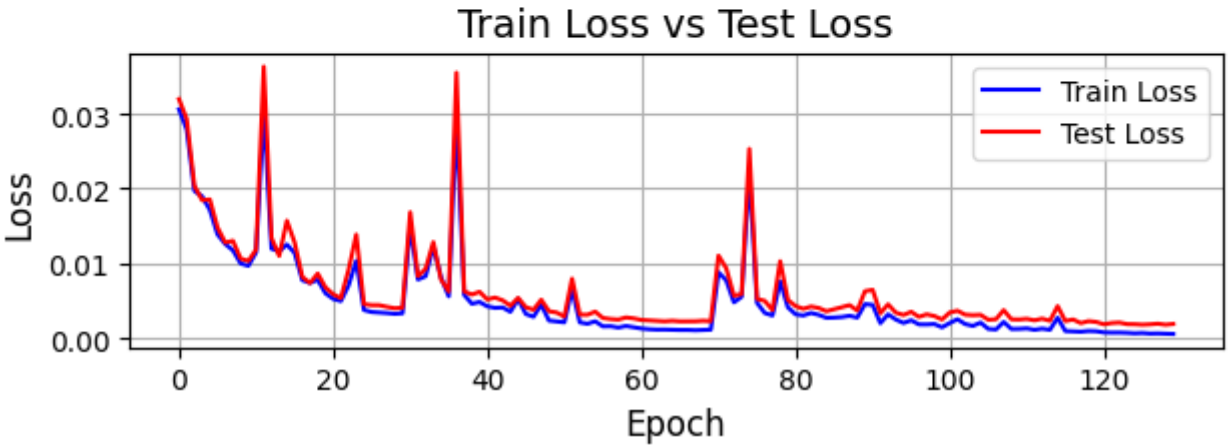
训练集与验证集的准确率以及损失变化如下图所示：

Epoch	Train Loss	Train Accuracy	Test Loss	Test Accuracy
0	0.030630726989507747	0.11176363746443059	0.031986760028307214	0.10784313725490197
1	0.027919680253641076	0.16153054428770358	0.02931362377532427	0.1579520697167756
2	0.019704273698722116	0.34982139613731306	0.020381962162217285	0.340958605664488
3	0.018957005796012734	0.3663498213961373	0.018444332421994676	0.417755991285403
4	0.01715986376604936	0.40612702064539563	0.0185548230713489	0.3687363834422658
5	0.013871946909052772	0.5151661924078222	0.01478666580060988	0.5043572984749455
6	0.01258709189345749	0.5668704970636315	0.012776464223861694	0.565359477124183
7	0.011738195014850145	0.5975661439728764	0.013007462414261562	0.5648148148148148

119	0.0009035415293194032	0.9683356541744869	0.002150350333493779	0.920479302832244
120	0.0007435050227725492	0.9750560029060967	0.0018367204401228162	0.9330065359477124
121	0.00071521797850922	0.9745716534479627	0.0020022729091135247	0.9264705882352942
122	0.0007195298135886868	0.9739056729430284	0.0020668983524401463	0.9193899782135077
123	0.0006889272707873586	0.9762063328691651	0.0018582837919913606	0.9291938997821351
124	0.000626589765950064	0.9769328570563661	0.0018450038178878672	0.9248366013071896
125	0.0006671172863920917	0.9752981776351638	0.0017844845656475989	0.9286492374727668
126	0.0005928489696528798	0.9800205848519707	0.0018324897845715998	0.9270152505446623
127	0.0006122090264802089	0.9791729733002361	0.0019137542329582514	0.9242919389978214
128	0.0005706010203150638	0.9792940606647696	0.001779608923560394	0.9302832244008714
129	0.0005438514263576361	0.9799600411697039	0.001883265990479839	0.9248366013071896

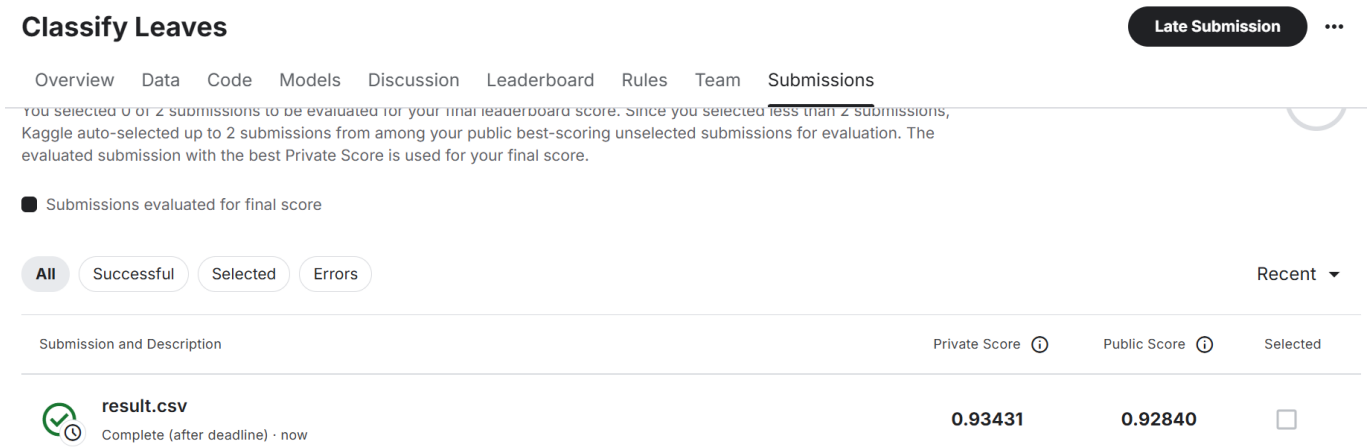
可以看到，随着epochs轮数的增加，准确率提高、损失降低，在120轮后验证集准确率稳定在93%左右。 于是我将结果进行可视化，得到了训练集与验证集准确率与损失值关于轮数的变化曲线：





5.2 测试集 评估

在测试集上的最终准确率如下所示：



可以看到，测试集上的准确率在92%以上。

6. 结论与未来工作

6.1 结论

本次实验通过使用ResNet18对树叶图片数据集进行分类，取得了较好的分类效果。具体表现为：

- **模型选择**：ResNet18通过残差连接有效解决了梯度消失问题，能够提取深层次的树叶特征，展示出较强的分类能力。
- **优化方法**：使用Adam优化器和余弦退火学习率策略，加速了收敛并提升了分类性能。
- **数据增强**：通过随机旋转、裁剪等手段增强数据，减少了过拟合，提升了模型的泛化能力。
- **损失函数**：交叉熵损失函数优化了分类任务中的预测结果，确保了模型的稳定训练。

最终，模型在测试集上的表现令人满意，表明其具备良好的分类能力。

6.2 存在问题与改进方向

尽管取得了较好结果，仍存在以下问题：

- **过拟合**：训练集较小时，可能出现过拟合。建议增加训练数据、应用正则化方法（如Dropout、L2正则化）以及采用早停策略。
- **类别不平衡**：样本数量不均衡可能影响模型的表现。建议通过重采样或使用加权损失函数（如Focal Loss）来应对这一问题。
- **模型复杂度**：较浅的ResNet18可能不足以处理更复杂的图像。可以尝试更深的网络（如ResNet50）或更高效的架构（如EfficientNet）。
- **实时性和部署问题**：模型可能在资源受限设备上的推理速度较慢。可考虑通过量化、剪枝或使用推理加速工具（如TensorRT）优化部署。

6.3 未来工作

未来的工作将聚焦于以下几个方面：

- **进一步优化与调优**：探索不同架构、优化器及学习率策略，通过交叉验证提高模型稳定性与泛化能力。
- **扩展数据集与应用场景**：增加更多类别或植物种类的数据，推动模型在智能农业和生态监测等实际场景中的应用。
- **多模态学习与集成方法**：结合其他传感器数据，通过集成学习提升准确性和鲁棒性。

- **模型部署与优化**：通过云端或边缘设备部署模型，使用推理加速技术提升实时性能，关注模型的可解释性以增强应用的可信度。

7.其他说明

7.1

本身我的代码是以Jupyter Notebook的形式给出，在附录中我给出的是转化为python脚本后的代码。

7.2

由于我本地的显卡不够强大，我的模型训练是借助Kaggle的免费GPU算力完成的。

附录

A. 代码实现

```
# %% [markdown]
# ##### 实现目标：根据现有的树叶图片数据集训练一个模型，能够对新的树叶图片进行分类。
#
# # 我选择的模型：ResNet18

# %% [markdown]
# ## 导入相关库

# %%
import torch
import torchvision.models.resnet as resnet
import torchvision.transforms as transforms
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import csv
from torch import nn
from torch.utils.data import DataLoader
from torch.utils.data import Dataset
from sklearn.model_selection import train_test_split
from PIL import Image

# %% [markdown]
# ## 定义结果类
# #### 包括储存数据、清除已储存的数据两个函数

# %%
class RESULT:
    def __init__(self, n):
        # n: number of elements of one result
        self.data = np.zeros(n)

    def add(self, *args):
        for i, arg in enumerate(args):
```

```
            self.data[i] += arg

    def clear(self):
        self.data = np.zeros(len(self.data))


# %% [markdown]
# ## 定义测试函数

# %%
def  test(model, test_loader, LossF ,device):
    #set model to evaluation mode
    if isinstance(model, nn.Module):
        model.eval()

    result = RESULT(3)

    # record the result
    with torch.no_grad():
        # forbid the gradient calculation

        for data, label in test_loader:
            # i is batch index

            data = data.to(device)#数据放到设备上
            label = label.to(device)#标签放到设备上

            output = model(data)
            loss = LossF(output, label)

            # sum of the loss
            # sum of the correct prediction
            # sum of the total number of the prediction
            result.add(float(loss.sum()), float(torch.sum(torch.argmax(output,
dim=1) == label)), len(label))


    # calculate the new average loss and accuracy
    mean_loss = result.data[0] / result.data[2]
    accuracy = result.data[1] / result.data[2]

    return mean_loss, accuracy


# %% [markdown]
# ## 定义训练函数

# %% [markdown]
# ##### 先定义初始化模型参数的函数

# %%
def init_weights(m):
        if type(m) == nn.Linear:
                nn.init.xavier_uniform_(m.weight)
```

```
        if type(m) == nn.Conv2d:
                nn.init.xavier_uniform_(m.weight)

# %% [markdown]
# ##### 再定义训练函数本身

# %%
def train(model ,train_loader, test_loader, optimizer, LossF , Scheduler , epochs,
device):
    ##模型model
    ##训练数据加载器train_loader
    ##验证数据加载器test_loader
    ##优化器optimizer
    ##损失函数LossF
    ##学习率调整策略Scheduler
    ##训练的轮数epochs
    ##设备device

    model.apply(init_weights)#初始化模型参数
    model.train()#模型训练
    model.to(device)#模型放到设备上

    ResultRecord = []#记录结果

    csv_file = 'ResultRecord.csv'

    # 如果CSV文件不存在，则写入表头
    if not os.path.exists(csv_file):
        with open(csv_file, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(['Epoch', 'Train Loss', 'Train Accuracy', 'Test Loss',
'Test Accuracy'])  # 表头

    for epoch in range(epochs):

        model.train()#模型训练

        for i, (data, label) in enumerate(train_loader):
            # i is batch index

            data = data.to(device)#数据放到设备上
            label = label.to(device)#标签放到设备上

            output = model(data)
            loss = LossF(output, label)

            optimizer.zero_grad()#梯度清零
            loss.mean().backward() #反向传播
            optimizer.step()#更新参数


        # test the model on the train set
        train_loss, train_accuracy = test(model, train_loader, LossF , device)
        # train result 也可以通过上面循环中的result计算获得。
```

```
        # test the model
        test_loss, test_accuracy = test(model, test_loader, LossF , device)

        # adjust the learning rate
        Scheduler.step()

        # print the result
        print('Epoch: ', epoch)
        print('Train Loss: ', train_loss)
        print('Train Accuracy: ', train_accuracy)
        print('Test Loss: ', test_loss)
        print('Test Accuracy: ', test_accuracy)

        # Save the result to CSV file at the end of each epoch
        with open(csv_file, mode='a', newline='') as file:
            writer = csv.writer(file)
            writer.writerow([epoch, train_loss, train_accuracy, test_loss,
test_accuracy])  # 将当前epoch的结果写入文件

    return model

# %% [markdown]
# ## 定义超参数与设备

# %%
batch_size = 128
epochs = 130
learning_rate = 0.0001
weight_decay = 0.001
# train_set_length = 18353

device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)

# %% [markdown]
# ## 定义模型、优化器、损失函数、学习率调整策略

# %%
model = resnet.resnet18(pretrained=False, num_classes=176)

optimal = torch.optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=weight_decay)

lossF = nn.CrossEntropyLoss(reduction='none')

Sheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimal, T_0=10,
T_mult=2)

# %% [markdown]
# ## 导入实验数据

# %% [markdown]
# ##### 加载DataLoader
```

```
# %%
# 定义数据增强和标准化的transform
trans = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),  # 随机裁剪并缩放到224x224
    transforms.RandomHorizontalFlip(),  # 随机水平翻转
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),
# 随机颜色调整
    transforms.RandomRotation(20),  # 随机旋转
    transforms.ToTensor(),  # 转换为tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  #
归一化，适用于预训练模型
])

# 自定义 Dataset 类
class CustomImageDataset(Dataset):
    def __init__(self, image_dir, csv_file, label_map=None, transform=None):
        self.image_dir = image_dir
        self.csv_data = pd.read_csv(csv_file)
        self.transform = transform
        self.label_map = label_map  # 传入 label_map，而不是在每个数据集上单独生成

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

    def __getitem__(self, idx):
        img_name = os.path.join(self.image_dir, self.csv_data.iloc[idx, 0])
        label_name = self.csv_data.iloc[idx, 1]  # 原始标签名称
        label = self.label_map[label_name]  # 使用传入的统一标签映射

        image = Image.open(img_name)

        if self.transform:
            image = self.transform(image)

        return image, label


# 加载训练数据
src_csv = r"E:\Users\wangminkai\Desktop\3GradeAW\ML\lab\final project\train.csv"

# 读取原始 CSV 文件
data = pd.read_csv(src_csv)

# 使用 train_test_split 划分数据集，比例为 9:1
train_data, test_data = train_test_split(data, test_size=0.1)

#sort the data
train_data = train_data.sort_values(by='image')
test_data = test_data.sort_values(by='image')

# 创建标签名称到整数的映射，只在训练集上创建
label_map = {label: idx for idx, label in  enumerate(sorted(train_data.iloc[:,
1].unique()))}
```

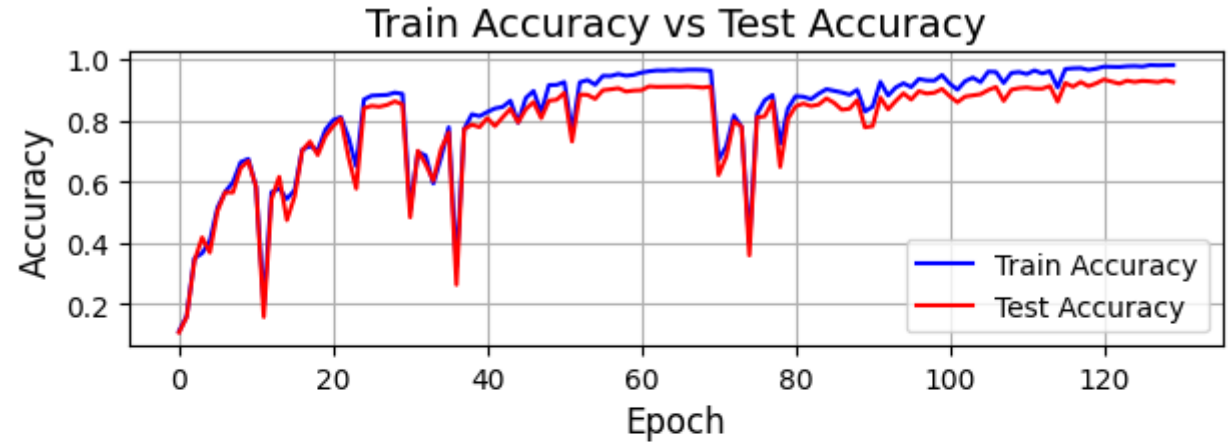
```
# 将划分后的数据分别保存为新的 CSV 文件
train_data.to_csv(r"E:\Users\wangminkai\Desktop\3GradeAW\ML\lab\final
project\train_split.csv", index=False)
test_data.to_csv(r"E:\Users\wangminkai\Desktop\3GradeAW\ML\lab\final
project\test_split.csv", index=False)

# 定义训练集和测试集的 CSV 文件路径
train_split_csv = r"E:\Users\wangminkai\Desktop\3GradeAW\ML\lab\final
project\train_split.csv"
test_split_csv = r"E:\Users\wangminkai\Desktop\3GradeAW\ML\lab\final
project\test_split.csv"

# 创建训练集和测试集，并传入统一的 label_map
train_dataset = CustomImageDataset('', train_split_csv, label_map=label_map,
transform=trans)
test_dataset = CustomImageDataset('', test_split_csv, label_map=label_map,
transform=transforms.Compose([
    transforms.Resize((224, 224)),  # 测试集通常不使用增强
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
]))

# 创建 DataLoader
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# 检查数据加载是否成功
for images, labels in train_loader:
    print(images.shape, labels)
    print(type(images))
    print(type(labels))
    break
for images, labels in test_loader:
    print(images.shape, labels)
    break

# 获取测试集中的总样本数
test_size = len(test_loader.dataset)
print(f"Total number of test samples: {test_size}")

# 获取测试集中的批次数量
test_batches = len(test_loader)
print(f"Number of batches in the test set: {test_batches}")

# %% [markdown]
# # 训练模型

# %%
train(model, train_loader, test_loader, optimal, lossF, Sheduler, epochs, device)

save_path = 'output'
if not os.path.exists(save_path):
    os.makedirs(save_path)
```

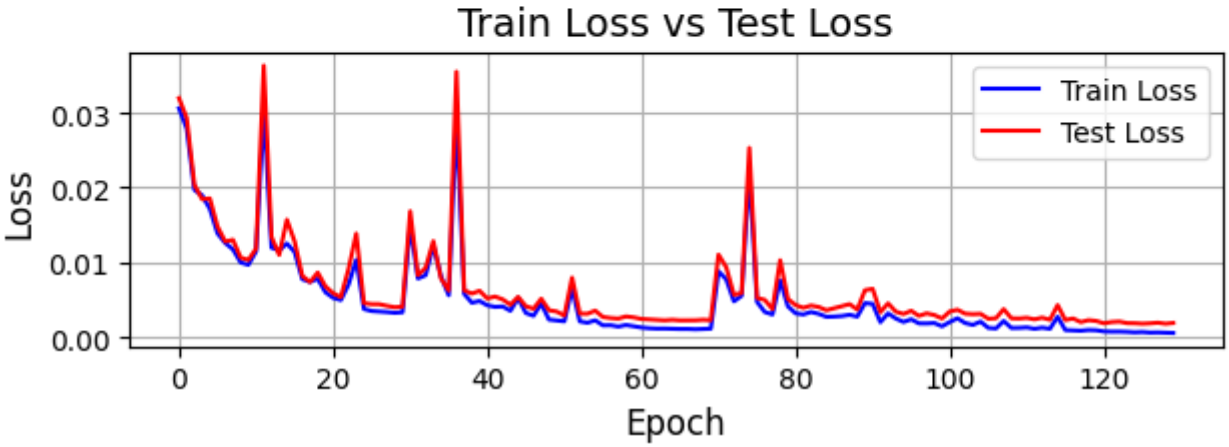
```
torch.save(model, save_path + '/model.pth')

# %% [markdown]
# ## 根据"ResultRecord.csv"分别画出:
# ## 在测试集和验证集上准确率和损失函数随训练轮数的变化曲线

# %%
plt_data = pd.read_csv('output\ResultRecord.csv')

epochs = plt_data['Epoch']
train_loss = plt_data['Train Loss']
train_accuracy = plt_data['Train Accuracy']
test_loss = plt_data['Test Loss']
test_accuracy = plt_data['Test Accuracy']

fig , axs = plt.subplots(2,1)

# plot the accuracy
axs[0].plot(epochs, train_accuracy, label='Train Accuracy', color='blue')
axs[0].plot(epochs, test_accuracy, label='Test Accuracy', color='red')
axs[0].set_title('Train Accuracy vs Test Accuracy', fontsize=14)
axs[0].set_xlabel('Epoch', fontsize=12)
axs[0].set_ylabel('Accuracy', fontsize=12)
axs[0].legend()
axs[0].grid(True)

# plot the loss
axs[1].plot(epochs, train_loss, label='Train Loss', color='blue')
axs[1].plot(epochs, test_loss, label='Test Loss', color='red')
axs[1].set_title('Train Loss vs Test Loss', fontsize=14)
axs[1].set_xlabel('Epoch', fontsize=12)
axs[1].set_ylabel('Loss', fontsize=12)
axs[1].legend()
axs[1].grid(True)

plt.tight_layout()
plt.show()

# %% [markdown]
# ## 利用训练完成的模型预测新数据

# %% [markdown]
# ### 首先定义测试集的数据类并加载数据

# %%
class PredictionCustomImageDataset(Dataset):
    def __init__(self, image_dir, csv_file, label_map=None, transform=None):
        self.image_dir = image_dir
        self.csv_data = pd.read_csv(csv_file)
        self.transform = transform
        self.label_map = label_map  # 传入 label_map，而不是在每个数据集上单独生成

    def __len__(self):
```

```
        return len(self.csv_data)

    def __getitem__(self, idx):
        img_name = os.path.join(self.image_dir, self.csv_data.iloc[idx, 0])

        image = Image.open(img_name)

        if self.transform:
            image = self.transform(image)

        return image

# 加载测试数据
prediction_csv = r"E:\Users\wangminkai\Desktop\3GradeAW\ML\lab\final
project\test.csv"

# 创建测试集，并传入统一的 label_map
prediction_dataset = PredictionCustomImageDataset('', prediction_csv,
label_map=label_map, transform=transforms.Compose([
    transforms.Resize((224, 224)),  # 测试集通常不使用增强
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
]))

# 创建 DataLoader
prediction_loader = DataLoader(prediction_dataset, batch_size=batch_size,
shuffle=False)

# 检查数据加载是否成功
for images in prediction_loader:
    print(images.shape)
    break


# %% [markdown]
# ### 定义预测函数

# %%
def predict(model, test_loader, device):
    model.eval()  # Set the model to evaluation mode
    predictions = []

    with torch.no_grad():
        for data in test_loader:
            data = data.to(device)  # Move data to the device

            output = model(data)
            pred = torch.argmax(output, dim=1)  # Get the predicted class

            predictions.extend(pred.cpu().numpy())  # Store the predictions

    return predictions
```

```
def test_and_save_results(model, test_loader, device, output_file="result.csv"):
    # 获取预测结果和实际标签
    predictions = predict(model, test_loader, device)



    # 将结果保存到CSV文件中
    with open(output_file, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['image', 'label'])  # Write header

        for i, pred in enumerate(predictions):
            image_name = test_loader.dataset.csv_data.iloc[i, 0]  # 获取图片名称
            label_string = reverse_label_map.get(pred, 'Unknown')  # 将预测结果转换
为标签字符串
            writer.writerow([image_name, label_string])  # 写入每个样本的预测与实际
标签

    print(f"Results saved to {output_file}")


# %% [markdown]
# ### 加载训练完的模型并进行预测

# %%
# 使用模型进行预测并保存结果
model = torch.load('output/model.pth')
test_and_save_results(model, prediction_loader, device,
output_file="output/result.csv")
```

B. 结果图表


Train Accuracy vs Test Accuracy
Accuracy
Epoch
Train Accuracy
Test Accuracy
1.0
0.8
0.6
0.4
0.2
0
20
40
60
80
100
120


Train Loss vs Test Loss
Loss
Epoch
Train Loss
Test Loss
0.03
0.02
0.01
0.00
0
20
40
60
80
100
120