人工智能大作业报告

1. 介绍

a. 手写数字识别

LeNet5 + MNIST被誉为深度学习领域的“Hello world”。本实验主要介绍使用MindSpore在MNIST手写数字数据集上开发和训练一个LeNet5模型，并验证模型精度。

b. 垃圾分类识别

随着人们对于自然的保护，垃圾处理成为人们日常生活中必不可少的一步。深度学习计算中，从头开始训练一个实用的模型通常非常耗时，需要大量计算能力。常用的数据如OpenImage、ImageNet、VOC、COCO等公开大型数据集，规模达到几十万甚至超过上百万张。网络和开源社区上通常会提供这些数据集上预训练好的模型。大部分细分领域任务在训练网络模型时，如果不使用预训练模型而从头开始训练网络，不仅耗时，且模型容易陷入局部极小值和过拟合。因此大部分任务都会选择预训练模型，在其上做微调（也称为Fine-Tune）。本实验以MobileNetV2+垃圾分类数据集为例，主要介绍如在使用MindSpore在CPU/GPU平台上进行Fine-Tune。

2. 原理介绍

a. 残差神经网络

残差神经网络的主要贡献是发现了“退化现象（Degradation）”，并针对退化现象发明了“快捷连接（Shortcut connection）”，极大的消除了深度过大的神经网络训练困难问题。

与传统的机器学习相比，深度学习的关键特征在于网络层数更深、非线性转换（激活）、自动的特征提取和特征转换，其中，非线性转换是关键目标，它将数据映射到高纬空间以便于更好的完成“数据分类”。随着网络深度的不断增大，所引入的激活函数也越来越多，数据被映射到更加离散的空间，此时已经难以让数据回到原点（恒等变换）。退化现象让我们对非线性转换进行反思，非线性转换极大的提高了数据分类能力，但是，随着网络的深度不断的加大，我们在非线性转换方面已经走的太远，竟然无法实现线性转换。显然，在神经网络中增加线性转换分支成为很好的选择。

ResNet是既有非线性转换、又有线性转换。

b. 学习率衰减策略

学习率是指导我们该如何通过损失函数的梯度调整网络权重的超参数，通过设置学习率控制参数更新的速度。

我们在做神经网络时，参数权重是通过不断的训练得出的，那么这个参数每一次增加或者减少的数值就是通过学习率来控制的。

学习率过小时，学习速度慢，而且容易出现过拟合，收敛速度慢，一般用在训练了一定次数之后使用

学习率过大时，学习速度快，但是容易出现震荡问题，使得训练值在最优值左右来回摆动，当学习率设为1时，效果就是参数在两个固定值来回摆动。而且容易造成参数爆炸增长或减小现象。

i. 指数衰减（exponential decay）

指数衰减：学习率以指数的形式进行衰减

```python
decay_type == "exponential":
                exponential_decay = (0.4)**(total_steps/decay_steps)
                lr = (lr_max - lr_end) * exponential_decay + lr_end
```

ii. 余弦衰减（cosine decay）

余弦衰减：学习率以cosine 函数曲线进行进行衰减.

```python
cosine_decay = 0.5 * (1 + math.cos(math.pi * (i - warmup_steps) / decay_steps))
lr = (lr_max - lr_end) * cosine_decay + lr_end
```

iii. 多项式衰减（polynomial decay）

多项式衰减：调整学习率的衰减轨迹以多项式对应的轨迹进行。

```python
frac = 1.0 - float(i - warmup_steps) / (total_steps - warmup_steps)
lr = (lr_max - lr_end) * (frac * frac) + lr_end
```

3. 具体实现

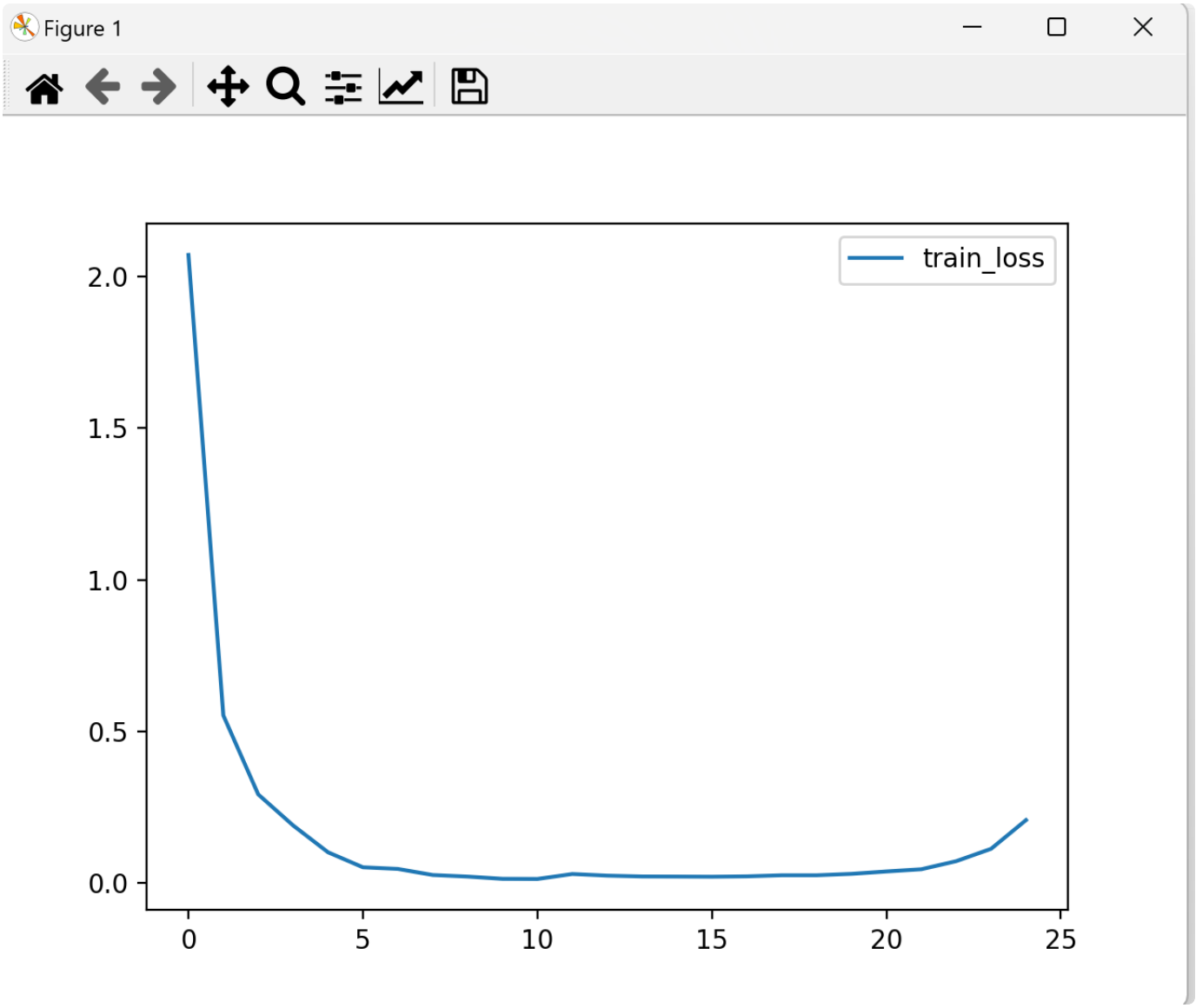
a. 垃圾分类的具体调参过程

i. epochs

迭代步数，生成模型的数量，关系到模型的拟合问题。该值过高会导致过拟合，过低会导致欠拟合

可以根据loss曲线进行分析。

如下图



可以很明显的发现，超过20之后，loss曲线开始上升，说明过拟合，需要降低。25发现开始过拟合，调节参数，但后面发现该参数对于准确度没有实质性的提高

ii. decay_type

该参数选择学习率衰减方式，有三种衰减方式，分别是cosine、square和exponential。本小组的创新点在于使用学习率指数衰减的方式，指数衰减的公式为

$$decayed\_learning\_rate = learning\_rate * decay\_rate^{\frac{global\_step}{decay\_steps}}$$

$$decayed\_learning\_rate = learning\_rate * decay\_rate^{\frac{global\_step}{deacy\_steps}}$$

具体实现代码如下

```python
elif decay_type == "exponential":
                exponential_decay = (0.4)**(total_steps/decay_steps)
                lr = (lr_max - lr_end) * exponential_decay + lr_end
```

衰减率是一个可以调节的参数，本小组经过调节发现，参数在0.3–0.5比较合适，最终确定参数为0.4

iii. momentum

该参量直接使用实验报告中的推荐参数0.9即可，效果较为良好，将本小组的准确度在初始阶段有了较大的提高

iv. weight_decay

对于准确度的提升有较大的帮助，使准确度迅速提高。weight_decay尽可能小会比较合适。

v. reduction

reduction 池化方式在本项目中有mean和max两种方式，即 AvgPooling or MaxPooling.

最大池化（Max Pooling）是将输入的图像划分为若干个矩形区域，对每个子区域输出最大值。即，取局部接受域中值最大的点。

平均池化（Average Pooling）为取局部接受域中值的平均值。

池化过程在一般卷积过程后。池化（pooling） 的本质，其实就是采样。Pooling 对于输入的 Feature Map，选择某种方式对其进行降维压缩，以加快运算速度。

池化的作用：

（1）保留主要特征的同时减少参数和计算量，防止过拟合。

（2）invariance(不变性)，这种不变性包括translation(平移)，rotation(旋转)，scale(尺度)。

Pooling 层说到底还是一个特征选择，信息过滤的过程。也就是说我们损失了一部分信息，这是一个和计算性能的一个妥协，随着运算速度的不断提高，我认为这个妥协会越来越小。

vi. 失败的尝试

1. 添加全连接层

加入全连接层

```
hidden_units_1 = 512
hidden_units_2 = 512
hidden_units_3 = 256
self.dense1 = nn.Dense(input_channel, hidden_units_1, weight_init=HeNormal(), has_bias=False)
self.dense2 = nn.Dense(hidden_units_1, num_classes, weight_init=HeNormal(), has_bias=False)
# self.dense3 = nn.Dense(hidden_units_2, hidden_units_3, weight_init=HeNormal(), has_bias=False)
# self.dense4 = nn.Dense(hidden_units_3, num_classes, weight_init=HeNormal(), has_bias=False)
self.relu = nn.ReLU()
self.sigmoid = nn.Sigmoid()
self.softmax = nn.Softmax()
# self.bn = nn.BatchNorm2d(num_classes)
self.bn = nn.BatchNorm1d(num_classes)
self.dropout = nn.Dropout(p=dropout_rate)
if activation == "Sigmoid":
    self.activation = nn.Sigmoid()
    self.need_activation = True
elif activation == "Softmax":
    self.activation = nn.Softmax()
    self.need_activation = True
else:
    self.need_activation = False
```

发现loss也保持不变不下降，未实现收敛

2. 手动加入激活函数

将def __init__(self, input_channel=1280, hw=7, num_classes=1000, reduction='mean', activation="none"):

中的activation 改为 相应的激活函数，发现loss保持不变

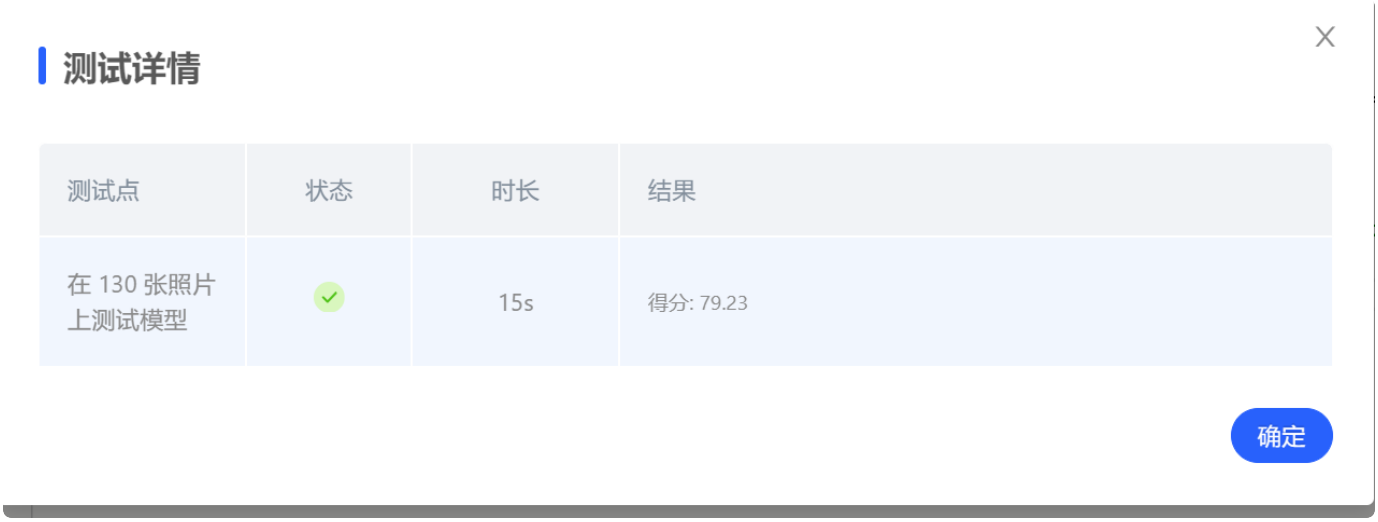
```
epoch: 1, time cost: 0.7754170894622803, avg loss: 3.2580950260162354
epoch: 2, time cost: 0.5906636714935303, avg loss: 3.2580950260162354
epoch: 3, time cost: 0.6129188537597656, avg loss: 3.2580950260162354
epoch: 4, time cost: 0.611717939376831, avg loss: 3.2580950260162354
epoch: 5, time cost: 0.5879716873168945, avg loss: 3.2580950260162354
epoch: 6, time cost: 0.5930564403533936, avg loss: 3.2580950260162354
epoch: 7, time cost: 0.5875554084777832, avg loss: 3.2580950260162354
epoch: 8, time cost: 0.6340069770812988, avg loss: 3.2580950260162354
epoch: 9, time cost: 0.5920095443725586, avg loss: 3.2580950260162354
epoch: 10, time cost: 0.5755884647369385, avg loss: 3.2580950260162354
```

尝试失败

vii. 更改模型评估的标准

在多次尝试后发现，虽然acc可以达到很高的值，但是在130中的提交的分数在76到79徘徊，说明模型的评价的方式出现偏差，需要重新定义模型的评估标准。

4. 结果



5. 误差分析

6. 几个失败的尝试

a. 搭建经典卷积神经网络

i. 尝试的方向

1. 归一化（normalization）

Normalization 有很多种，但是它们都有一个共同的目的，那就是把输入转化成均值为 0 方差为 1 的数据。我们在把数据送入激活函数之前进行 normalization（归一化），因为我们不希望输入数据落在激活函数的饱和区。可以防止梯度爆炸或弥散、可以提高训练时模型对于不同超参（学习率、初始化）的鲁棒性、可以让大部分的激活函数能够远离其饱和区域。

Batch Normalization (BN) 是最早出现的，也通常是效果最好的归一化方式。feature map： 包含 N 个样本，每个样本通道数为 C，高为 H，宽为 W。对其求均值和方差时，将在 N、H、W上操作，而保留通道 C 的维度。具体来说，就是把第1个样本的第1个通道，加上第2个样本第1个通道 ..... 加上第 N 个样本第1个通道，求平均，得到通道 1 的均值（注意是除以 N×H×W 而不是单纯除以 N，最后得到的是一个代表这个 batch 第1个通道平均值的数字，而不是一个 H×W 的矩阵）。求通道 1 的方差也是同理。对所有通道都施加一遍这个操作，就得到了所有通道的均值和方差。

2. 优化器选择

深度学习当中，通常模型比较大，参数较多，训练所需要数据的量也很大，得益于 GPU 加速计算的方式，近几年大放异彩。由于模型参数很多，参数空间很大，如何快速寻找到一个参数空间中相对较好的局部最优点就至关重要，由于模型的非线性和高维度，通过梯度下降的方式去优化一个模型参数成为主流。我们主要采用了一些两种优化器。

2.1. SGD+Momentum

- 使用动量(Momentum)的随机梯度下降法(SGD)，主要思想是引入一个积攒历史梯度信息动量来加速SGD。
- 从训练集中取一个大小为n的小批量{X(1),X(2),...,X(n)}样本，对应的真实值分别为Y(i)，则Momentum优化表达式为：

$$\begin{cases} v_t = \alpha v_{t-1} + \eta_t \Delta J(W_t, X^{(i_s)}, Y^{(i_s)}) \\ W_{t+1} = W_t - v_t \end{cases}$$

- 其中，v_t表示t时刻积攒的加速度。α表示动力的大小，一般取值为0.9（表示最大速度10倍于SGD）。$\Delta J(W_t, X^{(is)}, Y^{(is)})$含义见SGD算法。$W_t$表示$t$时刻模型参数。
- 动量主要解决SGD的两个问题：一是随机梯度的方法（引入的噪声）；二是Hessian矩阵病态问题（可以理解为SGD在收敛过程中和正确梯度相比来回摆动比较大的问题）。
- 理解策略为：由于当前权值的改变会受到上一次权值改变的影响，类似于小球向下滚动的时候带上了惯性。这样可以加快小球向下滚动的速度。

2.2. Adam

首先，Adam 中动量直接并入了梯度一阶矩（指数加权）的估计。其次，相比于缺少修正因子导致二阶矩估计可能在训练初期具有很高偏置的 RMSProp，Adam 包括偏置修正，修正从原点初始化的一阶矩（动量项）和（非中心的）二阶矩估计。

Adam算法的数学描述：

$$\begin{cases} m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t \\ v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2 \\ \hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\ W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t \end{cases}$$

3. 正则化方法

3.1. 我们采用了L2正则化的方式，即权值衰减（regularization）:在代价函数后面再加上一个正则化项。正则化项是指所有参数 w 的平方的和，除以训练集的样本大小 n，并乘以一个自定义的衰减系数。

L2正则化是一种减小模型复杂度的技术，它可以防止模型的参数过大，从而避免过拟合的问题。L2正则化的方法是在模型的损失函数中加上一个和参数平方和成正比的惩罚项，这样就会使参数趋于0，但不会变成0。L2正则化也可以从概率的角度理解，即假设参数服从高斯分布，然后求解最大后验概率。L2正则化可以使模型更加平滑，减少噪声的影响，提高模型的泛化能力。

3.2. 此外，我们还使用了dropout层。

Dropout的作用是防止神经网络过拟合，提高模型的泛化能力。它的原理是在训练过程中，随机地让一些神经元停止工作，从而减少神经元之间的依赖和共适应，使得每个神经元都能学习到更加鲁棒的特征。Dropout相当于对多个不同的子网络进行平均，可以降低模型的复杂度和方差，增加模型的稳定性。Dropout在训练和测试时的操作是不同的，训练时需要对激活值进行缩放或者权重进行调整，而测试时不需要做额外的处理。Dropout是一种简单而有效的正则化方法，它可以应用于各种类型的神经网络中。

4. 数据增强

减少过拟合的一种思路是增加训练集中的样本数量，尤其在训练深层神经网络的过程中，需要大量的训练数据才能得到比较理想的结果。但是增加样本是需要花费较多资源去搜集和标注更多的样本，因此可以基于现有的有限样本，可以通过数据增强（Data Augmentation）来增强数据，避免过拟合。

图形数据的增强主要是通过对图像进行转变、引入噪声等方法来增加数据的多样性，生成的假训练数据虽然无法包含像全新数据那么多的信息，但是代价几乎为零。常用增强的方法有：

- 旋转：将图像按顺时针或者逆时针方向随机旋转一定角度；
- 翻转：将图像沿水平或垂直方向随机翻转一定角度；
- 缩放：将图像沿水平或垂直方向平移一定步长；

- 加噪声：加入随机噪声；
- 一些有弹性的畸变（elastic distortions）；
- 截取（crop）原始图片的一部分。

5. 标签平滑

在数据增强中，是通过给样本特征加入随机噪声来避免过拟合的，而标签平滑是通过给样本的标签（输出）引入噪声来避免过拟合。假设训练数据集中，有些样本的标签是被错误标注的，那么最小化这些样本上的损失函数会导致权重往错误的方向去学习，从而导致过拟合。这时如果使用 softmax 分类器并采用交叉熵损失函数，那么最小化损失函数会使得正确类和其他类之间的权重差异变得非常大。

在我们的数据集中，图片的标签是准确的，且我一开始并没有给图片加噪声，所以在卷积神经网络中我并没有采用这种方式，而在残差神经网络中我使用了这种方式，具体实现在下一节中提到。

6. 数据迁移

我们从 ResNet-50 着手，这是一种由 50 层预先训练好的网络层构成的深度学习模型，其在识别猫狗方面有着很高的准确率。在这个神经网络中，各层被用来识别轮廓、曲线、线条和其它可以用来识别这些动物的特征点。这些层需要大量的标记训练数据，借助这些层来开发执行新任务的神经网络可以节省很多时间。

我们想将一个神经网络应用到一个没有充足数据的新领域当中，所以希望借助预先训练的数据集迁移到我们的任务中。所以我们删除了神经网络的最后一层（用于预测），将其替换成预测垃圾类别的层。

我们还使用了别的网络，这将在下一节中详细说明。

ii. 具体代码实现

导入必要的模块 Python

```python
import torch
from torch import nn, optim
from torch.utils.data import DataLoader
from torchvision import transforms, datasets, models
import time
import random
from matplotlib import pyplot as plt
import os
import math
from PIL import Image
from torch.optim.lr_scheduler import _LRScheduler
import torch.nn.functional as F
import torchvision
import numpy as np
```

定义加载数据集的类 Python

```python
class CustomImageFolder(torch.utils.data.Dataset):
    def __init__(self, img_dir, transform=None):
        self.img_dir = img_dir
        self.transform = transform
        self.classes = os.listdir(img_dir)
        self.classes.sort()
        self.imgs = [os.path.join(root, name)
                     for root, dirs, files in os.walk(img_dir)
                     for name in files]
        self.class_to_idx = {'00_00': 0, '00_01': 1, '00_02': 2, '00_03': 3, '00_04': 4, '00_05': 5, '00_06': 6, '00_07': 7,
                             '00_08': 8, '00_09': 9, '01_00': 10, '01_01': 11, '01_02': 12, '01_03': 13, '01_04': 14,
                             '01_05': 15, '01_06': 16, '01_07': 17, '02_00': 18, '02_01': 19, '02_02': 20, '02_03': 21,
                             '03_00': 22, '03_01': 23, '03_02': 24, '03_03': 25}
        self.idx_to_class = {0: 'Plastic Bottle', 1: 'Hats', 2: 'Newspaper', 3: 'Cans', 4: 'Glassware', 5: 'Glass Bottle', 6: 'Cardboard', 7: 'Basketball',
                             8: 'Paper', 9: 'Metalware', 10: 'Disposable Chopsticks', 11: 'Lighter', 12: 'Broom', 13: 'Old Mirror', 14: 'Toothbrush',
                             15: 'Dirty Cloth', 16: 'Seashell', 17: 'Ceramic Bowl', 18: 'Paint bucket', 19: 'Battery', 20: 'Fluorescent lamp', 21: 'Tablet capsules',
                             22: 'Orange Peel', 23: 'Vegetable Leaf', 24: 'Eggshell', 25: 'Banana Peel'}

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

    def __getitem__(self, idx):
        img_path = self.imgs[idx]
        label = self.class_to_idx[os.path.basename(os.path.dirname(img_path))]
        image = Image.open(img_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        # return image, self.idx_to_class[label]
        return image, label
```

定义神经网络模块 Python

```python
class ConvNet(nn.Module):
    def __init__(self, num_classes=26):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=5, stride=1, padding=2),
            # 使用归一化函数
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # 使用Dropout正则化
            nn.Dropout(0.5))
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            # nn.Softmax(dim=1),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(0.5))
        # self.fc = nn.Linear(56*56*64, num_classes)
        self.fc1 = nn.Linear(56*56*64, 1024)
        self.fc2 = nn.Linear(1024, num_classes)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
        out = F.relu(out)
        out = self.fc2(out)
        return F.softmax(out, dim=1)
```

自定义学习率下降方式 Python

```python
class CustomLR(_LRScheduler):
    def __init__(self, optimizer, last_epoch=-1):
        self.optimizer = optimizer
        super(CustomLR, self).__init__(optimizer, last_epoch)

    def get_lr(self):
        if self.last_epoch < 10:
            return self.base_lrs
        else:
            return [base_lr * 0.5 ** (self.last_epoch - 9) for base_lr in self.base_lrs]
```

训练模型 Python

```python
def train(train_iter, test_iter):
    # 训练模型
    num_epochs = 10
    loss_history = []

    #  创建模型实例
    model = ConvNet(num_classes=26).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-9)
    # 创建余弦退火学习率调度器
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs, eta_min=1e-7)
    # optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
    # scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min')
    # 创建学习率调度器
    # scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=4, gamma=0.1)
    # scheduler = CustomLR(optimizer)

    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(train_iter):
            images = images.to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if (i+1) % 5 == 0:
                loss_history.append(loss.item())
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                    .format(epoch+1, num_epochs, i+1, len(train_iter), loss.item()))

        # 在每个epoch结束后，更新学习率
        scheduler.step()
        # 输出当前的学习率
```

```
        current_lr = optimizer.param_groups[0]['lr']
        print('Epoch [{}/{}], Current learning rate: {:.4f}'.format(epoch+1, num_epochs, current_lr))

    plt.plot(loss_history)
    plt.xlabel('Step')
    plt.ylabel('Loss')
    plt.show()

    # 测试模型
    model.eval()  # eval mode (batchnorm uses moving mean/variance instead of mini-batch mean/variance)
    print('开始验证')
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in test_iter:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        print('Test Accuracy of the model on the test images: {} %'.format(100 * correct / total))

    # 保存模型
    torch.save(model.state_dict(), 'Convresults/model.pth')
```

main 函数 Python

```python
if __name__ == "__main__":
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print('代码运行在:',device)
    train_dir = "datasets/5fbdf571c06d3433df85ac65-momodel/garbage_26x100/train"
    test_dir = "datasets/5fbdf571c06d3433df85ac65-momodel/garbage_26x100/val"

    # 将图像调整为224×224尺寸并归一化
    mean = [0.485, 0.456, 0.406]
    std = [0.229, 0.224, 0.225]
    train_augs = transforms.Compose([
        transforms.RandomResizedCrop(size=224),
        # 随机翻转实现数据增强
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(90),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])
    test_augs = transforms.Compose([
        transforms.Resize(size=256),
        transforms.CenterCrop(size=224),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])

    train_set = CustomImageFolder(img_dir=train_dir, transform=train_augs)
    test_set = CustomImageFolder(img_dir=test_dir, transform=test_augs)

    batch_size = 24
    test_batch_size = 4
    train_iter = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    test_iter = DataLoader(test_set, batch_size=test_batch_size)

    # 是否训练模型
    is_train = 1
    if is_train == 1:
        train(train_iter, test_iter)

    # 是否显示图像
    pltshow = 0
    if pltshow == 1:
        # 获取一批图像和标签
        images, labels = next(iter(train_iter))
```

```
43
44          # 选择要显示的图像数量
45          n_images = 2
46
47          # 随机选择图像
48          indices = torch.randperm(images.size(0))[:n_images]
49          selected_images = images[indices]
50          selected_labels = labels[indices]
51          # selected_names = idx_to_class(labels[indices])
52          # 获取类别的名称
53          selected_names = [train_set.idx_to_class[label.item()] for label i
    n selected_labels]
54          print(selected_names)
55          # 使用make_grid函数创建一个网格图像
56          grid = torchvision.utils.make_grid(selected_images)
57
58          # 转换颜色通道
59          grid = grid.numpy().transpose((1, 2, 0))
60
61          # 反归一化
62          mean = np.array([0.485, 0.456, 0.406])
63          std = np.array([0.229, 0.224, 0.225])
64          grid = std * grid + mean
65          grid = np.clip(grid, 0, 1)
66
67          # 显示图像
68          plt.imshow(grid)
69          plt.title('Labels: ' + ', '.join(map(str, selected_labels.tolist()
    )))
70          plt.show()
```

iii. 运行结果

iv. 误差分析

b.搭建残差卷积神经网络

i. ResNet50残差网络具体代码实现

导入必要的库 Python

```python
import torch
from torch import nn, optim
from torch.utils.data import DataLoader
from torchvision import transforms, datasets, models
import time
import random
from matplotlib import pyplot as plt
import os
import math
from PIL import Image
from torch.optim.lr_scheduler import _LRScheduler
```

定义数据类 Python

```python
class CustomImageFolder(torch.utils.data.Dataset):
    def __init__(self, img_dir, transform=None):
        self.img_dir = img_dir
        self.transform = transform
        self.classes = os.listdir(img_dir)
        self.classes.sort()
        self.imgs = [os.path.join(root, name)
                     for root, dirs, files in os.walk(img_dir)
                     for name in files]
        self.class_to_idx = {'00_00': 0, '00_01': 1, '00_02': 2, '00_03': 3, '00_04': 4, '00_05': 5, '00_06': 6, '00_07': 7,
                             '00_08': 8, '00_09': 9, '01_00': 10, '01_01': 11, '01_02': 12, '01_03': 13, '01_04': 14,
                             '01_05': 15, '01_06': 16, '01_07': 17, '02_00': 18, '02_01': 19, '02_02': 20, '02_03': 21,
                             '03_00': 22, '03_01': 23, '03_02': 24, '03_03': 25}
        self.idx_to_class = {0: 'Plastic Bottle', 1: 'Hats', 2: 'Newspaper', 3: 'Cans', 4: 'Glassware', 5: 'Glass Bottle', 6: 'Cardboard', 7: 'Basketball',
                             8: 'Paper', 9: 'Metalware', 10: 'Disposable Chopsticks', 11: 'Lighter', 12: 'Broom', 13: 'Old Mirror', 14: 'Toothbrush',
                             15: 'Dirty Cloth', 16: 'Seashell', 17: 'Ceramic Bowl', 18: 'Paint bucket', 19: 'Battery', 20: 'Fluorescent lamp', 21: 'Tablet capsules',
                             22: 'Orange Peel', 23: 'Vegetable Leaf', 24: 'Eggshell', 25: 'Banana Peel'}

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

    def __getitem__(self, idx):
        img_path = self.imgs[idx]
        label = self.class_to_idx[os.path.basename(os.path.dirname(img_path))]
        image = Image.open(img_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        # return image, self.idx_to_class[label]
        return image, label
```

数据集预处理 Python

```python
# 定义数据增强的变换
train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])

test_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])

# 加载数据集
train_dir = "datasets/5fbdf571c06d3433df85ac65-momodel/garbage_26x100/train"
test_dir = "datasets/5fbdf571c06d3433df85ac65-momodel/garbage_26x100/val"
train_data = CustomImageFolder(train_dir, transform=train_transform)
test_data = CustomImageFolder(test_dir, transform=test_transform)

# 划分验证集
val_size = int(len(train_data) * 0.2)
train_size = len(train_data) - val_size
train_data, val_data = torch.utils.data.random_split(train_data, [train_size, val_size])

# 创建数据加载器
train_iter = DataLoader(train_data, batch_size=16, shuffle=True, num_workers=4)
val_iter = DataLoader(val_data, batch_size=16, shuffle=False, num_workers=4)
test_iter = DataLoader(test_data, batch_size=16, shuffle=False, num_workers=4)

# 定义设备
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

ResNet50残差网络模型训练 Python

```python
def train(train_iter, val_iter, test_iter):
    # 训练模型
    num_epochs = 10
    loss_history = []
    val_loss_history = []
    val_acc_history = []

    #  创建模型实例
    model = models.resnet50(pretrained=True) # 使用预训练的ResNet50
    for param in model.parameters():
        param.requires_grad = False # 冻结参数，不更新梯度
    model.fc = nn.Sequential( # 修改最后一层
        nn.Linear(model.fc.in_features, 26),
        nn.Softmax(dim=1) # 加上softmax层
    )
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.fc.parameters(), lr=0.001, weight_decay=0) # 使用权重衰减

    # 创建学习率调度器
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2, verbose=True) # 使用ReduceLROnPlateau

    for epoch in range(num_epochs):
        start = time.time()
        running_loss = 0.0
        for i, (images, labels) in enumerate(train_iter):
            images = images.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if (i + 1) % 10 == 0:
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 10))
                loss_history.append(running_loss / 10)
                running_loss = 0.0
        end = time.time()
        print('Epoch %d cost %.3f seconds' % (epoch + 1, end - start))
        # 在验证集上评估模型
        val_loss, val_acc = evaluate(model, val_iter, criterion)
```

```
        print('Validation loss: %.3f, Validation accuracy: %.3f' % (val_loss, val_acc))
        val_loss_history.append(val_loss)
        val_acc_history.append(val_acc)
        # 调整学习率
        scheduler.step(val_loss)
    print('Finished Training')
    # 在测试集上评估模型
    test_loss, test_acc = evaluate(model, test_iter, criterion)
    print('Test loss: %.3f, Test accuracy: %.3f' % (test_loss, test_acc))
    # 绘制损失和准确率曲线
    plot_loss_and_acc(loss_history, val_loss_history, val_acc_history)
    torch.save(model.state_dict(), 'Convresults/model3.pth')
```

模型评估 Python

```
def evaluate(model, data_iter, criterion):
    # 评估模型
    model.eval()
    with torch.no_grad():
        total_loss = 0.0
        total_acc = 0.0
        total_count = 0
        for images, labels in data_iter:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            total_loss += loss.item() * images.size(0)
            _, preds = torch.max(outputs, 1)
            total_acc += torch.sum(preds == labels).item()
            total_count += images.size(0)
        return total_loss / total_count, total_acc / total_count
```

可视化损失函数 Python

```python
def plot_loss_and_acc(loss_history, val_loss_history, val_acc_history):
    # 绘制损失和准确率曲线
    plt.figure(figsize=(12, 4))
    plt.subplot(121)
    plt.plot(loss_history, label='train loss')
    plt.plot(val_loss_history, label='validation loss')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.legend()
    plt.subplot(122)
    plt.plot(val_acc_history, label='validation accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()
```

Python

```python
if __name__ == "__main__":
    train(train_iter, val_iter, test_iter)
```

ii. ResNet18残差网络具体代码实现

仅需将ResNet50训练模型的代码块调整为下图所示即可，其余部分不需调整

ResNet18残差网络模型训练 Python

```python
def train(train_iter, val_iter, test_iter):
    # 训练模型
    num_epochs = 10
    loss_history = []
    val_loss_history = []
    val_acc_history = []

    #  创建模型实例
    model = models.resnet18(pretrained=True) # 使用预训练的ResNet18
    for param in model.parameters():
        param.requires_grad = False # 冻结参数，不更新梯度
    model.fc = nn.Sequential( # 修改最后一层
        nn.Linear(model.fc.in_features, 26),
        nn.Softmax(dim=1) # 加上softmax层
    )
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    # optimizer = torch.optim.AdamW(model.fc.parameters(), lr=0.001, weigh
t_decay=0.01) # 使用AdamW优化器
    optimizer = Lamb(model.fc.parameters(), lr=0.001, weight_decay=0.01)
# 使用LAMB优化器
    scaler = GradScaler() # 创建梯度缩放器

    # 创建学习率调度器
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode
='min', factor=0.1, patience=2, verbose=True) # 使用ReduceLROnPlateau

    for epoch in range(num_epochs):
        start = time.time()
        running_loss = 0.0
        for i, (images, labels) in enumerate(train_iter):
            images = images.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            with autocast(): # 开启自动混合精度
                outputs = model(images)
                loss = criterion(outputs, labels)
            scaler.scale(loss).backward() # 使用梯度缩放器
            scaler.step(optimizer) # 使用梯度缩放器
            scaler.update() # 使用梯度缩放器
            running_loss += loss.item()
            if (i + 1) % 10 == 0:
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 10))
                loss_history.append(running_loss / 10)
```

```
                running_loss = 0.0
        end = time.time()
        print('Epoch %d cost %.3f seconds' % (epoch + 1, end - start))
        # 在验证集上评估模型
        val_loss, val_acc = evaluate(model, val_iter, criterion)
        print('Validation loss: %.3f, Validation accuracy: %.3f' % (val_loss, val_acc))
        val_loss_history.append(val_loss)
        val_acc_history.append(val_acc)
        # 调整学习率
        scheduler.step(val_loss)
    print('Finished Training')
    # 在测试集上评估模型
    test_loss, test_acc = evaluate(model, test_iter, criterion)
    print('Test loss: %.3f, Test accuracy: %.3f' % (test_loss, test_acc))
    # 绘制损失和准确率曲线
    plot_loss_and_acc(loss_history, val_loss_history, val_acc_history)
    # 保存模型
    torch.save(model.state_dict(), 'Convresults/model4.pth')
```

此外我还尝试了其余几种残差神经网络，代码类似，就不在此赘述了，所有代码的结果会在下文给出。

ResNet是一种深度卷积神经网络，通过引入残差块来解决深度网络训练中的梯度消失和梯度爆炸问题。ResNet18和50的主要区别在于网络深度和参数数量。

ResNet18由18个卷积层和全连接层组成，相比于50，在计算资源有限的情况下，更容易训练和部署，但同时因为深度较浅，所以18的表达能力相对较弱，无法处理更复杂的任务。

ResNet18引入残差连接，通过跳跃连接来解决梯度消失和梯度爆炸问题。

iii. 运行结果

说明：所有程序均在以下初始条件中运行

epoch = 15;

scheduler(学习率下降方式)：余弦模拟退火下降

lr0（初始学习率）： 0.001

lrf（终止学习率）： 0.00005

Momentum（动量）: 0.937

weight_decay（权重衰减）： 0.0005

train_batch_size（训练集大小）： 24

test_batch_size（测试集大小）：15

optimizer（优化器）： SGD-Momentum

warm_up（预训练步数）： 4

warmup_momentum（预训练学习率）： 0.8

a.ResNet50运行结果

Epoch	GPU_mem	train_loss	val_loss	top1_acc	top5_acc
1/19	1.74G	3.2	3.29	0.0133	0.16:
2/19	1.74G	2.79	3.33	0.0667	0.227:
3/19	1.74G	1.63	4.1	0.0667	0.267:
4/19	1.74G	1.02	5.46	0.0667	0.267:
5/19	1.74G	1.19	5.81	0.0667	0.24:
6/19	1.74G	1.03	4.99	0.0667	0.267:
7/19	1.74G	0.935	4.23	0.0667	0.32:
8/19	1.74G	0.904	4.36	0.0667	0.307:
9/19	1.74G	0.808	4.76	0.0667	0.307:
10/19	1.74G	0.805	4.87	0.0667	0.32:
11/19	1.74G	0.928	4.68	0.0667	0.32:
12/19	1.74G	0.753	4.3	0.0667	0.307:
13/19	1.74G	0.73	4.1	0.0667	0.307:
14/19	1.74G	0.734	4.06	0.0667	0.32:
15/19	1.74G	0.817	4.17	0.0667	0.307:
16/19	1.74G	0.702	4.17	0.0667	0.293:
17/19	1.74G	0.697	4.18	0.0667	0.293:
18/19	1.74G	0.771	4.21	0.0667	0.307:
19/19	1.74G	0.73	4.21	0.0667	0.293:

b.ResNet18运行结果

Epoch	GPU_mem	train_loss	val_loss	top1_acc	top5_acc
1/19	0.614G	3.44	3.56	0.0267	0.107
2/19	0.614G	2.96	3.57	0.04	0.147
3/19	0.614G	1.94	4.09	0.0667	0.147
4/19	0.614G	1.03	5.19	0.0667	0.16
5/19	0.614G	1.16	5.73	0.0667	0.213
6/19	0.614G	1.23	5.16	0.0667	0.187
7/19	0.614G	0.851	4.61	0.08	0.213
8/19	0.614G	0.855	4.55	0.08	0.227
9/19	0.614G	1.1	4.72	0.08	0.187
10/19	0.614G	0.849	4.7	0.08	0.213
11/19	0.614G	0.728	4.53	0.08	0.227
12/19	0.614G	0.716	4.5	0.0933	0.253
13/19	0.614G	0.756	4.51	0.0933	0.253
14/19	0.614G	0.746	4.52	0.0933	0.253
15/19	0.614G	0.704	4.48	0.0933	0.227
16/19	0.614G	0.719	4.5	0.0933	0.24
17/19	0.614G	0.683	4.5	0.0933	0.227
18/19	0.614G	0.7	4.53	0.0933	0.227
19/19	0.614G	0.711	4.55	0.0933	0.213

c.ResNet101运行结果

Epoch	GPU_mem	train_loss	val_loss	top1_acc	top5_acc
1/19	2.54G	3.24	3.3	0.0667	0.2
2/19	2.56G	2.76	3.34	0.0667	0.2
3/19	2.56G	1.59	4.18	0.0667	0.227
4/19	2.56G	1.24	5.49	0.0667	0.253
5/19	2.56G	1.19	6.1	0.0667	0.24
6/19	2.56G	0.979	4.68	0.0667	0.24
7/19	2.56G	0.949	4.19	0.08	0.253
8/19	2.56G	0.936	4.4	0.0667	0.253
9/19	2.56G	1.02	4.88	0.0667	0.253
10/19	2.56G	0.804	4.72	0.0667	0.253
11/19	2.56G	0.844	4.6	0.0667	0.267
12/19	2.56G	0.784	4.35	0.0667	0.253
13/19	2.56G	0.724	4.14	0.0667	0.253
14/19	2.56G	0.723	4.09	0.08	0.253
15/19	2.56G	0.72	4.12	0.08	0.267
16/19	2.56G	0.707	4.21	0.0667	0.267
17/19	2.56G	0.715	4.34	0.0667	0.267
18/19	2.56G	0.77	4.47	0.0667	0.267
19/19	2.56G	0.687	4.41	0.0667	0.253