同濟大學

TONGJI UNIVERSITY

人工智能课程设计三

第三次实验报告

班级：国豪工科嘉定3班

姓名：倪雨舒

学号： 2353105

完成日期： 2025年5月13日

1.问题概述

1.1 问题背景

*任务1：*利用LeNet5网络模型处理手写数字识别问题；数字分类问题基于的卷积神经网络是LeNet5，可以说是未来卷积神经网络的老祖，LeNet5神经网络基本上包含了卷积神经网络中最基础的，以及最经典的组成部分，比如卷积层，池化层，全连接层，是一个非常适合我们初学者上手的简单项目。由于在作业要求使用Mindspore库，**其实在Mindspore库中已经存在相关更高级的封装好的LeNet5模型，但是没有内部详细的模式，无法了解到神经网络的构建过程，因此我选择手动实现的方式通过LeNet5模型完成垃圾分类项目。**

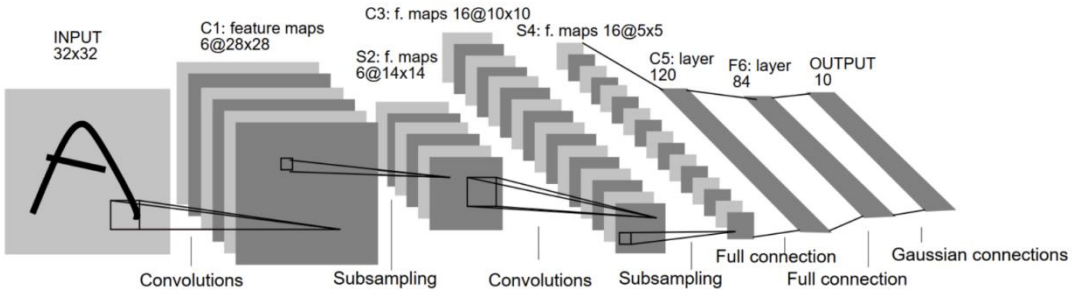
*任务2：*使用Mindspore框架实现26种垃圾分类。基于Github上的开源的Mindspore项目，对项目部分代码进行重构，讨论不同方法微调的效果以及效率并且对不同的微调方法进行对比分析。

1.2 问题思路

由于只涉及到对模型的微调所以有关神经网络构建的部分不需要我们来进行定义，首先需要对模型设定的一些确定的参数进行调整，调整之后跑通整个项目，并且对其中的比如优化器函数等内容进行修改，探讨修改之后的效果并进行对比分析。随后尝试不同的微调方法进行对比与说明。

2.算法设计

作为比较经典的神经网络的应用，因此先对算法的核心，卷积神经网络的基本原理进行说明：LeNet5的经典结构如下图所示，实际在进行分类任务的时候回和经典结构有所改变：



首先输入的是图像的灰度值，只有一个维度，大小为32*32，并且要对数据集进行预处理包含尺寸调整，归一化处理以及格式调整。第一个卷积层是C1，有6个卷积核大小为5*5，最终输出6个特征矩阵，进行特征提取，S2是池化层，池化窗口的大小为2*2，步长为2，最终输出的特征矩阵的大小为14*14，原始的LeNet5采用的是平均池化方式，但是目前的比较主流的方式是最大池化方式。

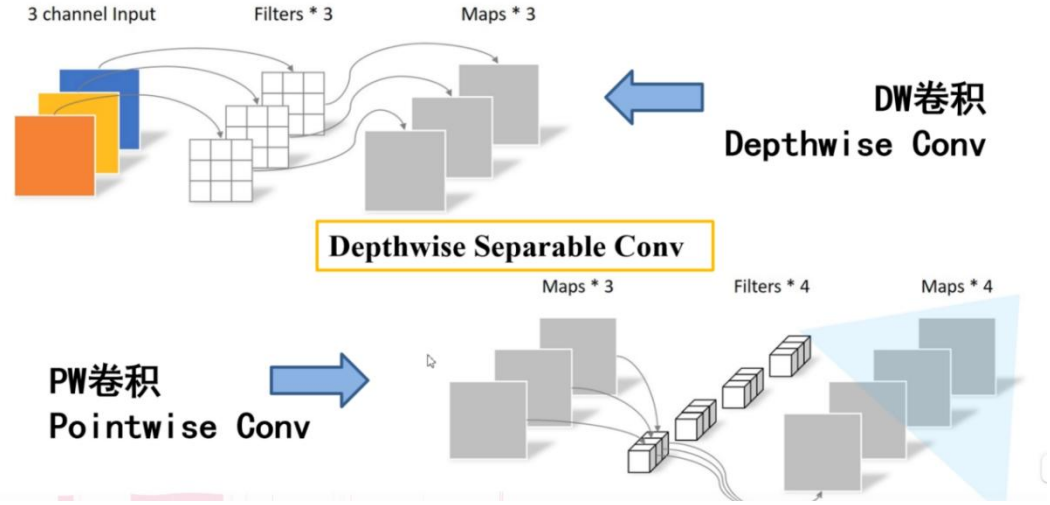
随后的C3和S4又分别是一个卷积层和一个池化层，最终得到了16个10*10的特征矩阵，得到的特征图在C5层转化为一个120维的向量，最终在F6全连接层中执行加权求和通过softmax激活函数输出最后的结果。最终输出层输出的就是0-9个数字可能的概率。

在原始的LetNet5中第二次卷积过程中选择了组合连接的方式计算出得到的16个特征矩阵，原来的文章说这样做的原因是减少参数，并且认为这种不对称的组合连接方式有利于提取多种组合特征。后续会对全连接方式与组合方式进行准确性上的对比。

在原论文的基础上，可以对神经网络进行进一步的优化，包括使用最大池化方式，全连接方式，ReLU激活函数。下表给出了在全过程中参数的数目以及含义：

	C1 层	S2 层（池化层）	C3 层	S4 层（池化层）	C5 层	F6 层（全连接层）
输入	32x32	28x28（6 个）	14x14（6 个）	10x10（16 个）	5x5（16 个）	c5 120 维向量
采样方式	无	2*2	无	2*2	无	无
卷积核	6-5*5	无	16-5*5	无	120-5*5	无
输出 featuremap	28x28（6 个）	14x14（6 个）	10x10（16 个）	5x5（16 个）	1x1（120 个）	10 个神经元（对应 0~9）
神经元数量	28*28*6	14*14*6	10*10*16	5*5*16	120	84
训练参数	(5*5+1)*6=156	2*6=12	6*(3*5*5+1)+6*(4*5*5+1)+3*(4*5*5+1)+1*(6*5*5+1)=1516	2*16=32	120*(16*5*5+1)=48120	84*(120+1)=10164
连接数	(5*5+1)*6*28*28=122304	(2*2+1)*6*14*14=5880	10*10*1516=151600	16*（2*2+1）*5*5=2000	120*(16*5*5+1)=48120	(120+1)*84=10164
Activation	tanh	tanh	tanh	tanh	tanh	softmax

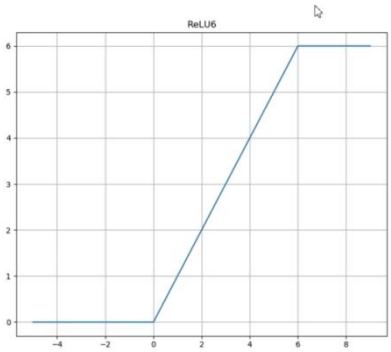
垃圾分类任务基于的是MobileNetV2深度神经网络，而MobileNetV2又是基于MobileNetV1网络改进而来，相较于传统的卷积神经网络，V1网络引入了DW卷积和PW卷积，二者共同构成了DS卷积，如下图所示。



传统的卷积神经网络要求卷积核的通道数等于输入特征矩阵的通道数，输出特征矩阵的通道数等于卷积核的个数，而DW卷积（Depthwise Conv）中卷积核的通道数永远为1，并且输入特征矩阵的通道数等于卷积核的个数等于输出的特征矩阵的通道数。一个卷积核只负责输入的特征矩阵一个通道的特征提取。PW（Pointwise Conv）卷积本质上就是一个正常的卷积过程，只不过是卷积核的大小为1*1，最终得到4个输出的特征图。

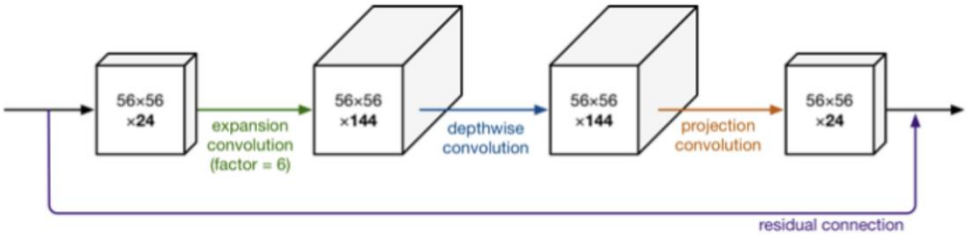
MobileNetV2最核心的创新点在于倒残差结构以及线性瓶颈的处理。在ResNet中引入了残差结构，即通过1*1的卷积核先实现降维，再进行卷积操作然后再通过1*1的卷积核进行升维，而MobileNetV2中的倒残差结构则是先用1*1的卷积核进行升维，然后再通过DW卷积进行卷积操作，最后再进行降维操作，并且利用ReLU（6）作为激活函数，以上就是倒残差结构的基本特征。

$$y = \mathrm{ReLU6}(x) = \min(\max(x, 0), 6)$$



MobileNetV1 使用了大量 深度可分离卷积（DW + PW），极大减少了计算量。然而，它将整个网络都限制在了低维通道空间（如 16、32、64），这造成了表达能力有限。深度学习的“特征表示”主要发生在通道维度，通道数太少 导致无法刻画复杂的语义信息，进而导致准确率受限

MobileNetV2 的想法在于：能不能既保持轻量（低计算），又提升表示能力，于是 MobileNetV2 做了两件事：在特征提取阶段临时“升维”，即用 1×1 卷积把低维通道扩展为高维（如 16 → 96）提供更多“冗余特征”，增强模型表达能力。再通过线性“降维”压缩回低维输出，这样保持模型整体轻量并且减少后续层的计算负担同时避免冗余传播
这样做会导致计算量飙升但为什么可以这样做这是因为：中间使用的是 DW卷积，它对通道是分开卷积的，计算量极小，标准卷积的计算量正比于 K×K×C_in×C_out，DW卷积：计算量正比于K×K×C_in，所以即使升维到很高，DW卷积的计算成本也不会很高。所以只有在DW卷积里，才有“先升维 → 然后卷积 → 再降维”的过程传统卷积计算成本过高，只能“先降维再升维”（如 ResNet 的 bottleneck）



在传统卷积中，每一层后面都会加激活函数（如 ReLU），增强非线性能力。但在 MobileNetV2 中，最后一层降维后不再加激活函数而是保持线性映射，因为ReLU会将小于 0 的值置 0，如果在“低维空间”使用 ReLU，容易导致信息完全丢失，破坏表示能力原论文中指出：“低维空间是信息瓶颈，加激活函数反而限制了信息传递”。

4. 算法实现

对于LeNet5垃圾分类任务，需要使用Mindspore的cell基类构建新的网络模型，这里首先尝试完全参照参照原始的LeNet5模型构建，代码如下：

```
class LeNet5(nn.Cell):
    def __init__(self, num_classes=26):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d( in_channels: 1,  out_channels: 6, kernel_size=5, pad_mode='valid')
        self.relu1 = nn.ReLU()
        self.pool1 = nn.AvgPool2d(kernel_size=(2, 2), stride=(2, 2))
        self.conv2 = nn.Conv2d( in_channels: 6,  out_channels: 16, kernel_size=5, pad_mode='valid')
        self.relu2 = nn.ReLU()
        self.pool2 = nn.AvgPool2d(kernel_size=(2, 2), stride=(2, 2))
        self.flatten = nn.Flatten()
        self.fc1 = nn.Dense(16 * 4 * 4,  out_channels: 120)
        self.fc2 = nn.Dense( in_channels: 120,  out_channels: 84)
        self.fc3 = nn.Dense( in_channels: 84, num_classes)

    def construct(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

定义损失函数以及优化器如下，分别为交叉熵损失函数以及随机梯度下降优化算法，更加接近原始模型的算法设计。

```
# 4. 损失函数与优化器
loss_fn = nn.CrossEntropyLoss()
optimizer = nn.SGD(model.trainable_params(), learning_rate: 1e-2)
```

为了便于观察训练过程的准确率以及损失值，手动定义训练函数。训练过程中会对输出结果的标签与真实标签进行比对用于计算准确率：

```
def train(model, dataset):
    size = dataset.get_dataset_size()           # 获取训练集中 batch 总数（用于 tqdm 显示）
    model.set_train()                           # 设置模型为训练模式（启用 Dropout/BN 等）
    correct = 0                                 # 统计预测正确的样本数
    total = 0                                   # 样本总数
    total_loss = 0                              # 累加 loss
    for batch, (data, label) in enumerate(tqdm(dataset.create_tuple_iterator(), total=size, desc="Training")):
        loss = train_step(data, label)          # 执行一次 train_step（前向+反向+更新）
        total_loss += loss.asnumpy()            # 累加损失（转为 numpy 标量）
        logits = model(data)                    # 再前向一次用于评估准确率
        correct += (logits.argmax(1) == label).asnumpy().sum()  # argmax 取预测类别，与标签对比
        total += label.shape[0]                 # 样本总数
        if batch % 10 == 0:                     # 每 10 个 batch 输出一次中间 loss
            print(f"loss: {loss.asnumpy():>7f}  [{batch:>3d}/{size:>3d}]")

    avg_loss = total_loss / size                # 平均每 batch 的损失
    accuracy = correct / total                  # 总体准确率
    print(f"Train: Accuracy: {(100*accuracy):>0.1f}%, Avg loss: {avg_loss:>8f}\n")
    return avg_loss, accuracy                   # 返回给主循环记录
```

Mindspore中的MobileNetV2网络结构如下图所示：鉴于git上的原始的项目结构更加完整，因此在原始项目的基础上对项目的整体架构进行详细说明。在附加的文件中是一个参照官方示例使用LeNet5实现的垃圾分类的项目，但以下说明仍旧采用原始的项目进行说明。

输入	操作	t(扩张倍数expand_ratio)	c(输出通道数)	n(重复次数)	s(步长stride)
$3*224*224$	conv2d	-	32	1	1
$32*112*112$	InvertedResidual	1	16	1	1
$16*112*112$	InvertedResidual	6	24	2	2
$24*56*56$	InvertedResidual	6	32	3	2
$32*28*28$	InvertedResidual	6	64	4	2
$64*28*28$	InvertedResidual	6	96	3	1
$96*14*14$	InvertedResidual	6	160	3	2
$160*7*7$	InvertedResidual	6	320	1	1
$320*7*7$	conv2d 1*1	-	1280	1	1
$1280*7*7$	avgpool 7*7	-	-	1	-
$1*1*k$	conv2d 1*1	-	k	-	-

主要存在三类层，conv2d就是普通的卷积层，InvertedResidual模块是上面提到过的倒残差结构，内部要先有1*1的卷积升维的过程，然后利用DW卷积提取空间特征，最后再降维，最后的输出层要先进行升维然后进行全局平均池化，最后分类输出。

观察Mindspore实现的MobileNetV2网络，发现其将整个神经网络分为了两大部分，其中bachbone部分是网络的核心，总的目的就是用于提取图像的通用特征，是整个网络的核心部分，head部分用于生成分类结果，微调模型的过程就是调整的就是这部分的参数。

鉴于整个项目的代码过多并且大部分已经是现有的已经写好的代码，因此只对项目的核心部分mobilenetV2.py进行详细说明，介绍是如何实现深度卷积操作和倒残差结构的构建等核心内容。

首先介绍深度卷积功能是如何实现的。构造函数的参数的含义为in_planes (int): 输入通道数；out_planes (int): 输出通道数（标准卷积用）；kernel_size (int): 卷积核

大小，默认为 3；stride (int)：卷积步长，默认为 1；groups (int)：卷积组数。标准卷积时为 1，深度卷积时为 in_planes（每通道独立卷积）

```
    示例:
        >>> ConvBNReLU(16, 256, kernel_size=1, stride=1, groups=1)  # 标准卷积
        >>> ConvBNReLU(32, 32, kernel_size=3, stride=1, groups=32)  # 深度卷积
    """

    def __init__(self, in_planes, out_planes, kernel_size=3, stride=1, groups=1):
        super(ConvBNReLU, self).__init__()
```

具体实现如下面的代码所示，主要调用Mindspore内置的标准库的如卷积层等进行组合和修改，最终从代码方面实现深度卷积功能：

```
    def __init__(self, in_planes, out_planes, kernel_size=3, stride=1, groups=1):
        super(ConvBNReLU, self).__init__()
        # 卷积边缘填充，使输出尺寸与输入保持一致（在 stride=1 时）
        padding = (kernel_size - 1) // 2
        # 保存输入/输出通道信息
        in_channels = in_planes
        out_channels = out_planes
        # 根据 groups 值判断是标准卷积还是深度卷积
        if groups == 1:
            # 标准卷积：groups=1，输出通道数由 out_planes 决定
            conv = nn.Conv2d(
                in_channels, out_channels, kernel_size, stride,
                pad_mode='pad', padding=padding
            )
        else:
            # 深度卷积：groups=in_channels，每个输入通道独立卷积，不进行通道混合
            out_channels = in_planes  # 通道数保持不变
            conv = nn.Conv2d(
                in_channels, out_channels, kernel_size, stride,
                pad_mode='pad', padding=padding, group=in_channels
            )
        # 构造卷积块：Conv → BN → ReLU6
        layers = [
            conv,
            nn.BatchNorm2d(out_planes),  # 批归一化
            nn.ReLU6()                   # 激活函数（限制最大值为 6，适合量化）
        ]

        # 使用 MindSpore 的 SequentialCell 打包成顺序网络模块
        self.features = nn.SequentialCell(layers)
```

然后介绍倒残差结构是如何实现的，构造函数的参数的含义为输入通道数目，输出通道数目，stride步长，控制为1-2，expand_ratio作为扩展倍数

```
    示例:
        >>> InvertedResidual(32, 64, stride=1, expand_ratio=6)
    """

    def __init__(self, inp, oup, stride, expand_ratio):
        super(InvertedResidual, self).__init__()
        assert stride in [1, 2]  # stride 只能为 1 或 2，确保网络结构合法
```

具体实现代码如下，注意其中有对捷径的阐述，如果步长为1并且输入输出通道数目相等的话就可以直接使用残差链接，随后分别进行升维，DW卷积过程，降维过程，最后将他们打包。

```
hidden_dim = int(round(inp * expand_ratio))
# 判断是否使用残差连接：
# 条件：stride=1 且 输入输出通道数一致，才能直接加法相连
self.use_res_connect = stride == 1 and inp == oup
layers = []  # 用于保存各层结构
#Expansion Layer：升维（如果 expand_ratio > 1）
if expand_ratio != 1:
    layers.append(ConvBNReLU(inp, hidden_dim, kernel_size=1))  # 1×1 卷积升维
#Depthwise Convolution：空间特征提取
layers.extend([
    ConvBNReLU(hidden_dim, hidden_dim, stride=stride, groups=hidden_dim),
    # DW卷积：组卷积中 groups=输入通道数，表示每个通道单独卷积
])
# Projection Layer：降维（Linear Bottleneck，无激活）
layers.extend([
    nn.Conv2d(hidden_dim, oup, kernel_size=1, stride=1, has_bias=False),  # 1x1 卷积降维
    nn.BatchNorm2d(oup),  # BN（无激活），线性输出
])
# 将所有层用 SequentialCell 打包
self.conv = nn.SequentialCell(layers)
```

基于上述结构介绍backbone部分，backbone部分由ConvBNReLU卷积层、InvertedResidual倒残差结构组成，代码如下：

```
参数:
    width_mult (float): 宽度乘子，用于按比例缩放网络通道数。默认值为 1。
    inverted_residual_setting (list): 倒残差结构的配置列表，默认为 None 使用标准配置。
    round_nearest (int): 通道数向上取整到最接近的整数倍（如 8、16），默认 8。
    input_channel (int): 初始输入通道数，默认为 32。
    last_channel (int): 最后输出通道数，默认为 1280。
输出:
    Tensor: 提取后的特征图（通常是 shape = [B, 1280, 7, 7]）
"""
def __init__(self, width_mult=1., inverted_residual_setting=None, round_nearest=8,
             input_channel=32, last_channel=1280):
    super(MobileNetV2Backbone, self).__init__()
```

构造函数的参数含义依次为：width_mult：宽度缩放系数（用来调整通道数）；inverted_residual_setting：手动配置倒残差结构的列表（默认使用标准配置）；round_nearest：通道数对齐倍数，确保通道数能整除硬件友好的数字（如 8）；input_channel：输入初始卷积层的通道数；last_channel：网络最后输出通道数（一般为1280）。

然后是倒残差结构的设置，包含了t扩展倍数，c输出通道数量，n重复次数，s步长，代码如下：与之前表格中的定义保持一致

```
# 设置倒残差结构配置，如果外部没传，就使用默认标准配置
self.cfgs = inverted_residual_setting
if inverted_residual_setting is None:
    self.cfgs = [
        # [扩展倍数 t, 输出通道 c, 重复次数 n, 步长 s
        [1, 16, 1, 1],
        [6, 24, 2, 2],
        [6, 32, 3, 2],
        [6, 64, 4, 2],
        [6, 96, 3, 1],
        [6, 160, 3, 2],
        [6, 320, 1, 1],
    ]
```

然后是倒残差结构的设置，包含了t扩展倍数，c输出通道数量，n重复次数，s步长，代码如下：接下来处理输入通道与输出通道数目以及构建第一个卷积层，代码如下，其中

卷积层的输入为RGB图像也就是个3通道的特征矩阵，卷积的步长为2，下采样至112*112，其中的ConvBNReLU是一个组合层，包含了Conv2D的二维卷积层、BatchNorm2D的批归一化层对每个通道的输出进行归一化处理使其均值接近0，方差接近1，提高训练稳定性。ReLu激活函数层，三者共同构成一个卷积操作单元。

```
        # 根据缩放系数和对齐规则调整输入、输出通道数
        input_channel = _make_divisible(input_channel * width_mult, round_nearest)
        self.out_channels = _make_divisible(last_channel * max(1.0, width_mult), round_nearest)

        # 构建第一层卷积（3x3，stride=2）
        features = [ConvBNReLU( in_planes: 3, input_channel, stride=2)]
```

然后构建 InvertedResidual 块堆叠，对每个倒残差配置：根据 width_mult 计算出实际输出通道重复 n 次堆叠该模块第一次使用设定步长 s，其余使用 stride=1 保持尺寸每个 block 更新 input_channel代码如下，构建完成之后最后添加一个1*1的卷积层用来统一输出维度，将他们通过SequentialCell打包，最后将网络层的参数进行初始化。

```
# 构建 InvertedResidual 模块（堆叠多个 block）
for t, c, n, s in self.cfgs:
    output_channel = _make_divisible(c * width_mult, round_nearest)
    for i in range(n):
        # 第一次使用给定 stride，其余重复层 stride=1
        stride = s if i == 0 else 1
        # 添加一个 InvertedResidual 模块
        features.append(block(input_channel, output_channel, stride, expand_ratio=t))
        # 更新输入通道为当前输出通道
        input_channel = output_channel
# 添加最后一个 1x1 卷积层用于统一输出通道数
features.append(ConvBNReLU(input_channel, self.out_channels, kernel_size=1))
# 将所有层组合成一个顺序结构（类似 PyTorch 的 nn.Sequential）
self.features = nn.SequentialCell(features)
# 权重初始化
self._initialize_weights()
```

构造函数之外，类中还有前向传播函数将张量x按照定义好的网络向前计算。权值初始化函数对每个子模块进行初始化，保证网络开始时训练稳定，代码如下：

```
    def construct(self, x):
        """
        前向传播函数（MindSpore 中替代 PyTorch 的 forward）
        参数:
            x (Tensor): 输入图像张量
        返回:
            x (Tensor): 特征图（例如 [B, 1280, 7, 7]）
        """
        x = self.features(x)
        return x

    def _initialize_weights(self):
        """
        初始化网络中所有卷积和归一化层的参数
        Conv2d: 使用 He 正态初始化（适合 ReLU）
        BatchNorm: gamma 初始化为 1，beta 初始化为 0
        """
        self.init_parameters_data()
        for _, m in self.cells_and_names():
            if isinstance(m, nn.Conv2d):
                # He初始化：根据输出通道调整标准差
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.set_data(Tensor(np.random.normal( loc: 0, np.sqrt(2. / n),
                                                          m.weight.data.shape).astype("float32")))
                if m.bias is not None:
                    m.bias.set_data(Tensor(np.zeros(m.bias.data.shape, dtype="float32")))
            elif isinstance(m, nn.BatchNorm2d):
                m.gamma.set_data(Tensor(np.ones(m.gamma.data.shape, dtype="float32")))
                m.beta.set_data(Tensor(np.zeros(m.beta.data.shape, dtype="float32")))
```

接下来对head网络部分进行解析与介绍，head网络接收backbone处理之后的矩阵，因此输入的通道数是固定的1280，has_dropout用于防止过拟合，如果不需要防止过拟合的话那么就将全局池化层直接连接到全连接层，并且最终判断是否还需要激活函数，代码如下。最后利用上述的所有的类，实现了MobileNetV2类以及MobileNetV2combine类分别作为标准的网络以及自定义的网络，辅助构造函数：mobilenet_v2(backbone, head)直接返回MobileNetV2Combine(backbone, head) 实例，便于后续调用。

```
class MobileNetV2Head(nn.Cell):
    def __init__(self, input_channel=1280, num_classes=1000, has_dropout=False, activation="None"):
        super(MobileNetV2Head, self).__init__()
        # 定义分类头结构（顺序组合）
        if not has_dropout:
            # 无 Dropout 时：全局池化 → 全连接层
            head = [
                GlobalAvgPooling(),                          # 将 [B, C, H, W] → [B, C]
                nn.Dense(input_channel, num_classes, has_bias=True)  # 全连接层输出 num_classes 类
            ]
        else:
            # 使用 Dropout 时：全局池化 → Dropout → 全连接层
            head = [
                GlobalAvgPooling(),
                nn.Dropout(0.2),  # 保留概率 0.8，丢弃 20%
                nn.Dense(input_channel, num_classes, has_bias=True)
            ]

        # 使用 MindSpore 的顺序网络容器
        self.head = nn.SequentialCell(head)

        # 根据是否需要激活函数确定后续处理方式
        self.need_activation = True
        if activation == "Sigmoid":...
        elif activation == "Softmax":...
        else:...
        # 初始化权重
        self._initialize_weights()
```

成功定义MobileNetV2网络之后在model.py中调用相关函数，形成了构建网络的接口函数，代码如下所示，在train文件中也要调用define_net函数生成所需的MobileNetV2神经网络。

```
def define_net(config, is_training):
    backbone_net = MobileNetV2Backbone()
    activation = config.activation if not is_training else "None"
    head_net = MobileNetV2Head(input_channel=backbone_net.out_channels,
                               num_classes=config.num_classes,
                               activation=activation)
    net = mobilenet_v2(backbone_net, head_net)
    return backbone_net, head_net, net
```

以下解释数据预处理的相关代码，首先的create_dataset函数根据是训练集还是验证集、运行平台、以及配置参数来生成可供模型训练/评估使用的数据集。参数定义如下：

```
# dataset_path   数据集所在路径（按文件夹分类存放）
# do_train   是否为训练模式（决定是否启用增强）
# config     包含平台、图像尺寸、batch_size 等配置
# repeat_num     数据集重复次数（默认 1）
def create_dataset(dataset_path, do_train, config, repeat_num=1):
```

如果是训练数据预处理过程对图像进行了如下操作：对图像数据进行预处理与增强，包括解码、随机裁剪、翻转、颜色扰动、标准化和格式转换，为模型训练准备格式一致且更具鲁棒性的输入数据。

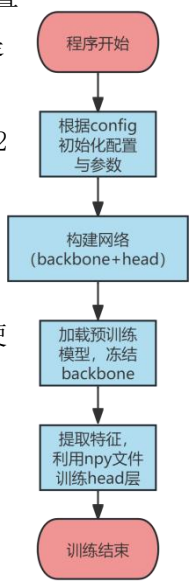
操作	描述
RandomCropDecodeResize	随机裁剪解码+resize（用于仿射增强）
RandomHorizontalFlip(prob=0.5)	以 50% 概率水平翻转图像
RandomColorAdjust	调整亮度、对比度、饱和度（颜色扰动）
Normalize	按 ImageNet 均值方差归一化（RGB）
HWC2CHW	通道格式转换：高宽通道 → 通道高宽（适配模型）

如果是验证数据预处理过程对图像进行了如下操作，最后应用数据增强与映射操作，打乱数据顺序，应用分批处理，最终得到处理好的数据集。

操作	描述
Decode	解码图像为 RGB 数组
Resize	统一缩放图像到 256×256
CenterCrop	从中心裁剪为指定大小（如 224×224）
Normalize	同上，进行图像归一化
HWC2CHW	同上，转换图像通道顺序

```
    # define map operations
    decode_op = C.Decode()
    resize_crop_op = C.RandomCropDecodeResize(resize_height, scale=(0.08, 1.0), ratio=(0.75, 1.333))
    horizontal_flip_op = C.RandomHorizontalFlip(prob=0.5)

    resize_op = C.Resize((256, 256))
    center_crop = C.CenterCrop(resize_width)
    rescale_op = C.RandomColorAdjust(brightness=0.4, contrast=0.4, saturation=0.4)
    normalize_op = C.Normalize(mean=[0.485 * 255, 0.456 * 255, 0.406 * 255],
                               std=[0.229 * 255, 0.224 * 255, 0.225 * 255])
    change_swap_op = C.HWC2CHW()

    if do_train:
        trans = [resize_crop_op, horizontal_flip_op, rescale_op, normalize_op, change_swap_op]
    else:
        trans = [decode_op, resize_op, center_crop, normalize_op, change_swap_op]

# 应用数据增强和映射操作
ds = ds.map(operations=trans, input_columns="image", num_parallel_workers=8)
ds = ds.map(operations=type_cast_op, input_columns="label", num_parallel_workers=8)
# shuffle打乱数据顺序，避免训练过拟合于样本顺序
# batch按配置batch_size进行分批处理（不足则丢弃）
# repeat将数据集重复若干次，用于多epoch训练
ds = ds.shuffle(buffer_size=buffer_size)
ds = ds.batch(config.batch_size, drop_remainder=True)
ds = ds.repeat(repeat_num)
```

实际的训练过程如流程图所示，通过预先设定好的参数，对模型的配置进行初始化，初始化配置如下所示：num_classes = 26代表我们的任务是 26 类垃圾分类，activation = "Softmax"证明分类头使用 Softmax 激活，输入数据相关中的图像尺寸为224 x 224，batch_size = 32，每次读取32张图片进行训练，训练超参数中epoch_size = 30，证明要总共训练 30 轮。warmup_epochs = 0证明不启用 warmup（学习率预热）。学习率策略使用的是调度函数get_lr()在lr_generator.py文件中有定义，并且支持 warmup 和衰减，lr_init = 0.0，lr_max = 0.03，lr_end = 0.03。使用 Momentum 优化器：momentum = 0.9，weight_decay = 4e-5



损失函数控制方面，label_smooth = 0.1观察训练脚本中的内容表明启用了标签平滑交叉熵。loss_scale = 1024，证明启用混合精度训练时的

loss scale 因子，每个 epoch 保存一次 checkpoint，最多保留 20 个保存路径为当前目录（"./"），platform：由命令行参数决定（如 CPU / GPU）run_distribute = False：当前训练不启用多卡并行

在训练脚本中的定义如下所示，首先定义网络结构并且加载预训练好的模型

```
# 定义网络结构
backbone_net, head_net, net = define_net(config, args_opt.is_training)
# 加载预训练的backbone并冻结其参数
if args_opt.pretrain_ckpt != "":
    load_ckpt(backbone_net, args_opt.pretrain_ckpt, trainable=False)
```

然后构建损失函数，标签平滑可以使得模型拥有更高的泛化能力，代码如下：

```
# 构建损失函数
if config.label_smooth > 0:
    loss = CrossEntropyWithLabelSmooth(
        smooth_factor=config.label_smooth, num_classes=config.num_classes)
else:
    loss = SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
```

学习率调度使用自己定义的学习率控制函数，便于在不同阶段产生不同的学习率，加快收敛速度，优化器也同时进行定义，代码如下所示：

```
# 构建损失函数
if config.label_smooth > 0:
    loss = CrossEntropyWithLabelSmooth(
        smooth_factor=config.label_smooth, num_classes=config.num_classes)
else:
    loss = SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')

# 学习率调度
epoch_size = config.epoch_size
lr = Tensor(get_lr(global_step=0,
                   lr_init=config.lr_init,
                   lr_end=config.lr_end,
                   lr_max=config.lr_max,
                   warmup_epochs=config.warmup_epochs,
                   total_epochs=epoch_size,
                   steps_per_epoch=step_size))

# 只优化分类头 head_net 的参数
opt = Momentum(filter(lambda x: x.requires_grad, head_net.get_parameters()),
               lr, config.momentum, config.weight_decay)
```

最后实现训练过程的封装，实现前向计算+损失+反向传播的全过程。

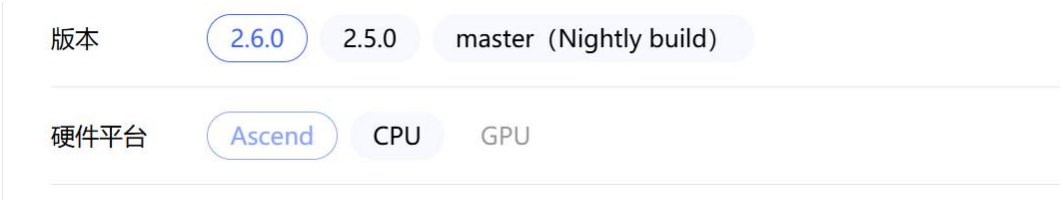
```
# 构建带 loss 的训练网络
network = WithLossCell(head_net, loss)
network = TrainOneStepCell(network, opt)
network.set_train()
```

实际训练过程要加载 .npy 特征向量和标签执行一次前向 + 反向更新（TrainOneStepCell）实际训练并记录 loss

```
# 开始训练
for epoch in range(epoch_size):
    random.shuffle(idx_list)
    epoch_start = time.time()
    losses = []
```

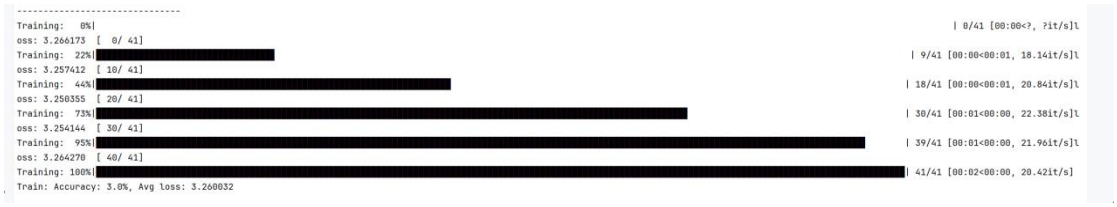
4. 实验结果

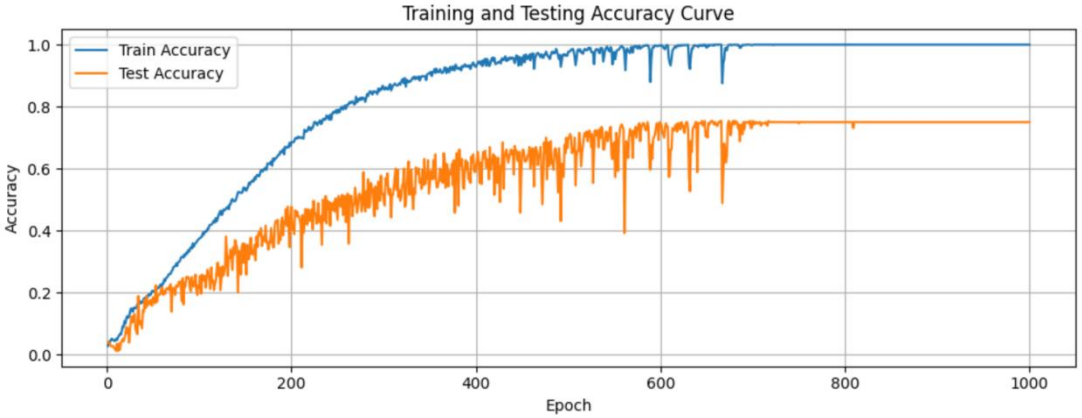
首先配置Mindspore的环境，由于目前版本的Mindspore框架（2.5，2.6）不支持英伟达系列的GPU，因此下面所示的实验都是基于CPU进行训练得到的。

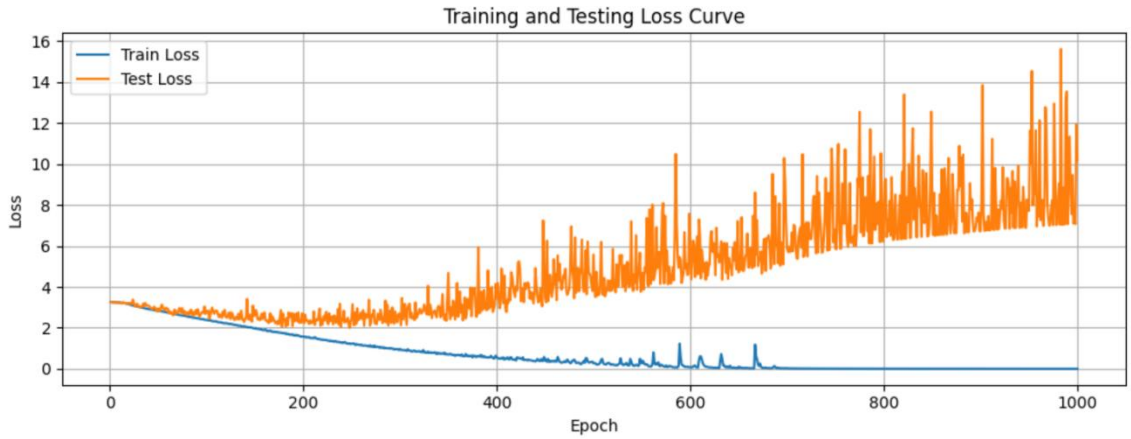



```
(mindspore_py39) C:\Users\32450>python -c "import mindspore;mindspore.set_device(device_target='CPU');mindspore.run_chec
k()"
MindSpore version:  2.5.0
The result of multiplication calculation is correct, MindSpore has been installed on platform [CPU] successfully!

(mindspore_py39) C:\Users\32450>python --version
Python 3.9.21
```

基于原始的LeNet5模型训练得到的结果如下所示，在训练过程中记录准确率与损失，训练过程如下所示，最终得到的曲线如下所示：



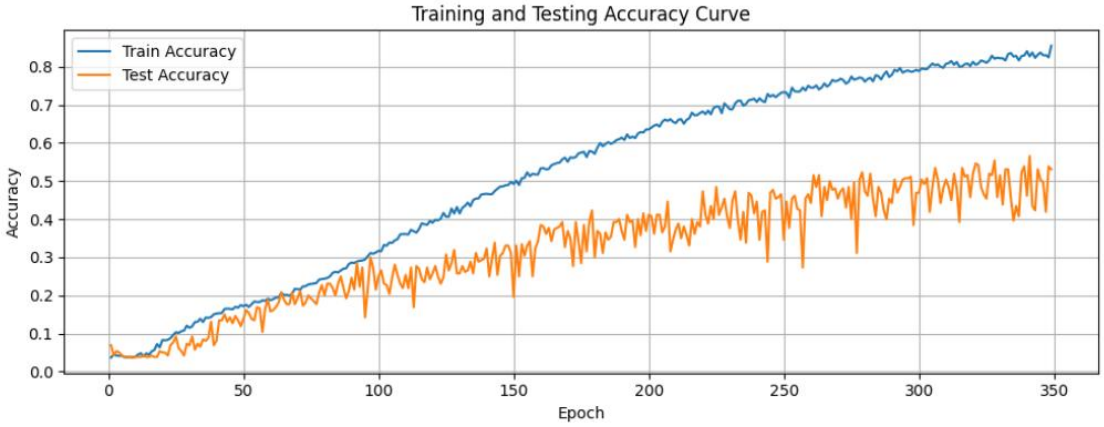


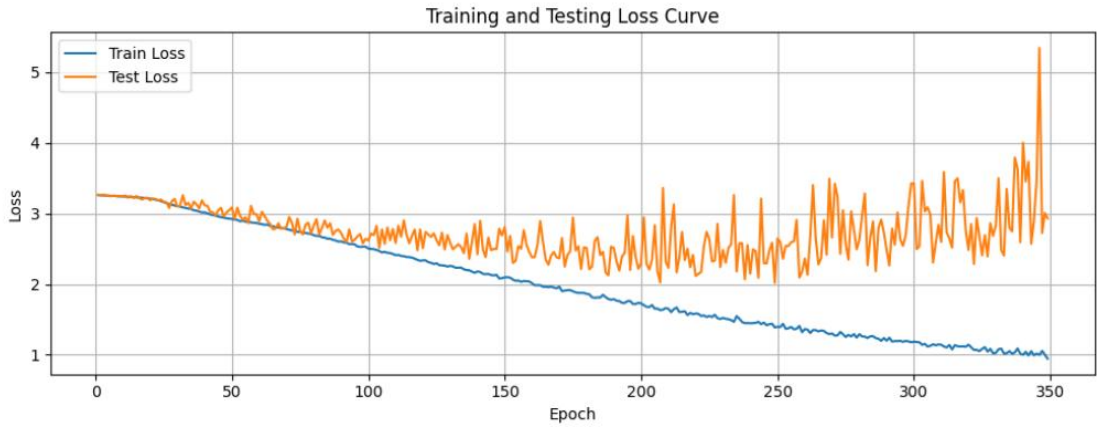


可以发现在训练1000轮次的条件下得到的在训练集上的准确率已经达到了100%但是这也意味着我们出现了过拟合现象，在下图中的loss可以看出，在400轮次之后的loss就出现爆炸的情况，证明我们的模型出现了过拟合现象，为此我们引入早停机制减小过拟合对模型的影响，修改训练循环的代码如下，加入早停机制：这里我们设置容忍轮次为100，观察上面的两张图可以发现大约在300-350轮次的时候得到了较高的准确率并且具有较低的loss值，容忍值设置为100时正好命中这个区间范围内，如果使用早停机制得到的准确率值与损失曲线如下所示：

```python
for t in range(1000):  # 最多训练 1000 轮
    print(f"Epoch {t+1}\n-------------------------------")
    train_loss, train_acc = train(model, train_dataset)
    test_loss, test_acc, conf_matrix = test(model, test_dataset, loss_fn)

    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
    test_losses.append(test_loss)
    test_accuracies.append(test_acc)
    final_conf_matrix = conf_matrix

    # EarlyStopping 判断与模型保存
    if test_loss < best_loss:
        best_loss = test_loss
        early_stop_counter = 0
        save_checkpoint(model, ckpt_file_name: "best_model.ckpt")
        print(" Best model updated.")
    else:
        early_stop_counter += 1
        print(f" No improvement in test loss for {early_stop_counter} epoch(s).")
        if early_stop_counter >= patience:
            print(f" Early stopping triggered at epoch {t+1}")
            break
```

```
⌛No improvement in test loss for 100 epoch(s).
 Early stopping triggered at epoch 346
混淆矩阵已保存
模型已保存到 model.ckpt
✅最佳模型在验证集上的准确率为：44.23%
(mindspore_py39) PS D:\PythonProject\lenet5>
```



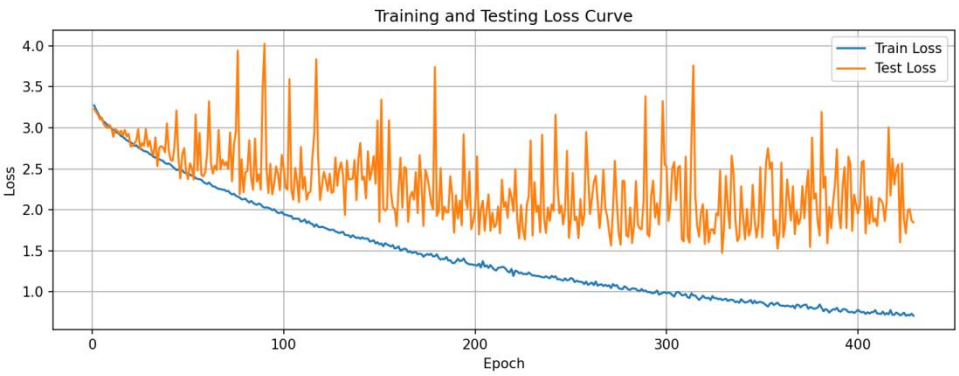


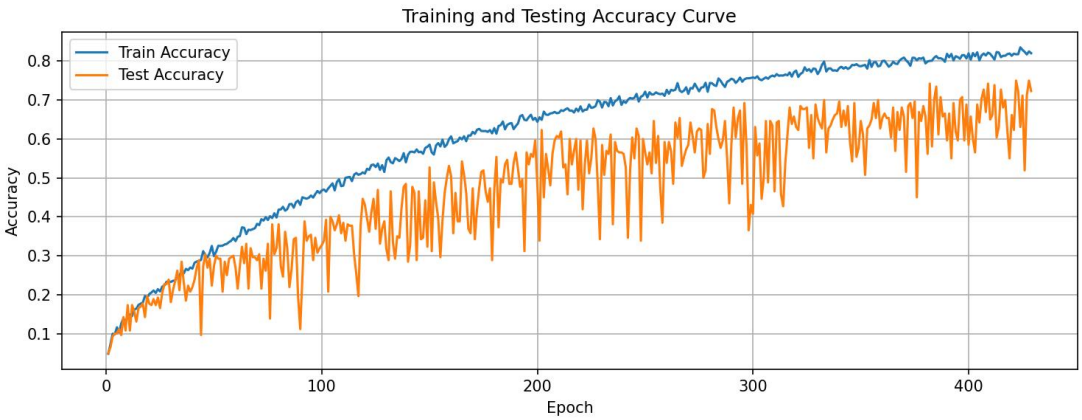
可见引入早停机制之后，loss的值有所减小但仍旧存在爆炸的倾向，并且此时模型在测试集上的准确率有所下降，并且采用最小loss的模型的准确率为44%。这样的准确率实属有点低，因此下面对LeNet5的模型结构进行优化，结果如下所示：

```python
class LeNet5(nn.Cell):
    def __init__(self, num_classes=26):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d( in_channels: 1,  out_channels: 6, kernel_size=5, pad_mode='valid')
        self.bn1 = nn.BatchNorm2d(6)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d( in_channels: 6,  out_channels: 16, kernel_size=5, pad_mode='valid')
        self.bn2 = nn.BatchNorm2d(16)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)

        self.flatten = nn.Flatten()
        self.fc1 = nn.Dense(16 * 4 * 4,  out_channels: 84)  # ✅ 降低维度
        self.dropout1 = nn.Dropout(keep_prob=0.5)
        self.fc2 = nn.Dense( in_channels: 84, num_classes)

    def construct(self, x):
        x = self.pool1(self.relu1(self.bn1(self.conv1(x))))
        x = self.pool2(self.relu2(self.bn2(self.conv2(x))))
        x = self.flatten(x)
        x = self.dropout1(self.fc1(x))
        x = self.fc2(x)
        return x
```

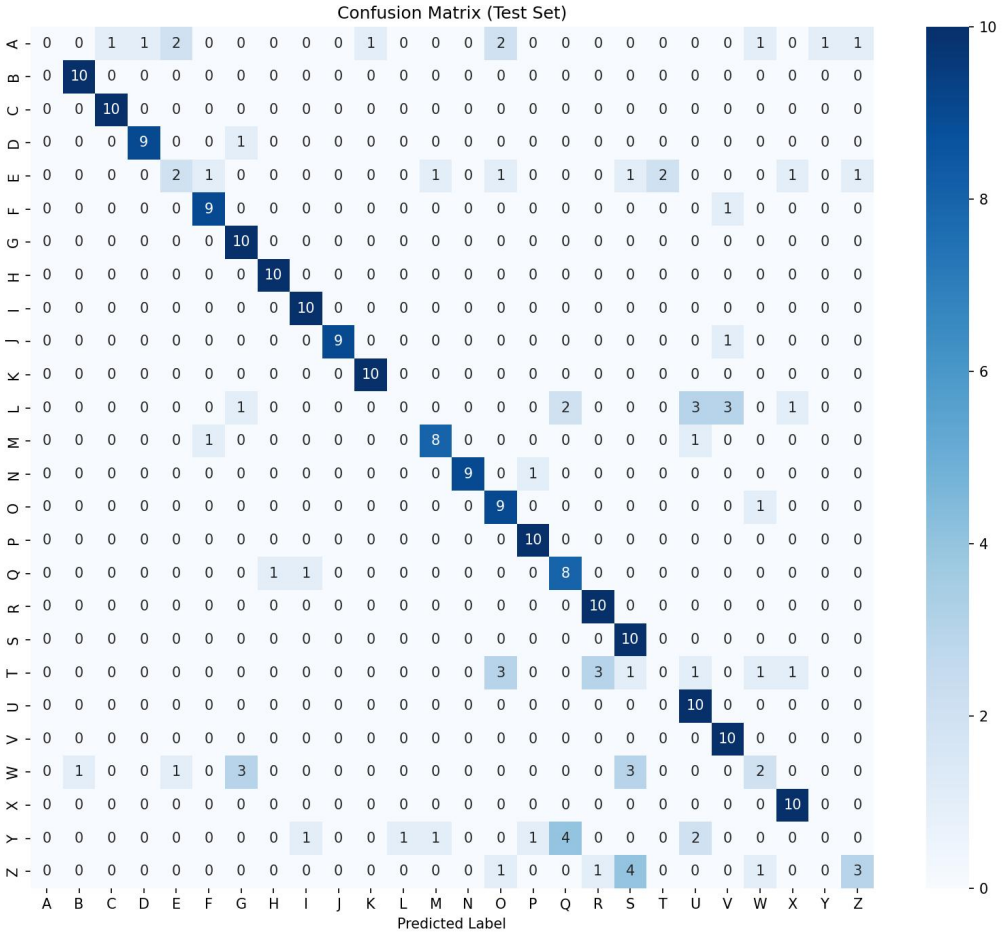
对比原始的LeNet5网络，采在每个模块的位置加入BatchNorm,使得激活值的分布更加稳定并且可以防止梯度震荡，并且为了降低过拟合的风险，将fc1层的维度降低，控制全连接层参数量，并且加入dropout部分，部分抑制神经元提升泛化效果。同时依旧保留早停机制，最后的结果如下所示：

```
Test: Accuracy: 72.3%, Avg loss: 1.845883

 No improvement in test loss for 100 epoch(s).
 Early stopping triggered at epoch 429
混淆矩阵已保存
模型已保存到 model.ckpt
[WARNING] ME(35716:17368,MainProcess):2025-05-2
✅最佳模型在验证集上的准确率为：68.85%
```





可以发现改进模型之后我们在测试集上的准确率达到了68%，并且测试集的损失曲线对比之前的损失曲线没有超过4，更加平缓，在训练集上的损失曲线下降更快更加平缓，证明我们的改进是有效果的，虽然达不到之前的1000轮次下最佳的准确率75%但是不论是loss以及训练时间都更优。最后给出我们绘制的混淆矩阵，如图所示：



对于垃圾分类任务修改默认的参数其实原始项目的参数已经给咱们修改好了，只需要简单地提高总的训练轮次就可以获得较高的准确率，修改之后的参数如上所示：

```python
def set_config(args):
    config_cpu = ed({
        # 数据相关
        "num_classes": 26,  # 垃圾分类的类别数，共有 26 类
        "image_height": 224,  # 输入图像的高度（MobileNetV2 默认输入 224×224）
        "image_width": 224,  # 输入图像的宽度
        "batch_size": 32,  # 每次训练的样本批量大小
        # 训练控制
        "epoch_size": 30,  # 总共训练的轮数（Epoch 数）
        "warmup_epochs": 0,  # 学习率预热的轮数（warmup 设为 0 表示无 warmup）
        # 学习率调度（一般为 cosine、step、linear 等调度策略）
        "lr_init": .0,  # 学习率初始值（预热阶段的起点），如果没有预热可为 0
        "lr_end": 0.03,  # 学习率训练结束时的值（一般用于 cosine 衰减）
        "lr_max": 0.03,  # 学习率的最大值（主学习率）
        # 优化器参数
        "momentum": 0.9,  # Momentum 优化器的动量参数
        "weight_decay": 4e-5,  # 权重衰减（L2 正则项系数），防止过拟合
        # 损失函数设置
        "label_smooth": 0.1,  # 标签平滑系数（用于 CrossEntropyWithLabelSmooth）
        # 防止模型过于自信，提升泛化能力。0 表示不使用标签平滑。
        # 数值稳定性与精度
        "loss_scale": 1024,  # 混合精度训练时的损失缩放因子，避免梯度下溢（特别用于 float16）
        # Checkpoint（模型保存）相关
        "save_checkpoint": True,  # 是否保存训练过程中的模型
        "save_checkpoint_epochs": 1,  # 每训练多少个 epoch 保存一次模型
        "keep_checkpoint_max": 20,  # 最多保留多少个最近的 checkpoint 文件
        "save_checkpoint_path": "./",  # 保存 checkpoint 的路径
        "platform": args.platform,  # 当前运行平台（如 CPU、GPU、Ascend 等）
        "run_distribute": False,  # 是否启用分布式训练（一般 CPU 设为 False）
        "activation": "Softmax"  # 分类 head 的输出激活函数类型（Softmax / Sigmoid / None）
    })
```

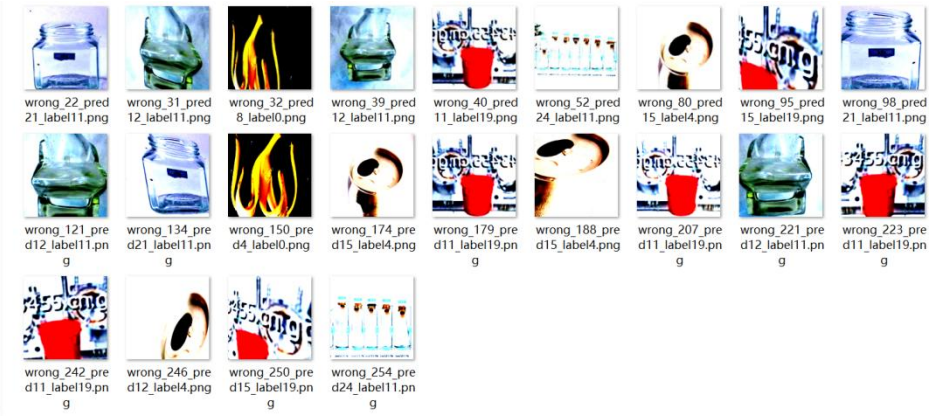
上述参数是反复尝试之后的结果，比如训练的总轮数以及批量样本大小，最终得到了较高的准确率。在控制台中输入参数，训练过程如下所示：

```
epoch[28/30], iter[81] cost: 2465.107, per step time: 30.433, avg loss: 0.760
epoch[29/30], iter[81] cost: 2356.244, per step time: 29.089, avg loss: 0.765
epoch[30/30], iter[81] cost: 2262.117, per step time: 27.927, avg loss: 0.756
total cost 85.9219 s
```

可以看到平均损失在不断降低，证明训练效果不错，最终花费了一分多钟完成训练。验证的结果如下：可见准确率很高，达到令我们满意的效果。

```
result:{'acc': 0.9140625}
pretrain_ckpt=./ckpt_0/mobilenetv2_30.ckpt
(mindspore_py39) PS D:\PythonProject\mobilenetv2\code>
```

我还修改了一下估计脚本，使得最终模型可以输出错误的被分类的图片，结果如下所示：



可以发现在预处理数据集的时候还对图像进行了增强，比如颜色扰动等，但是最后依旧出现了一些问题，比如说第一个图像把玻璃瓶识别成了塑料瓶，第二个图把玻璃瓶识别成了玻璃器皿，其实二者的区别不是很大，出现错误也是情有可原，能达到91%的准确率证明模型的识别效果依旧很优秀。

5.加分项

5.1 Lora微调

上述基础的部分采用的是冻结backbone层只对head层进行微调但是效率较低，30轮次下的训练时间在1分钟以上，准确率达到了91%，是否还有训练效率更高的方法呢？其实是有的就是Lora微调。假设模型中的某个全连接层的权重矩阵为W，标准微调下会更新这个矩阵，Lora微调则将其改写为原始矩阵和一个由两个低秩矩阵（A、B）相乘的矩阵加和的结果，即只训练A、B，并将其输出作残差叠加到原来的模型输出中。

假设原始的全连接层的参数量为d1*d2，Lora引入的参数量为r*（d1+d2）当r远小于d的时候参数量会大大下降，并且原始的权重W被完全冻结，不占用反向传播占用显存。以下使用使用使用Lora方法代替最终的Dense层因为Dense层的参数量很大，并且Dense层决定最终输出，改动的话对精度1影响比较显著。

首先定义lora神经网络，由于Mindspore中没有现成的lora模型，因此调用cell基类手动构造lora层，代码如下：

```python
class LoRALinear(nn.Cell):
    def __init__(self, in_features, out_features, r=4, alpha=1.0):
        super(LoRALinear, self).__init__()
        self.r = r
        self.alpha = alpha
        self.scale = alpha / r if r > 0 else 1

        # 原始全连接层权重（被冻结，不训练）
        self.weight = nn.Dense(in_features, out_features, has_bias=False)
        self.weight.to_float(mindspore.float32)
        self.weight.weight.requires_grad = False  # 冻结主干参数

        # LoRA A 和 B 层：低秩适配模块
        self.lora_a = nn.Dense(in_features, r, has_bias=False)
        self.lora_b = nn.Dense(r, out_features, has_bias=False)

    def construct(self, x):
        # 原始权重部分 + LoRA 部分
        return self.weight(x) + self.scale * self.lora_b(self.lora_a(x))
```

然后在mobilenetV2.py中修改模型结构使其支持lora，并且通过输入的参数控制是使用原始的head层还是使用lora层，代码如下：

```python
# 选择 Dense 或 LoRA 线性层
if use_lora:
    dense_layer = LoRALinear(input_channel, num_classes, r=r, alpha=alpha)
else:
    dense_layer = nn.Dense(input_channel, num_classes, has_bias=True)

if not has_dropout:
    head = [
        GlobalAvgPooling(),
        dense_layer
    ]
else:
    head = [
        GlobalAvgPooling(),
        nn.Dropout(0.2),
        dense_layer
    ]
self.head = nn.SequentialCell(head)
self._initialize_weights()
```

然后再models.py中的define_net函数中再加入新的使用lora的代码，使得在构建模型的时候可以正确加载lora层的部分，代码如下所示：

```python
# 网络结构构建函数（支持 use_lora 控制）
def define_net(config, is_training):
    # Backbone: MobileNetV2 特征提取
    backbone_net = MobileNetV2Backbone()

    # 分类头激活函数（训练阶段一般不用激活）
    activation = config.activation if not is_training else "None"

    # Head: 分类头，支持是否使用 LoRA
    head_net = MobileNetV2Head(
        input_channel=backbone_net.out_channels,
        num_classes=config.num_classes,
        activation=activation,
        has_dropout=config.has_dropout if hasattr(config, "has_dropout") else False,
        use_lora=config.use_lora if hasattr(config, "use_lora") else False,
        r=config.lora_rank if hasattr(config, "lora_rank") else 4,
        alpha=config.lora_alpha if hasattr(config, "lora_alpha") else 16
    )

    # Combine 网络
    net = mobilenet_v2(backbone_net, head_net)
    return backbone_net, head_net, net
```

在train.py部分，注入了lora的参数以及使用Adam优化器，代替原来的Momentum优化器，其更适合lora微调方法，最后在args.py添加lora命令行参数支持，包括训练阶段以及验证阶段。添加的代码如下：

```
def train_parse_args():
    for fine tune or incremental learning')
    train_parser.add_argument( *name_or_flags: '--freeze_layer', type=str, default="", choices=["", "none", "backbone"], \
        help="freeze the weights of network from start to which layers")
    train_parser.add_argument( *name_or_flags: '--run_distribute', type=ast.literal_eval, default=True, help='Run distribute')

    train_parser.add_argument( *name_or_flags: '--use_lora', type=bool, default=False, help='Enable LoRA in classification head')
    train_parser.add_argument( *name_or_flags: '--lora_rank', type=int, default=4, help='LoRA rank (low-rank dim)')
    train_parser.add_argument( *name_or_flags: '--lora_alpha', type=float, default=16.0, help='LoRA scaling factor')

    train_args = train_parser.parse_args()
    train_args.is_training = True
    return train_args

def eval_parse_args():
    eval_parser = argparse.ArgumentParser(description='Image classification eval')
    eval_parser.add_argument( *name_or_flags: '--platform', type=str, default="Ascend", choices=("Ascend", "GPU", "CPU"), \
        help='run platform, only support GPU, CPU and Ascend')
    eval_parser.add_argument( *name_or_flags: '--dataset_path', type=str, required=True, help='Dataset path')
    eval_parser.add_argument( *name_or_flags: '--pretrain_ckpt', type=str, required=True, help='Pretrained checkpoint path \
        for fine tune or incremental learning')
    eval_parser.add_argument( *name_or_flags: '--run_distribute', type=ast.literal_eval, default=False, help='If run distribute in GPU.')
    eval_parser.add_argument( *name_or_flags: '--use_lora', type=bool, default=False, help='Whether to use LoRA structure')
    eval_parser.add_argument( *name_or_flags: '--lora_rank', type=int, default=4, help='LoRA rank')
    eval_parser.add_argument( *name_or_flags: '--lora_alpha', type=float, default=16.0, help='LoRA alpha scaling')
```

```
# 只优化分类头 head_net 的参数
#opt = Momentum(filter(lambda x: x.requires_grad, head_net.get_parameters()),
opt = Adam(filter(lambda x: x.requires_grad, head_net.get_parameters()),
           learning_rate=lr,
           weight_decay=config.weight_decay)

if args_opt.pretrain_ckpt != "":
    load_ckpt(backbone_net, args_opt.pretrain_ckpt, trainable=False)

    # 冻结 backbone 的参数（确保只训练 head 或 LoRA）
    for param in backbone_net.get_parameters():
        param.requires_grad = False
else:
    raise ValueError("Pretrained checkpoint required for fine-tuning head only.")
```

Lora方法最终的准确率要低于原先的微调方法，可能是因为Lora采用的参数量较少，或者是分类任务本身很简单，Lora层在分类头的位置的话效果反而可能不如原始的Dense层的分类效果，训练30轮的效果最后的准确率仅为66%，过低因此不在这里展示。训练50轮之后的准确率如下：训练时间对比之前的冻结head层微调大大降低，此时的秩为4/8，缩放因子为16。准确率已经较高并且花费时间是之前冻结head层微调的一半都不到可见Lora方法是是十分有效的。

```
epoch[50/50], iter[81], cost: 606.381 ms, step: 7.486 ms, avg loss: 1.740
total cost 32.7283 s
```

秩为4的时候损失过大，因此不进行验证

```
epoch[50/50], iter[81], cost: 608.920 ms, step: 7.518 ms, avg loss: 0.905
total cost 32.2523 s
```

秩为8的时候损失小于1，因此进行验证

```
[Eval Finished] Total: 256, Correct: 220, Acc: 0.8594
[Wrong Samples Saved to]: ./wrong_samples
```

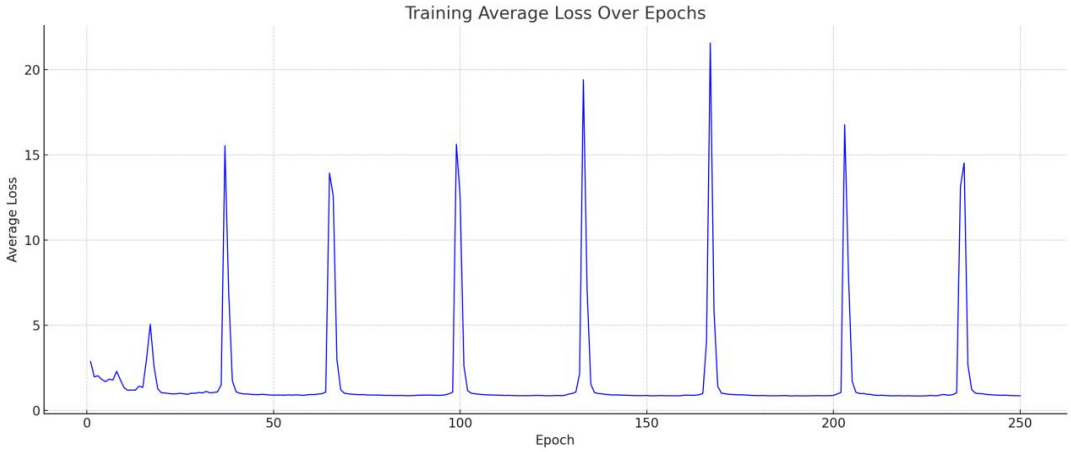
秩为8缩放因子为16的情况下使用lora微调的准确率已经达到了85.9%

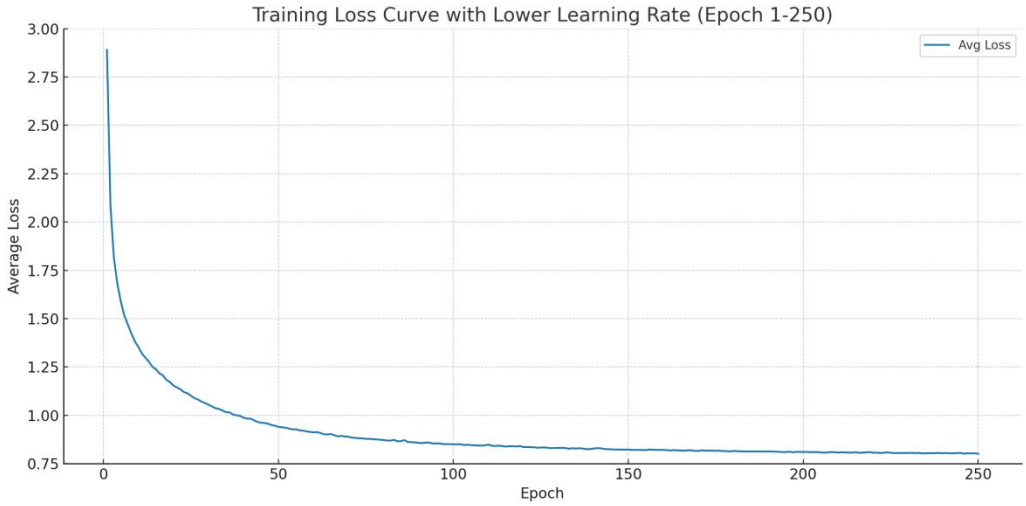
```
epoch[249/250], iter[81], cost: 981.484 ms, step: 12.117 ms, avg loss: 0.872
epoch[250/250], iter[81], cost: 950.867 ms, step: 11.739 ms, avg loss: 0.869
total cost 171.9642 s
[Eval Finished] Total: 256, Correct: 229, Acc: 0.8945
```

最终模型的损失趋近于0.85较优效果的模型的准确率可以接近90%

这里给出测试Lora微调下250轮次的平均损失的折线图，可以看到存在若干异常的loss经过排查发现可能是由于学习率设置过大（0.03）忘记修改导致的，如果学习率不改变的话在多epoch下会出现异常突升的损失值，修改学习率为0.001之后损失曲线恢复正常的收敛模式，下面给出了250轮次的非正常/正常的损失值曲线图及此时50轮次的模型的准确率。

```
[Eval Finished] Total: 256, Correct: 230, Acc: 0.8984
[Wrong Samples Saved to]: ./wrong_samples
```





5.2 全量微调

全量微调其实指的是解冻backbone层，在原始权重模型的基础上进行微调，但是这样微调的结果却出现问题，如果加载原始的参数模型的话就出现loss爆炸的情况，过程截图如下：

```
epoch: [  1/100], step:[   12/   23], loss:[14.831/9.093], time:[2933.124], lr:[0.030]
epoch: [  1/100], step:[   13/   23], loss:[14.988/9.514], time:[2915.779], lr:[0.030]
epoch: [  1/100], step:[   14/   23], loss:[14.488/9.846], time:[2856.017], lr:[0.030]
epoch: [  1/100], step:[   15/   23], loss:[14.485/10.136], time:[2742.085], lr:[0.030]
epoch: [  1/100], step:[   16/   23], loss:[13.459/10.331], time:[2833.706], lr:[0.030]
epoch: [  1/100], step:[   17/   23], loss:[15.395/10.612], time:[2751.660], lr:[0.030]
epoch: [  1/100], step:[   18/   23], loss:[14.460/10.815], time:[2747.320], lr:[0.030]
```

第二轮开始的时候loss就已经发生了爆炸，接下来继续训练已经没有意义了。推测原因可能为如下原因：首先最有可能是预训练权重与当前任务差异过大，特征分布不匹配，原始的预训练权重来源于ImageNet数据集，现在的数据集是26类垃圾，此时预训练网络已经习惯于捕捉ImageNet中有用的特征，而对于新的数据集，对于新的图像会导致从输入到输出的整个路径上的特征全部失效。这会导致梯度方向混乱，此外全量微调属于强更新，这样带来的梯度震荡。反而导致出现梯度爆炸。采用warm_up策略之后依旧没有很好改善上述问题。

```
epoch: [  1/100], step:[   65/  186], loss:[8.971/6.848], time:[407.327], lr:[0.008]
epoch: [  1/100], step:[   66/  186], loss:[6.526/6.843], time:[422.614], lr:[0.008]
epoch: [  1/100], step:[   67/  186], loss:[9.406/6.881], time:[428.090], lr:[0.008]
epoch: [  1/100], step:[   68/  186], loss:[9.838/6.924], time:[411.467], lr:[0.008]
epoch: [  1/100], step:[   69/  186], loss:[9.960/6.967], time:[411.397], lr:[0.008]
epoch: [  1/100], step:[   70/  186], loss:[4.395/6.931], time:[442.530], lr:[0.008]
epoch: [  1/100], step:[   71/  186], loss:[9.198/6.962], time:[473.088], lr:[0.008]
```

5.3 部分冻结微调

在基础项中已经通过冻结backbone层得到了令人满意的效果，下面探究解冻backbone部分的部分层对模型的影响。原始的MobileNetV2模型具有19层，现在解冻后三层观察模型训练情况，训练过程中发现损失值依旧不收敛，并且100轮次训练之后的损失值依然大于2，证明之前全量微调出现的“灾难性遗忘”错误依旧出现。下面的结果是在极低的学习率以及使用Adam优化器之后的结果，如果保持较高的学习率或者Momentum优化器，将在起始的轮次就出现极高的loss爆炸的情况，但是目前如下所示的解冻之后的微调训练已经丧失了意义，证明在加载原始模型的参数的基础上再继续微调会导致灾难性遗忘，证明至少在本项目中采用解冻backbone的策略是不合适的。

```
epoch time: 61366.748, per step time: 757.614, avg loss: 2.829
============== 训练结束 ==============
total cost 5556.1089 s
```

5.4 不同微调方法的对比

基于上述分析可以发现，在我们的任务下使用全量微调以及部分冻结微调都是不合适的，原因在上面已经讨论过了。为了便于对比不同模型的效果，这里给出从头完整训练即不使用原始模型的参数进行训练的模型作为baseline，baseline模型的准确率如下所示：可见我们的Lora微调以及冻结微调都比从头训练的效果要更优。

```
result:{'acc': 0.46875}
pretrain_ckpt=../code/ckpt_0/mobilenetv2_2-100_186.ckpt
(mindspore_py39) PS D:\PythonProject\mobilenetv2_change\code>
```

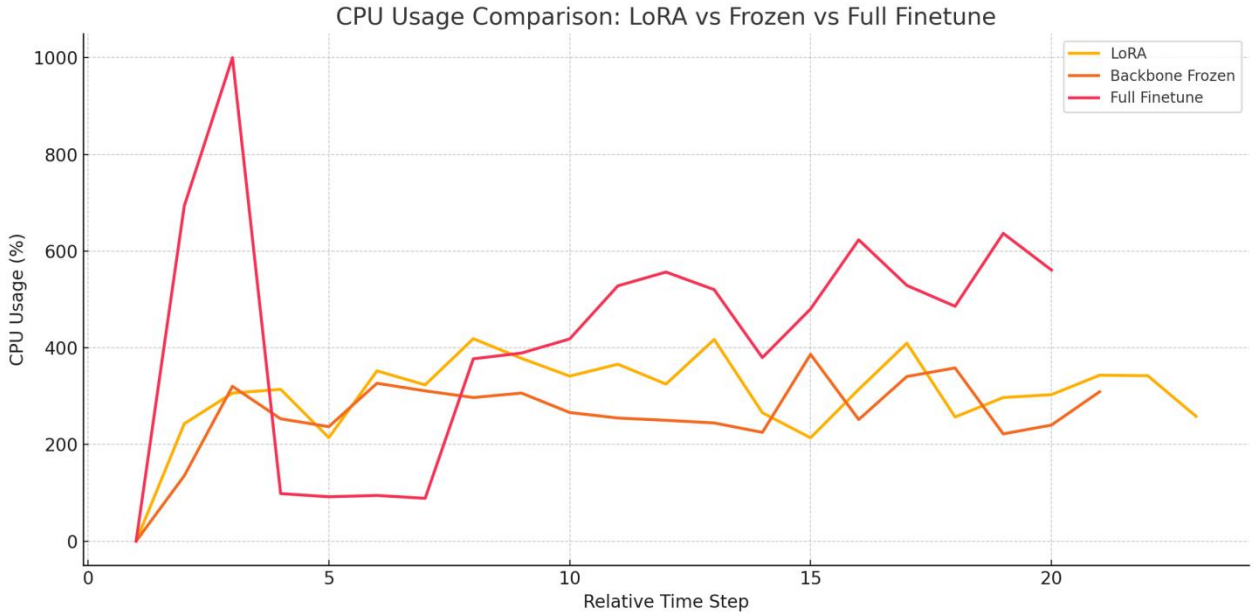
```
epoch[30/30], avg loss: 1.055, time: 599.658 ms
Total time: 19.5152s

[Eval Finished] Total: 256, Correct: 219, Acc: 0.8555
[Wrong Samples Saved to]: ./wrong_samples
```

采用Lora微调的30轮次下不算特征提取的时间接近20s，此时的学习率为0.001

```
epoch[30/30], avg loss: 0.756, time: 591.717 ms
Total time: 19.2553s

[Eval Finished] Total: 256, Correct: 235, Acc: 0.9180
[Wrong Samples Saved to]: ./wrong_samples
```

采用冻结微调30轮次下的时间接近Lora微调，此时的学习率为0.03，大于Lora

这里给出冻结微调、Lora微调、全量微调CPU使用率的对比



LoRA微调的峰值约为 418.8%，相对稳定在 250%~420% 区间，虽然主干冻结，但由于在每层加入LoRA模块，仍需在主干中计算权重插值，因此相较于纯 head 微调，CPU压力略高。CPU利用率有一定波动，体现出在不同 batch 下参数更新和图构建开销不同。

Backbone Frozen（分类头微调）峰值约为386.4%，波动范围为200%~350%特征：仅参与前向与反向传播的参数数量最少（只有 head），计算图简洁。整体CPU占用偏低，波动也较平稳。是三种方式中计算最轻的一种，适用于资源受限或快速实验场景。

全量微调峰值达到了 1000.2%，远超其他方式，高频波动在 400%~700%，瞬时峰值超过 900%所有层参数都在反向传播中参与梯度计算和更新，导致计算图最大，内存和计算开销双高。频繁触发GC、数据传输、缓存刷新等系统级开销，使得CPU利用率高且不稳定。虽然能带来最充分的性能挖掘，但开销代价极高。

5.5 模型的优化以及讨论

查阅资料发现，MobileNetV3网络在V2网络问世一年之后就出现了，这里引用其中的分类头的部分对模型进行改进，给出改进之后的head部分如下所示：

```
class MobileNetV2HeadV3Style(nn.Cell):
    def __init__(self, input_channel=1280, mid_channel=1280, num_classes=1000,

        # 可选输出激活函数
        self.need_activation = True
        if activation == "Sigmoid":
            self.activation = P.Sigmoid()
        elif activation == "Softmax":
            self.activation = P.Softmax()
        else:
            self.need_activation = False

        layers = [
            GlobalAvgPooling(keep_dims=True),  # 输出为 [B, C, 1, 1]
            nn.Conv2d(in_channels=input_channel, out_channels=mid_channel,
                      kernel_size=1, has_bias=False, pad_mode='pad'),
            nn.BatchNorm2d(mid_channel),
            nn.HSwish()
        ]

        if has_dropout:
            layers.append(nn.Dropout(0.2))

        layers.append(nn.Conv2d(in_channels=mid_channel,
                                out_channels=num_classes,
                                kernel_size=1, has_bias=True, pad_mode='pad'))

        self.squeeze = P.Squeeze(axis=(2, 3))  # 去除 H, W
        self.head = nn.SequentialCell(layers)

        self._initialize_weights()
```

用1×1卷积代替Dense，直接作用在[B, C, 1, 1]特征上，不需要reshape，减少内存拷贝；此外增加中间通道扩展（mid_channel）nn.Conv2d(input_channel → mid_channel) → BN → HSwish并且引入瓶颈结构（类似于 ResNet bottleneck block）增强非线性建模能力，类似于 MobileNetV3 中的最后几层处理。激活函数从 ReLU转化为HSwish原 V2 可能使用默认Dense无激活或仅 Softmax；V3风格使用HSwish（Hard Swish）是一种轻量但连续的非线性函数；在小模型中相比 ReLU / ReLU6 表现更好，更易训练，不会产生梯度消失。保持 GlobalAvgPooling，不变保留全局平均池化操作作为特征压缩方式（[B, C, H, W] → [B, C, 1, 1]）；保证模型轻量且无全连接层依赖。 5. 仍支持 Softmax/Sigmoid 可选激活保留原有分类激活逻辑；可根据 activation 参数决定是否附加激活。

最终改进之后的模型的准确率与之前持平但是训练时间有所缩短，这与我之前的推测有所出入，推测可能是替代dense层为卷积层使得卷积层可以原地处理图像信息，不需要reshape以及内存复制的操作，并且CPU对卷积层的适配率也更高，换用HSwish后开销并没有明显的增长。

原始冻结微调方法

```
epoch[30/30], avg loss: 0.756, time: 743.572 ms
Total time: 23.8832s
```

改进之后的方法

```
epoch[30/30], avg loss: 0.756, time: 650.781 ms
Total time: 22.1843s
```

6.总结与分析

通过此次实验使我对微调大模型有了更加直观的了解，对于第一个手写数字识别，在了解LeNet5模型的基础上使用该模型对垃圾分类任务进行了完成。针对第二个实验由于要求的是微调，并且本身的模型规模不是很大，因此直接可以利用CPU进行微调训练最后也得到了不错的结果。

通过实验也有看出在有限资源下，采取模型微调训练远比从头训练要好的多，并且由于我们采用的是一个轻量级的模型，因此计算开销也不是很高，这也启发我们在资源不足但是仍旧需要模型支持的时候，微调大模型是一个很好的方法，下一个实验也即将用到微调模型的方法，希望我会有一个更加直观与全面的理解。

基于原始git上的项目做了如下修改，首先对项目进行扩展改写train部分使其适配更多任微调模式，并且对错误分类的结果进行了可视化，对原始模型进行了Lora扩展使其可以进行lora微调训练，参照V3模型的思路对原始模型进行了优化处理，并且在训练效率上有所提升，准确率方面由于本身的任务限制并没有提升。

7.参考文献

[深度学习 --- 卷积神经网络CNN（LeNet-5网络详解）-CSDN博客]

[course/fine_tune at master • mindspore-ai/course]

[7.1 MobileNet网络详解_哔哩哔哩_bilibili]