第3章

EfficientDet与美食场景检测

当读完本章时，应该能够：

- 熟悉并理解美食数据集的结构特点。
- 了解解决目标检测问题的技术路线。
- 掌握一种为数据集做标签的方法。
- 理解并掌握 EfficientDet-D0～EfficientDet-D7 模型的体系结构与工作原理。
- 基于 TFLite Model Maker 做迁移学习。
- 基于 TFLite Task Library 在 Android 上部署 TFLite 模型。
- 基于 mAP 指标评价目标检测模型。
- 民以食为天，即刻拥有在美食领域创业的冲动与梦想。

3.1 项目动力

美食是人类追求美好生活的应有之义。美食关系健康，例如，人体必需的八种氨基酸不能体内合成，需要从食物中摄取。在中国数千年的饮食文化岁月里，美食是区域文化符号，体现了区域特色，也体现了人们的创造与追求。中央电视台一度热播的纪录片《舌尖上的中国》将美食与健康、美食与文化、人们对美食的创造与演绎表达得淋漓尽致。

大千世界，美食多姿多彩，美食背后蕴含的知识也是海量的。如果人们在一起聚会聊天时，借助 AI 技术，对着餐桌上的美食拍一下，对那些即便不太熟悉的食材，也能迅速得知其产地习性、历史传承、营养成分、烹饪方法、饮食禁忌等知识，着实令人神往。

基于上述项目初心，本章案例将从零起步，从数据集的采集与标签定义，到 EfficientDet 模型解读，再到模型训练、评估、迁移、部署和应用，实现美食场景检测中最富创造力的一个环节，即自动区分食材类别。

正确界定食材类别是构建手机版美食应用的关键。关于食材的其他相关知识，可以通过构建数据库的方式完成，限于篇幅，数据库的设计不作为本章项目的内容。

3.2　技术路线

目标检测主要有两种技术路线：一种是 Two-Stage 检测方法；另一种是 One-Stage 检测方法。

（1）Two-Stage 检测方法。将检测逻辑划分为两个阶段，首先产生候选区域，然后对候选区域进行校正和分类。这类算法的典型代表是基于候选区域的 R-CNN 系列算法，如 R-CNN、Fast R-CNN、Faster R-CNN、Mask R-CNN 等。

（2）One-Stage 检测方法。不需要产生候选区域（Region Proposal）阶段，直接产生目标的坐标值和类别概率值，经典的算法如 SSD、YOLO 和 EfficientDet 等。

EfficientDet 采用的骨干分类网络是 EfficientNet，正如 EfficientNet 是一个系列模型（EfficientNet-B0～EfficientNet-B7），同时 EfficientDet 也是一个适应不同规模需求的模型系列，包括 EfficientDet-D0～EfficientDet-D7。

图 3.1 显示了 EfficientDet 系列模型与其他目标检测模型在计算量与准确率两个维度上的对比。

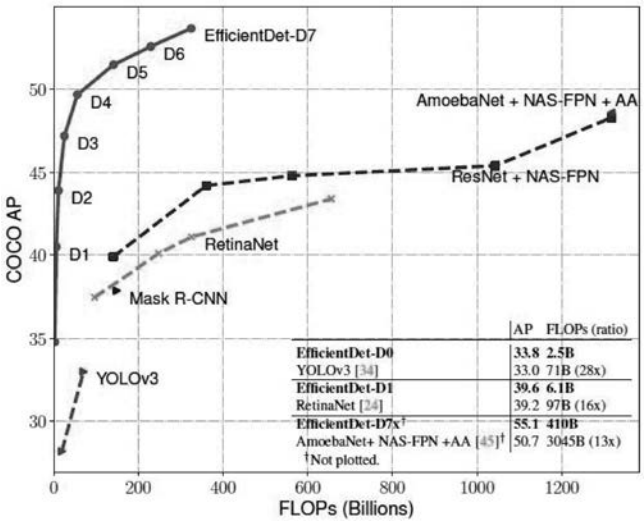


图 3.1　EfficientDet 与其他模型对比

EfficientDet 在计算量与准确率两个指标上，显著领先于之前的其他经典模型。EfficientDet-D0 的准确率比 YOLOv3 稍高，但是计算量只有其 1/28。从 EfficientDet-D4 开始，在计算量相当或较低的情况下，其准确率已经显著领先于 Mask R-CNN、RetinaNet、ResNet＋NAS-FPN 等模型。

3.3 MakeSense 定义标签

本节介绍的数据集标注工具软件 MakeSense 是一款为数据集打标签的免费在线软件，不需要本地安装，入手简单，支持多种数据集格式。

MakeSense 支持分类任务或者目标检测任务。输出的文件格式包括 YOLO、VOC XML、VGG JSON 和 CSV 等。对于目标检测问题，支持的标签类型包括点、线段、矩形框和多边形。官方网站工作地址为 https://www.makesense.ai/。

在官方网站首页右下角有一个名称为 Get Started 的按钮，单击该按钮，打开工作界面，如图 3.2 所示，该界面提供了目标检测和图像识别两种工作模式。

将需要做标注的图片拖放到中央的大矩形框中，单击 Object Detection 按钮，首先会弹出一个询问界面，要求用户给定数据集标签列表，如图 3.3 所示，用户既可以一次性导入数据集的标签列表，也可以单击左上角的“+”按钮，临时定义标签列表。

图 3.2 MakeSense 首页工作界面

图 3.3 定义数据集标签列表

创建标签列表后，即可为指定的图片做标签。可选择一批图片上传到 MakeSense 中，如图 3.4 所示，从左侧列表中选择图片，在中央工作区拖动鼠标，定义矩形框，框住目标，在中央工作区的右侧，右上角有标签选择栏，右下角有边界形状选择栏，共同确定本次标注内容的位置和类型。

本节视频教学中随机完成了 5 幅图像的标注工作，当完成全部图片标注时，单击 MakeSense 顶部导航栏 Actions 中的 Export Annotations 命令，弹出如图 3.5 所示的对话框，选择导出文件的格式，执行 Export 命令，导出数据集标签文件。

图 3.4 用 MakeSense 定义标签

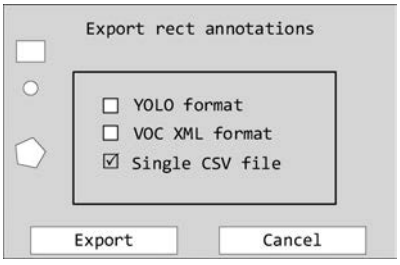
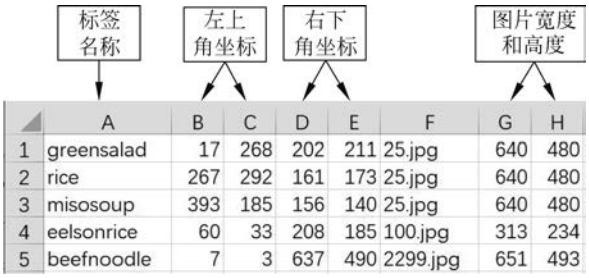


图 3.5 导出数据集标签文件

打开数据集标签文件，内容如图 3.6 所示，其中只包含做过标注的图片，没有做标注的图片不在其中。



	A	B	C	D	E	F	G	H
1	greensalad	17	268	202	211	25.jpg	640	480
2	rice	267	292	161	173	25.jpg	640	480
3	misosoup	393	185	156	140	25.jpg	640	480
4	eelsonrice	60	33	208	185	100.jpg	313	234
5	beefnoodle	7	3	637	490	2299.jpg	651	493

图 3.6　数据集标签文件结构

A 列为标签的名称，B、C、D、E 4 列依次是矩形框的左上角（x1，y1）与右下角（x2，y2）坐标，F 列表示文件名称，G、H 列分别表示图片的宽度与高度。

显然，当数据量很大时，数据标注是一项耗费人力和时间的工作。

3.4　定义数据集

虽然可以采用 3.3 节的方法为数据集做标签，但是采集足够多的数据是一项富有挑战性的工作，事实上，本项目落地的一个前提即是构建超大的美食数据集。为了演示需要，本章项目采用的美食数据集来自 UEC FOOD 100 数据集，由日本电子通信大学食品识别研究小组发布，数据集下载地址为 http://foodcam.mobi/dataset.html。

UEC FOOD 100 数据集定义了 100 种美食对应的图片和标签。解压下载的数据集文件，目录列表如图 3.7 所示。每一种美食对应一个 Bounding Box 标签。

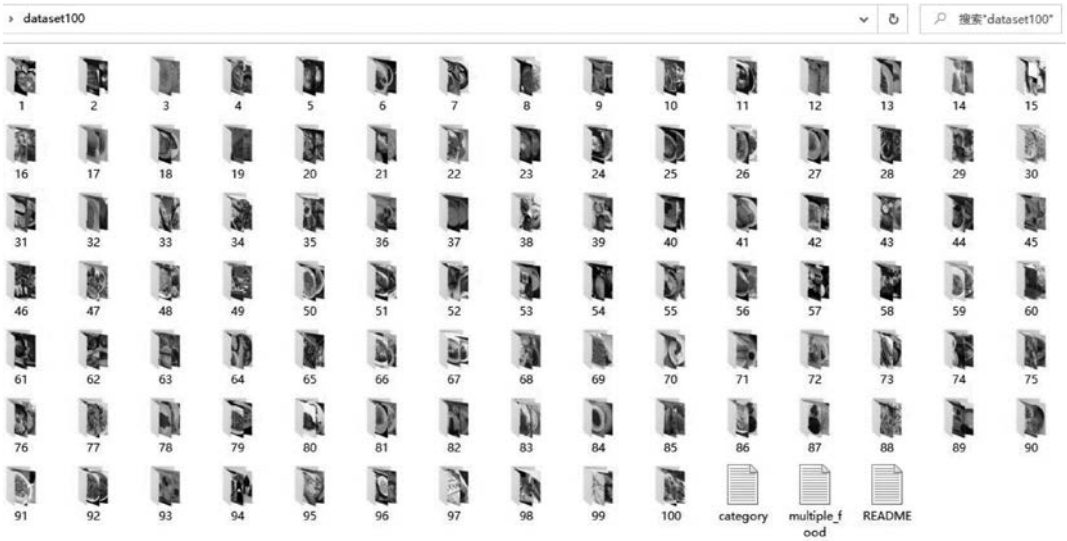


图 3.7　UEC FOOD 100 数据集目录列表

以目录100为例,其包含的部分图片样本如图3.8所示。每一个目录均有一个名称为bb_info.txt的文件,存储该目录下所有图片的位置标签。文件bb_info.txt包含5列数据,依次是图像的ID(即文件名称),矩形框的左上角和右下角坐标x1、y1、x2、y2。

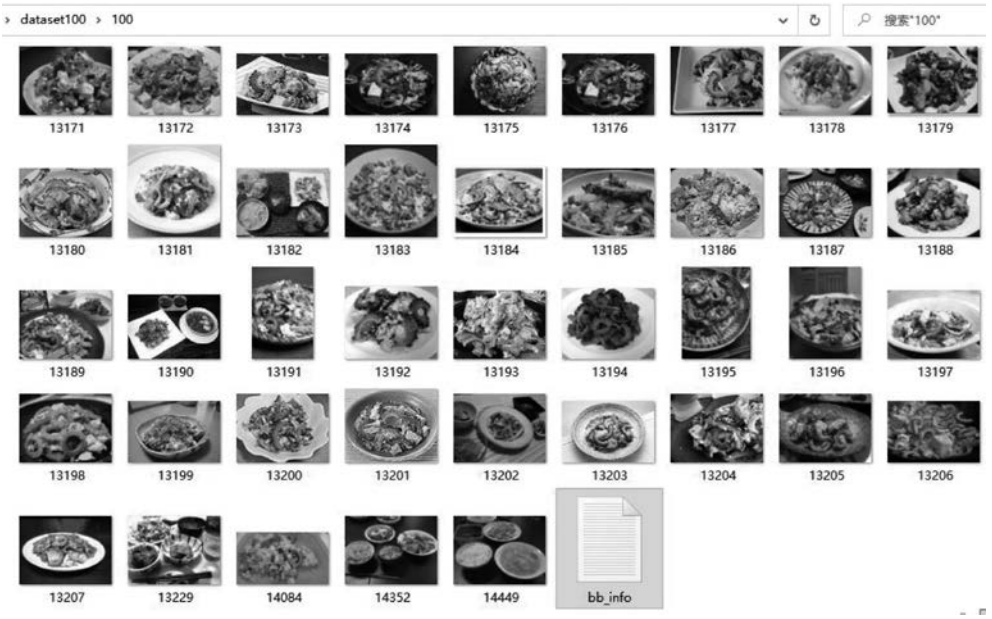


图3.8 目录100包含的部分图片样本

数据集目录结构及功能描述如表3.1所示。

表3.1 数据集目录结构与功能描述

目录或文件名称	功能描述	样本规模
目录1～100	以数字1～100命名的100个目录,每个目录下存放同一种类型的美食图片,图片文件采用数字命名	总样本数量为14 611,类别总数为100
bb_info.txt	存放于每一个目录下,记录该目录下每一幅图片的Bounding Box标签	100个目录,共100个bb_info.txt文件
category.txt	类别标签文件,100种类别对应的数字与英文名称	100种类别的名称与索引
multiple_food.txt	包含多个分类目标的图片id及其标签	共1174幅图片

为了便于后续建模工作,上述数据集需要做进一步的预处理。用PyCharm打开本教材的项目TensorFlow_to_Android,在根目录下创建子目录EfficientDet,将图3.7所示的数据集目录dataset100移动到EfficientDet目录下。

在EfficientDet目录下新建程序dataset.py,完成数据集的划分与标签预处理工作,编码逻辑如程序源码P3.1所示。

程序源码P3.1 dataset.py对数据集做预处理,划分训练集、验证集和测试集

```
1 import numpy as np
2 import pandas as pd
```

```
from sklearn.utils import shuffle
from PIL import Image
all_foods = []                                          # 存放所有样本标签
# 读取所有类别名称
category = pd.read_table('./dataset100/category.txt')
# 列表中列的顺序
column_order = ['type', 'img', 'label', 'x1', 'y1', 'x2', 'y2']
# 遍历目录 1～100,读取所有图片的标签信息,汇集到 all_foods 列表
for i in range(1,101,1):
    # 读取当前目录 i 的标签信息
    foods = pd.read_table(f'./dataset100/{i}/bb_info.txt',
                          header = 0,
                          sep = '\s + ')
    # 将图像 ID 映射为对应的文件路径
    foods['img'] = foods['img'].apply(lambda x: f'./dataset100/{i}/' + str(x) + '.jpg')
    # 新增一列 label,标注图片类别名称
    foods['label'] = foods.apply(lambda x: category['name'][i - 1], axis = 1)
    foods['type'] = foods.apply(lambda x: '', axis = 1)
    foods = foods[column_order]
    # 保存当前类别的标签文件
    foods.to_csv(f'./dataset100/{i}/label.csv',
                 index = None,
                 header = ['type', 'img', 'label', 'x1', 'y1', 'x2', 'y2'])
    # 汇聚到列表 all_foods
    all_foods.extend(np.array(foods).tolist())
# 保存列表到文件中
df_foods = pd.DataFrame(all_foods)
df_foods.to_csv('./dataset100/all_foods.csv',
                index = None,
                header = ['type', 'img', 'label', 'x1', 'y1', 'x2', 'y2'])
# 随机洗牌,打乱数据集排列顺序,划分为 TRAIN、VALIDATE、TEST 三部分
datasets = pd.read_csv('./dataset100/all_foods.csv')    # 读数据
datasets = shuffle(datasets,random_state = 2022)        # 洗牌
datasets = pd.DataFrame(datasets).reset_index(drop = True)
rows = datasets.shape[0]                                # 总行数
test_n = rows //40                                      # 测试集样本数
validate_n = rows //5                                   # 验证集样本数
train_n = rows - test_n - validate_n                    # 训练集样本数
print(f'测试集样本数:{test_n},验证集样本数:{validate_n},训练集样本数:{train_n}')
# 按照一定比例对数据集进行划分
for row in range(test_n):                               # 标注测试集
    datasets.iloc[row, 0] = 'TEST'
for row in range(validate_n):                           # 标注验证集
    datasets.iloc[row + test_n, 0] = 'VALIDATE'
for row in range(train_n):                              # 标注训练集
    datasets.iloc[row + test_n + validate_n, 0] = 'TRAIN'
# 将 Bounding Box 的坐标改为浮点类型,取值范围为[0,1]
print('开始对 BBox 坐标做归一化调整,请耐心等待...')
for row in range(rows):
```

```
    img = Image.open(datasets.iloc[row, 1])          # 读取图像
    (width, height) = img.size                       # 图像宽度与高度
    width = float(width)
    height = float(height)
    datasets.iloc[row, 3] = round(datasets.iloc[row, 3] / width, 3)
    datasets.iloc[row, 4] = round(datasets.iloc[row, 4] / height, 3)
    datasets.iloc[row, 5] = round(datasets.iloc[row, 5] / width, 3 )
    datasets.iloc[row, 6] = round(datasets.iloc[row, 6] / height, 3)
    datasets.insert(datasets.shape[1], 'Null1', '')   # 插入空列
    datasets.insert(datasets.shape[1], 'Null2', '')   # 插入空列
# 调整列的顺序,为以后数据集划分做准备
order = ['type', 'img', 'label', 'x1', 'y1', 'Null1', 'Null2', 'x2', 'y2']
datasets = datasets[order]
print(datasets.head())
datasets.to_csv('./dataset100/datasets.csv', index = None, header = None)
print('数据集构建完毕!')
```

运行程序 dataset.py,查看 dataset100 目录下新生成的数据集文件 datasets.csv,观察测试集样本数、验证集样本数和训练集样本数,可以根据实验环境的计算能力,适当调整数据集规模与比例划分。

程序源码 P3.1 的划分结果:测试集样本数为 365,验证集样本数为 2922,训练集样本数为 11 324。

3.5 EfficientDet 解析

EfficientDet 模型参见论文 *Efficientdet: Scalable and efficient object detection* (TAN M,PANG R,LE Q V.2020),它是谷歌研究团队借鉴 EfficientNet 分类模型的体系架构,在目标检测领域取得的创新性进展。

EfficientDet 的主要创新点有两个:一是采用双向加权特征金字塔网络(a weighted bidirectional feature pyramid network, BiFPN),实现多尺度特征提取与融合;二是采用复合缩放法,同时对所有主干网络、特征网络、目标定位网络和分类网络的分辨率、深度、宽度统一缩放。基于上述创新点,得到了 EfficientDet 模型系列,即 EfficientDet-D0～EfficientDet-D7。

正如作者所强调的那样,EfficientDet 模型的研发动力来自机器人、自动驾驶等对视觉模型精度和响应速度的严苛要求。机器视觉领域往往关注了速度,就会牺牲精度;或者关注了精度,又会拖累速度。

EfficientDet 的目标是在可伸缩架构和高精度之间取得平衡,开发更为高效的并适应多场景需求的目标检测网络,最终形成独特的网络设计,由主干网络、特征融合网络、定位网络和分类网络构成的 EfficientDet 模型如图 3.9 所示。

EfficientDet 模型从输入层开始,依次经历了主干网络、特征融合网络和分类/定位网络三个阶段,是一个端到端的网络结构。主干网络采用 EfficientNet 网络,特征融合网络采用 BiFPN 网络,分类和定位网络采用卷积网络。

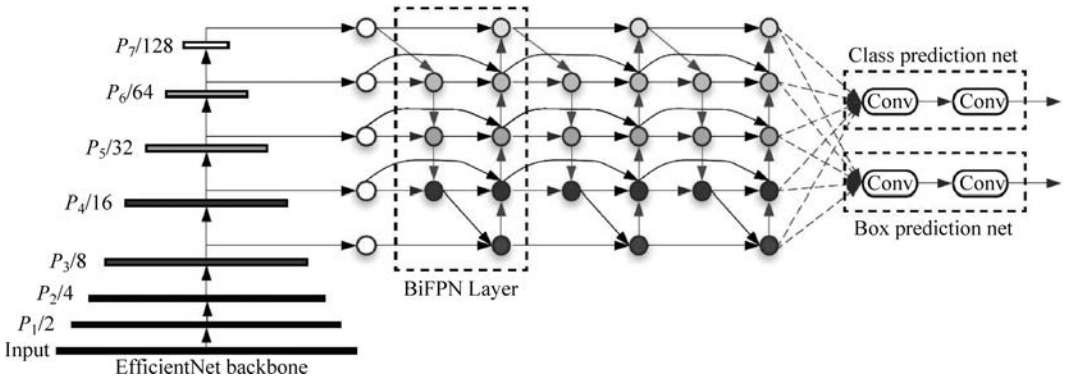


图 3.9　EfficientDet 模型

来自目标检测场景的一个非常现实的问题是，有的目标看起来很大，有的目标看起来很小，同一种类型的目标由于观察距离或者视角的问题也会出现大小差异。如何处理这些大小不一的目标是一个挑战。有的模型可能对大目标识别度好，对小目标识别度差；或者关注了小目标，大目标的误差又会偏大。对此，一种常见的解决方案是采用多尺度特征融合。EfficientDet 在此基础上创新设计出了双向加权特征金字塔网络（BiFPN）并配合 EfficientNet 网络来更好地解决特征提取这个关键问题。

为了寻找最佳网络规模，研究发现，过往的模型只对主干网络和输入图像缩放，事实上对特征网络、定位网络和分类网络缩放也至关重要。对网络整体统一缩放，全局性更强。

EfficientNet 强大的特征提取能力和分类能力及其匹配多种需求的优势，在本书第 1 章已经有系统的描述，此处不再赘述。

下面重点介绍 EfficientDet 的 BiFPN 技术和模型的复合缩放技术。图 3.10 给出了四种特征融合网络设计模式。

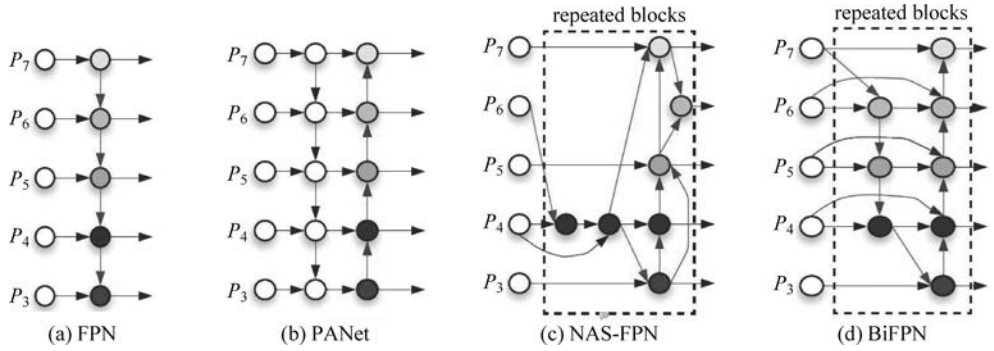


图 3.10　特征融合网络设计

图 3.10(a)展示 FPN，FPN 对来自 $P_3 \sim P_7$ 的多尺度特征进行自顶向下的多尺度特征融合。图 3.10(b)展示 PANet，在 FPN 基础上叠加了自底向上的特征融合路径，即将 FPN 的单向特征融合变为双向特征融合。图 3.10(c)展示 NAS-FPN，它是基于机器自动学习模式搜索一个网络结构用于特征融合。图 3.10(d)展示 BiFPN，它是一种高效的双

向跨尺度交叉连接和加权特征融合网络。

多尺度特征融合的前提是多尺度特征提取,图 3.9 展示了 EfficientDet 多尺度特征提取过程。主干网络采用 EfficientNet,读者可以回看 EfficientNetV2 的模型结构(见表 1.9),包含八层模块,去掉最后的输出层,EfficientNetV2 的第 1～7 层是特征提取层,图 3.9 中的主干网给出的 P_1～P_7,代表 EfficientNetV2 的七个模块层。但是在输入到特征融合网络时,只采用了其中的 P_3～P_7 这五个模块层进行特征融合。

以 FPN 为例,其特征融合逻辑可以表示为式(3.1)。

$$\begin{aligned} P_7^{\text{out}} &= \text{Conv}(P_7^{\text{in}}) \\ P_6^{\text{out}} &= \text{Conv}(P_6^{\text{in}} + \text{Resize}(P_7^{\text{out}})) \\ P_5^{\text{out}} &= \text{Conv}(P_5^{\text{in}} + \text{Resize}(P_6^{\text{out}})) \\ P_4^{\text{out}} &= \text{Conv}(P_4^{\text{in}} + \text{Resize}(P_5^{\text{out}})) \\ P_3^{\text{out}} &= \text{Conv}(P_3^{\text{in}} + \text{Resize}(P_4^{\text{out}})) \end{aligned} \tag{3.1}$$

再看 BiFPN 的特征融合逻辑。对于 FPN、PANet 而言,跨尺度特征之间叠加时,没有权重分配的问题,认为不同尺度的特征同等重要。而对于 BiFPN,则采取了加权叠加方式,即认为不同尺度的特征,在特征融合时所占权重不同。

EfficientDet 论文中给出了三种加权方法,分别如下。

(1) 无边界融合(Unbounded Fusion)。计算方法如式(3.2)所示。

$$O = \sum_i w_i \cdot I_i \tag{3.2}$$

其中,w_i 表示对每一个输入 I_i 施加一个可学习的权重参数 w_i,区分不同尺度特征 I_i 的重要性后再叠加在一起,得到输出 O。实验表明,这个加权方式缺乏模型稳定性,因为权重 w_i 的取值自由度过大。

(2) 基于 Softmax 函数的融合(Softmax-based Fusion)。计算逻辑如式(3.3)所示。

$$O = \sum_i \frac{e^{w_i}}{\sum_j e^{w_j}} \cdot I_i \tag{3.3}$$

将权重 w_i 用 Softmax 函数变换一下,约束到 0～1 这个区间内。基于 Softmax 的特征融合,将权重的取值限制为 0～1,表达出了不同尺度特征的重要性,稳定性比无边界融合方法要好。但是实验表明,该方法明显拖累了 GPU 的运行速度。

(3) 快速归一化融合(Fast Normalized Fusion)。计算逻辑如式(3.4)所示。

$$O = \sum_i \frac{w_i}{\varepsilon + \sum_j w_j} \cdot I_i \tag{3.4}$$

显然,式(3.4)简化了式(3.3)的计算。实验表明,式(3.4)与式(3.3)在取得相似精度的前提下,GPU 的计算速度提升了 30%。

以 BiFPN 中的 P_6 层的特征融合为例,其计算逻辑如式(3.5)所示。

$$P_6^{\text{td}} = \text{Conv}\left(\frac{w_1 \cdot P_6^{\text{in}} + w_2 \cdot \text{Resize}(P_7^{\text{in}})}{w_1 + w_2 + \varepsilon}\right)$$

$$P_6^{\text{out}}=\text{Conv}\left(\frac{w'_1\cdot P_6^{\text{in}}+w'_2\cdot P_6^{\text{td}}+w'_3\cdot \text{Resize}(P_5^{\text{out}})}{w'_1+w'_2+w'_3+\varepsilon}\right) \tag{3.5}$$

其中，P_6^{td} 是 P_6 层的中间计算结果，P_6^{out} 是 P_6 层对应的最终输出结果。

观察图 3.9 的网络模型，不难看出 BiFPN 模块层往往需要重复多次，这就涉及最佳重复次数问题。

现在讨论模型的整体缩放。EfficientDet 首先确定了一个基准模型。

对于主干网络，采用 EfficientNet-B0～EfficientNet-B6 作为基准参照。

对于 BiFPN 网络，其宽度与深度的缩放采用式(3.6)计算。

$$W_{\text{bifpn}}=64\times(1.35^{\phi}),\quad D_{\text{bifpn}}=3+\phi \tag{3.6}$$

其中，W_{bifpn} 表示网络宽度(通道数)，D_{bifpn} 表示网络深度(层数)。参数 1.35 是通过对参数列表{1.2，1.25，1.3，1.35，1.4，1.45}做网格搜索得到的最佳值。ϕ 是缩放因子。

对于定位网络和分类网络，采用的缩放方法如式(3.7)所示。

$$D_{\text{box}}=D_{\text{class}}=3+\lfloor\phi/3\rfloor \tag{3.7}$$

输入图像分辨率的缩放如式(3.8)所示。

$$R_{\text{input}}=512+128\phi \tag{3.8}$$

根据式(3.6)～式(3.8)，用一个系数 ϕ 可以完成对整个 EfficientDet 网络的缩放，例如 $\phi=0$ 得到模型 EfficientDet-D0，$\phi=7$ 得到模型 EfficientDet-D7，如表 3.2 所示。

表 3.2 EfficientDet 系列模型参数

模型	输入 R_{input}	主干网络	BiFPN 网络		定位网络和分类网络 D_{class}
			W_{bifpn}	D_{bifpn}	
D0($\phi=0$)	512	B0	64	3	3
D1($\phi=1$)	640	B1	88	4	3
D2($\phi=2$)	768	B2	112	5	3
D3($\phi=3$)	896	B3	160	6	4
D4($\phi=4$)	1024	B4	224	7	4
D5($\phi=5$)	1280	B5	288	7	4
D6($\phi=6$)	1280	B6	384	8	5
D7($\phi=7$)	1536	B6	384	8	5
D7x	1536	B7	384	8	5

关于模型更多解析，参见本节视频教程。

3.6 EfficientDet-Lite 预训练模型

第 2 章的鸟类识别案例采用的是一种传统的 TFLite 建模方法，模型训练与部署路径：MobileNetV3 建模→模型训练→模型评估→用 TFLiteConverter 将模型转换为 TFLite 版→添加 TFLite 元数据→将 TFLite 版模型部署应用到 Android 上。

本章案例尝试一种更为简单的方案，直接基于已经训练好的 EfficientDet-Lite 版模

型做迁移学习，完成美食场景检测模型的训练与评估。技术路径：EfficientDet-Lite 版预训练模型→用 TensorFlow Lite Model Maker 完成 TFLite 模型的训练与评估→得到迁移学习后的 TFLite 新模型→将 TF Lite 版模型部署应用到 Android 上。两种技术路径的对比关系如图 3.11 所示。

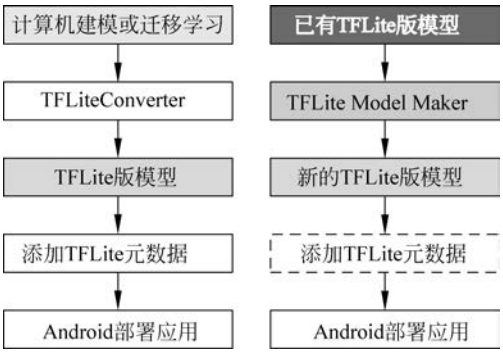


图 3.11　TFLite 版模型建模路径

显然，两种建模路径的起点不同，基于 TensorFlow Lite Model Maker 库的建模路径，要求已经拥有预训练好的 TFLite 模型，而且 TFLite 模型训练完成后，不需要单独添加模型元数据信息，因为此前的 EfficientDet-Lite 预训练模型已经包含相关元数据的结构信息。

本章案例采用的 EfficientDet-Lite 预训练模型全部来自 TensorFlow Hub，模型基于 COCO 2017 数据集训练，其性能表现如表 3.3 所示。

表 3.3　EfficientDet-Lite 模型性能表现

Model Architecture	Size/MB	Latency/ms	Average Precision/%
EfficientDet-Lite0	4.4	37	25.69
EfficientDet-Lite1	5.8	49	30.55
EfficientDet-Lite2	7.2	69	33.97
EfficientDet-Lite3	11.4	116	37.70
EfficientDet-Lite4	19.9	260	41.96

表 3.3 中数据来自 TensorFlow Hub 网站，其中：

Size/MB：表示采用整数量化后的模型大小。

Latency/ms：表示模型在 4 核 CPU 的 Pixel 4 手机上的单幅图像的时间延迟。

Average Precision/%：表示模型在 COCO 2017 验证集上的平均精度(mean Average Precision，mAP)。

以 EfficientDet-Lite2 模型为例，模型对输入图像的尺寸要求是：Height×Width×3，Height=448，Width=448，像素取值范围为[0，255]。

模型输出的内容如下。

(1) num_detections：一次最多可检测的目标对象数量，最大值为 25。

(2) detection-boxes：定位目标的矩形框坐标。

（3）detection-classes：目标分类。

（4）detection-scores：目标置信度。

为了便于读者学习，表 3.3 中的 5 个 EfficientDet-Lite 版预训练模型已经放到了本章项目文件夹 EfficientDet\pretraining 中。读者也可以自行到 TensorFlow Hub 官方网站下载。

3.7　美食版 EfficientDet-Lite 训练

由于案例中使用了 TensorFlow Lite Model Maker 库和 COCO 2017 数据集的标签，因此需要在当前项目环境安装必需的软件包。用 PyCharm 打开当前项目，转到 Terminal 窗口，执行下述两条命令。

```
pip install tflite-model-maker
pip install pycocotools
```

在当前项目 EfficientDet 根目录下创建程序 model.py。基于迁移学习的模型训练逻辑如程序源码 P3.2 所示。

程序源码 P3.2　model.py 美食版 EfficientDet-Lite 迁移学习训练

```
import json
from absl import logging
from tflite_model_maker import model_spec
from tflite_model_maker import object_detector
import tensorflow as tf
assert tf.__version__.startswith('2')
tf.get_logger().setLevel('ERROR')
logging.set_verbosity(logging.ERROR)
spec = model_spec.get('efficientdet_lite0')                    # 指定模型
print('数据集划分需要读取 14611 幅图像,可能花费几分钟时间。请耐心等待!')
train_data, validation_data, test_data = \
            object_detector.DataLoader.from_csv('./dataset100/datasets.csv')
print('开始模型训练...')
# 训练模型,指定训练参数
model = object_detector.create(train_data,
                               model_spec=spec,
                               epochs=30,
                               batch_size=16,
                               train_whole_model=True,
                               validation_data=validation_data)
# 将训练好的模型导出为 TFLite 模型并保存到当前工作目录下.默认采用整数量化方法
print('正在采用默认优化方法,保存 TFLite 模型...')
model.export(export_dir='.')                                   # 保存 TFLite 模型
model.summary()
# 保存与模型输出一致的标签列表
classes = ['???'] * model.model_spec.config.num_classes
label_map = model.model_spec.config.label_map
```

```
for label_id, label_name in label_map.as_dict().items():
    classes[label_id-1] = label_name
print(classes)
with open('labels.txt', 'w') as f:                          # 模型标签保存到文件
    for i in range(len(classes)):
        for label in classes:
            f.write(label + "\r")
# 在测试集上评测训练好的模型
dict1 = {}
print('开始在测试集上对计算机版模型评估...')
dict1 = model.evaluate(test_data, batch_size = 16)
print(f'计算机版模型在测试集上评估结果:\n {dict1}')
# 加载 TFLite 格式的模型,在测试集上做评估
dict2 = {}
print('开始在测试集上对优化后的 TFLite 模型评估...')
dict2 = model.evaluate_tflite('model.tflite', test_data)
print(f'优化后的 TFLite 模型在测试集上评估结果: \n {dict2}')
# 保存模型的评估结果
for key in dict1:
    dict1[key] = str(dict1[key])
    print(f'{key}: {dict1[key]}')
with open('dict1.txt','w') as f :
    f.write(json.dumps(dict1))
# 保存优化后的 TFLite 模型在测试集上的评估结果
print('真实版的 TFLite 模型测试结果...')
for key in dict2:
    dict2[key] = str(dict2[key])
    print(f'{key}: {dict2[key]}')
with open('dict2.txt','w') as f :
    f.write(json.dumps(dict2))
```

运行程序 model.py,数据集加载完成后,模型开始训练。注意,第 17 行语句和第 18 行语句指定的 epochs 参数和 batch_size 参数,可以根据配置的计算能力进行修改。如果内存低于 32GB,建议将 batch_size 设置为 8。

本章项目训练采用的主机配置如下。

(1) CPU: Intel Core i7,8 核。

(2) RAM: 32GB。

(3) GPU: NVIDIA GeForce RTX 3070,8GB。

训练 30 代,大约需要 3 小时。读者可根据个人主机配置情况,调整模型训练参数。

训练过程演示及测试指标讲解参见本节视频教程。

3.8 评估指标 mAP

目标检测领域通常采用 mAP 作为模型评价的主要指标,例如 Faster R-CNN、SSD、EfficientDet、YOLO 等算法均采用 mAP 指标作为模型的评价标准。目标检测领域有两

个经典数据集，分别是 Pascal VOC 和 MS COCO。mAP 在这两个数据集上的计算逻辑有所区别，所以，有时会特别指出 mAP 遵循的计算方法，例如 Pascal VOC 的 mAP 指标或者 MS COCO 的 mAP 指标。

在 COCO 数据集关于 mAP 的解释中，通常将 mAP 与 AP(Average Precision，平均精度)不做区分。AP 的含义是当召回率(Recall Rate)在[0，1]这个区间变化时，对应的精确率(Precision Rate)的平均值。

显然，AP 与精确率和召回率有关，那么什么是精确率和召回率呢？

以多分类问题中的类别 A 为例，精确率是预测结果正确的比例。精确率越高，意味着误报率越低，因此，当误报的成本较高时，精确率指标有助于判断模型的好坏。

召回率是正确预测的样本占该类样本总数的比例。召回率越高，意味着模型漏掉的目标越少，当漏掉的目标成本很高时，召回率指标有助于衡量模型的好坏。

$$\text{精确率：Precision}=\frac{\text{预测结果为 A 且正确的数量}}{\text{预测结果为 A 的数量}}$$

$$\text{召回率：Recall}=\frac{\text{预测结果为 A 且正确的数量}}{\text{类别 A 的总数量}}$$

要确定对某个目标对象的预测是否正确，通常采用 IoU 判断。IoU 被定义为预测 Bounding Box 和实际 Bounding Box 的交集除以它们的并集。如果 IoU＞阈值，则认为预测正确；如果 IoU≤阈值，则认为预测错误。

当 IoU＞0.5 的预测被认为是正确预测时，这意味着 IoU＝0.6 或者 IoU＝0.9 的两个预测具有相同的权重。因此，固定某个阈值会在评估指标中引入偏差。解决这个问题的一个思路是对一定范围内的 IoU 阈值，计算其 mAP。

以 COCO 数据集上定义的 mAP 为例。当只考虑 IoU 阈值为 0.5 的情况时，平均精度记作 AP50 或者 mAP50。同理，当只考虑 IoU 阈值为 0.75 时，平均精度可以记作 AP75 或者 mAP75。

单个类别的平均精度通常添加一个表示类别名称的后缀。例如，米饭和鸡肉米饭两种美食的平均精度可以分别表示如下。

```
AP_/rice: 0.42938477
AP_/chicken rice: 0.7119283
```

COCO 数据集上，mAP(或者 AP)将 IoU 的阈值以 0.05 为步长，覆盖了[0.5：0.95]的 10 个数值，其计算逻辑如式(3.9)所示。

$$\text{mAP}_{\text{COCO}}=\frac{\text{mAP}_{0.50}+\text{mAP}_{0.55}+\cdots+\text{mAP}_{0.95}}{10} \tag{3.9}$$

事实上，mAP 的计算逻辑包含三次平均计算过程。还是以 COCO 数据集采用的 mAP 为例。

步骤 1：对于每个类别(共 80 个类别)，计算不同的 IoU 阈值下的 AP，取它们的平均值，得到该类别的 AP。计算逻辑如式(3.10)所示。

$$\text{AP}[\text{class}]=\frac{1}{\#\text{thresholds}}\sum_{\text{IoU}\in\text{thresholds}}\text{AP}[\text{class},\text{IoU}] \tag{3.10}$$

步骤 2：通过对不同类别的 AP 进行平均来计算最终的 AP，如式(3.11)所示。

$$\mathrm{AP}=\frac{1}{\sharp\,\mathrm{classes}}\sum_{\mathrm{class}\in\,\mathrm{classes}}\mathrm{AP}[\mathrm{class}] \tag{3.11}$$

除了 AP 指标，COCO 数据集上还定义了其他一些指标，用于反映模型的性能，如表 3.4 所示。

表 3.4　COCO 数据集上反映模型性能的 12 个指标

指 标 名 称	功 能 描 述
AP：平均精度	
AP	最基本的评价指标，在 IoU=0.50:0.05:0.95 区间计算 AP
$AP^{IoU=0.50}$	固定 IoU 的阈值为 0.50
$AP^{IoU=0.75}$	固定 IoU 的阈值为 0.75
AP Across Scales：不同尺寸目标的平均精度	
AP^{small}	针对小目标(像素数量$<32^2$)的平均精度
AP^{medium}	针对中目标($32^2<$像素数量$<96^2$)的平均精度
AP^{large}	针对大目标(像素数量$>96^2$)的平均精度
Average Recall(AR)：平均召回率	
$AR^{max=1}$	每幅图像最多给出 1 个检测目标
$AR^{max=10}$	每幅图像最多给出 10 个检测目标
$AR^{max=100}$	每幅图像最多给出 100 个检测目标
AR Across Scales：不同尺寸目标的平均召回率	
AR^{small}	针对小目标(像素数量$<32^2$)的平均召回率
AR^{medium}	针对中目标($32^2<$像素数量$<96^2$)的平均召回率
AR^{large}	针对大目标(像素数量$>96^2$)的平均召回率

3.9　美食版 EfficientDet-Lite 评估

3.7 节模型训练程序 model.py 中已经给出了 12 个综合评价指标(见表 3.4)以及每个类别(共 100 个类别)的平均精度值。

为便于直观观察模型效果，程序 evaluation.py 完成了 EfficientDet-Lite 计算机版与移动版 TFLite 模型之间的对比。

选取了如下三个比较维度：

(1) 计算机版 EfficientDet-Lite 与移动版 EfficientDet-Lite 的 12 项指标对比。

(2) 按照各类别的 AP 排序，前 20 名 AP 对比。

(3) 按照各类别的 AP 排序，后 20 名 AP 对比。

在当前项目中新建程序 evaluation.py，编程逻辑如程序源码 P3.3 所示。

程序源码 P3.3　evaluation.py 美食版 EfficientDet-Lite 模型评估

```
1  import json
2  import pandas as pd
```

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
# 读取计算机版 TFLite 模型评估数据
dict1 = [json.loads(line) for line in open(r'dict1.txt','r')]
for key in dict1[0]:
    dict1[0][key] = float(dict1[0][key])
df1 = pd.DataFrame(dict1)
print(df1.head())
# 读取移动版 TFLite 模型评估数据
dict2 = [json.loads(line) for line in open(r'dict2.txt','r')]
for key in dict2[0]:
    dict2[0][key] = float(dict2[0][key])
df2 = pd.DataFrame(dict2)
print(df2.head())
# 取前 12 项指标
columns = ['AP', 'AP50', 'AP75', 'APs', 'APm', 'APl',
           'ARmax1', 'ARmax10', 'ARmax100', 'ARs','ARm','ARl']
df1_12 = df1.iloc[0, 0:12]
df2_12 = df2.iloc[0, 0:12]
sns.barplot(x = np.array(df1_12).tolist(), y = columns)       # 计算机版 TFLite
plt.show()
sns.barplot(x = np.array(df2_12).tolist(), y = columns)       # 移动版 TFLite
plt.show()
# 100 个类别 mAP 指标的条形图
df1.drop(columns = columns, inplace = True, axis = 1)
df1 = df1.stack()                                             # 行列互换
df1 = df1.unstack(0)
df1.sort_values(by = 0, axis = 0, ascending = False, inplace = True)
df2.drop(columns = columns, inplace = True, axis = 1)
df2 = df2.stack()                                             # 行列互换
df2 = df2.unstack(0)
df2.sort_values(by = 0, axis = 0, ascending = False, inplace = True)
# 根据需要,只显示 mAP 值最高的前 20 个类别
sns.barplot(x = df1[0][0:20], y = df1.index[0:20])            # 计算机版 TFLite
plt.show()
sns.barplot(x = df2[0][0:20], y = df2.index[0:20])            # 移动版 TFLite
plt.show()
# 只显示 mAP 值最低的 20 个类别
sns.barplot(x = df1[0][ - 20:], y = df1.index[ - 20:])        # 计算机版 TFLite
plt.show()
sns.barplot(x = df2[0][ - 20:], y = df2.index[ - 20:])        # 移动版 TFLite
plt.show()
```

执行程序源码 P3.3,观察计算机版 TFLite 模型与移动版 TFLite 模型的对比效果。图 3.12 给出了计算机版模型的 12 项指标条形图分布。各指标含义参见表 3.4。

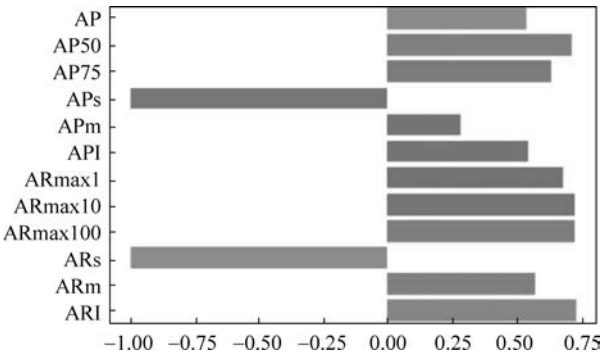


图 3.12　计算机版 TFLite 模型的 12 项指标条形图分布

图 3.13 给出了移动版 TFLite 模型的 12 项指标条形图分布。

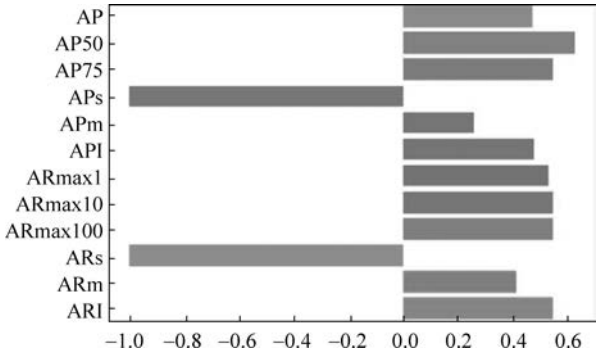


图 3.13　移动版 TFLite 模型的 12 项指标条形图分布

图 3.12 和图 3.13 中，APs(小目标平均精确率)和 ARs(小目标平均召回率)的值均为−1，表示测试集中不存在小目标图像。

不难看出，由于移动版 TFLite 模型做了量化优化，各项指标值均低于计算机版 TFLite 模型。追求计算速度的同时，损失精确率与召回率在所难免。但是移动版 TFLite 模型仍然整体上保持了较高的精确率和召回率，例如其 AP50 超过了 0.6。

图 3.14 给出了计算机版 TFLite 模型前 20 名类别的 AP 条形图分布，AP 值均超过 0.8。

图 3.15 给出了移动 TFLite 模型前 20 名类别的 AP 条形图分布，虽然依旧保持了较高的 AP 值，但是排名顺序发生变化，而且目标对象并不完全一致。

图 3.14 中有 4 种美食 AP_/Japanese-style pancake、AP_/pork cutlet on rice、AP_/sushi、AP_/sashimi bowl 不包括在图 3.15 中，取而代之的是另外 4 种美食 AP_/tempura、AP_/hamburger、AP_/spicy chili-flavored tofu、AP_/dipping noodles。而且即使对于同一种美食，其排名顺序也不一定相同。以 AP_/pizza 为例，在图 3.15 中排在第 20 名，而在图 3.14 中，却排到了第 3 名。

观察前 20 种美食的排名，计算机版 TFLite 模型与移动版 TFLite 模型差异显著。可见，量化优化以后，对模型精度的影响是显著的。

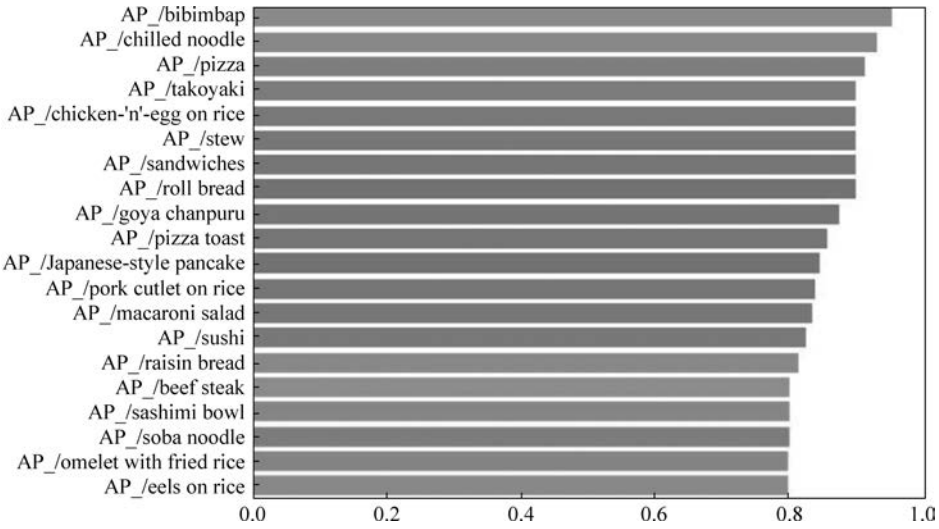


图 3.14　计算机版 TFLite 模型前 20 名类别的 AP 条形图分布

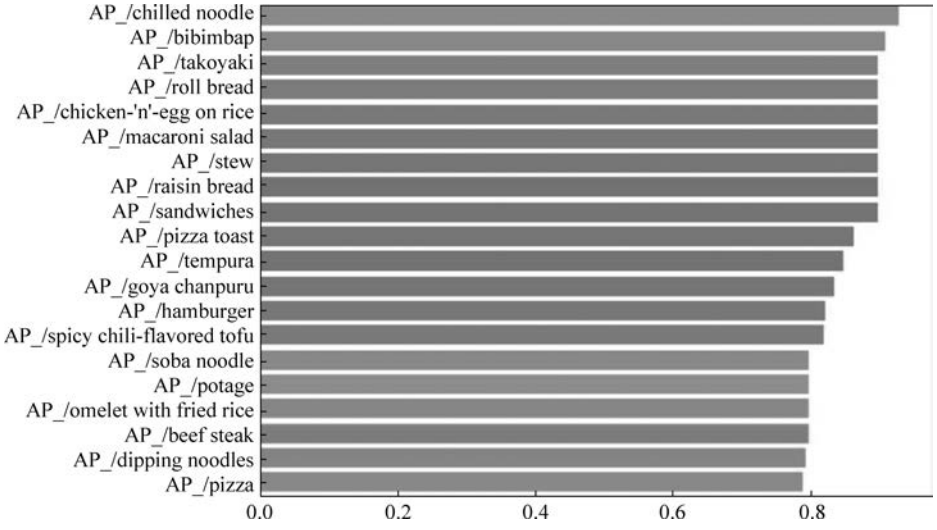


图 3.15　移动版 TFLite 模型前 20 名类别的 AP 条形图分布

图 3.16 给出了计算机版 TFLite 模型后 20 名类别的 AP 条形图分布。检查数据集文件 dataset.csv 发现，AP_/fried fish 在测试集中包含 3 个样本，AP_/vegetable tempura 在测试集中只包含 1 个样本，故其 AP 值非常小。测试集中不包含 AP_/sauteed vegetables 和 AP_/croissant，故其 AP 值为－1。

图 3.17 给出了移动版 TFLite 模型后 20 名类别的 AP 条形图分布。移动版 TFLite 的排名，除了类别列表上的差异，有更多的类别的 AP 值接近于 0，这说明这些类别在测试集中的样本数量过少，同时也说明，对比计算机版 TFLite，移动 TFLite 版在精度上的损失显著增加了。

事实上，程序源码 P3.1 随机划分数据集时，对于 100 个类别来讲，测试集只包含 365

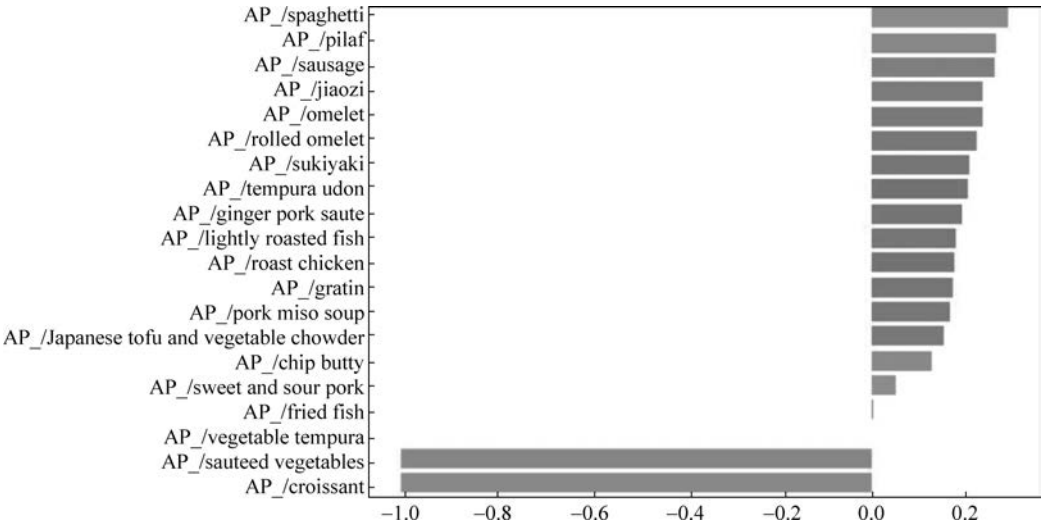


图 3.16　计算机版 TFLite 模型后 20 名类别的 AP 条形图分布

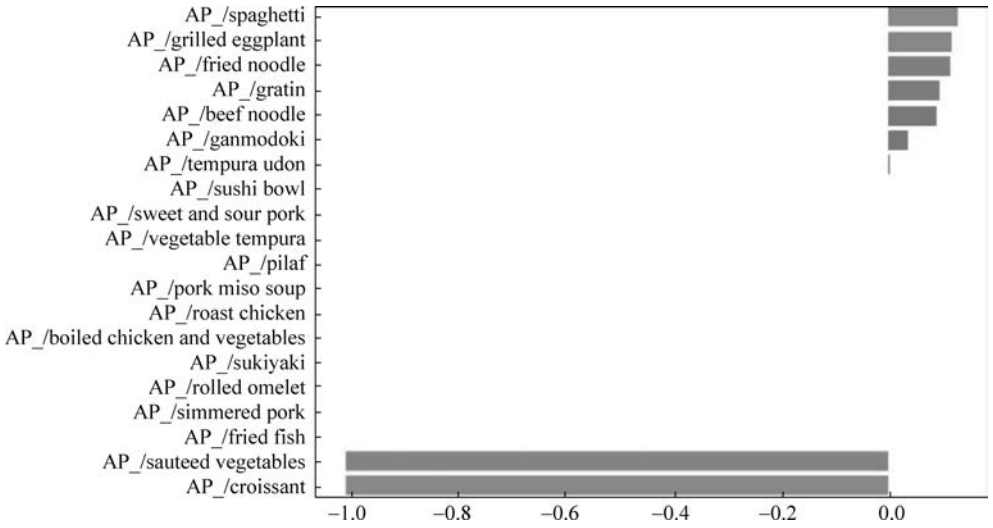


图 3.17　移动版 TFLite 模型后 20 名类别的 AP 条形图分布

个样本，平均每个类别 3.65 个样本，确实太少了。这也是为了增强教学演示效果、说明相关问题而刻意为之的一个举措。就本章案例而言，测试集和验证集的样本数各占 2000 左右，训练集为 10 000 左右比较合理。

3.10　美食版 EfficientDet-Lite 测试

在将移动版 TFLite 模型部署到 Android 手机上之前，首先在计算机里对模型做样本实证观察。在当前项目中新建程序 predict.py，编码逻辑如程序源码 P3.4 所示。

程序源码 P3.4 predict.py 美食版 EfficientDet-Lite 模型测试

```
import cv2
import tensorflow as tf
from PIL import Image
import numpy as np
model_path = 'model.tflite'                                    # 预训练模型
with open('labels.txt','r') as f:                              # 读取模型标签文件
    classes = f.readlines()
for i in range(len(classes)):                                  # 取出标签中的换行符
    classes[i] = classes[i].replace('\n','')
# 图像预处理
def preprocess_image(image_path, input_size):
    img = tf.io.read_file(image_path)                          # 读取指定图像
    img = tf.io.decode_image(img, channels = 3)                # 解码
    img = tf.image.convert_image_dtype(img, tf.uint8)          # 数据类型
    original_image = img                                       # 原始图像
    resized_img = tf.image.resize(img, input_size)             # 图像缩放
    resized_img = resized_img[tf.newaxis, :]                   # 增加维度,表示样本数量
    resized_img = tf.cast(resized_img, dtype = tf.uint8)       # 数据类型
    return resized_img, original_image                         # 裁剪后的图像与原始图像
def detect_objects(interpreter, image, threshold):
    """
    用指定的模型和置信度阈值,对指定的图像检测
    :param interpreter: 推理模型
    :param image: 待检测图像
    :param threshold: 置信度阈值
    :return: 返回检测结果(字典列表)
    """
    # 推理模型解释器
    signature_fn = interpreter.get_signature_runner()
    # 对指定图像做目标检测
    output = signature_fn(images = image)
    # 解析检测结果
    count = int(np.squeeze(output['output_0']))                # 检测到的目标数量
    scores = np.squeeze(output['output_1'])                    # 置信度
    class_curr = np.squeeze(output['output_2'])                # 类别
    boxes = np.squeeze(output['output_3'])                     # Bounding Box 坐标
    results = []
    for i in range(count):                                     # 所有目标组织为列表
        if scores[i] >= threshold:                             # 只返回超过阈值的目标
            result = {                  # 以字典格式组织单个检测结果
              'bounding_box': boxes[i],
              'class_id': class_curr[i],
              'score': scores[i]
            }
            results.append(result)
    return results                                             # 返回检测结果(字典列表)
def run_odt_and_draw_results(image_path, interpreter, threshold = 0.5):
```

```
    """
    用指定模型在指定图片上根据阈值做目标检测并绘制检测结果
    :param image_path: 待检测图像
    :param interpreter: 推理模型
    :param threshold: 置信度阈值
    :return: 绘制 Bounding Box、类别和置信度的图像数组
    """
    # 根据模型获得输入维度
    _, input_height, input_width, _ = interpreter.get_input_details()[0]['shape']
    # 加载图像并做预处理
    preprocessed_image, original_image = preprocess_image(
        image_path,
        (input_height, input_width)
      )
    # 对图像做目标检测
    results = detect_objects(interpreter,
                             preprocessed_image,
                             threshold = threshold)
    # 在图像上绘制检测结果(Bounding Box,类别,置信度)
    original_image_np = original_image.numpy().astype(np.uint8)
    for obj in results:
        # 根据原始图像尺寸(高度和宽度),将 Bounding Box 的坐标调整为整数
        ymin, xmin, ymax, xmax = obj['bounding_box']
        xmin = int(xmin * original_image_np.shape[1])
        xmax = int(xmax * original_image_np.shape[1])
        ymin = int(ymin * original_image_np.shape[0])
        ymax = int(ymax * original_image_np.shape[0])
        # 当前类别的 ID
        class_id = int(obj['class_id'])
        # 用指定颜色绘制 Bounding Box
        color = [0,255,0]                                   # 颜色
        cv2.rectangle(original_image_np,
                      (xmin, ymin),
                      (xmax, ymax),
                      color, 1)
        # 调整类别标签的纵向坐标,保持可见
        y = ymin - 5 if ymin - 5 > 15 else ymin + 20
        # 类别标签和置信度显示为字符串
        label = "{}: {:.0f}% ".format(classes[class_id], obj['score'] * 100)
        color = [255,255,0]                                 # 标签文本颜色
        cv2.putText(original_image_np, label, (xmin + 5, y),
                    cv2.FONT_ITALIC, 0.5, color, 1)    # 绘制标签
    # 返回绘制结果的图像
    original_uint8 = original_image_np.astype(np.uint8)
    return original_uint8
# 随机选择图像进行测试
# TEMP_FILE = './dataset100/25.jpg'
TEMP_FILE = './dataset100/11156.jpg'
DETECTION_THRESHOLD = 0.13                                  # 置信度阈值,可以调整
```

```
97  im = Image.open(TEMP_FILE)                                    # 打开图像
98  im.thumbnail((512, 512), Image.ANTIALIAS)                     # 缩放
99  im.save(TEMP_FILE)                                            # 保存缩放后的图像
100 # 加载 TFLite 推理模型
101 interpreter = tf.lite.Interpreter(model_path = model_path)
102 interpreter.allocate_tensors()
103 # 进行目标检测并绘制检测结果
104 detection_result_image = run_odt_and_draw_results(
105     TEMP_FILE,
106     interpreter,
107     threshold = DETECTION_THRESHOLD
108 )
109 # 显示检测结果
110 Image.fromarray(detection_result_image).show()
```

运行程序 predict.py，修改第 96 行语句设定的置信度阈值，观察输出结果。图 3.18 所示为图片 25.jpg 在置信度阈值为 0.13 时的测试结果。

图 3.19 所示为图片 11156.jpg 在置信度阈值为 0.13 时的测试结果。注意，其中的 french fries 检测到了两个目标框，因为其置信度阈值均超过了 0.13。

图 3.18 图片 25.jpg 在置信度阈值为 0.13 时的测试结果(5 种美食全部检出)(见彩插)

图 3.19 图片 11156.jpg 在置信度阈值为 0.13 时的测试结果(检测到 4 种美食)

3.11 新建 Android 项目

新建 Android 项目，项目模板选择 Empty Activity，项目名称为 Foods，项目包可自由定义，本章设置为 cn.edu.ldu.foods，编程语言选择 Kotlin，SDK 最小版本号设置为 API 21：Android 5.0(Lollipop)，如图 3.20 所示，单击 Finish 按钮，完成项目创建和初始化。

打开项目资源列表中的 strings.xml 文件，修改 app_name 属性的值为“美食场景检测”：

```
<string name = "app_name">美食场景检测</string>
```

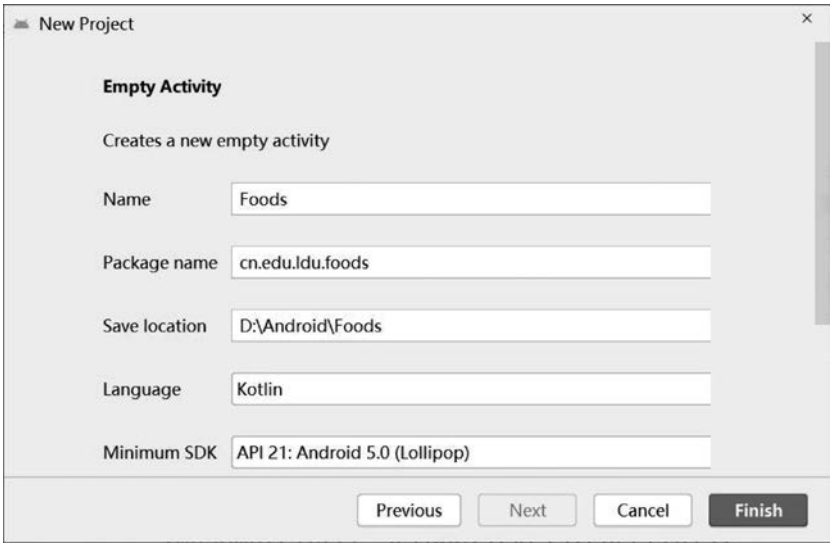


图 3.20　项目初始化与参数配置

右击项目视图中的 app 节点，在弹出的快捷菜单中执行 New→Image Asset 命令，在弹出的对话框中选择一幅素材图片作为程序图标，调整图标大小，完成图标定制，如图 3.21 所示。

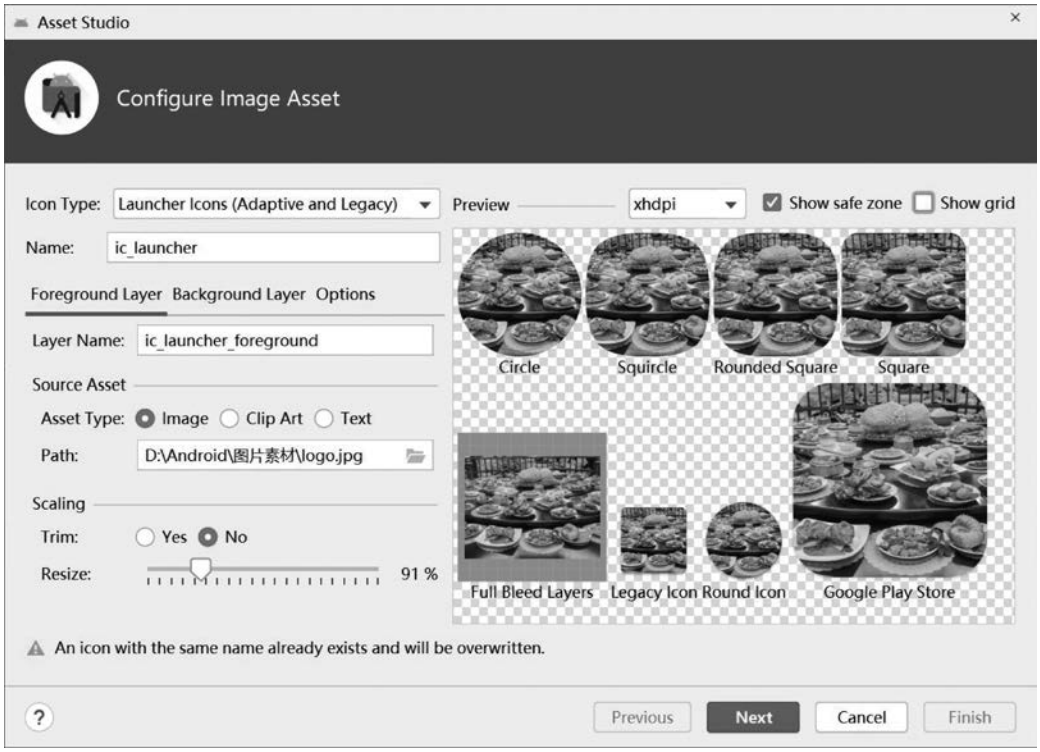


图 3.21　定制项目图标

选择 app 节点，在鼠标右键的快捷菜单中执行 New→Folder→Assets Folder 命令，创建 assets 目录，复制 model.tflite 模型文件到项目的 assets 目录下。

执行 Refactor→Migrate to AndroidX 命令，将项目支持库转变为 AndroidX 模式。

在项目的 build.gradle 文件中，添加 TFLite Task Library 库依赖：

```
implementation 'org.tensorflow:tensorflow-lite-task-vision:0.3.1'
```

在 AndroidManifest.xml 清单文件开启相机拍照功能。

```
<queries>
        <intent>
            <action android:name="android.media.action.IMAGE_CAPTURE" />
        </intent>
</queries>
```

在 AndroidManifest.xml 中添加 provider 元素，定义 FileProvider。

```
<manifest>
   ...
   <application>
       ...
       <provider
           android:name="androidx.core.content.FileProvider"
            android:authorities="cn.edu.ldu.objectdetection.fileprovider"
            android:exported="false"
            android:grantUriPermissions="true">
            <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/file_paths" />
     </provider>
   </application>
</manifest>
```

在 res/xml 节点下创建 file_paths.xml 文件，定义外部存储路径。

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <external-files-path name="my_images" path="Pictures" />
</paths>
```

此时，项目结构如图 3.22 所示。其中清单文件 AndroidManifest.xml、模块依赖文件 build.gradle、照片存储路径文件 file_paths.xml 和 TFLite 模型文件 model.tflite 已经部署完成。

接下来的工作是完成界面设计和主程序逻辑设计。

图 3.22　项目结构

3.12　Android 界面设计

打开 activity_main.xml 文件，定义界面布局，如图 3.23 所示。自顶向下包含四个控件，依次是文本提示控件 tvPlaceholder、图像视图控件 imageView、相机控件 btnCapture、相册控件 btnPicture。图 3.24 为模拟器测试的初始界面。

图 3.23　界面布局设计

图 3.24　模拟器测试的初始界面

界面脚本如程序源码 P3.5 所示。

程序源码 P3.5　activity_main.xml 界面布局设计

```
<?xml version = "1.0" encoding = "utf - 8"?>
< androidx.constraintlayout.widget.ConstraintLayout
xmlns:android = "http://schemas.android.com/apk/res/android"
    xmlns:app = "http://schemas.android.com/apk/res - auto"
    xmlns:tools = "http://schemas.android.com/tools"
    android:layout_width = "match_parent"
    android:layout_height = "match_parent"
    tools:context = ".MainActivity">
    < FrameLayout
        android:layout_width = "match_parent"
        android:layout_height = "match_parent"
        android:layout_above = "@ + id/btnCamera"
        app:layout_constraintStart_toStartOf = "parent"
        app:layout_constraintTop_toTopOf = "parent">
        < TextView
            android:id = "@ + id/tvPlaceholder"
            android:layout_width = "match_parent"
            android:layout_height = "wrap_content"
            android:layout_marginTop = "10dp"
            android:text = "此处显示检测结果"
            android:textAlignment = "center"
            android:textSize = "36sp" />
        < ImageView
            android:id = "@ + id/imageView"
            android:layout_width = "match_parent"
            android:layout_height = "464dp"
            android:adjustViewBounds = "true"
            android:contentDescription = "@null"
            android:scaleType = "fitCenter"
            app:srcCompat = "@mipmap/ic_launcher_foreground" />
    </FrameLayout >
    < Button
        android:id = "@ + id/btnCamera"
        android:layout_width = "100dp"
        android:layout_height = "80dp"
        android:layout_marginStart = "60dp"
        android:layout_marginBottom = "60dp"
        android:background = "@android:drawable/ic_menu_camera"
        app:layout_constraintBottom_toBottomOf = "parent"
        app:layout_constraintStart_toStartOf = "parent"
        tools:ignore = "SpeakableTextPresentCheck" />
    < Button
        android:id = "@ + id/btnPicture"
        android:layout_width = "90dp"
        android:layout_height = "70dp"
```

```
46          android:layout_marginEnd = "60dp"
47          android:layout_marginBottom = "65dp"
48          android:background = "@android:drawable/ic_menu_gallery"
49          app:layout_constraintBottom_toBottomOf = "parent"
50          app:layout_constraintEnd_toEndOf = "parent"
51          tools:ignore = "SpeakableTextPresentCheck" />
52  </androidx.constraintlayout.widget.ConstraintLayout>
```

在模拟器或者真机上做项目测试,此时,只能看到初始界面,单击“相机”按钮和“相册”按钮,系统没有响应。

3.13 Android 逻辑设计

本节完成主程序 MainActivity.kt 的编程设计,程序逻辑如图 3.25 所示,包括 10 个模块函数和一个实体类,矩形框内为模块函数名称或实体类名称,旁边给出了模块功能的简单描述。

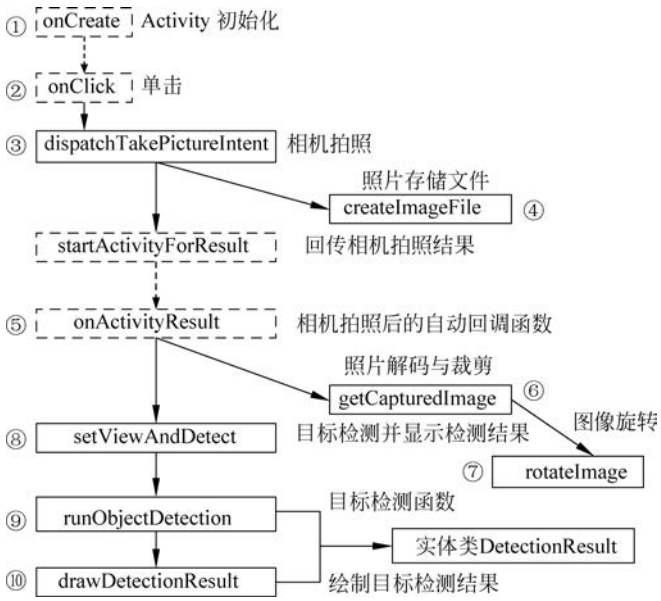


图 3.25 程序逻辑

模块之间的箭头连线表示了其调用关系。虚线箭头表示不是直接调用关系,但存在间接的逻辑关联或者事件关联。

虚线框表示该模块是系统函数模块,需要重写或者调用。实线框表示该模块是由用户新定义完成的模块。

编码逻辑如程序源码 P3.6 所示,其中模块名称和实体类名称加了粗体标注。

程序源码 P3.6　MainActivity.kt 程序主逻辑

```
package cn.edu.ldu.foods
import android.app.Activity
import android.content.ActivityNotFoundException
import android.content.Intent
import android.graphics.*
import android.net.Uri
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.Environment
import android.provider.MediaStore
import android.util.Log
import android.view.View
import android.widget.Button
import android.widget.ImageView
import android.widget.TextView
import androidx.core.content.FileProvider
import androidx.exifinterface.media.ExifInterface
import androidx.lifecycle.lifecycleScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import org.tensorflow.lite.support.image.TensorImage
import org.tensorflow.lite.task.vision.detector.Detection
import org.tensorflow.lite.task.vision.detector.ObjectDetector
import java.io.File
import java.io.IOException
import java.text.SimpleDateFormat
import java.util.*
import kotlin.math.max
import kotlin.math.min
class MainActivity : AppCompatActivity(), View.OnClickListener {
    companion object {                              // 定义常量
        const val TAG = "TFLite Object Detection"
        const val REQUEST_IMAGE_CAPTURE: Int = 2022
        private const val MAX_FONT_SIZE = 96F
    }
    private lateinit var btnCamera: Button
    private lateinit var inputImageView: ImageView
    private lateinit var tvPlaceholder: TextView
    private lateinit var currentPhotoPath: String
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        btnCamera = findViewById(R.id.btnCamera)
        inputImageView = findViewById(R.id.imageView)
        tvPlaceholder = findViewById(R.id.tvPlaceholder)
        btnCamera.setOnClickListener(this)
    }
```

```
48      // 相机拍照返回后的回调函数
49      override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
50          super.onActivityResult(requestCode, resultCode, data)
51          if (requestCode == REQUEST_IMAGE_CAPTURE &&
52              resultCode == Activity.RESULT_OK
53          ) {
54              setViewAndDetect(getCapturedImage()) // 显示检测结果
55          }
56      }
57      // onClick(v: View?), 检测 Activity 上的单击事件
58      override fun onClick(v: View?) {
59          when (v?.id) {
60              R.id.btnCamera -> {
61                  try {
62                      dispatchTakePictureIntent()
63                  } catch (e: ActivityNotFoundException) {
64                      Log.e(TAG, e.message.toString())
65                  }
66              }
67          }
68      }
69      // 目标检测函数,完成对指定图像的目标检测
70      private fun runObjectDetection(bitmap: Bitmap) {
71          // Step 1: 创建 TFLite's TensorImage 对象
72          val image = TensorImage.fromBitmap(bitmap)
73          // Step 2: 初始化目标检测器对象
74          val options = ObjectDetector.ObjectDetectorOptions.builder()
75              .setMaxResults(5)
76              .setScoreThreshold(0.3f)            // 更改置信度阈值,会影响检测结果
77              .build()
78          val detector = ObjectDetector.createFromFileAndOptions(
79              this,
80              "model.tflite",                     // 此前训练好的 TFLite 模型文件
81              options
82          )
83          // Step 3: TensorImage 格式的图像传给检测器,开始检测
84          val results = detector.detect(image)
85          // Step 4: 分析检测结果并显示
86          val resultToDisplay = results.map {
87              // 获取排名第一的类别,构建显示文本
88              val category = it.categories.first()
89              val text = "${category.label}, ${category.score.times(100).toInt()}%"
90              // 创建数据对象,存储检测结果
91              DetectionResult(it.boundingBox, text)
92          }
93          // 在位图上绘制检测结果
94          val imgWithResult = drawDetectionResult(bitmap, resultToDisplay)
95          // 将检测结果更新到视图
96          runOnUiThread {
```

```
97                inputImageView.setImageBitmap(imgWithResult)
98            }
99        }
100       // 将图像显示到视图中,并对其做目标检测
101       private fun setViewAndDetect(bitmap: Bitmap) {
102           // 显示图像
103           inputImageView.setImageBitmap(bitmap)
104           tvPlaceholder.visibility = View.INVISIBLE      // 隐藏文本提示
105           // 目标检测是一个同步过程,为避免界面阻塞,将检测过程定义为协程模式
106           lifecycleScope.launch(Dispatchers.Default) { runObjectDetection(bitmap) }
107       }
108       // 对相机返回的图像解码并根据图像视图的大小进行裁剪
109       private fun getCapturedImage(): Bitmap {
110           // 视图的宽度与高度
111           val targetW: Int = inputImageView.width
112           val targetH: Int = inputImageView.height
113           val bmOptions = BitmapFactory.Options().apply {
114               inJustDecodeBounds = true
115               BitmapFactory.decodeFile(currentPhotoPath, this)
116               // 图像的宽度与高度
117               val photoW: Int = outWidth
118               val photoH: Int = outHeight
119               // 计算裁剪比例因子
120               val scaleFactor: Int = max(1, min(photoW / targetW, photoH / targetH))
121               inJustDecodeBounds = false
122               inSampleSize = scaleFactor
123               inMutable = true
124           }
125           // 获取照片的属性信息
126           val exifInterface = ExifInterface(currentPhotoPath)
127           val orientation = exifInterface.getAttributeInt(
128               ExifInterface.TAG_ORIENTATION,
129               ExifInterface.ORIENTATION_UNDEFINED
130           )
131           val bitmap = BitmapFactory.decodeFile(currentPhotoPath, bmOptions)
132           return when (orientation) { // 根据照片方向做适当旋转变换
133               ExifInterface.ORIENTATION_ROTATE_90 -> {
134                   rotateImage(bitmap, 90f)
135               }
136               ExifInterface.ORIENTATION_ROTATE_180 -> {
137                   rotateImage(bitmap, 180f)
138               }
139               ExifInterface.ORIENTATION_ROTATE_270 -> {
140                   rotateImage(bitmap, 270f)
141               }
142               else -> {
143                   bitmap
144               }
145           }
```

```
    }
    // 对图像进行旋转变换
    private fun rotateImage(source: Bitmap, angle: Float): Bitmap {
        val matrix = Matrix()
        matrix.postRotate(angle)
        return Bitmap.createBitmap(
            source, 0, 0, source.width, source.height,
            matrix, true
        )
    }
    // 创建图像文件,为相机拍摄的照片写入做准备
    @Throws(IOException::class)
    private fun createImageFile(): File {
        // 图像文件名称及其路径
        val timeStamp: String = SimpleDateFormat("yyyyMMdd_HHmmss").format(Date())
        val storageDir: File? = getExternalFilesDir(Environment.DIRECTORY_PICTURES)
        return File.createTempFile(
            "JPEG_${timeStamp}_", /* prefix */
            ".jpg", /* suffix */
            storageDir /* directory */
        ).apply {
            // 返回图像文件保存路径
            currentPhotoPath = absolutePath
        }
    }
    // 调用相机拍照
    private fun dispatchTakePictureIntent() {
        Intent(MediaStore.ACTION_IMAGE_CAPTURE).also { takePictureIntent ->
            // 确保有 camera activity 处理 intent
            takePictureIntent.resolveActivity(packageManager)?.also {
                // 创建存储相机数据的图像文件
                val photoFile: File? = try {
                    createImageFile()
                } catch (e: IOException) {
                    Log.e(TAG, e.message.toString())
                    null
                }
                // 如果文件创建成功
                photoFile?.also {
                    val photoURI: Uri = FileProvider.getUriForFile(
                        this,
                        "cn.edu.ldu.objectdetection.fileprovider",
                        it
                    )
                    // 保存图像
                    takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, photoURI)
                    // 回传相机拍照结果
                    startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
                }
```

```
195                 }
196             }
197         }
198         // 绘制检测结果,包括 Bounding Box、类别名称、置信度
199         private fun drawDetectionResult(
200             bitmap: Bitmap,
201             detectionResults: List<DetectionResult>
202         ): Bitmap {
203             val outputBitmap = bitmap.copy(Bitmap.Config.ARGB_8888, true)
204             val canvas = Canvas(outputBitmap)
205             val pen = Paint()
206             pen.textAlign = Paint.Align.LEFT
207             detectionResults.forEach {
208                 // 绘制 Bounding Box
209                 pen.color = Color.GREEN
210                 pen.strokeWidth = 8F
211                 pen.style = Paint.Style.STROKE
212                 val box = it.boundingBox
213                 canvas.drawRect(box, pen)
214                 val tagSize = Rect(0, 0, 0, 0)
215                 // 字体设置
216                 pen.style = Paint.Style.FILL_AND_STROKE
217                 pen.color = Color.YELLOW
218                 pen.strokeWidth = 2F
219                 pen.textSize = MAX_FONT_SIZE
220                 pen.getTextBounds(it.text, 0, it.text.length, tagSize)
221                 val fontSize: Float = pen.textSize * box.width() / tagSize.width()
222                 // 调整字体大小,让文本显示在框内
223                 if (fontSize < pen.textSize) pen.textSize = fontSize
224                 var margin = (box.width() - tagSize.width()) / 2.0F
225                 if (margin < 0F) margin = 0F
226                 canvas.drawText(
227                     it.text, box.left + margin,
228                     box.top + tagSize.height().times(1F), pen
229                 )
230             }
231             return outputBitmap                          // 返回绘制检测结果的图像
232         }
233     }
234     // 实体类,存储检测到的对象的可视化信息
235     data class DetectionResult(val boundingBox: RectF, val text: String)
```

第 70～99 行实现的函数模块 runObjectDetection,基于 TFLite Task Library 编写 Android 目标检测逻辑,通过 4 个步骤轻松完成,确实非常简单。第 96 行语句用线程模式更新界面,第 106 行语句用协程模式完成后台目标检测的推理过程,避免界面阻塞。

程序源码 P3.6 略去了从相册选择图片做目标检测的逻辑设计,该项功能也留到本章的课后习题,读者不难根据照相机的逻辑设计,自行完成相册的目标检测。更多解释参见本节视频讲解。

3.14 Android 手机测试

修改程序源码 P3.6 的第 76 行语句 setScoreThreshold(0.13f)的置信度阈值参数，可以影响返回的检测结果。

为了与 3.10 节的程序源码 P3.4 做对照，这里仍然采用 0.13 的阈值参数，并且选取 25.jpg 和 11156.jpg 作为教学演示图片。使用 Android 真机测试，检测结果分别如图 3.26 和图 3.27 所示。

图 3.26 25.jpg 图像置信度阈值为 0.13 的检测结果

图 3.27 11156.jpg 图像置信度阈值为 0.13 的检测结果

图 3.26 与图 3.18 均采用 25.jpg 图像做目标检测测试。图 3.26 所示的 Android 真机检测只发现了 3 个正确的目标，而在置信度阈值同为 0.13 的情况下，图 3.18 给出的正确检测结果是 5 个。事实上，可以把这种差异归结为手机对着屏幕拍照时屏幕的反光、抖动或其他光影效果对成像质量的影响造成的。

图 3.27 与图 3.19 均采用 11156.jpg 图像做目标检测测试。有意思的是，仍然是对着屏幕拍照，在置信度阈值相同的情况下，图 3.27 的检测效果却更好，显示检出了 5 个目标，比图 3.21 多出了 jiaozi。同时，对于汤品的认定也不同，图 3.27 给出的结果是 chinese soup，而图 3.19 给出的结果是 miso soup。

表 3.5 给出的实证测试对比，从一定程度上说明 EfficientDet 模型的健壮性好，泛化能力强。

表 3.5 两种场景检测效果对比(置信度阈值为 0.13)

类 别	真机对屏幕场景	TFLite 对图片场景
rice	17%，正确检测	37%，正确检测
cold tofu	31%，正确检测	21%，正确检测
miso soup	27%，chinese soup	34%，正确检测
french fries	29%，正确检测	27%，正确检测
jiaozi	19%，正确检测	没有检测到

图 3.28 是将置信度阈值调整为 0.1 后的检测结果，与图 3.26 做对比，虽然多出了 jiaozi 这个目标，然而并不正确，只是说明了阈值对结果的影响。图 3.29 给出了正确的检测结果。

图 3.28　置信度阈值为 0.1 的检测结果

图 3.29　置信度阈值为 0.2 的检测结果

3.15　小结

本章以美食场景中的食材检测为切入点，以 EfficientDet 模型应用于美食场景检测的方法路径为主线，完成了基于 MakeSense 的数据集标注、EfficientDet 论文深度解析、基于 TensorFlow Lite Model Maker 实现 TFLite 模型的训练与评估、基于 TFLite Task Library 实现 Android 版的美食场景检测。

3.16　习题

1. 目标检测常见的技术路线有哪些？
2. 如何为目标检测数据集定义标签？试举例说明。
3. EfficientDet 模型的主要创新点包括哪些？
4. 双向加权特征金字塔网络(BiFPN)与其他特征融合模式相比优势有哪些？
5. EfficientDet 模型的复合缩放方法是如何实现的？
6. EfficientDet 与哪些经典模型做了对比？论文给出的实验结论是什么？
7. 描述 EfficientDet 的结构，解析这种结构的优势。
8. TFLite 版模型建模路径有哪些？
9. 描述 EfficientDet-Lite 版模型做迁移学习的基本步骤。
10. 目标检测问题为什么会选择 mAP 作为评估指标？
11. 描述 mAP 指标的计算逻辑。

12. 计算机版 TFLite 模型与移动版 TFLite 模型的差异说明了什么问题？

13. 结合美食版 EfficientDet-Lite 模型的建模、训练与测试，侧重从健康美食的角度谈谈你对美食类 App 应用前景的瞻望。

14. 结合本章项目设计，谈谈在 Android 上部署应用 TFLite 模型的方法和步骤。

15. 美食版 TFLIte 模型与鸟类版 TFLite 模型部署有何不同？

16. 根据相机拍照的检测逻辑设计，自行完成相册的目标检测设计。