第 17 章 水果坏果检测

随着市场规模的不断扩张，食品加工产业正加速向大规模化、自动化转型。在这一背景下，产品质检环节，特别是水果加工中的质量检测，面临着前所未有的挑战。由于水果在采摘、运输、仓储等过程中易受损且易腐，传统的人工分拣方式已难以满足高效、精准的需求。因此，本案例致力于引入目标检测技术，以实现水果分拣的自动化，旨在提升产品质量，保障食品安全，并降低企业运营成本，推动食品加工产业的进一步发展。

17.1 问题分析

本案例聚焦于水果加工领域的自动化检测与监控，内容具体涵盖三个方面：

1. 实时目标检测，即要求系统能够快速且高精度地识别出多种不同品质的水果。
2. 是进行水果计数，准确统计出各类水果的数量。
3. 可视化检测过程，确保能够实时监控并直观展示检测情况。

为了满足这些需求，本案例选择了 YOLOv5s 模型作为目标检测算法，该模型以其高效性与准确性被广为使用。进一步地，为了提升处理速度，将此模型部署在算能平台上，并利用 TPU（Tensor Processing Unit）进行加速处理，从而确保系统在实际应用中能够满足实时性与高效性。

17.2 数据预处理

为了能够精准实现水果种类的识别及其新鲜度的检测，从 kaggle 网站选取了涵盖不同信息程度（如颜色、纹理、形状等）的多种水果图像，确保数据集的代表性。采集了 928 张图像，这些图像被细致地划分为 5 个类别：新鲜苹果、新鲜橙子、腐烂苹果、腐烂橙子等。其中，新鲜苹果图像 228 张，新鲜橙子图像 162 张，腐烂苹果图像 162 张，腐烂橙子图像 211 张。部分图像如图 17.1 所示。

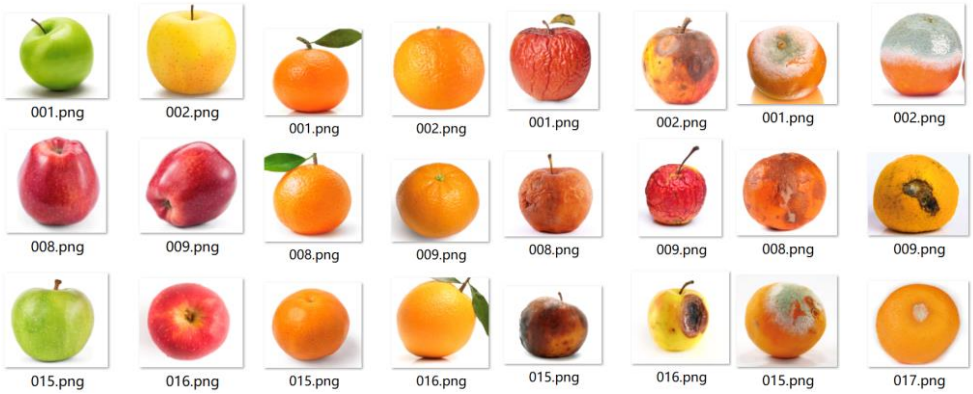

图 17.1 数据集部分图像

为了保证 YOLOv5s 预训练模型的学习效果，对上述数据集进行预处理，尤其是进行了数据增强，提升了模型的泛化能力。

当然，以下是保留段落格式并梳理后的文字：

1. 数据清洗

为了使数据集合成效果更好，只保留了仅包含一个物体，且背景较为清晰的图像。这一步骤确保了数据集的质量和一致性，为后续处理打下了良好的基础。

2. 图像随机选择

由于每种类别的图像数量不一致，直接随机选取会造成数据不平衡。结合实际情况中新鲜水果出现的概率更高这一特点，按照 3:3:2:2 的权重随机抽取了新鲜苹果、新鲜橙子、腐烂苹果、腐烂橙子类别下的图像。这样处理后，新鲜水果每种的标签数量约为 6000 个，腐烂水果的标签数量每种约为 4000 个。每张待生成的图像计划包含 5-10 个物体，以确保数据集的多样性和训练的有效性。

3. 图像数据增强

为了增加数据集的多样性和模型的泛化能力，将选取出的单个物体图像进行了随机旋转、缩放等数据增强操作。这些操作有助于模型学习到更加鲁棒的特征。

4. 图像融合

采用 OpenCV 的 seamlessClone 图像无缝融合函数，将多张单目标图像融合到同一画布中。这一函数的优势在于能够去除目标背景，使得多张图像间的融合更加自然，图像间的遮挡现象大大减少。

5. 随机放置图像

设置了一个 448 x 448 像素的空白图像作为背景，将上一步骤中挑选出的图像进行随机放置。为了保证图像间重叠部分不能过大，在放置当前物体前会计算其与画布上所有现有物体的 IoU（交并比）值。当该图像与画布上所有物体的平均 IoU 值大于某一设定值时，会重新随机生成放置坐标。这一步骤有效避免了多个水果重叠面积过大的问题，确保了生成图像的质量。

6. 标注信息

采用合成数据集的一个显著好处是，可以直接将图像坐标转换为目标检测的 bounding box，无需进行繁琐的手工标注。标注数据按照 YOLO 的格式存储，每张图像的标注对应一个 txt 文件。文件的每一行都包含了一个目标的信息，具体包括目标类别、中心点 x 坐标、中心点 y 坐标、宽度和高度，如图 17.2 所示。

```
1 0.662946 0.305804 0.183036 0.183036
2 0.214286 0.167411 0.171875 0.171875
1 0.571429 0.691964 0.198661 0.198661
3 0.334821 0.395089 0.178571 0.178571
3 0.816964 0.194196 0.165179 0.165179
1 0.290179 0.725446 0.165179 0.165179
```

图 17.2 标注信息示意图

梳理后的段落如下：

7. 数据集划分

在完成数据清洗、图像随机选择、数据增强、图像融合、随机放置图像等步骤后，共生成了 5000 组数据。为了更有效地利用这些数据，按照 6:2:2 的比例将其划分为训练集、验证集和测试集。具体地，生成了 train.txt、val.txt、test.txt 三个文件，分别记录了对应数据集（训练集、验证集、测试集）的图像路径，这些文件的格式遵循 YOLO 的规范。

尽管通过融合拼接方式生成的数据集已经具备了一定的泛化性，但在实际应用中，还希望模型能够对不同大小的水果有更好的识别能力。因此，在训练前的预处理阶段，对数据集

进行了进一步的“马赛克”增强处理。这一处理过程涉及选取 4 张图像，通过随机裁剪、缩放、旋转等操作，最终将它们合成为一张图像。这种方法不仅增加了单张图像中的目标数量，还补充了边缘遮挡的目标数据，有助于提升模型对复杂场景的适应能力。

此外，考虑到在本案例中，水果的新鲜度与颜色特征密切相关，特别对色调（HSV）进行了微调。具体来说，对色相（H）进行了微小调整，对饱和度（S）和明度（V）则分别进行了一定的调整。这种细微的颜色调整有助于模型更好地捕捉水果颜色变化中的细微差异，从而提高对水果新鲜度的判断能力。调整后的图像如图 17.3 所示。

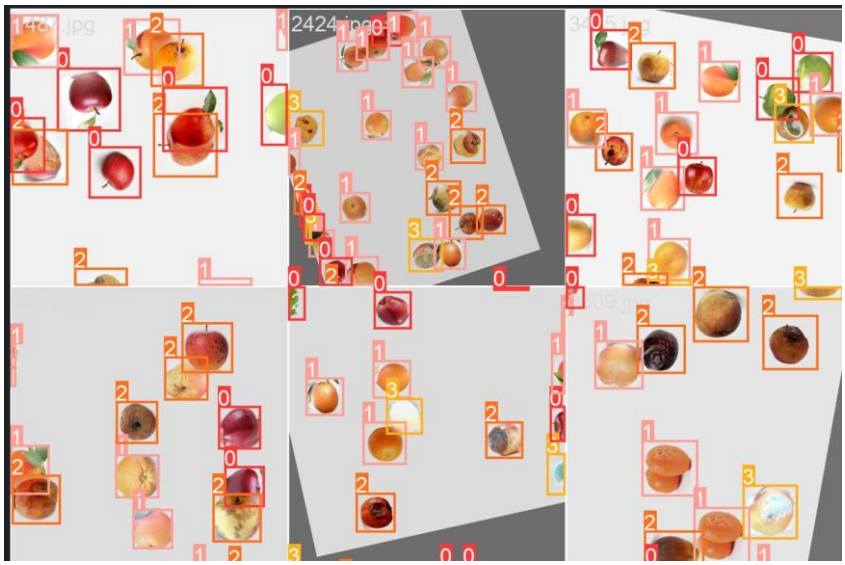

图 17.3 马赛克后的数据集

17.3 模型训练

本案例识别具有特定特征的目标图像，包括大小规整、类别较少、目标间重叠与差异小但数量较多等特点。鉴于这些特点，以及实时推理和成本控制的需求（使用性能较弱的边缘端设备），技术选型聚焦于小目标识别能力强、推理速度快且硬件部署方案成熟的 YOLOv5s 模型。

本案例采用 YOLOv5s 模型，该模型已在 COCO 数据集上预训练，支持 80 个物体类别的检测。为兼顾预测精度与推理速度，案例选用了 YOLOv5s 预训练模型，并通过迁移学习进行定制化训练：

1. 加载预训练模型权重

从 GitHub 仓库（https://github.com/ultralytics/yolov5）下载 YOLOv5s 模型的预训练权重文件，这些权重在 COCO 80 类别数据集上训练得到，具备强大的图像特征提取能力。

2. 冻结模型参数

在迁移学习过程中，为了加速训练过程并保留预训练模型的特征提取能力，选择冻结 backbone 网络的参数，避免对整个模型进行重新训练。

3. 超参数设置

训练配置基本遵循 YOLOv5s 的默认设置，采用 SGD 优化算法，学习率设定为 0.01。损失函数由三部分组成，分别针对 anchor 与真实框（gt_box）的 CIOU 损失、检测网格的置信度损失以及目标分类损失，三者的权重比设置为 0.05:0.5:1.0，以平衡不同损失项对模型训练的影响。

4、开始训练

完成上述配置后，使用 YOLOv5 自带的训练脚本开始训练。

```
# 5s
python3 ./ultralytics_yolov5/train.py --img 448 \    # 输入尺寸
    --batch 16 \
    --epochs 20 \
    --data ./data/fruit.yaml \  # 自定义数据集
    --weights ./ultralytics_yolov5/yolov5s.pt \ # 初始化权重
    --cache ram  \  # 启用缓存（加快数据读取）
    --name 5s_freeze \
    --cfg ./configs/yolov5s.yaml \  # 模型设置
    --hyp ./configs/hyp.yaml \  # 超参数
    --freeze 10     # 冻结前10层参数（backbone）
```

5. 训练过程

对于 YOLOv5s 模型训练 20 个 epoch，每个 epoch 之后更新在验证集上评估并调整学习率，并保存在当前最有的模型。

17.4 模型评估

在训练过程中， loss 曲线显示了模型学习的有效性，如图 17.4 所示。具体而言，无论是在训练集还是验证集上，所记录的三种不同类型的 loss 均呈现出持续下降的趋势，并最终达到了收敛状态。值得注意的是，cls_loss（分类损失）的下降最为显著，这表明该模型在训练过程中能够非常有效地捕捉并区分不同类别之间的图像特征差异，从而实现了对目标类别的准确识别。

图 17.4 YOLOv5s 训练和校验过程中 loss 曲线

图 17.5 显示了 YOLOv5s 模型在训练过程种 metrics 的变化。可见随着模型的迭代，模型预测性能提升显著。

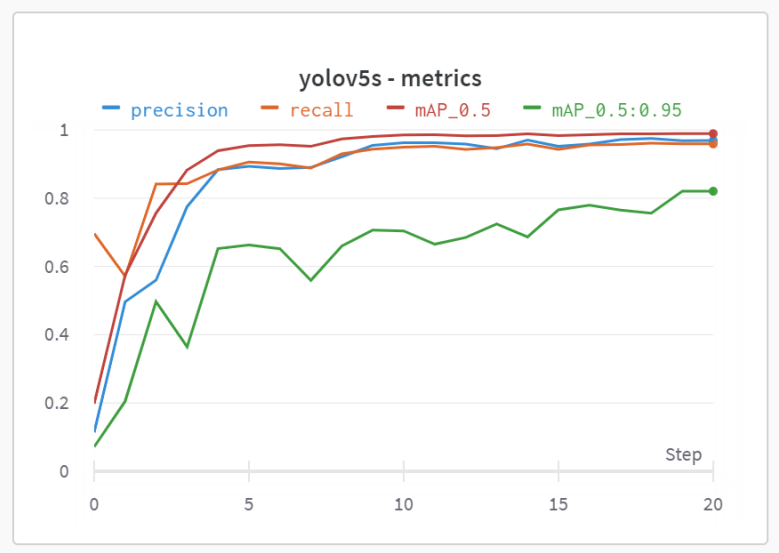

图 17.5 YOLOv5s 模型评估指标

17.5 模型加速

在工业应用场景中，鉴于推理设备资源有限且模型训练完成可能很少进行再训练，因此直接采用训练时所用的动态图模型框架（如 PyTorch）进行推理并非最佳选择。为了提高推理效率，一种常见的做法是将动态图模型转换为静态图模型（如 ONNX 格式），进而转化为更适合硬件加速的模型版本。这一转换过程旨在优化模型在资源受限环境下的执行效率。

本案例在算能 SOPHNET 平台提供的 TPU 服务器上进行模型迁移与部署。具体步骤如下：首先将 PyTorch 框架下的 YOLOv5 动态图模型导出为 ONNX 格式的静态图模型。利用 YOLOv5 官方提供的导出程序 export 来完成这一转换过程，以便后续在 TPU 服务器上高效地进行模型部署和推理。

```
# 导出onnx模型
python3 ./ultralytics_yolov5/export.py \
  --img 448 \
  --weights ./runs/train/5s_freeze/weights/best.pt \
  --include onnx

# 将导出的onnx模型`best.onnx`拷贝到`${YOLOv5}/build`文件夹下
mkdir ./build
cp ./runs/train/5s_freeze/weights/best.onnx ./build/yolov5s_v7.0_1output.onnx
```

为了能够使用 TPU 进行模型推理，需要进一步将模型转换为硬件设备支持的格式。使用平台提供的转换工具进行转换，并验证模型。结果如图 17.6 所示。

```
# 直接转32位浮点
python3 -m bmneto \
  --model=./build/yolov5s_v7.0_1output.onnx \
  --shapes="[1,3,448,448]" \
  --target="BM1684" \
  --outdir=./build/fp32model

# 验证模型信息
bm_model.bin --info ./build/fp32model/compilation.bmodel
```

```
root@a03a7fbcfb44:/tmp/fruit_detect# bm_model.bin --info ./build/fp32model/compilation.bmodel
bmodel version: B.2.2
chip: BM1684
create time: Sat Dec 24 21:54:49 2022

==========================================
net 0: [yolov5s_v7.0_1output.onnx]  static
------------
stage 0:
input: images, [1, 3, 448, 448], float32, scale: 1
output: output0, [1, 12348, 9], float32, scale: 1
```

图 17.6 浮点模型信息

转换后的模型结构已成功与 ONNX 格式保持一致，且由于原始模型采用 Float32 精度，转换后的模型也保留了这一精度设置，其模型大小为 27MB。然而，在模型训练阶段虽常用 32 位浮点数来确保高精度，但在实际应用中，尤其是计算资源受限、功耗敏感的移动设备上，这种高精度并非总是必要。为此，模型量化技术应运而生，它通过减少表示参数所需的比特数来降低资源消耗，尽管这会伴随少量精度的损失。

本例采用了 cali 工具将上述 ONNX 模型进一步转换为 INT8 模型，以优化其在移动设备上的表现。值得注意的是，量化过程中可能存在精度损失，因此量化工具需依据特定数据集对量化模型进行反复调整，以确保量化后的模型输出与原始模型尽可能接近。

为实现这一目标，编写了一个 bash 脚本，从验证集中自动抽取了 200 张图像作为量化过程的基准数据集，以便量化工具能够基于这些数据对模型进行精细调整。

```
# 构建量化数据集
dst_img_dir=./data/val200
if [ ! -d "$dst_img_dir" ]; then
    echo "create val200 dir: $dst_img_dir"
    mkdir -p $dst_img_dir
fi

n=0
for line in `cat ./data/val.txt`
do
    file_real_path=$(cd data && readlink -f $line)
    ln -s $file_real_path $dst_img_dir

    n+=1
    if((n>=200)); then
        break
    fi
done
```

转换脚本如下，执行后将生成量化后的模型，结果如图 17.7 所示。

```
# 量化
#!/bin/bash

CURDIR=$(cd $(dirname ${BASH_SOURCE[0]})/../ ); pwd )
auto_cali_dir=$CURDIR/auto_cali

if [ ! -d "${auto_cali_dir}" ]; then
    echo "create data dir: ${auto_cali_dir}"
    mkdir -p ${auto_cali_dir}
fi
pushd ${auto_cali_dir}
python3 -m ufw.cali.cali_model \
 --net_name yolov5s_cali \
 --model $CURDIR/build/yolov5s_v7.0_1output.onnx \
 --cali_image_path $CURDIR/data/val200 \
 --cali_image_preprocess 'resize_h=448,resize_w=448;scale=0.003921569,bgr2rgb=True' \
 --input_shapes "[1,3,448,448]" \
 --cali_iterations=1
popd

mv $CURDIR/build/yolov5s_cali $CURDIR/build/int8model
```

```
root@a03a7fbcfb44:/tmp/fruit_detect# bm_model.bin --info ./build/int8model/compilation.bmodel
bmodel version: B.2.2
chip: BM1684
create time: Sun Dec 25 00:17:30 2022

=========================================
net 0: [yolov5s_cali]  static
------------
stage 0:
input: images, [1, 3, 448, 448], int8, scale: 132.533
output: output0, [1, 12348, 9], float32, scale: 1
```

图 17.7 量化模型信息

通过模型量化，模型的输入格式从 FP32 转换为了 INT8，这一转变显著减小了模型体积，从原来的大小缩减了近 3/4，最终仅为 7.5MB。为了评估量化对模型性能的具体影响，分别使用浮点模型和量化模型对相同图像进行了推断对比，如图 17.8 所示（左图为 FP32 格式）。

在识别精度方面，浮点模型展现出了其高准确性，能够准确无误地识别出图像中的所有目标，且置信度均保持在 0.9 以上。相比之下，量化模型虽然整体上仍能识别出大部分目标，但却出现了误判情况，即将一个“苹果”错误地识别为“橘子”，且其置信度和包围框的精确度均有所下降，包围框尺寸甚至超出了物体的实际范围。然而，在推断速度上，量化模型展现出了其显著优势。与浮点模型相比，量化模型的推断速度从 0.09 秒大幅提升至了 0.01 秒，这对于资源紧缺、实时性要求较高的应用场景而言尤为关键。

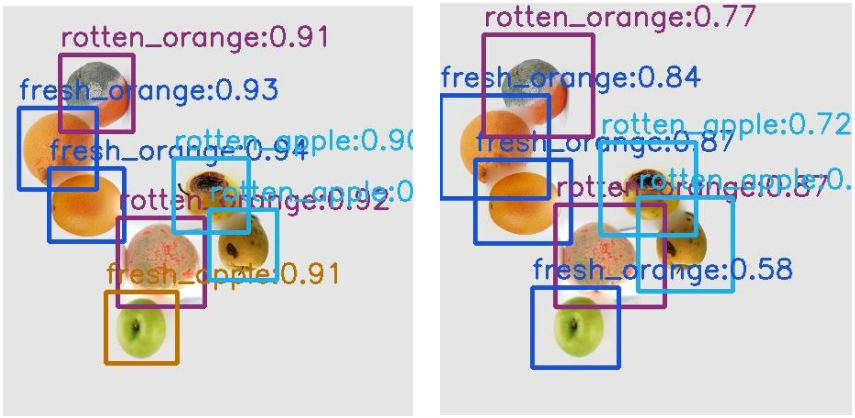

图 17.8 量化模型推断效果对比