

学号 2019302120317

密级 _____

武汉大学本科毕业论文

面向边缘设备的目标检测方法

院（系）名称：电子信息学院

专业名称：电子信息工程

学生姓名：杨宁福

指导教师：梅天灿 副教授

二〇二三年五月

郑重声明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：杨宁福

日期：2023年5月1日

摘要

近年来，目标检测在自动驾驶、人脸识别、安防监控、医学影像等多个领域应用广泛。尽管深度网络模型在目标检测中取得了巨大成功，但是由于模型规模庞大，使得深度网络目标检测模型难以部署至边缘端设备进行实时检测任务。因此，本文的目标是提出一种面向边缘设备的目标检测方法，并对目标检测模型的性能进行测试和对比。

首先，通过对目前主流的目标检测模型进行对比，选择了 YOLOv8 作为本文的目标检测模型，同时对 YOLOv8 网络结构中的 C2f 和 Detect 层进行修改后在 PC 端训练好模型，然后将其转换为 NCNN 格式并进行简化和优化，结合 YOLOv8NCNN 和 YOLOXNCNN 两个框架并修改相关参数将 YOLOv8 模型部署到 Android 端，实现了摄像头实时检测和图片检测。除此之外，通过修改部署框架，使多个不同数据集训练的 YOLOv8 模型可以同时部署在 Android 上。最后通过修改图片检测功能，使其在图片检测的选择图片时可以进行多选，并将检测结果(类别、置信度和预测框坐标 x, y, w, h)以及检测时间输出到 Android 端文档路径下的 txt 文档。同时在 COCO 数据集的验证集中选取 500 张图片进行测试，将检测结果和检测时间传输到 PC 端，并将检测结果转化成 JSON 文件，在 Python 中利用 pycocotools 库进行 mAP 值的计算；通过对检测时间取平均值，可以得到模型对数据集的平均检测时间，并将该结果作为 FPS 值。

最终可以得到本文的 YOLOv8n 模型在 Android 端使用 CPU 进行目标检测的 FPS 值为 40.66，mAP 值为 20.06%，使用 GPU 进行目标检测时的 FPS 值为 23.62，mAP 值为 19.64%；YOLOv8s 模型在 Android 端使用 CPU 进行目标检测的 FPS 值为 26.25，mAP 值为 28.07%，使用 GPU 进行目标检测时的 FPS 值为 19.50，mAP 值为 28.78%；YOLOv8m 模型在 Android 端使用 CPU 进行目标检测的 FPS 值为 15.22，mAP 值为 35.10%，使用 GPU 进行目标检测时的 FPS 值为 16.22，mAP 值为 35.21%。通过对比 TensorFlow 官方提供的模型以及 YOLOv5 模型在 Android 端的性能，YOLOv8 模型在速度和精度上均有着非常不错的表现。

关键词：目标检测；Android 设备；YOLOv8 模型；平均精度均值；帧率

ABSTRACT

In recent years, object detection has been widely used in various fields such as autonomous driving, face recognition, security monitoring, and medical imaging. Despite the huge success of deep network models in object detection, the large model size makes it difficult to deploy deep network object detection models to edge devices for real-time detection tasks. Therefore, the goal of this paper is to propose an object detection method for edge devices and test and compare the performance of object detection models.

Firstly, after comparing the mainstream object detection models, YOLOv8 was selected as the object detection model for this paper. The C2f and Detect layers in the YOLOv8 network structure were modified and the model was trained on a PC. Then, it was converted to the NCNN format and simplified and optimized. The YOLOv8 model was deployed on Android by combining the YOLOv8NCNN and YOLOXNCNN frameworks and modifying the relevant parameters, enabling real-time detection and image detection. In addition, by modifying the deployment framework, multiple YOLOv8 models trained on different datasets can be deployed on Android at the same time.

Finally, by modifying the image detection function, multiple selections can be made when selecting images for detection, and the detection results (category, confidence, and predicted bounding box coordinates x, y, w, h) and detection time are output to a txt file in the Android document path. 500 images were selected from the COCO dataset for testing, and the detection results and detection time were transmitted to a PC. The detection results were then converted into a JSON file, and the mAP value was calculated using the pycocotools library in Python. By taking the average of the detection time, the average detection time of the model on the dataset was obtained, which was used as the FPS value.

In conclusion, the YOLOv8n model achieved an FPS value of 40.66 and an mAP value of 20.06% when performing object detection using the CPU on Android. When using the GPU, the FPS value was 23.62 and the mAP value was 19.64%. The YOLOv8s model achieved an FPS value of 26.25 and an mAP value of 28.07% when performing object detection using the CPU on Android. When using the GPU, the FPS value was 19.50 and the mAP value was 28.78%. The YOLOv8m model achieved an FPS value of 15.22 and an mAP value of

35.10% when performing object detection using the CPU on Android. When using the GPU, the FPS value was 16.22 and the mAP value was 35.21%. By comparing the performance of TensorFlow's official models and the YOLOv5 model on Android, the YOLOv8 model demonstrates excellent performance in terms of speed and accuracy.

Keywords: Object Detection;Android device;YOLOv8 model;mAP;FPS

目录

| | |
|------------------------------------|----|
| 1 绪论 | 1 |
| 1.1 研究背景 | 1 |
| 1.1.1 目标检测的应用 | 1 |
| 1.1.2 目标检测的计算部署 | 2 |
| 1.1.3 目标检测的计算量庞大 | 2 |
| 1.2 国内外研究现状 | 2 |
| 1.3 研究内容 | 4 |
| 1.4 论文的组织结构 | 5 |
| 2 相关技术 | 6 |
| 2.1 目标检测的主流模型 | 6 |
| 2.1.1 二阶段方法 | 6 |
| 2.1.2 一阶段方法 | 6 |
| 2.1.3 YOLOv8 算法 | 7 |
| 2.2 深度学习框架 | 9 |
| 2.3 模型格式介绍与转换 | 9 |
| 2.3.1 PyTorch | 9 |
| 2.3.2 ONNX | 10 |
| 2.3.3 NCNN | 11 |
| 2.3.4 模型格式转换 | 11 |
| 2.4 深度网络模型在 Android 设备上的部署框架 | 12 |
| 2.4.1 框架介绍 | 12 |

| | |
|-----------------------------------|-----------|
| 2.4.2 部署流程 | 13 |
| 2.5 目标检测模型的性能指标 | 14 |
| 2.5.1 IOU | 14 |
| 2.5.2 精度指标 mAP | 14 |
| 2.5.3 速度指标 FPS | 15 |
| 3 边缘设备上目标检测系统的设计与测试 | 17 |
| 3.1 总体实验流程 | 17 |
| 3.2 移动端的 YOLOv8 模型 | 18 |
| 3.2.1 YOLOv8 模型结构修改 | 18 |
| 3.2.2 YOLOv8 模型训练 | 19 |
| 3.2.3 模型格式转换 | 20 |
| 3.3 模型在 Android 设备上的部署 | 21 |
| 3.3.1 模型推理 | 21 |
| 3.3.2 部署多种模型 | 22 |
| 3.4 Android 设备上的模型测试 | 22 |
| 3.4.1 精度测试 | 22 |
| 3.4.2 速度测试 | 23 |
| 4 测试与验证 | 24 |
| 4.1 不同模型的 mAP 值和 FPS 值测试及对比 | 24 |
| 4.2 摄像头实时检测 | 26 |
| 4.3 图片检测 | 28 |
| 4.4 多模型测试 | 29 |

| | |
|---------------|----|
| 5 总结和展望 | 31 |
| 5.1 总结 | 31 |
| 5.2 展望 | 32 |
| 参考文献 | 33 |
| 致谢 | 36 |

1 緒論

目标检测是计算机视觉领域中的一个重要任务，它的主要目的是在图像或视频中自动识别和定位感兴趣的物体。近年来，随着物联网和移动计算的快速发展，越来越多的智能设备需要进行目标检测，这些设备包括智能手机、摄像头、智能门锁等边缘设备。

传统的目标检测算法通常需要大量的计算资源和存储空间，因此难以在边缘设备上实现实时的目标检测。但是，随着深度学习算法的发展，越来越多的轻量化目标检测算法被提出，这些算法可以在边缘设备上进行实时的目标检测，并具有较高的准确率和实用性。

1.1 研究背景

1.1.1 目标检测的应用

目标检测是计算机视觉中的一个重要任务，它的应用非常广泛，包括但不限于以下领域：

1) 自动驾驶：在自动驾驶汽车中，目标检测用于识别交通标志、行人、车辆和其他障碍物，以便自动驾驶系统能够做出相应的决策和行动。例如，深度学习模型 YOLO (You Only Look Once) 在自动驾驶中的应用被证明非常有效^[1]。

2) 人脸识别：目标检测可以用于人脸检测和识别。在人脸检测中，目标检测模型可以在图像中自动识别人脸，从而实现自动化的人脸识别。文献^[2] 中提出的 MTCNN (Multitask Cascaded Convolutional Networks) 方法在人脸识别领域中取得了良好的表现。

3) 安防监控：目标检测可用于安防监控，以便在实时视频中识别出人、车、物等目标。这可以帮助安全人员及时发现异常情况，从而保护公众的安全。文献^[3] 中提出的 Faster R-CNN 模型已经被广泛用于安防监控中的物体识别。

4) 医学影像：在医学影像领域中，目标检测可用于识别和定位病变区域，以辅助医生进行疾病诊断。文献^[4] 中，作者使用卷积神经网络进行糖尿病视网膜病变和老年性黄斑变性的自动检测。

总而言之，目标检测在计算机视觉领域中具有广泛的应用前景，可以帮助人们更

好地利用图像和视频数据，提高自动化程度和工作效率。

1.1.2 目标检测的计算部署

对于目标检测应用的部署而言，由于其根本的深度神经网络需要大量的计算资源，所以目标检测的推理计算多数在云端数据中心部署^[5]。云端部署指的是将目标检测算法部署在云服务器上进行计算和处理。

但是随着边缘设备的不断发展，其计算能力也在不断增强，越来越多的目标检测已在边缘设备上部署。边缘设备部署的优点包括：

- 1) 低延迟。将目标检测算法部署在边缘设备上，可以避免数据传输时延，能够实现更低的响应延迟，提高用户体验。
- 2) 高安全性。由于数据在本地进行处理，可以避免数据在网络传输过程中被窃取或篡改的风险，提高数据的安全性。
- 3) 节省带宽。将目标检测算法部署在边缘设备上，可以避免数据在网络传输过程中占用大量的带宽，降低了网络传输的负担。

总体来说，云端部署适合处理大规模的图像数据，同时也适合需要灵活扩展计算资源的业务场景；而边缘设备部署适合对数据响应延迟和数据安全性要求较高的业务场景，同时也适合需要节省带宽和减少网络传输负担的场景。

1.1.3 目标检测的计算量庞大

由于目标检测需要在大量的图像中进行目标检测和定位，因此需要处理大量的图像数据，并对每个图像进行多次卷积和池化操作。这些计算操作需要大量的计算资源和时间，而且在实际应用中，往往需要在有限的时间和资源内完成目标检测任务，因此需要对目标检测算法进行计算优化。

1.2 国内外研究现状

目标检测的发展可以分为两个阶段：传统目标检测算法和基于深度学习的目标检测算法。传统的目标检测方法包括基于特征提取和分类器的方法，这些方法在小规模数据集上表现良好，但在大规模数据集上性能较差。随着深度学习技术的不断发展，基于深度学习的目标检测方法也随之出现。最初的方法是基于卷积神经网络的物体检测方法，比如 R-CNN^[6] 首次使用深度模型提取图像特征，以 49.6% 的准确率开创了检

测算法的新时代。随后 Fast R-CNN^[7]、Faster R-CNN^[8]、Mask R-CNN^[9] 等相继被提出，这些网络均属于二阶段 (two stage) 方法，需要在检测前生成一系列候选框，再对这些候选框进行分类和位置回归，计算量较大，无法满足实时检测的要求。最终一阶段 (one stage) 方法的出现又将检测领域带到一个新的高度，一阶段方法以 YOLO^{[10][11][12]} 和 SSD^[13] 为代表，它们通过一个神经网络直接对整张图片进行分类预测和位置回归，相较于二阶段方法准确率较低，但是速度更快，可以做到真正意义上的实时效果。

随着着物联网和人工智能的迅猛发展，将深度学习网络部署到边缘设备中已经成为研究的热点之一。边缘设备的深度学习网络部署研究现状涉及到深度学习模型设计、模型压缩和模型部署等多个方面，以下将从这些方面介绍目前的研究进展和技术趋势。

1) 模型设计

针对边缘设备的深度学习网络设计，目前主要的思路是通过精心设计网络结构，提高模型的计算效率和存储效率。例如，一些研究者提出了基于网络宽度和深度的模型设计方法，例如 MobileNet^[14]、ShuffleNet^[15] 等，这些模型通过减少模型参数和计算量，实现了在边缘设备上高效运行的目标检测模型。

2) 模型压缩

边缘设备计算能力和存储空间有限，因此需要通过模型压缩来减少模型大小和计算量，提高模型在边缘设备上的运行效率。常用的模型压缩技术包括剪枝、量化和知识蒸馏。其中，剪枝技术通过删除冗余的神经元和连接来压缩模型^[16]。量化技术通过减少参数的表示精度来减小模型大小，如 8 位整型量化^[17]。而知识蒸馏技术则通过利用大型网络的知识来指导小型网络的训练，从而实现模型压缩^[18]。

3) 模型部署

模型部署是将训练好的深度学习模型部署到边缘设备上，实现目标检测任务的关键环节。目前常用的模型部署技术包括离线部署和在线部署。离线部署是将训练好的模型编译成可执行文件，部署到边缘设备上，可以实现较高的运行效率；在线部署是通过将模型部署在云端服务器上，将数据传输到云端进行处理后再返回结果，可以实现更高的灵活性和可扩展性。近年来，一些开源工具和平台也推出了针对边缘设备深度学习网络部署的工具和平台，例如 TensorFlow Lite、Caffe2、MXNet、Keras 以及 NCNN^[19] 等。这些框架和工具可以使模型在边缘设备上快速运行，并且可以自动处理一些部署相关的细节问题，例如模型转换、加速等。同时，也可以通过针对具体设备

的定制化优化来实现模型部署，例如 MobileNet^[20]、ShuffleNet^[21]、Tiny-YOLO^[22] 等。这些模型都是通过对特定设备的性能和限制进行优化，从而实现在边缘设备上高效运行。

总而言之，边缘设备的深度学习网络部署研究现状包括模型设计、模型压缩和模型部署等多个方面，研究者通过提出新的模型结构、模型压缩技术和模型部署方案等，不断提高模型在边缘设备上的运行效率和精度

1.3 研究内容

本文主要研究面向边缘设备的目标检测方法，而 Android 作为一种非常常见的边缘设备，因此本文的主要目标是研究面向 Android 设备的目标检测的相关技术。本文的研究内容主要由以下三点组成。

1) 获取可部署在移动端的目标检测模型：通过对目前主流的目标检测模型进行对比，选择 YOLOv8 作为本文的目标检测模型，同时因为选择了 NCNN 作为模型在 Android 端的部署框架。为了模型在 NCNN 框架上部署后能够正常检测，需要对 YOLOv8 网络结构中的 C2f 和 Detect 层进行修改，然后利用修改后的模型来进行训练自己的目标检测模型；在 PC 端训练好 YOLOv8 模型后，需要将其从 Pytorch 格式转换为 ONNX 格式并对转换后的模型利用 ONNXsim 工具对其进行简化，去除冗余，然后利用 onnx2ncnn 工具将模型从 ONNX 格式转换成 NCNN 格式，最后利用 ncnnoptimize 工具将模型进行优化，并输出为 fp16 半精度浮点数格式。

2) 模型部署：由于 YOLOv8NCNN 框架只实现了摄像头实时检测功能，没有图片检测功能，在后续无法测试部署到 Android 端的目标检测模型的 mAP 值以及不同模型对于处理单张图片的速度和精度对比，而 YOLOXNCNN 框架实现了图片检测，因此本文结合上述两个框架将 YOLOv8 部署到 Android 端，并实现了摄像头实时检测、图片检测以及检测结果保存功能。同时，通过修改置信度阈值以及 NMS 的阈值使模型达到了一个比较好的检测效果。最后，通过修改框架，可以使多个不同数据集训练的 YOLOv8 模型同时部署在 Android 上，在 APP 的用户界面中通过下拉栏选择对应模型进行检测。

3) 模型测试：通过修改图片检测功能，使其在图片检测的选择图片时可以进行多选，至多选择 100 张图片，并将检测结果(类别、置信度和预测框坐标 x, y, w, h)以及检测时间输出到 Android 端文档路径下的 txt 文档，检测结果的文档名称为检测

图片的名称，检测时间保存至 run_time.txt 文档中。同时在 COCO 数据集的验证集中选取 500 张图片进行测试，将 500 个保存着检测结果的 txt 文档和记录每次检测所需时间的 run_time.txt 文档传输到 PC 端，并将检测结果转化成 JSON 文件，在 Python 中利用 pycocotools 库进行 mAP 值的计算。最后通过对检测时间取平均值，可以得到模型对数据集的平均检测时间，并将该结果作为 FPS 值。

1.4 论文的组织结构

关于本文的组织结构，说明如下。

第一章，通过研究背景、国内外研究现状、研究内容三个方面说明了本文的主旨。研究背景中说明了目标检测应用和在边缘设备上部署目标检测模型的优势以及所面临的挑战。国内外研究现状中阐述了边缘计算和目标检测技术的目前技术水平。研究内容中概述了本文要研究的面向边缘设备的目标检测的技术内容。此外还介绍了本文的组织结构。

第二章，主要由五部分内容组成。一，对目标检测的主流算法进行比较分析，并详细的介绍了 YOLOv8 算法；二，介绍了 YOLOv8 工程使用的深度学习框架 PyTorch；三，对实验过程中的相关模型格式及其转换进行了介绍；四，对目前一些深度学习网络模型在 Android 设备上的部署框架进行对比，并详细的介绍了 NCNN 框架；五，介绍了目标检测中 IOU 的含义和计算以及目标检测模型的性能指标 mAP 和 FPS。

第三章，首先从总体实验流程的角度说明本文在 Android 端实现目标检测并设计算法对其性能进行测试的技术路线，然后对研究内容的三个方面进行了详细的介绍。

第四章，首先从检测速度和检测精度两个方面，测试并对比了 YOLOv8 模型在 Android 端的性能；然后展示了 YOLOv8 模型部署到 Android 端进行实时检测和图片检测的效果，并对相关结果进行了分析；最后展示了多模型检测的效果，同时说明了该功能的应用场景。

2 相关技术

2.1 目标检测的主流模型

2.1.1 二阶段方法

R-CNN、Fast R-CNN、Faster R-CNN 和 Mask R-CNN 都是深度学习中用于目标检测的二阶段方法，其中 R-CNN (Region-based Convolutional Neural Network) 是一个基于区域的卷积神经网络算法，它通过对图像中每个区域进行分类来实现目标检测。R-CNN 的缺点是计算量非常大，因为需要对每个区域进行独立的卷积计算，因此速度比较慢；Fast R-CNN 是对 R-CNN 的改进，它使用了全卷积网络来替代 R-CNN 中的卷积操作，从而减少了计算量。Fast R-CNN 同时也引入了 RoI 池化层，可以有效地处理不同大小的区域，提高了检测速度和准确率；Faster R-CNN 是对 Fast R-CNN 的改进，它引入了 Region Proposal Network (RPN) 来代替 Selective Search 等方法，从而实现了端到端的训练，使得检测速度更快，同时准确率也有所提高；Mask R-CNN 是对 Faster R-CNN 的改进，它在 Faster R-CNN 的基础上增加了一个分支网络来生成每个检测框内部的像素级别的语义分割，从而实现了目标检测和语义分割的联合训练，进一步提高了模型的精度。总的来说，这些算法都是基于卷积神经网络的目标检测算法，它们在不同方面进行了改进，从而逐步提高了检测速度和准确率，为目标检测任务的应用提供了有力的支持。

但是二阶段目标检测方法通常包括区域提取和分类两个步骤，需要进行多次计算和复杂的图像处理。与之相比，一阶段目标检测方法通常只需要一次前向传递就可以完成目标检测，因此其速度更快，更适合在边缘设备上部署。

2.1.2 一阶段方法

RetinaNet^[23]、SSD 和 YOLO 都是现在流行的一阶段目标检测算法。

RetinaNet 是由 FAIR (Facebook AI Research) 提出的一种基于单阈值的目标检测模型，它通过引入 Focal Loss 解决了目标检测中正负样本不平衡的问题。RetinaNet 将 FPN 网络与特征金字塔的思想相结合，同时在每个金字塔层次上引入两个分支，一个

用于回归边界框，另一个用于预测物体的类别，从而实现了高精度的目标检测。

SSD (Single Shot MultiBox Detector) 由 Google 在 2016 年提出，其主要思想是在输入图像中同时预测出多个目标框，并对这些目标框进行分类和回归。SSD 的特点是快速、准确，并且只需要一个网络即可完成整个目标检测过程。在 SSD 中，输入图像被经过多层卷积和池化之后，通过多个不同大小的卷积核进行特征提取，从而得到多个特征图。接着，每个特征图中的每个位置都会生成若干个预测框，预测框的数量和大小取决于特征图的大小和比例。

YOLO 由 Joseph Redmon 在 2015 年提出，其主要思想也是在输入图像中同时预测出多个目标框，并对这些目标框进行分类和回归。与 SSD 不同的是，YOLO 是通过单个卷积神经网络来实现目标检测，这个网络会同时输出目标的类别和位置。在 YOLO 中，输入图像会被分成若干个网格，每个网格负责预测包含在其内部的目标框。YOLO 还使用了一种特殊的损失函数，可以同时考虑目标框的位置和类别信息，从而在目标检测中取得了不错的效果。

虽然 RetinaNet 在目标检测精度方面表现非常出色，但它需要更多的计算资源和更大的模型大小。因此，在计算资源较为有限的 Android 设备上部署可能会遇到性能瓶颈。SSD 在处理小物体时表现较好，并且具有较好的可扩展性，可以处理多尺度输入。相比之下，YOLO 系列最新的算法 YOLOv8 在速度和准确度方面表现更出色，尤其是在大目标检测方面。同时，YOLOv8 采用了较轻量化的模型结构，因此其模型大小相对较小，更适合在资源受限的 Android 设备上使用。

2.1.3 YOLOv8 算法

YOLOv8 是一个 SOTA 模型，它建立在以前 YOLO 版本的成功基础上，并引入了新的功能和改进，以进一步提升性能和灵活性。具体创新包括一个新的骨干网络、一个新的 Ancher-Free 检测头和一个新的损失函数。

首先，YOLOv8 的检测策略仍是分而治之，即将一幅图像分成 $n \times n$ 个网格 (grid cell)，每个 grid cell 要预测 B 个边界框 (bounding box)，每个 bounding box 要预测框的位置和大小 (x, y, w, h) 以及置信度 confidence 共 5 个值，同时每个 grid cell 还要预测一个类别信息，即每个类别的概率，用 grid cell 的类别概率，乘以 grid cell 自己生成的 bounding box 的置信度，就获得该 bounding box 对应各个类别的概率。最后在后处理中使用阈值过滤和非极大值抑制 (NMS) 将类别概率小于设定阈值的 bounding box

和与类别概率较大的 bounding box 重合度高于设定阈值的 bounding box 剔除，可以得到最后的检测结果。

然后是 YOLOv8 的网络结构，如下图 2.1 所示。

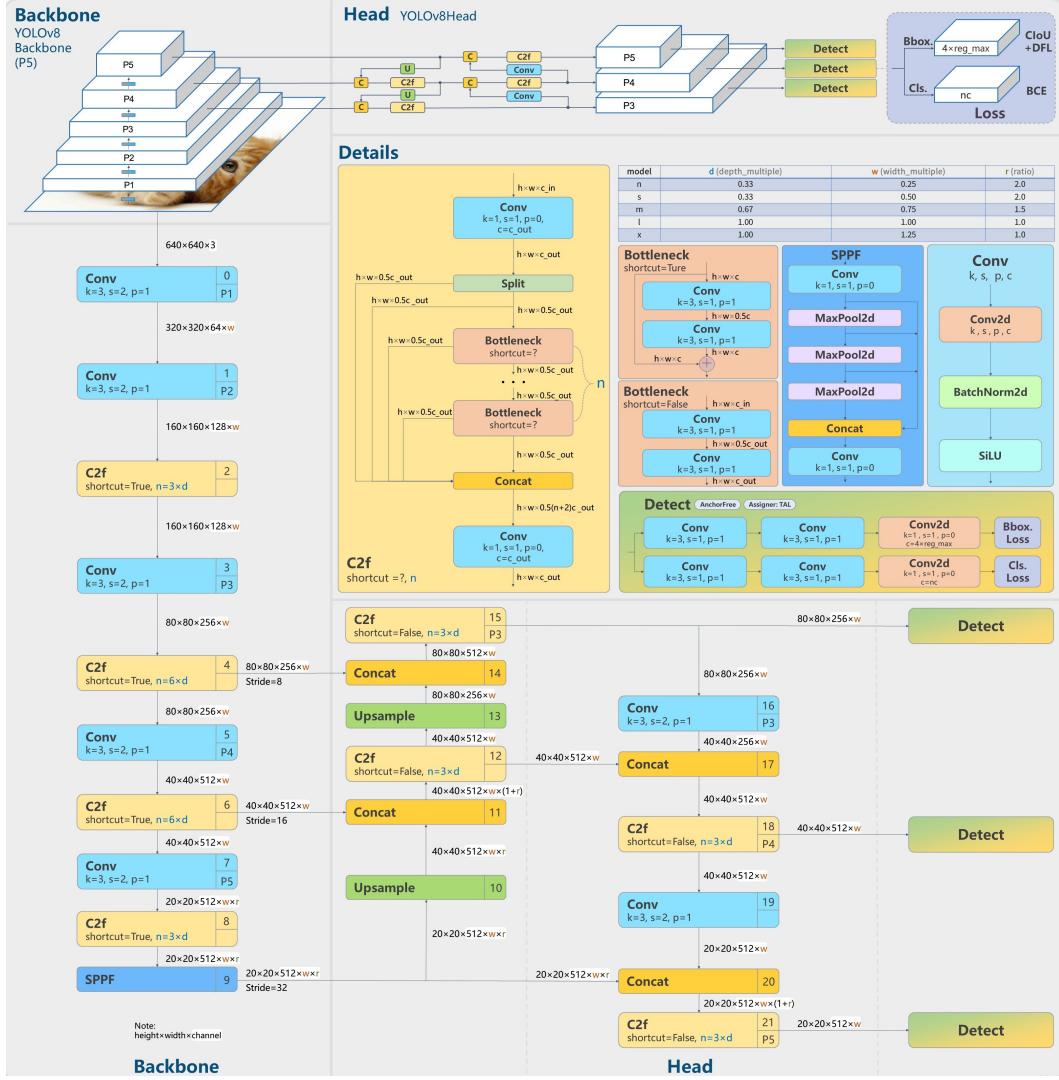


图 2.1 YOLOv8 网络结构图

YOLOv8 的骨干网络和 Neck 部分参考了 YOLOv7 ELAN 设计思想，将 YOLOv5 的 C3 结构换成了梯度流更丰富的 C2f 结构，并对不同尺度模型调整了不同的通道数。除此之外还将 YOLOv5 的 Coupled-Head 换成了目前主流的 Decoupled-Head。

其次，YOLOv8 从 Anchor-Based 换成了 Anchor-Free，减少了预测框的数量的同时也加速了 NMS 非极大值抑制。

最后，YOLOv8 的损失函数抛弃了以往的 IOU 匹配或者单边比例的分配方式，而是使用了 Task-Aligned Assigner 正负样本匹配方式。同时，Loss 计算抛弃了 YOLOv5

的 Obj 分支，只包括 2 个分支：Cls 分类和 Bbox 回归分支，其中分类分支依然采用 BCE Loss，而回归分支使用了 Distribution Focal Loss 的同时还使用了 CIoU Loss。3 个 Loss 采用一定权重比例加权即可。

2.2 深度学习框架

YOLOv8 使用的深度学习框架是 PyTorch。PyTorch 由 Facebook 开发并维护。它的特点是易于使用、灵活性高、性能强大、支持动态图和静态图，以及可扩展性强。

PyTorch 的核心是一个多维数组库，称为 Tensor（张量），它与 NumPy 的数组类似。PyTorch 还支持自动求导（autograd）功能，这使得编写神经网络变得更加容易和直观。用户可以通过定义计算图来实现自动求导，这使得训练深度学习模型变得更加简单和高效。

PyTorch 支持动态图和静态图两种模式，可以根据用户的需求进行选择。动态图模式适用于灵活的计算需求，用户可以使用标准的 Python 控制流（例如 if 语句和循环）来构建计算图。静态图模式适用于需要高度优化的计算需求，用户可以使用 PyTorch 的 JIT 编译器来将计算图编译成高效的机器码。

此外，PyTorch 还提供了许多高级功能，例如分布式训练、GPU 加速、模型保存和加载、可视化工具等。这些功能使得 PyTorch 成为一个全面的机器学习框架，能够满足各种复杂的计算需求。

总而言之，PyTorch 是一个功能强大、易于使用、灵活性高的机器学习框架，可以满足各种不同的计算需求。

2.3 模型格式介绍与转换

2.3.1 PyTorch

PyTorch 的.pt 文件是一种用于保存 PyTorch 训练好的神经网络模型的二进制文件格式。它可以保存模型的架构和参数，并可以在不同的平台和设备之间移植，无需重新训练模型。

.pt 文件中包含了以下内容：

1) 模型的架构：即神经网络的结构，包括各个层的类型、输入输出维度、激活函数等信息。这部分信息通常被保存在模型定义的 Python 代码中，也可以通过 PyTorch

提供的保存和加载接口保存在.pt 文件中。

2) 模型的参数：即神经网络中各个层的权重和偏置等可训练参数。这些参数在训练过程中被优化，保存在.pt 文件中，以便在推理阶段使用。

3) 计算图信息：由于 PyTorch 使用动态计算图机制，即在每次前向传播时动态构建计算图，因此.pt 文件中也包含了完整的计算图信息。这部分信息在模型训练和调试中非常有用，可以用于分析和调试模型。

总而言之，PyTorch 的.pt 文件格式是一种功能强大的模型保存格式，它可以保存模型的架构、参数、计算图等完整信息，并且具有可移植性、灵活性、安全性和易于调试等特点，为模型的部署、共享和调整提供了很大的便利性。同时，通过结合 PyTorch 提供的各种工具和技术，可以进一步提高模型的性能和效率。

2.3.2 ONNX

ONNX (Open Neural Network Exchange) 是一种用于表示深度学习模型的开放式标准格式。它是由微软和 Facebook 等公司联合推出的，旨在提高深度学习模型的可移植性和互操作性。

ONNX 的设计初衷是让深度学习框架之间可以共享模型，并且可以在不同框架之间进行迁移。它采用了一种基于计算图的方式来表示模型，每个节点代表一个操作，如卷积、池化、激活函数等。节点之间的边表示数据流动的方向和依赖关系。这种基于计算图的方式，使得 ONNX 格式可以在多种深度学习框架之间共享模型，包括 TensorFlow、PyTorch、Caffe、CNTK 等。

ONNX 格式还支持多种硬件加速库，包括 Intel MKL-DNN、nGraph、TensorRT 等，可以在不同硬件平台上实现加速推理。同时，ONNX 格式也支持对模型进行优化和转换，以减小模型的大小、提高推理效率等。这包括剪枝、量化、融合等多种优化技术，使得模型可以更好地适应不同的硬件环境和应用场景。

总而言之，ONNX 格式是一个具有可移植性、互操作性、优化性和开放性的深度学习模型表示格式，它可以在不同的框架和平台之间共享模型，并可以实现多种硬件加速和优化技术。通过使用 ONNX 格式，用户可以更好地满足不同的应用需求和硬件环境，提高模型的效率和性能。

2.3.3 NCNN

NCNN (Ni Hao Caffe Neural Network) 是一个基于 C++ 编写的深度学习框架，专门针对移动设备和嵌入式设备进行优化。NCNN 格式是 NCNN 框架所支持的一种模型格式，NCNN 格式的模型文件包含了模型的结构和参数，以及模型所需的所有计算操作。NCNN 框架将模型文件转换为高效的 C++ 代码，并在移动设备和嵌入式设备上进行高效推理。

NCNN 格式的模型文件通常由两个文件组成：.param 和 .bin 文件。.param 文件包含了模型的结构信息，而 .bin 文件包含了模型的参数。

.param 文件是一个文本文件，其内容描述了模型的网络结构，包括输入输出尺寸、卷积核大小、卷积核数量、池化大小、激活函数类型等。这些信息可以帮助 NCNN 框架正确地构建模型，并在推理时对输入数据进行正确的处理。.param 文件的格式类似于 C++ 的代码，使用了 NCNN 框架提供的结构体和函数。

.bin 文件是一个二进制文件，其内容描述了模型的参数，包括卷积核权重、偏置、归一化参数等。这些参数是在训练模型时得到的，可以直接用于模型推理。.bin 文件的格式是按照数据类型和存储顺序进行排列的。

在使用 NCNN 框架进行模型推理时，首先需要读取.param 文件，解析出模型的结构信息，并根据这些信息构建模型。然后读取.bin 文件，将其中的参数加载到模型中，使得模型能够进行推理。因此，.param 文件和 .bin 文件是 NCNN 模型文件中不可缺少的两个部分，两者配合使用，才能构建出一个完整的 NCNN 模型。

总的来说，NCNN 格式是一个非常适合在移动设备和嵌入式设备上进行深度学习推理的模型格式。它具有高效性、轻量级、可移植性和灵活性等特点，可以适应不同的应用场景和需求。

2.3.4 模型格式转换

本文需要将 YOLOv8 网络模型从 PyTorch 格式转换成 ONNX 格式，再由 ONNX 格式转换成 NCNN 格式。一般来说，模型格式的转换包括计算图的转换和参数的转换两个方面。比如将模型从 PyTorch 格式转换成 ONNX 格式需要进行以下两步：

- 1) PyTorch 模型的计算图转换为 ONNX 计算图。在 PyTorch 中，每个模型都有一个计算图，描述了模型的结构和参数，包括计算图中的节点和边，节点表示计算操

作，边表示计算操作之间的数据流动。在转换为 ONNX 格式时，需要将 PyTorch 计算图转换为 ONNX 计算图。

2) PyTorch 模型的参数转换为 ONNX 参数。PyTorch 模型的参数通常以张量的形式存储，而 ONNX 模型的参数通常以二进制文件的形式存储。在转换过程中，需要将 PyTorch 模型的参数转换为 ONNX 参数，并将其存储为二进制文件。

最后，ONNX 模型转换为 NCNN 模型的基本原理也是将 ONNX 模型的计算图转换为 NCNN 模型的计算图，并将模型参数转换为 NCNN 模型的.param 和.bin 文件。

2.4 深度网络模型在 Android 设备上的部署框架

2.4.1 框架介绍

深度网络模型在 Android 设备上的部署框架有很多种，比如 PyTorch Mobile、TensorFlow Lite、NetEase Computer Neural Network 等。

PyTorch Mobile 是 PyTorch 框架的移动端版本。它允许将训练好的 PyTorch 模型导出到移动设备上，并在本地运行。PyTorch Mobile 提供了一个轻量级的 API，可用于在移动设备上加载和执行模型，同时保持高度的模块化和可扩展性。

TensorFlow Lite 是 Google 开发的 TensorFlow 框架的移动端版本。TFLite 支持在移动设备、嵌入式设备和 IoT 设备上部署深度学习模型。与 PyTorch Mobile 一样，TFLite 也允许将训练好的 TensorFlow 模型导出到移动设备上，并在本地运行。TFLite 还提供了一些工具，可以帮助用户压缩模型大小，提高模型性能，并优化模型以在特定硬件上运行。

而 NetEase Computer Neural Network(NCNN) 是一个由腾讯 AI 平台部开发的高性能深度学习框架，主要面向 ARM 架构的移动设备、嵌入式设备以及服务器端的高性能计算。与 PyTorch Mobile 和 TensorFlow Lite 相比，NCNN 的优点包括：

- 1) 高性能：NCNN 框架在 ARM 架构下进行了高度优化，能够利用 ARM 平台上的硬件加速器（如 NEON、Vulkan）以及多线程技术，实现高效的计算。
- 2) 轻量化：NCNN 框架采用了轻量级的设计思路，可以非常有效地压缩模型大小。NCNN 提供了一些工具，可以对模型进行裁剪、量化等操作，使得模型更加适用于嵌入式设备等资源受限的环境。
- 3) 跨平台支持：NCNN 框架支持多种操作系统，包括 Android、iOS、Linux 等，

同时还支持多种编程语言，包括 C++、Python、Java 等。

4) 易用性：NCNN 框架的 API 设计简单易用，可以快速地部署和运行深度学习模型。同时，NCNN 还提供了一些示例程序，方便用户学习和使用。

总体来说，NCNN 框架是一个高性能、轻量化、跨平台的深度学习框架，适用于移动设备、嵌入式设备以及服务器端的高性能计算。相比于 PyTorch Mobile 和 TensorFlow Lite，NCNN 在 ARM 架构下有着更高的性能和更小的模型大小，同时也更加灵活和易于使用。

由于 YOLOv8 模型需要高性能的硬件支持，而 NCNN 在移动设备和嵌入式设备上的性能比 PyTorch Mobile 和 TensorFlow Lite 更优秀。因此，使用 NCNN 来部署 YOLOv8 模型可以提供更快的推理速度和更高的精度。同时，NCNN 的灵活性比 PyTorch Mobile 和 TensorFlow Lite 更高。NCNN 支持多种模型和算法，并且可以自由定制网络结构和推理流程，可以更加精细地控制模型的大小和性能。此外，NCNN 提供了图形用户界面和 C++ 接口，可以更方便地进行模型部署和优化。因此，本文选择使用 NCNN 框架来在 Android 端部署 YOLOv8 模型。

2.4.2 部署流程

NCNN 框架的部署流程通常包括以下步骤：

- 1) 安装依赖：在部署 NCNN 之前，需要先安装相关的依赖库和工具链。具体的依赖库和工具链可能因为不同的操作系统和硬件平台而有所不同。
- 2) 编译 NCNN：在安装完依赖库和工具链后，需要编译 NCNN 框架。NCNN 提供了基于 CMake 的编译脚本，可以根据用户的需求自定义编译选项，例如选择使用哪些硬件加速器或者关闭不必要的模块等。
- 3) 导入模型：在编译好 NCNN 框架之后，需要将训练好的深度学习模型转换为 NCNN 格式并导入到 NCNN 中。NCNN 支持多种深度学习框架的模型转换，包括 Caffe、ONNX、TensorFlow 等。
- 4) 部署模型：一旦将模型导入到 NCNN 中，就可以在目标设备上部署模型。在移动设备和嵌入式设备上，可以使用 NCNN 提供的 C++ API 调用模型，实现高效的预测和推理。

2.5 目标检测模型的性能指标

2.5.1 IOU

在目标检测中，IOU (Intersection over Union, 交并比) 是一种用来评估模型检测结果和真实标签之间重叠程度的指标。该指标通常用来评估两个边界框之间的重叠程度，以确定它们是否表示同一个物体。

IOU 的计算方法是：将两个边界框的交集面积除以它们的并集面积。假设第一个边界框的面积为 A，第二个边界框的面积为 B，它们的交集面积为 C，则 IOU 可以表示为：

$$IOU = \frac{C}{A + B - C} \quad (2.1)$$

IOU 的取值范围为 0 到 1 之间，其中 0 表示两个边界框没有重叠，1 表示两个边界框完全重合。一般来说，如果两个边界框的 IOU 值大于一个预先设定的阈值，则可以将它们视为同一个物体。

在目标检测任务中，模型的预测结果和真实标签之间的 IOU 通常被用来评估模型的性能，并用来计算精确度、召回率等指标。同时，IOU 还被用来确定非极大值抑制的阈值，以去除重复的检测结果。

2.5.2 精度指标 mAP

在目标检测任务中，模型需要在图像中定位和识别出多个不同类别的物体。而 mAP(Mean Average Precision, 平均精度均值) 就是用来衡量模型在这个任务中的整体性能。通过计算每个类别的 AP 并取平均值，mAP 可以反映模型对不同类别的检测效果，并且考虑到了检测框的数量和位置等因素，是一种比较客观的评价方式。因此，在目标检测领域，mAP 是一个广泛使用的指标，它被用于评估算法的性能，并且被用作比较不同算法之间的效果。它的计算方法可以分为以下几个步骤：

首先，对于每个类别，模型需要根据预测结果和真实标注来计算 AP 值。具体而言，模型会对每个预测框计算一个置信度得分，然后按照置信度得分的高低对预测框进行排序。接下来，模型会从置信度得分最高的预测框开始，计算精度和召回率，其中精度指的是检测结果中正确预测的目标数量占检测结果总数的比例。例如，如果一

个类别的检测结果中有 5 个预测正确的目标，而总共有 10 个检测结果，那么该类别的精度就是 0.5。召回率指的是在所有真实目标中被正确检测到的目标数量占所有真实目标的比例。例如，如果一个类别有 10 个真实目标，算法检测到了其中 8 个，那么该类别的召回率就是 0.8，可以根据这些计算得到一个精度-召回曲线。

接下来，需要根据精度-召回曲线计算每个类别的平均精度 (AP)。计算 AP 的过程可以通过对精度-召回曲线进行积分得到。但是，在实际计算中，通常采用一种简化的方式，即对召回率进行插值，并将插值点的精度取最大值。例如，可以在召回率为 0、0.1、0.2、...、1 这 11 个点上计算插值精度，并将这些精度取最大值作为该点的 AP 值。然后，对这些 AP 值进行平均，即可得到该类别的平均精度 (AP)。

最后对所有类别的 AP 值求平均：对于多个类别的目标检测任务，需要将所有类别的 AP 值进行平均得到 mAP 值。具体而言，将所有类别的 AP 值相加，然后除以类别总数得到 mAP 值。

总而言之，mAP 是目标检测任务中常用的精度指标之一，它通过计算所有类别的 AP 值求平均来综合反映模型在多个类别上的检测和识别性能，可以用于评估模型的整体精度，mAP 值越高说明模型的整体精度越高。

2.5.3 速度指标 FPS

目标检测中的 FPS (Frames Per Second) 指标是衡量目标检测算法速度的重要指标。它表示算法每秒钟可以处理的图像帧数，即在单位时间内算法可以处理的图像数量。通常，越高的 FPS 值表示算法的处理速度越快。在实际应用中，fps 是一个非常关键的指标，因为它直接决定了算法能够处理多快的数据流。如果算法的处理速度不能满足实际需求，那么它就无法应用于实际场景中。因此，在实际应用中，除了考虑算法的精度，还需要考虑算法的速度和效率，以便在满足精度要求的前提下，提高算法的处理速度和效率，从而实现实际应用的需求。

在目标检测中，FPS 与许多因素有关。其中，一些主要因素包括以下几个方面：

- 1) 硬件设备：计算机硬件设备的性能是影响 FPS 的重要因素。比如，处理器、显卡、内存等硬件的质量和性能直接影响算法的运行速度和效率。
- 2) 检测器的复杂度：不同的目标检测算法和模型的复杂度不同，这也会对 FPS 产生影响。比如，一些高效的目标检测算法可以在保证一定准确率的情况下，实现较高的 FPS 值。

3) 图像大小: 处理大尺寸的图像需要更多的计算资源和时间, 因此会降低算法的 FPS。相反, 处理小尺寸的图像可以提高算法的 FPS。

总而言之, 在目标检测中, FPS 也是一个重要的指标。因此, 在选择目标检测模型时, 除了考虑精度和召回率等指标外, 还需要综合考虑模型的 FPS 表现。

3 边缘设备上目标检测系统的设计与测试

3.1 总体实验流程

总体实验流程如图 3.1 所示。

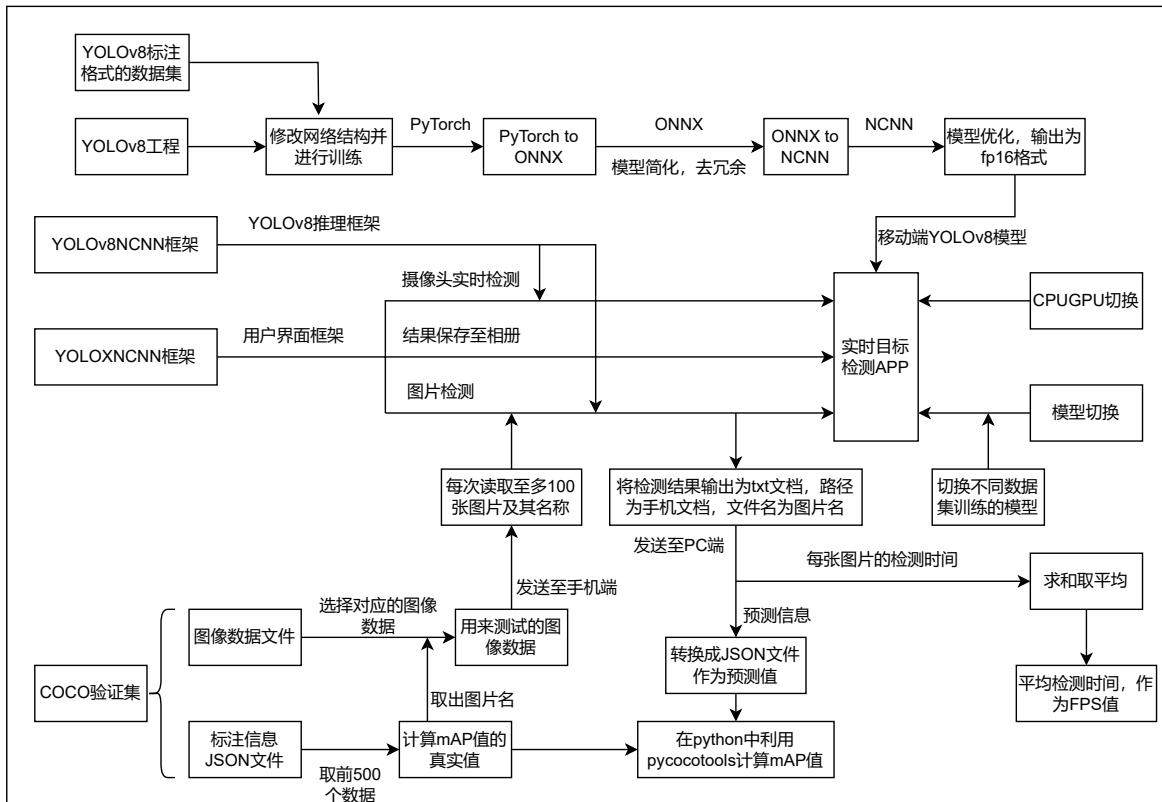


图 3.1 总体实验流程

整个实验的流程可以分为三部分：

- 1) 下载 YOLOv8 工程和 YOLOv8 标注格式的数据集，然后修改 YOLOv8 的部分网络结构之后利用数据集对模型进行训练，将训练好的 YOLOv8 模型从 PyTorch 格式转化为 ONNX 格式，然后进行模型简化，将简化后的模型从 ONNX 格式转换为可以在 Android 上部署的 NCNN 格式，最后进行模型优化，输出为半精度浮点数格式，得到本文最终在 Android 端部署的 YOLOv8 模型。
- 2) 将 YOLOv8NCNN 框架的 YOLOv8 推理部分和 YOLOXNCNN 框架的用户界面和功能实现部分相结合，利用上述的 YOLOv8 模型实现 YOLOv8 在 Android 端的目标实时检测、图片检测以及将图片检测结果保存至相册的功能，其中目标实时检测

会自动计算 FPS 值；同时该 APP 也支持 CPU 和 GPU 的切换，最后还对模型切换部分进行了适当修改，让不同数据集训练的 YOLOv8 模型可以同时部署到 Android 端，可以通过用户界面自由的切换模型进行检测。

3) 下载 COCO 数据集，利用验证集的 JSON 文件取出 500 张图片的信息以及标注信息 Ground Truth，利用图片信息中的图片名称到图片数据文件夹中取出对应的 500 张图片，然后发送至手机端作为本文模型性能测试的图片数据；修改图片检测功能，使其在读取图片时保存图片名称，同时支持图片多选功能，每次至多选择 100 张图片。在检测完成后将检测结果以文本文档的格式输出到手机端的文档路径，文件名与图片名对应，同时将每张图片的检测时间写到手机文档路径下的 run_time.txt 文档。最后将 500 张图片全部检测完成后将检测结果和检测时间发送至 PC 端并将检测结果转换成 JSON 文件，与上述的 Ground Truth 真实值一起在 python 中利用 pycocotools 工具计算模型部署到 Android 端的 mAP 值，然后利用保存检测时间的文档可以计算数据集中图片的平均检测时间，并将其作为 FPS 值。

3.2 移动端的 YOLOv8 模型

该部分主要介绍本文如何得到一个可部署到移动端的 YOLOv8 模型，在进行后续的操作前需要先下载 YOLOv8 工程并安装好所需的依赖库，同时 YOLOv8 的使用需要 Python 版本不低于 3.7 以及 PyTorch 版本不低于 1.7。

3.2.1 YOLOv8 模型结构修改

由于 YOLOv8 网络模型在后续的模型转化过程中，并不是所有的层都能够被直接转换为 NCNN 中的层。有些层可能需要使用自定义层来实现，或者需要进行额外的转换和优化才能在 NCNN 上运行。所以需要对 YOLOv8 的网络进行一些修改来保证后续部署到 NCNN 框架上能够正常运行。

1) change C2f split to slice

YOLOv8 在处理网络中的数据时，使用了 split 操作将数据分成几个部分进行处理，而 split 操作在 NCNN 中没有直接对应的实现方式，所以需要使用另一个与 split 作用相似的操作 slice。

split 操作是将一个 tensor 沿着指定的维度分割成若干个子 tensor，每个子 tensor 具有相同的 shape。它通常用于在训练神经网络时将数据 batch 分成多个部分，以便于

在 GPU 上并行处理。split 操作的输出是一个元素个数等于指定数量的 tensor 列表。

slice 操作是从一个 tensor 中按照指定的索引范围截取一个子 tensor。与 split 不同的是，slice 操作可以在一个 tensor 中选择不同的子区域，而且每个子 tensor 的 shape 也可以不同。这通常用于在网络的某些层中进行特定形式的数据处理，比如图像的裁剪或变形。slice 操作的输出是一个 tensor。

而 slice 操作可以用 ncnn::Crop 层实现，可以在不产生额外内存开销的情况下进行张量的分割，这与使用 Reshape 和 Crop 两个层来实现 split 操作相比，更加高效。因此，在后续的模型转换过程中，使用 slice 操作替换 split 操作，可以提高模型的性能和效率。

2) change class Detect output

YOLOv8 的 Detect 层负责将网络的特征图转换为检测框的位置和置信度信息，但是在 ONNX 转换为 NCNN 的过程中，并不会自动转换 Detect 层，需要使用 NCNN 的自定义层功能，通过编写自定义层来实现 Detect 层的功能，所以在进行模型转换前还需要修改 Detect 层的输出。

同时，为了模型能够正常进行训练和验证，可以定义一个 export 属性，默認為 False，在模型从 PyTorch 格式转换成 ONNX 格式前令 export 等于 True。当 export 等于 False 时正常使用 Detect 层来将网络的特征图转换为检测框的位置和置信度信息；而当 export 等于 True 时，则使用修改后的 Detect 层来进行模型格式转换。

3.2.2 YOLOv8 模型训练

1) 数据准备：下载可用于 YOLOv8 模型训练的数据集，包括图像和标注信息。由于本文主要研究深度网络模型在 Android 端部署的技术路线，对数据集或者说检测目标没有特别的要求，因此本文主要使用的数据集就是比较常用的 COCO 数据集。

2) 配置文件：YOLOv8 的训练需要一个配置文件 yaml，这个配置文件描述了模型的架构和训练参数，包括学习率、优化器、批大小等，这些参数的选择对训练过程和模型性能都有很大的影响。

3) 模型选择：YOLOv8 提供了几种不同的模型大小(YOLOv8n、YOLOv8s、YOLOv8m、YOLOv8l、YOLOv8x)，其中 N/S 和 L/X 两组模型只是改了缩放系数，较小的模型具有较少的卷积层和参数，较大的模型则具有更多的卷积层和参数，但是 S/M/L 等骨干网络的通道数设置不一样，没有遵循同一套缩放系数，如此设计的原因应该是同

一套缩放系数下的通道设置不是最优设计。由于本文是需要将 YOLOv8 网络部署到 Android 端，因此选择了较小的三种模型 YOLOv8n、YOLOv8s 以及 YOLOv8m。

4) 模型训练和评估：在完成上述步骤后可以直接开始 YOLOv8 网络的训练，训练完成后还需要对模型进行评估，评估过程主要是通过计算模型在验证集上的 mAP 指标来完成，最终可以得到一个 PyTorch 文件，这个就是训练好的 YOLOv8 模型，可以直接使用这个模型进行目标检测，但是 Pytorch 格式的文件无法部署到 Android 端，还需要在后续进行格式上的转换。

3.2.3 模型格式转换

为了将 YOLOv8 部署到 Android 端，需要将 YOLOv8 模型转换成 NCNN 格式，但是模型无法直接从 PyTorch 格式转换成 NCNN 格式，所以需要先将 YOLOv8 模型从 PyTorch 格式转换成 ONNX 格式，然后利用 ONNXsim 工具对导出的 ONNX 模型进行简化去冗余，再将简化后的 ONNX 模型转换成 NCNN 格式，最后对 NCNN 格式的模型进行优化并导出为 fp16 格式。具体过程如下：

1) PyTorch 转换成 ONNX

YOLOv8 工程中提供了 PyTorch 格式导出为 ONNX 格式的工具，按照前面的方法修改完 YOLOv8 的网络结构之后，利用 YOLOv8 工程中的 `export` 工具可以直接将训练好的模型直接导出 ONNX 格式并输出到同一路径下。

2) 模型简化

安装 ONNXsim，通过使用 ONNXSim 工具对模型进行简化，以便减少模型的复杂性、大小和计算需求，这通常包括去除一些冗余或不必要的操作，从而使模型更加高效和易于部署，最终帮助加速模型的推理速度、减少内存使用、降低功耗等，从而使模型更加适用于实际应用场景。

3) ONNX 转换成 NCNN

本文的该部分是在 Ubuntu 上面完成，在进行 ONNX 格式转 NCNN 格式之前需要先编译 onnx2ncnn 工具。首先，需要从官网下载 NCNN，然后将下载的 NCNN 解压到本地目录。同时，onnx2ncnn 工具的编译需要用到 Protobuf 和 CMake，所以需要先安装这两个依赖。安装完成后进入到 ONNX2NCNN 的根目录来编译 ONNX2NCNN 工具，编译完成后，会生成一个名为 onnx2ncnn 的可执行文件。利用 onnx2ncnn 工具可以直接将 ONNX 格式的模型转换成 NCNN 格式 (.param 和 .bin) 文件。

4) 模型优化

通常情况下，深度学习模型中的参数和激活值是使用 32 位浮点数（即 fp32）进行存储和计算的，而 16 位浮点数（即 fp16）则只使用一半的存储空间，可以显著减少模型在内存和带宽上的开销。虽然将精度降低到 fp16 会导致一定的数值误差，但对于目标检测这类对数值精度要求不高的应用场景，使用 fp16 可以在不影响模型性能的情况下显著提高推理速度和降低能耗。

同时，在完成上述的编译后，除了会生成 onnx2ncnn 工具之外，还会生成一些其他的 NCNN 工具，比如 NCNN 模型优化工具 ncnoptimize 和 NCNN 模型量化工具 quantize。其中 ncnoptimize 可以进行模型压缩、精度缩减和结构优化等操作，以提高模型的性能和部署效率。利用该工具对 NCNN 模型进行优化并使用半精度浮点数 fp16 来进行模型参数储存，得到本文最终部署到 Android 端的 YOLOv8 模型。

3.3 模型在 Android 设备上的部署

3.3.1 模型推理

将 YOLOv8NCNN 框架的模型推理部分与 YOLOXNCNN 框架的用户界面部分结合之后对 YOLOv8 模型进行推理，最终将 YOLOv8 模型部署到 Android 端，并同时实现摄像头实时检测、图片检测以及图片检测结果保存功能。

具体来说，YOLOv8NCNN 和 YOLOXNCNN 这两个框架的推理部分都是基于 C++ 实现的，只是它们的算法实现方式以及在一些细节上有一些不同，包括但不限于以下方面：

- 1) 网络结构：YOLOv8NCNN 和 YOLOXNCNN 的网络结构不同，YOLOv8NCNN 采用了改进的 CSPDarknet 作为主干网络，YOLOXNCNN 则使用了 PP-YOLOv2。
- 2) 特征图的生成和处理：YOLOv8NCNN 和 YOLOXNCNN 的特征图的生成和处理方法也有所不同，包括特征图的尺寸、通道数、缩放比例等等。
- 3) 前后处理：在目标检测的推理过程中，YOLOv8NCNN 和 YOLOXNCNN 也有不同的前后处理方式，例如 NMS 的实现方式、anchor 的生成方式等。

同时，YOLOv8NCNN 和 YOLOXNCNN 这两个框架的用户界面及功能都是基于 Java 实现的，但是 YOLOv8NCNN 框架只实现了摄像头实时检测，而 YOLOXNCNN 框架还实现了图片检测和结果保存。因此，通过对比 YOLOv8NCNN 和 YOLOXNCNN

的实现细节，并将 YOLOv8NCNN 的推理部分的代码实现替换到 YOLOXNCNN 框架中。替换完成后，对后处理的置信度阈值和 NMS 的 IOU 阈值进行调整，然后进行充分的测试和评估，最终设置置信度阈值为 0.3，NMS 的 IOU 阈值为 0.4。

3.3.2 部署多种模型

YOLOv8NCNN 框架本身支持部署多个 YOLOv8 模型，包括 YOLOv8n、YOLOv8s 以及 YOLOv8m 等等，但是部署的这些模型检测目标的类别数量和种类名称都必须是相同的，因为在 NCNN 框架中，目标检测的类别数量是一个常量，而种类名称就是一个长度为类别数量的一维数组，保存了每个类别的名称。

因此，若想同时部署检测目标类别数量或者种类名称不同的 YOLOv8 模型，则只需要将上述的类别数量定义为一个一维数组常量，对应的保存着每个模型检测的类别数量；同时将种类名称定义为一个二维数组，对应的保存着每个模型检测的种类名称。最后修改相关的类和相关的函数，使得程序在运行时会根据当前选择的模型来进行目标检测以及显示对应的检测结果。

3.4 Android 设备上的模型测试

将在 PC 端训练好的 YOLOv8 模型部署到 Android 端前，进行了模型格式转换、模型简化、模型优化等操作，同时由于 Android 设备资源限制，因此需要测试 YOLOv8 模型在 Android 设备上的性能表现。

3.4.1 精度测试

精度测试即为测试 YOLOv8 模型在 Android 端部署后进行目标检测的 mAP 值。由于在 Android 端没有直接测试目标检测模型的 mAP 值的方法，需要在 Android 端对数据集中的图片一张张进行测试并将结果保存，然后将所有的检测结果发送至 PC 端，与数据集中的 GroundTruth 一起利用 pycocotools 工具来计算该模型部署到 Android 端的 mAP 值。测试 mAP 值的整体流程比较繁琐，因此本文只选取了 COCO 验证集十分之一的数据（500 张图片）来进行检测和计算 mAP 值。具体步骤如下：

- 1) 选取 COCO 验证集中保存着标注信息的 JSON 文件的前 500 份数据作为计算 mAP 值的真实值，同时取出相应的图片信息。

2) 利用上述图片信息中的图片名称在 COCO 验证集的 5000 张图片中筛选出对应的 500 张图片作为测试图片，然后发送至手机端等待检测。

3) 修改图片检测功能，使其在读取图片时会读取图片名称，同时启用图片多选功能，每次至多选择 100 张图片进行检测，但只会在所有图片检测完成后显示最后一次的检测结果，然后在检测过程中添加记录每次检测结果的功能，即每张图片检测完成后会将其检测结果以 txt 文档的格式写到手机文档目录下的路径，具体来说是将每张图片检测到的物体标签、置信度以及预测框的位置(左上角 x 坐标、左上角 y 坐标、宽度 w、高度 h)信息写入以图片名称命名的 txt 文档，同时将每张图片的检测时间写入 run_time.txt 文档，单位是 ms，最后在将 500 张图片全部检测完成后将检测结果发送至 PC 端来进行 mAP 值的计算。

4) 将保存着检测结果的 500 个 txt 文档转化为计算 mAP 值所需的 JSON 文件作为预测值，然后利用上述保存真实值的 JSON 文件和保存预测值的 JSON，在 pycharm 上面调用 pycocotools 库来进行 mAP 值的计算。

3.4.2 速度测试

对于 Android 设备上 YOLOv8 模型的速度测试分为三种方式：单张图片检测时间、多张图片检测的平均时间以及摄像头实时检测的 FPS 值。

具体来说，对于单张图片的检测比较简单，检测一张图片，然后可以直接观察这张图片的检测时间，非常的直观；对于多张图片的检测就是利用上述精度测试中保存的 run_time.txt 文档来计算检测这 500 张图片的平均时间，也可以用 1000 除以平均时间并将其作为每秒平均处理的图片的数量 FPS 值；最后就是摄像头实时检测，其在检测时除了会显示检测到的物体的名称、置信度和预测框，还会显示实时检测的 FPS 值，因此也可以非常直观的观察到模型的检测速度。同时，上述所有的步骤均可以通过切换不同模型、CPU 和 GPU 来进行对比和分析。

4 测试与验证

以下测试使用的手机型号是 iQOO 10，处理器为 3.2GHz 第一代骁龙 8+ 八核，运行内存为 12GB，手机存储为 256GB。

4.1 不同模型的 mAP 值和 FPS 值测试及对比

本文这部分测试了 YOLOv8n、YOLOv8s 和 YOLOv8m 部署到 Android 端分别使用 CPU 和 GPU 的 mAP 值和 FPS 值，具体结果如下表 4.1 所示。其中测试数据使用的是 COCO 2017 验证集标注文件的前 500 个数据，每次测试结果中平均检测时间和 FPS 均有一定变化，下表中是测试两次并取平均的结果，而 mAP 值在每次测试中均保持不变。表 4.2 是 YOLOv8 作者提供的各种 YOLOv8 模型在 PC 端的表现。

表 4.1 不同模型的 mAP 和 FPS 对比

| | 平均检测时间/ms | FPS | mAP(50-95) |
|-------------|-----------|-------|------------|
| YOLOv8n/CPU | 24.59 | 40.66 | 20.06% |
| YOLOv8s/CPU | 38.09 | 26.25 | 28.07% |
| YOLOv8m/CPU | 65.69 | 15.22 | 35.10% |
| YOLOv8n/GPU | 42.34 | 23.62 | 19.64% |
| YOLOv8s/GPU | 51.27 | 19.50 | 28.78% |
| YOLOv8m/GPU | 61.67 | 16.22 | 35.21% |

表 4.2 YOLOv8 模型在 PC 端的表现

| 模型 | Speed(CPU ONNX)/ms | Speed(A100 TensorRT)/ms* | mAP(50-95)** |
|---------|--------------------|--------------------------|--------------|
| YOLOv8n | 80.4 | 0.99 | 37.3% |
| YOLOv8s | 128.4 | 1.20 | 44.9% |
| YOLOv8m | 234.7 | 1.83 | 50.2% |
| YOLOv8l | 375.2 | 2.39 | 52.9% |
| YOLOv8x | 479.1 | 3.53 | 53.9% |

*Speed averaged over COCO val images using an Amazon EC2 P4d instance.

**mAPval values are for single-model single-scale on COCO val2017 dataset.

由测试结果可知，模型部署到 Android 端后其 mAP 值相较于 PC 端均有不少下降，这也在意料之中，因为在模型转换的过程对模型进行了一些简化操作，使模型提升在 Android 端的推理速度的同时也会降低模型一定的精度。同时，通过对比三种模型分别使用 CPU 和 GPU 的表现来看，虽然 GPU 通常比 CPU 具有更高的并行处理能力，但是在 YOLOv8n 模型在使用 GPU 时不仅 mAP 值降低了 0.42%，其速度甚至

降低了将近一半，其中可能的原因是 YOLOv8 模型比较简单，GPU 的并行计算能力可能无法得到充分利用，从而导致性能下降；与之相反的是 YOLOv8m 模型，其使用 GPU 相较于使用 CPU 时，其 mAP 值提升了 0.11%，FPS 也提升了 1；而介于这两种模型之中的 YOLOv8s 模型在使用 GPU 时 mAP 值提升了 0.71%，但是 FPS 值降低了 6.75。最后对这三种模型的性能进行对比，YOLOv8n 模型最小，其检测速度最快，但是精度最低；而 YOLOv8m 模型最大，其检测速度最低，但是精度最高；YOLOv8s 模型介于两者之中，结果比较合理。

除此之外，本文还寻找了两种其他部署在 Android 设备上的目标检测模型的性能表现来进行对比，第一个是 TensorFlow Lite 官方提供的目标检测模型，其测试结果以及说明如下表 4.3 所示。

表 4.3 每个 EfficientDet-Lite 模型的性能相互比较

| 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% |

* Size of the integer quantized models.

** Latency measured on Pixel 4 using 4 threads on CPU.

*** Average Precision is the mAP (mean Average Precision) on the COCO 2017 validation dataset.

第二个是 Github 上一个开源的项目，作者将 YOLOv5s 部署到 Android 端并基于 Xiaomi Mi11 进行了测试，同时他们还通过使用 NNAPI (qti-dsp) 来加速推理过程，并将计算卸载到 Hexagon DSP，其测试结果如下表 4.4 所示。

表 4.4 yolov5s 在 android 上的性能

| delegate | Latency(ms) | device, model, delegate | Accuracy(mAP) |
|-----------------------|-------------|-----------------------------------|---------------|
| None (CPU,fp32) | 249 | host GPU (Tflite + PyTorch, fp32) | 27.8% |
| NNAPI (qti-gpu, fp32) | 156 | host CPU (Tflite + PyTorch, int8) | 26.6% |
| NNAPI (qti-gpu, fp16) | 92 | NNAPI (qti-gpu, fp16) | 28.5% |
| None (CPU,int8) | 95 | CPU (int8) | 27.2% |

通过对比 YOLOv8 模型和上述两种模型在 Android 的速度和 mAP 值来看，TensorFlow Lite 官方提供的目标检测模型中最大的两种模型的 mAP 要高于 YOLOv8n/s/m，但是其检测速度甚至要比 YOLOv8m 慢不少，由于它们测试使用的设备是 Pixel 4 并

且使用了 4 线程的 CPU，因此对于其模型与 YOLOv8 模型的检测速度的对比并不是非常清晰。而 YOLOv5s 模型在 Android 端的表现相较于 YOLOv8 模型来说，不管是在检测速度上还是检测精度上均要落后一点。

总体上来说，YOLOv8 模型不管是在 PC 端还是部署到 Android 端均有着非常不错的性能。

4.2 摄像头实时检测

YOLOv8 模型部署到 Android 端分别使用 CPU 和 GPU 进行实时目标检测的效果如下图 4.1 和图 4.2 所示。



图 4.1 YOLOv8 模型使用 CPU 实时检测结果对比

可以看出，越大的 YOLOv8 模型可以检测出更多的物体，而且其检测的位置框的整体置信度也要更高，但是越小的 YOLOv8 模型进行实时检测的 FPS 值更高。同时，在使用 GPU 进行实时检测时，相较于使用 CPU，YOLOv8n 模型和 YOLOv8s 模型检测的 FPS 值均有所下降，而 YOLOv8m 模型进行实时检测的 FPS 有所提升，这与上述 mAP 值和 FPS 值测试中得到的结论一致。但是本文测试的所有模型使用 CPU 和 GPU 时实时目标检测的 FPS 值均要低于使用数据集图片测试计算得到的 FPS 值，这可能是由于以下几个原因导致的：

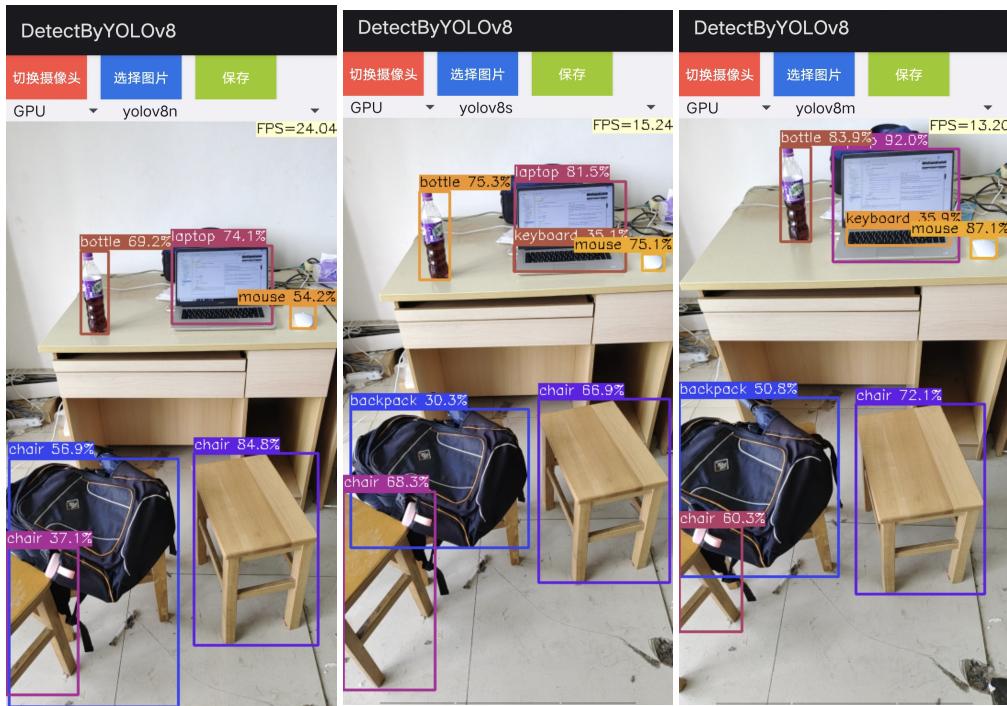


图 4.2 YOLOv8 模型使用 GPU 实时检测结果对比

- 1) 摄像头的输入是一个连续的视频流，而非静态图片。这意味着每个时间步骤（通常是 1/30 或 1/60 秒）都需要检测一个新的图像，这需要比对一张静态图片进行检测更多的计算。
- 2) 摄像头的输入通常会有更高的分辨率，即使将输入的分辨率降低到与静态图片相同，因为它需要连续进行处理，所以它需要更多的计算资源。
- 3) 摄像头实时检测的输出需要进行绘图操作，而上述图片检测没有计算绘制预测框的时间，因此这也会在一定程度上使得实时检测的 FPS 值更低，尤其是在检测到过多的物体的情况下。

综上所述，即使使用相同的模型，通过对静态图片进行检测计算出的 FPS 值和实时目标检测的 FPS 值可能会有所不同。因此，在进行模型性能评估时，应该考虑模型的输入数据类型和性质。

4.3 图片检测

YOLOv8 模型部署到 Android 端分别使用 CPU 和 GPU 进行图片检测的效果如下图 4.3 和图 4.4 所示，检测时间如下表 4.5 所示，其中对单张图片检测的时间是进行了三次测试之后取平均的结果。

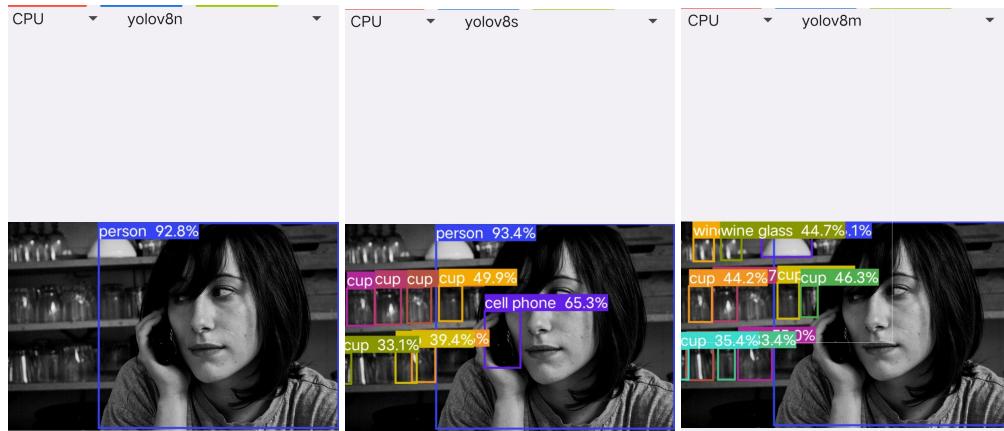


图 4.3 YOLOv8 模型使用 CPU 进行图片检测结果对比

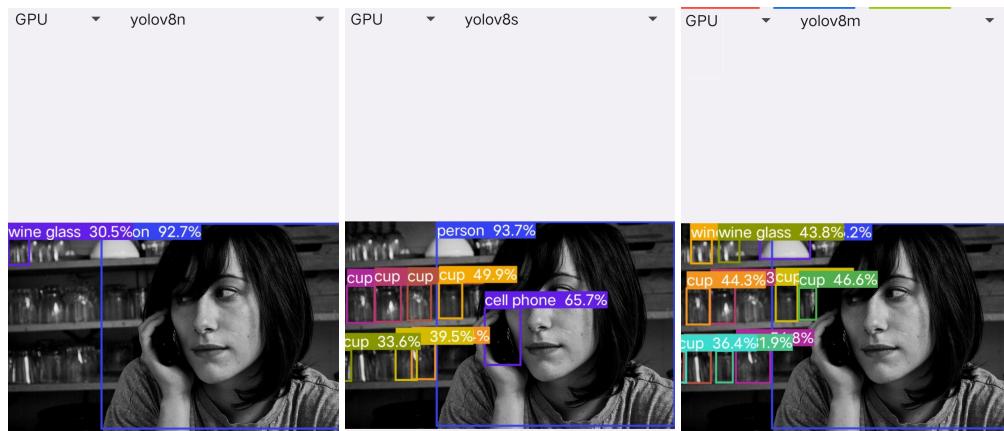


图 4.4 YOLOv8 模型使用 GPU 进行图片检测结果对比

表 4.5 不同模型检测单张图片的时间对比

| 单位/ms | YOLOv8n | YOLOv8s | YOLOv8m |
|-------|---------|---------|---------|
| CPU | 19.6194 | 39.4585 | 68.5239 |
| GPU | 37.0132 | 88.1489 | 132.392 |

从检查结果可以看出，越大的 YOLOv8 模型可以检测出更多的物体，但是检测的时间更慢，这与上述结论一致，但是在不同模型检测单张图片的时间上，YOLOv8n 模型使用 CPU 和 GPU 检测单张图片的时间均比检测整个数据集的平均时间要更低，

而 YOLOv8s 模型和 YOLOv8m 模型使用 CPU 和 GPU 检测单张图片的时间均比检测整个数据集的平均时间要更高，这可能是由于以下几个原因导致的：

1) 模型的复杂性和大小：对于较小的模型，例如最小的 YOLOv8n 模型，它的参数较少，计算量较小，因此对单张图片进行目标检测的检测时间可能比对 500 张图片进行检测的平均时间短。而对于较大的模型，例如 YOLOv8s 模型和 YOLOv8m 模型，它们的参数和计算量较多，因此对单张图片进行目标检测的检测时间可能比对 500 张图片进行检测的平均时间长。

2) GPU 和 CPU 的使用情况：对于使用 GPU 进行目标检测的情况，GPU 的使用情况也可能会影响检测时间。例如，当 GPU 在执行其他任务时，可能会降低对目标检测任务的优先级，导致检测时间变长。

3) 系统资源占用情况：对于 Android 系统而言，还有其他进程和任务也需要占用 CPU、GPU 等系统资源。在同时运行多个任务时，系统资源的分配可能会对单张图片目标检测的检测时间产生影响。

综上所述，对于单张图片目标检测的检测时间差异，可能受到多个因素的影响。因此，在进行模型性能评估时，为了得出准确的结论，最好不要使用单张图片检测的检测时间来进行对比。

4.4 多模型测试

除了使用 COCO 2017 数据集训练的 YOLOv8 模型以外，本文还测试了将另一个数据集 MaskDataSet 训练的 YOLOv8n 模型一起部署在 Android 端，该模型可以检测人们是否佩戴了口罩，其效果对比如下图 4.5 所示。

将多个数据集训练的模型同时部署的优点是可以提高系统的泛用性，通过训练不同模型来适应不同的场景。事实上，当应用场景对类别需求不是太大时，可以通过选择合适的数据集对所有的类别进行标注并训练成一个模型，例如 COCO 数据集标注了 80 个常用的物体。但是数据集标注的类别数目与模型的性能和准确度之间并非呈线性关系。在某些情况下，标注类别过多可能会导致模型过度拟合，从而降低模型的性能。此外，标注类别过多也意味着需要更多的人力和时间成本来完成标注工作。因此，当目标类别数量更多、目标之间的差异很大，或者存在不同的背景和环境条件，那么使用不同的模型来进行检测将会是一个更好的选择，这样可以更好地适应不同的场景和需求，同时还可以提高模型的准确性和性能。

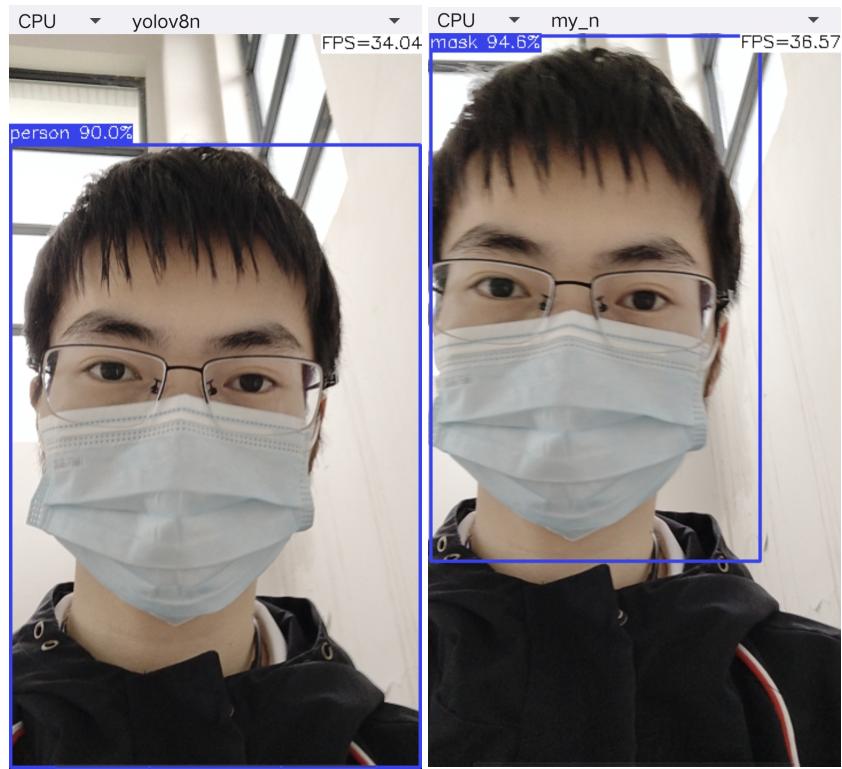


图 4.5 不同数据集训练的 YOLOv8n 模型检测结果对比

总而言之，本文提供了在 Android 系统上部署多种模型的方法，至于选择使用一个模型还是多个模型，以及类别数量的分配等问题，需要根据目标之间的差异和相似性，以及数据集的大小和特征等因素来进行权衡和取舍。

5 总结和展望

5.1 总结

本文主要研究了 YOLOv8 模型在 Android 端的部署以及性能测试。

首先，通过对目前主流的目标检测模型进行对比，选择了 YOLOv8 作为本文的目标检测模型，同时对 YOLOv8 网络结构中的 C2f 和 Detect 层进行了修改，然后利用修改后的模型来进行训练自己的目标检测模型；在 PC 端训练好 YOLOv8 模型后，将其从 Pytorch 格式转换为 ONNX 格式并对转换后的模型利用 ONNXsim 工具对其进行简化，去除冗余，然后利用 onnx2ncnn 工具将模型从 ONNX 格式转换成 NCNN 格式，最后利用 ncnnoptimize 工具将模型进行优化，并输出为 fp16 半精度浮点数格式。

结合 YOLOv8NCNN 和 YOLOXNCNN 两个框架将 YOLOv8 部署到 Android 端，实现了摄像头实时检测、图片检测以及检测结果保存功能。同时，通过修改置信度阈值为 0.3 以及 NMS 的 IOU 阈值为 0.4 使模型达到了一个比较好的检测效果。最后，通过修改部署框架，可以使多个不同数据集训练的 YOLOv8 模型同时部署在 Android 上，在 APP 的用户界面中通过下拉栏选择对应模型进行检测。

通过修改图片检测功能，使其在图片检测的选择图片时可以进行多选，至多选择 100 张图片，并将检测结果（类别、置信度和预测框坐标 x, y, w, h）以及检测时间输出到 Android 端文档路径下的 txt 文档，检测结果的文档名称为检测图片的名称，检测时间保存至 run_time.txt 文档中。同时在 COCO 数据集的验证集中选取 500 张图片进行测试，将 500 个保存着检测结果的 txt 文档和记录每次检测所需时间的 run_time.txt 文档传输到 PC 端，并将检测结果转化成 JSON 文件，在 Python 中利用 pycocotools 库进行 mAP 值的计算。除此之外，通过对检测时间取平均值，可以得到模型对数据集的平均检测时间，并将该结果作为 FPS 值。

最终可以得到本文的 YOLOv8n 模型在 Android 端使用 CPU 进行目标检测的 FPS 值为 40.66，mAP 值为 20.06%，使用 GPU 进行目标检测时的 FPS 值为 23.62，mAP 值为 19.64%；YOLOv8s 模型在 Android 端使用 CPU 进行目标检测的 FPS 值为 26.25，mAP 值为 28.07%，使用 GPU 进行目标检测时的 FPS 值为 19.50，mAP 值为 28.78%；YOLOv8m 模型在 Android 端使用 CPU 进行目标检测的 FPS 值为 15.22，mAP 值为 35.10%，使用 GPU 进行目标检测时的 FPS 值为 16.22，mAP 值为 35.21%。通过对比

TensorFlow 官方提供的模型以及 YOLOv5 模型在 Android 端的性能，YOLOv8 模型在速度和精度上均有着非常不错的表现。

5.2 展望

由于设备限制，本文只在本人的手机上进行了 YOLOv8 模型的性能测试，后续可以尝试使用不同性能的手机来进行测试，对比 YOLOv8 模型在不同性能的 Android 设备上的表现。

同时，将 YOLOv8 模型部署到 Android 设备前可以进行一些轻量化操作，比如使用 PyTorch 中的 `prune` 方法对模型进行剪枝操作，或者利用 NCNN 中的 `quantize` 工具对模型进行量化。由于 YOLOv8 模型的网络结构比较复杂，对网络各部分的剪枝权值不容易确定，无法保证剪枝后模型的性能，对模型的量化也是一样，需要对轻量化后的模型进行反复的性能测试，而由于时间的限制，这部分只能留待后续进行。

最后，在今年 4 月 17 日惊闻发布了 DETRs^[24]，其在目标检测上的检测精度和检测速度比 YOLO 系列的所有算法都要优秀。但由于此时本文写作已经开始，而且目前留给作者研究的时间和精力有限，因此没有深入研究 DETRs 在 Android 端部署的方法以及测试其在 Android 端的性能，留待后续进一步的探索。

参考文献

- [1] Redmon, Joseph, and Ali Farhadi."YOLO9000: better, faster, stronger." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.
- [2] Zhang, Kaipeng, et al."Joint face detection and alignment using multitask cascaded convolutional networks." IEEE Signal Processing Letters 23.10 (2016): 1499-1503.
- [3] Ren, Shaoqing, et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." IEEE Transactions on Pattern Analysis and Machine Intelligence 39.6 (2017): 1137-1149.
- [4] Dai, Yun, et al. "Automatic detection of diabetic retinopathy and age-related macular degeneration in digital fundus images." Investigative Ophthalmology & Visual Science 59.9 (2018): 4128-4138.
- [5] 陈健健. 面向边缘计算环境的轻量级目标检测技术研究 [D]. 上海交通大学,2019.DOI:10.27307/d.cnki.gsjtu.2019.004109.
- [6] Girshick R, Donahue J, Darrell T, et al.Richfeature hierarchies for accurate object detection and semantic segmentation[C]//2014 IEEE Conference on Computer Vision and Pattern Recognition, June 23-28, 2014, Columbus, OH, USA. New York:IEEE Press, 2014: 580-587.
- [7] Girshick R. Fast R-CNN[C]//2015 IEEE International Conference on Computer Vision (ICCV), December 7-13, 2015, Santiago, Chile. New York: IEEE Press,2015: 1440-1448.
- [8] Ren S Q, He K M, Girshick R, et al.Faster R-CNN: towards real-time object detection with region proposal networks[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017, 39(6):1137-1149.
- [9] He K M, Gkioxari G, Dollár P, et al.Mask R-CNN [C]//2017 IEEE International Conference on Computer Vision (ICCV), October 22-29, 2017, Venice, Italy. New York: IEEE Press, 2017: 2980-2988.

- [10] Redmon J, Divvala S, Girshick R, et al. You only look once: unified, real-time object detection[C]// 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 27-30, 2016, Las Vegas, NV, USA. New York: IEEE Press, 2016: 779-788.
- [11] Redmon J, Farhadi A. YOLO9000: better, faster, stronger[C]// 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), July 21-26, 2017, Honolulu, HI, USA. New York: IEEE Press, 2017: 6517-6525.
- [12] Redmon J, Farhadi A. YOLOv3: an incremental improvement[EB/OL]. (2018-04-08 [2021-02-01]).<https://arxiv.org/abs/1804.02767>.
- [13] Liu W, Anguelov D, Erhan D, et al. SSD: single shot MultiBox detector[M]// Leibe B, Matas J, Sebe N, et al. Computer vision-ECCV 2016. Lecture notes in computer science. Cham: Springer, 2016, 9905: 21-37.
- [14] Howard A G, Zhu M, Chen B, et al. MobileNets: Efficient convolutional neural networks for mobile vision applications[J]. arXiv preprint arXiv:1704.04861, 2017.
- [15] Zhang X, Zhou X, Lin M, et al. ShuffleNet: An extremely efficient convolutional neural network for mobile devices[J]. Proceedings of the IEEE conference on computer vision and pattern recognition, 2018: 6848-6856.
- [16] Han S, Mao H, Dally W J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding[J]. arXiv preprint arXiv:1510.00149, 2015.
- [17] Courbariaux M, Bengio Y, David J P. Binarynet: Training deep neural networks with weights and activations constrained to + 1 or -1[J]. arXiv preprint arXiv:1602.02830, 2016.
- [18] Hinton G, Vinyals O, Dean J. Distilling the knowledge in a neural network[J]. arXiv preprint arXiv:1503.02531, 2015.
- [19] Tencent. ncnn: a high-performance neural network inference framework optimized for mobile and other platforms. <https://github.com/Tencent/ncnn>.

- [20] Howard A G, Zhu M, Chen B, et al. MobileNets: Efficient convolutional neural networks for mobile vision applications[J]. arXiv preprint arXiv:1704.04861, 2017.
- [21] Zhang X, Zhou X, Lin M, et al. ShuffleNet: An extremely efficient convolutional neural network for mobile devices[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018: 6848-6856.
- [22] Redmon J, Divvala S, Girshick R, et al. You only look once: Unified, real-time object detection[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016: 779-788.
- [23] Lin T Y, Goyal P, Girshick R, et al. Focal loss for dense object detection[C]//Proceedings of the IEEE international conference on computer vision. 2017: 2980-2988.
- [24] Lv W, Xu S, Zhao Y, et al. Detrs beat yolos on real-time object detection[J]. arXiv preprint arXiv:2304.08069, 2023.

致谢

大学求学生涯即将告一段落，在此向给我帮助的老师、家人和同学致以最真挚的谢意。

首先，我要感谢我的导师梅天灿老师。在论文选题时，老师鼓励我选择自己感兴趣的课题，并在研究过程中给予了很多帮助和建议。在我遇到问题和困难时，老师总是悉心指导和引导我，让我能够克服困难并取得进展。此外，老师还在论文写作和修改方面给予了我很多有益的意见和建议。在此衷心感谢梅老师对我的教导与帮助。

其次，我要感谢我的家人。在整个大学求学期间一直给予我无私的支持和鼓励。你们的理解和支持让我能够专心学习和研究，感谢你们一直以来的陪伴和支持。

同时，我要感谢我的同学们和朋友们。你们在生活和学习上的支持和帮助是我求学生涯中的重要力量。感谢你们一直以来的支持和陪伴。

最后，我要感谢我的学校和所有教育工作者。感谢您们为我们提供了良好的学习环境和条件，感谢您们为我们传授了宝贵的知识和经验，感谢您们为我们打开了未来的大门。感谢教务处和各位老师的辛勤付出和关心，感谢您们让我有机会接受高质量的教育和研究。

再次感谢所有支持和帮助过我的人，我将倍加珍惜所得到的一切，继续不断努力，为更好的未来奋斗。