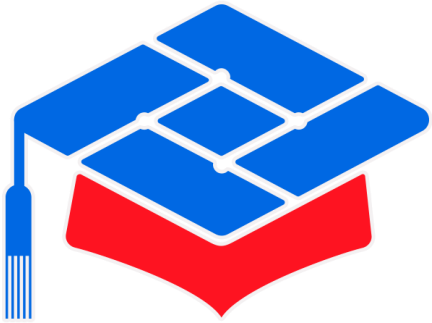


第七届

全国大学生集成电路创新创业大赛

报告类型*：设计报告

参赛杯赛*：飞腾杯

作品名称*：基于飞腾 ft2000/4 实现自动驾驶的目标检测

队伍编号*：CICC5989

团队名称*：headforest 保护小组

目录

摘要

本文针对自动驾驶目标检测中低光照条件下的识别精度不足的问题，采用了一种低光照图像增强算法，通过引入这一算法，从图像预处理环节入手，优化输入图像质量，提高了 YOLOv8 目标检测算法在低光照环境下的精度和鲁棒性。通过利用低光照图像增强算法和 YOLOv8 目标检测算法的结合这一创新的解决方案，可以有效地解决传统的目标检测算法在低光照条件下面临目标不清晰、细节丢失等问题，以提高自动驾驶系统的性能。

首先，我们参考与复现所采用低光照图像增强算法中的原理技术，通过自适应的对比度增强和亮度调整技术，有效提升低光照图像的可见度和细节信息。

接下来，将增强后的低光照图像输入 YOLOv8 目标检测算法进行目标检测。借助 YOLOv8 强大的检测能力和实时性，我们能够在低光照条件下准确、快速地检测出道路、车辆、行人等关键目标，为自动驾驶系统提供更可靠的感知能力。

最后，我们在 FT-2000/4 这块开发板上测验该复合系统的检测速度，通过对稳定视频源的处理，我们成功对低光照度下的视频进行了增强操作并准确地做了目标检测。

综上所述，本文的研究为自动驾驶系统在低光照条件下的目标检测问题提供了一种创新的解决方案。通过利用低光照图像增强算法和 YOLOv8 目标检测算法的结合，我们能够实现更高精度的自动驾驶目标检测，为实现安全、高效的自动驾驶技术迈出了重要的一步。

1. 系统工作原理与关键技术原理分析

1.1 基本概念

1.1.1 低光照度图像增强算法简介

低光照度处理算法是一种用于改善低光条件下图像质量的算法。它通过增强图像亮度、对比度和细节，以提高图像的清晰度和可视性。常见的处理方法包括图像增强、噪声降低、多帧融合和使用深度学习网络。我们采用“Toward Fast, Flexible, and Robust Low-Light Image Enhancement”[1]进行实验。该算法提出了一个轻量级而有效的框架：自校准照明 (SCI)，用于针对不同的现实世界场景进行低光照图像增强来表现 SOTA 模型，性能明显优于 LIME: Low-Light Image Enhancement via Illumination Map Estimation[2]算法，故采用此算法作为本实验的增强算法。

1.1.2 目标检测算法 YOLOv8 简介

YOLO(You Only Look Once)是一种基于深度学习的目标检测算法，它能够快速而准确地检测图像或视频中的多个对象。与传统的目标检测算法不同，YOLO 采用单阶段(one-stage)的检测方法，将对象检测问题转化为一个回归问题。它通过将输入图像划分为网格，并在每个网格单元中预测对象的边界框和类别概率。这意味着 YOLO 一次性处理整个图像，而不需要在多个阶段中进行区域提取和分类，既简便又高效。YOLO 的主要特点包括：

1. 快速：YOLO 能够实时地处理图像或视频流，因为它在单个前向传播中完成了所有的检测操作，避免了多次的区域提取和分类过程。
2. 精准：YOLO 利用深度卷积神经网络（CNN）来提取图像特征，并通过回归框位置和类别概率来准确地预测对象。
3. 多尺度特征融合：YOLO 使用不同尺度的特征图来检测不同大小的对象，并通过特征融合来提高检测的精度。
4. 多类别检测：YOLO 可以同时检测多个类别的对象，每个对象都有相应的类别概率。

而本文采用的 YOLOv8 是 Ultralytics 的最新版本 YOLO。作为目前最先进

的尖端 （SOTA） 模型，YOLOv8 建立在以前版本成功的基础上，引入了新功能和改进，与以往的一众 YOLO 模型（如 YOLOv7）做对比，明显增强了性能、灵活性和效率。该算法支持支持全方位的视觉 AI 任务，包括检测、分割、姿势估计、跟踪和分类。这也是我们小组选择该算法进行目标检测的主要原因。然后该算法论文尚未发表，仅公开技术文档，许多模块尚不完备，这对我们这群初学者来说也是不小的挑战。

1.2 处理流程

1. 输入低光照图像：将低光照图像作为算法的输入。
2. 低光照图像增强处理：对低光照图像进行增强处理，以提升图像的亮度、对比度和清晰度等。可以使用先前提到的光照度校正、对比度增强、噪声降低等技术。
3. YOLOv8 目标检测：使用 YOLOv8 算法对经过增强处理的图像进行目标检测。该算法可以实现实时目标检测，并能够同时检测多个对象类别。
4. 目标框绘制：对于检测到的目标，根据其边界框的位置和类别信息，在图像上绘制出相应的边界框和类别标签。
5. 结果输出：将处理后的图像（带有绘制的目标框）作为输出，可以保存、显示或进一步应用于其他任务。

1.3 数学建模

1.3.1 低光照增强算法

"Toward Fast, Flexible, and Robust Low-Light Image Enhancement"开发了一种新的自校准照明 (SCI) 学习框架，用于在现实世界的弱光场景中实现快速、灵活和稳健的增亮图像。

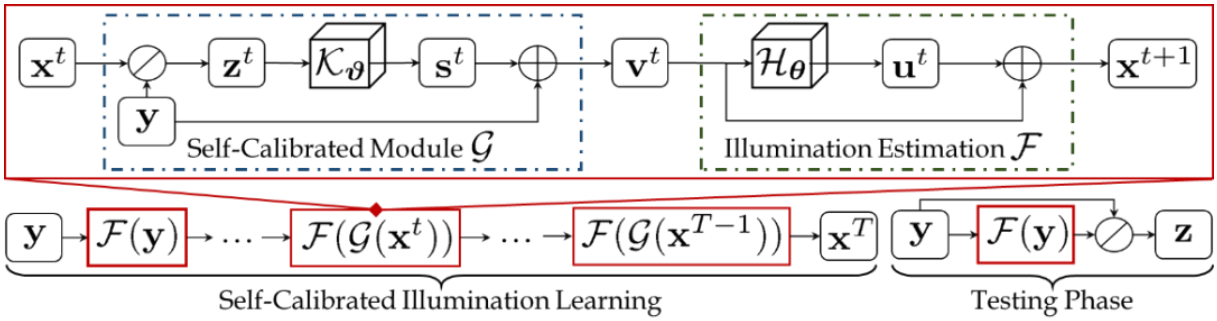


图 1 SCI 的整体框架

如图 1 所示，在训练阶段，SCI 框架是由光照估计和自校准模块组成。将自校准模块图添加到原始低光输入，作为下一阶段的照度估计的输入。注意，这两个模块分别是整个训练过程中的共享参数。在测试阶段，只利用一个单一的照明估计模块。

具体来说，通过建立了一个具有权重共享的级联光照学习过程来处理这个任务。构建了自校准模块以实现每个阶段结果之间的收敛，产生仅使用单个基本块进行推理的增益，这将大大降低计算成本。

接着，建立自校准模块。该模块的目的在于从分析每个阶段之间的关系入手，确保在训练过程中的不同阶段的输出均能够收敛到相同的状态。自校准模块的公式表达如下所示：

$$\mathcal{G}(\mathbf{x}^t): \begin{cases} \mathbf{z}^t = \mathbf{y} \oslash \mathbf{x}^t, \\ \mathbf{s}^t = \mathcal{K}_{\boldsymbol{\vartheta}}(\mathbf{z}^t), \\ \mathbf{v}^t = \mathbf{y} + \mathbf{s}^t, \end{cases}$$

其中 $\mathbf{v}^t$ 是校准后的用于下一阶段的输入。也就是说，原本的光照学习过程中第二阶段及以后的输入变成了由上述公式得到的结果（总的计算流程如图 1 所示），即光照优化过程的基本单元被重新公式化为：

$$\mathcal{F}(\mathbf{x}^t) \rightarrow \mathcal{F}(\mathcal{G}(\mathbf{x}^t)).$$

实际上，该自校准模块通过引入物理规律（即 Retinex 理论），逐步校正了每一阶段的输入来间接地影响了每一阶段的输出，进而实现了阶段间的收敛。图 2 探究了自校准模块的作用，可以发现，自校准模块的引入使得不同阶段的结果能够很快地收敛到相同状态（即三个阶段的结果重合）。

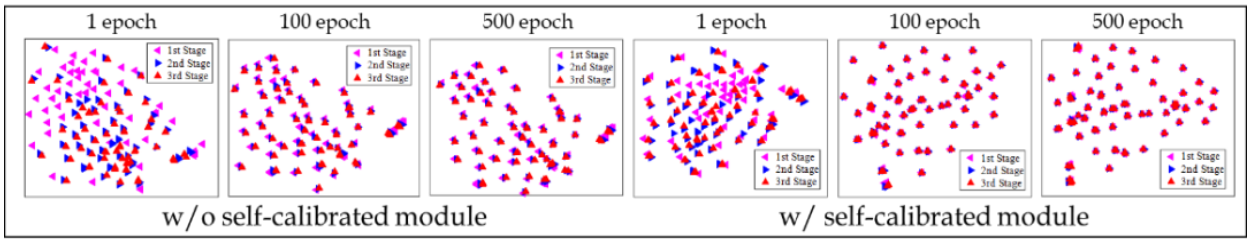


图 2 关于测试阶段是否采用自校准模块的增强结果 t-SNE 分布对比（阶段 3）

为了更好地训练提出的学习框架，该部分设计了一种无监督损失函数，以约束每一阶段的光照估计，公式表示如下：

$$\mathcal{L}_{total} = \alpha \sum_{t=1}^{T} \|\mathbf{x}^t - (\mathbf{y} + \mathbf{s}^{t-1})\|^2 + \beta \sum_{i=1}^{N} \sum_{j \in \mathcal{N}(i)} w_{i,j} |\mathbf{x}_i^t - \mathbf{x}_j^t|,$$

其中前一项与后一项分别代表数据保真项及平滑正则项（关于各个变量的详细说明请参见附件中的论文）。

1.3.2 YOLOv8 算法

YOLOv8 主要涉及到以下几个方面：backbone 使用 C2f 模块，检测头使用了 anchor-free + Decoupled-head，损失函数使用了分类 BCE、回归 CIOU + VFL（新增项目）的组合，框匹配策略由静态匹配改为了 Task-Aligned Assigner 匹配方式、最后 10 个 epoch 关闭 Mosaic 的操作、训练总 epoch 数从 300 提升到了 500。

整体的算法框架图和流程如图 3 所示：

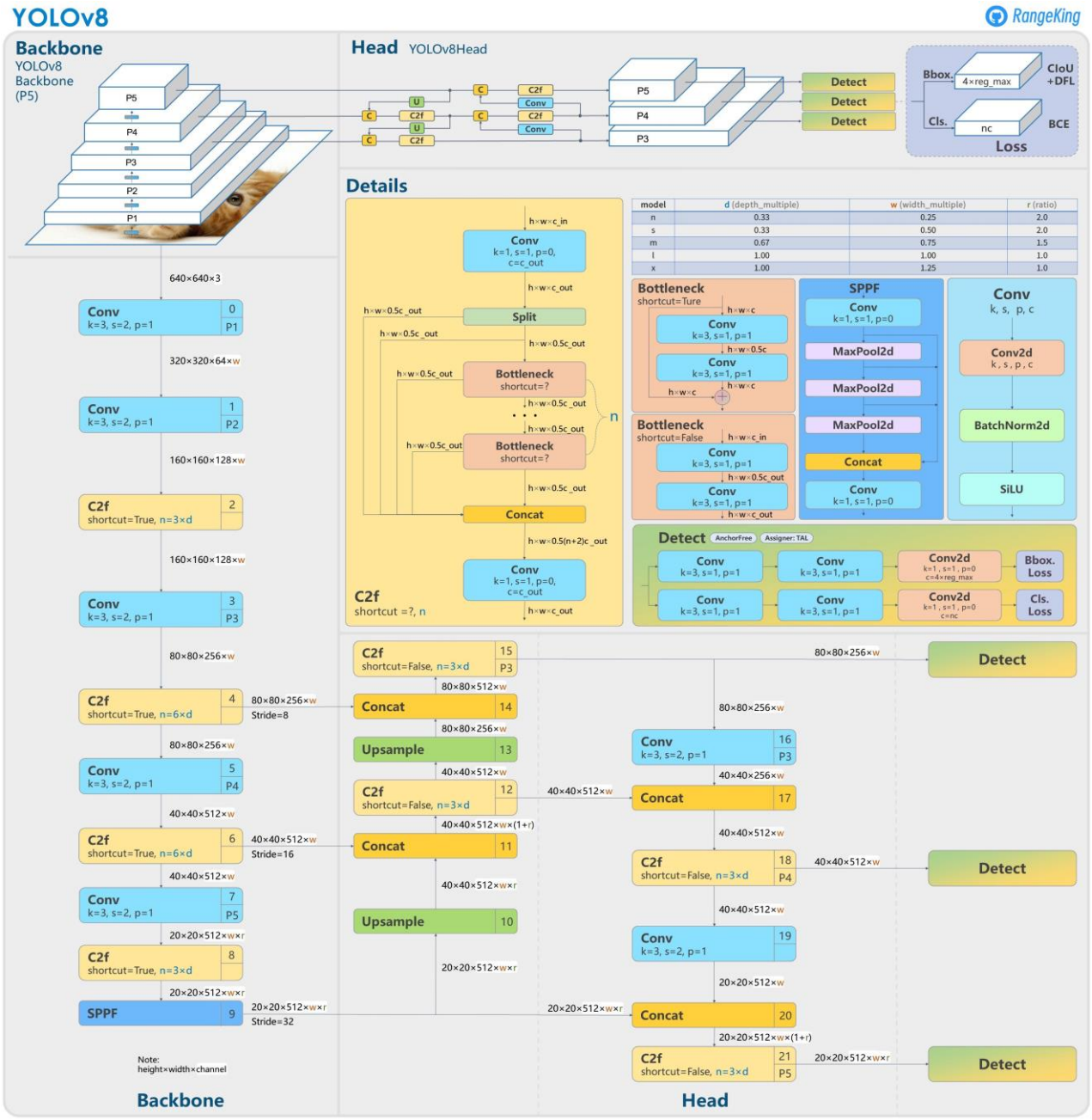


图 3　YOLOv8 结构模型设计[4]

具体细节：

1. 输入要求以及预处理：基础输入为 640*640，推理的预处理为 letterbox（根据参数配置可以为不同的缩放填充模式，主要用于 resize 到 640）+转换 rgb、chw、int8(0-255)->float(0-1)，注意没有归一化操作。
2. 主干网络：连续使用两个 3*3 卷积降低了 4 倍分辨率，
3. 检测头以及匹配机制：检测和分类的卷积是解耦的（decoupled），如图 4 所示，上面一条支路是框的卷积，框的特征图 channel 为 4*regmax，并

不是 anchor；分类的 channel 为类别数。

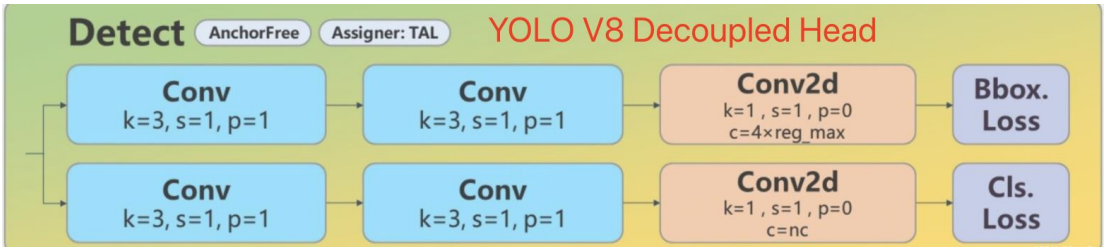


图 4　YOLOv8 检测头示意图

4. 损失函数设计

Loss 计算包括 2 个分支： 分类和回归分支，没有了之前的 objectness 分支。分类分支依然采用 BCE Loss。回归分支使用了 Distribution Focal Loss（DFL Reg_max 默认为 16）+ CIoU Loss。3 个 Loss 采用一定权重比例加权即可。

目前被广泛使用的 bbox 表示可以看作是对 bbox 方框坐标建模了单一的狄拉克分布。但是在复杂场景中，一些检测对象的边界并非十分明确。如下图左面所示，对于滑板左侧被水花模糊，引起对左边界的预测分布是任意而扁平的，对右边界的预测分布是明确而尖锐的。对于这个问题，有学者提出直接回归一个任意分布来建模边界框，使用 softmax 实现离散的回归，将狄拉克分布的积分形式推导到一般形式的积分形式来表示边界框。 狄拉克分布可以认为在一个点概率密度为无穷大，其他点概率密度为 0，如图 4 所示，这是一种极端地认为离散的标签时绝对正确的。

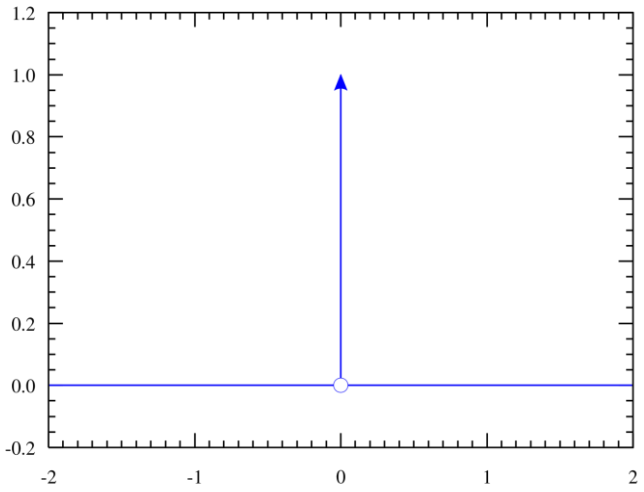


图 5 狄拉克分布函数图

因为标签是一个离散的点，如果把标签认为是绝对正确的目标，那么学习出的就是狄拉克分布，概率密度是一条尖锐的竖线。然而真实场景，物体边界并非是十分明确的，因此学习一个宽范围的分布更为合理。我们需要获得的分布虽然不再像狄拉克分布那么极端（只存在标签值），但也应该在标签值附近。因此学者提出 Distribution Focal Loss 损失函数，目的让网络快速聚焦到标签附近的数值，是标签处的概率密度尽量大。思想是使用交叉熵函数，来优化标签 y 附近左右两个位置的概率，是网络分布聚焦到标签值附近。如下公式。Si 是网络的 sigmod 输出，*yi* 和 *yi+1* 是上图的区间顺序，*y* 是 label 值。

$$DEL(S_i, S_{i+1}) = -((y_{i+1} - y)\log(S_i) + (y - y_i)\log(S_{i+1}))$$

具体而言，针对我们将 *DFL* 的超参数 *Reg_max* 设置为 16 的情况下：

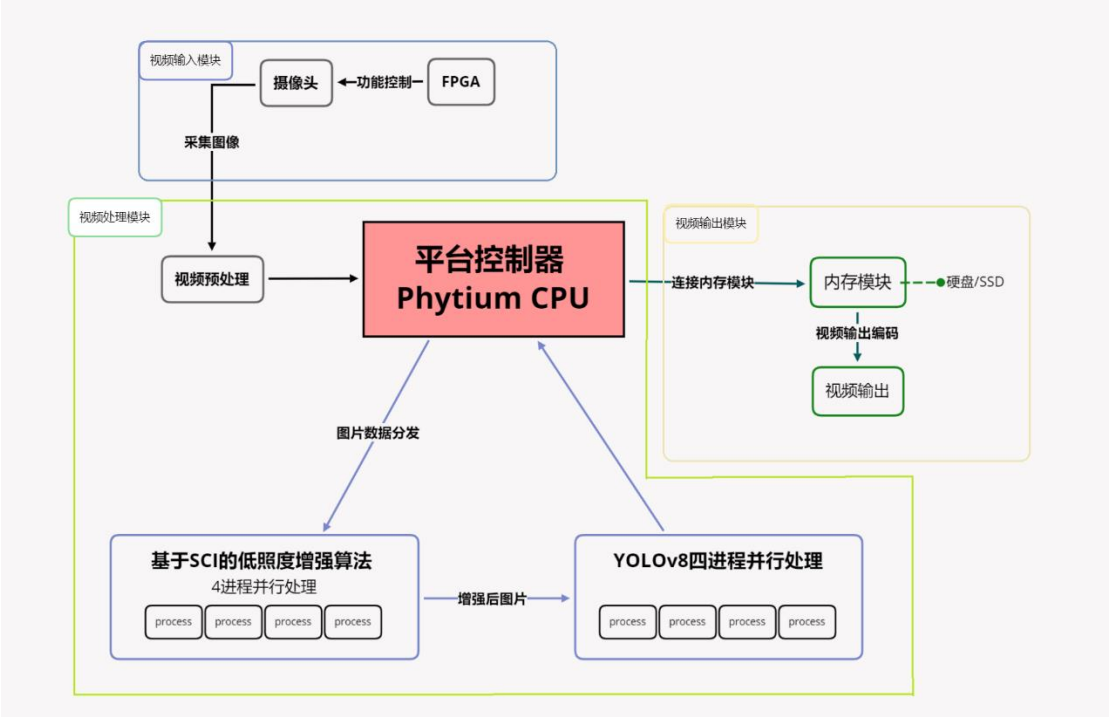
- 训练阶段：我们以回归 left 为例：目标的 label 转换为 *ltrb* 后，y = （ left - 匹配到的 anchor 中心点 x 坐标）/ 当前的下采样倍数，假设求得 3.2。那么 *i* 就应该为 3，$yi = 3$ ，$yi+1 = 4$。
- 推理阶段：因为没有 label，直接将 16 个格子进行积分（离散变量为求和，也就是期望）结果就是最终的坐标偏移量（再乘以下采样倍数+ 匹配到的 anchor 的对应坐标）。

总之，DEL 的实现方式就是一个卷积过程。

5. trics 训练：C2f 模块、Decoupled-Head、Anchor-Free、BCE Loss 作为分类损失 VFL Loss + CIOU Loss 作为回归损失、Task-Aligned Assigner 匹配方式、最后 10 个 epoch 关闭 Mosaic 的操作。
6. 进入测试阶段。

2.系统体系结构设计

2.1 系统体系结构和模块划分示意图



2.2 技术选择

2.2.1 硬件配置

飞腾教育开发板

1. CPU：FT2000/4，集成 4 个 FTC663 内核，兼容 64 位 ARMv8 指令集，集成 4MB 二级缓存， 4MB 三级缓存，主频 2.6GHz；
2. GPU：AMD Radeon R5 230 MXM，显存容量：1024MB，显存位宽：64bit，核心频率：625MHz，显存频率：1066MHz；
3. 内存：标配 8GB DDR4 笔记本内存条；
4. 储存：标配 64GB mSATA 硬盘；
5. 接口：6 个 USB3.0 接口，1 个 HDMI 接口，1 个 DP 接口、2 个千兆网口等；

摄像头

1. 分辨率：1280*720

2.2.2 系统组件

1. 操作系统：银河麒麟(kylin-Phytium-FT2000-4 4.4.131-20200710.kylin.desktop-generic)
2. Linux 内核：5.4.18-53-generic
3. 编译工具链：gcc 9.3.0

2.2.3 软件与环境配置

1. 编译环境：pyhton3.8 及以上版本，编译语言为 python
2. 编程 IDE：pycharm community
3. 环境配置：具体环境配置要求详见附件中 requirements.txt 和 README.txt

2.3 接口描述

2.3.1USB 3.0 接口

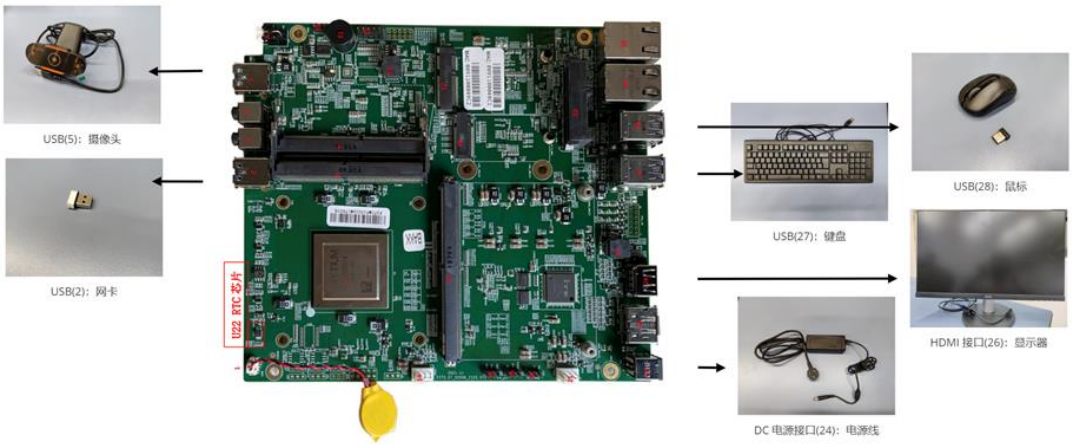
1. USB(2)：网卡；
2. USB(5)：摄像头；
3. USB(27)：鼠标；
4. USB(28)：键盘；

2.3.2HDMI 接口

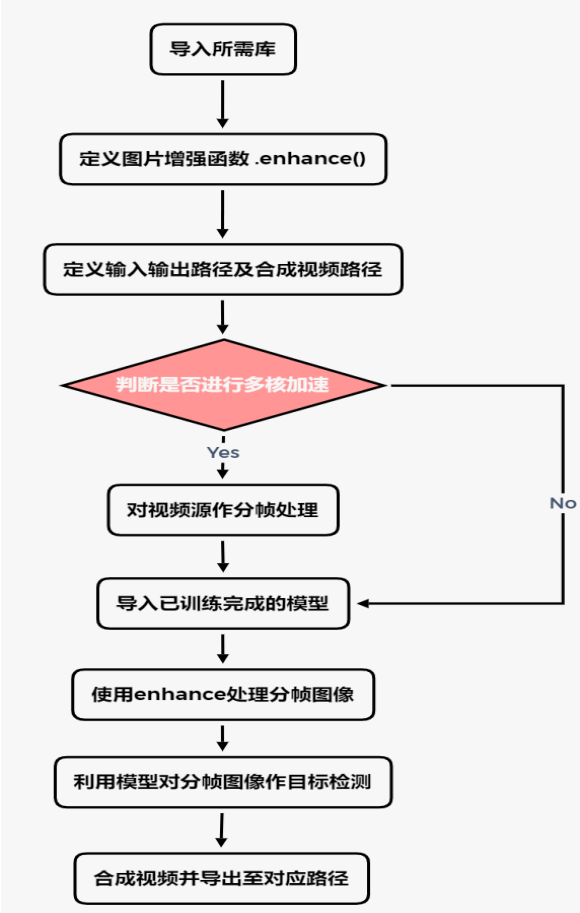
HDMI 接口：显示器

3.详细设计与实现

3.1 开发板连线图



3.2 软件流程图



3.3 关键代码分析

3.3.1 代码组成架构

图 6 代码组成架构示意图

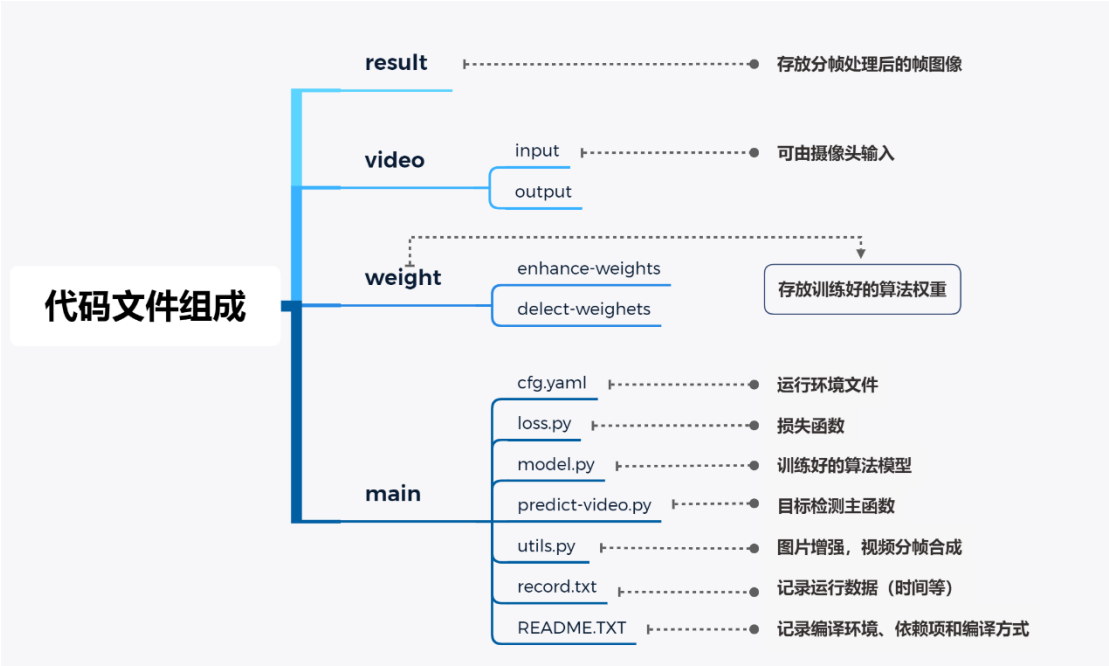


图 6 代码组成架构

如图 6 所示，代码主要分为 video、weigths 和 main 目录，video 路径下存放 input 和 output 目录，为视频输入输出路径；weigths 目录存放 enhance_weigths 和 detect_weigths 两个子目录，分别存放进行图像预处理算法和目标检测算法训练好的权重；main 目录为主模块，存放损失函数文件 loss.py、模型文件 model.py、目标检测算法实现文件 predict_video.py(main)、图片增强处理、提取帧合成视频文件 utils.py。

其中 utils.py 中的 video2framea()模块为视频做分针处理并将图片转为.jpg 格式（本实验支持任意的图片输入，此处仅以.jpg 格式为例），

predict_video.py 为算法实现文件，负责对分帧好的图形做光照增强，并实现目标检测。

predict_camera.py 为摄像头检测文件，负责获取稳定视频源文件。

utils.py 为视频分帧文件，负责对处理完的图像进行拼接并输出到指定 output 路径。

注意：代码运行完后会自行在当前目录下产生 result 目录用以存放每一帧处理

完的图像（.jpg）

3.3.2 主要功能代码分析

1. 主函数 predict_video.py 分析：

在主函数中首先，设置进程启动方式，并载入检测部分模型，通过设置进程开启数界定是否进行加速（进程开启数默认为 1，即表示未做加速；本实验采用进程开启数位 2，4，6 三种情况来表示多核加速）。

```
mp.set_start_method('spawn')     # 设置进程启动方式
dmodel = YOLO(cfg['d_model']) # 载入检测部分模型
process = cfg['process']              # 确定开启的进程数量
```

在视频的的分解帧操作之后，载入图片增强模型，并启动进程来进行图片增强。

```
model = Network(cfg['e_model']).to("cpu")# 载入 enhance 模型
video_path = cfg['video_path']     # 定义视频文件路径
out_frame_path = "video/output_frames/" # 定义输出图片的路径
load_path = cfg['load_path']         # 合成视频的路径
    output_folders, enhance_folders, detect_folders = create(process) # 将视频分解成帧
width, height, fps = video2frame(video_path, out_frame_path, output_folders)
processes = []                                # 存放进行的进程
for i_path, o_path, d_path in zip(output_folders, enhance_folders, detect_folders):
    d_path = d_path.split('/')[-2]
    p = mp.Process(target=run, args=(i_path, model, o_path, d_path, dmodel,))
    processes.append(p)
    p.start()   # 进程启动
for p in processes:
    p.join()   # 结束进程
# 从帧合成视频
frame2video(detect_folders, width, height, fps, load_path)
print("Tatol Cost Time:", time.time() - s_t)   # 记录运行时间
```

最后，通过将帧合并，获取完整的处理后视频。

```
# 从帧合成视频
frame2video(detect_folders, width, height, fps, load_path)
print("Tatol Cost Time:", time.time() - s_t)   # 记录运行时间
```

2. utils.py 分析：

在该文件中，主要实现视频分解功能，并逐帧保存为图片。以下为主要功能模块

```
# 逐帧读取视频并将帧保存为图片
```

```
for i in range(total_frames):
    ret, frame = cap.read()
    if ret:
        # 将帧保存为图片
        cv2.imwrite(out_frame_path + "frame{:04d}.jpg".format(i), frame)
    else:
        break
```

3. predict_camera.py 分析：

通过摄像头读取帧并做处理后保存到对应路径。

```
# 摄像头读取祯进程

def read_frame(frame_queue):
    cap = cv2.VideoCapture(0)
    while True:
        if frame_queue.qsize() > 4:
            frame_queue.get()
        else:
            pass
        frame_queue.put(cap.read()[1])
```

4. 系统测试与分析

4.1 执行时间测试与分析

执行时间记录在附录中的 record.txt 文件中，以下为测试时间结果：

分帧算法执行时间 anl Time：20.403006553649902s

整机执行时间

进程数：1 Total Cost Time: 1475.2765502929688s

进程数：2 Total Cost Time:1363.9979326725006s

进程数：4 Total Cost Time:1369.0619387626648s

根据测试结果可以得出以下结论：

1. 使用多进程加速处理可以显著降低整体执行时间。从进程数为 1 到进程数为 2 和 4 时，总执行时间都有所减少。
2. 进程数为 2 时，相较于进程数为 1，总执行时间减少了约 111 秒，表示多进程并行处理有明显的性能提升。

3. 进程数增加到 4 时，总执行时间与进程数为 2 时相比略有增加，这可能是由于资源竞争或调度开销导致的。

4.3 系统功能与其他性能测试与分析

4.3.1 结果展示与性能分析

图 7 源视频片段截取

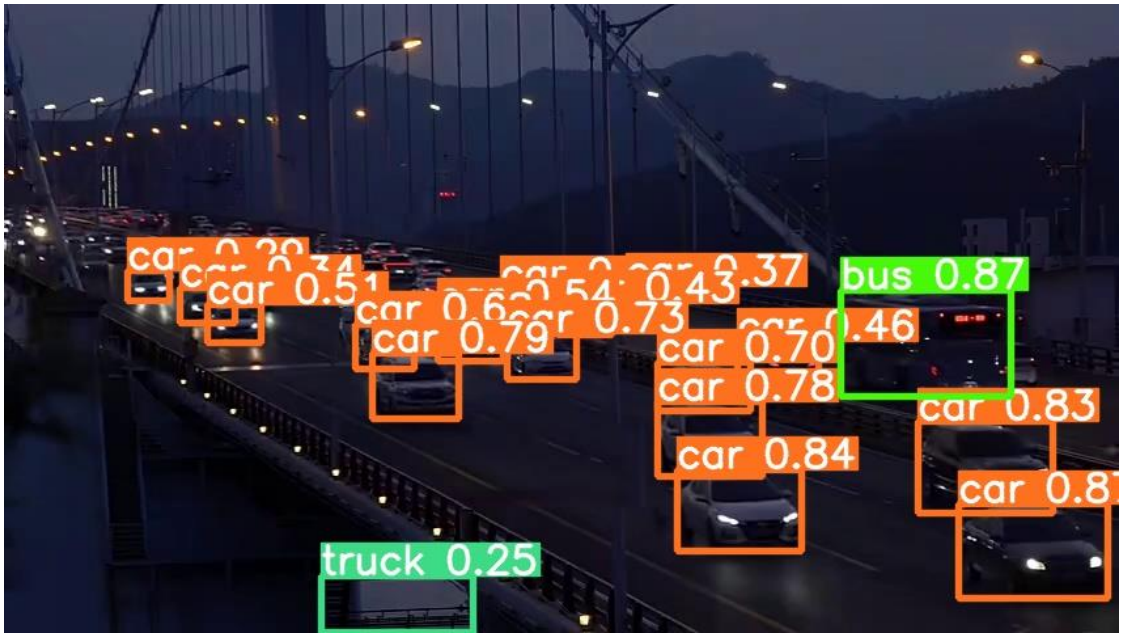


图 8 未增强图片的目标检测结果

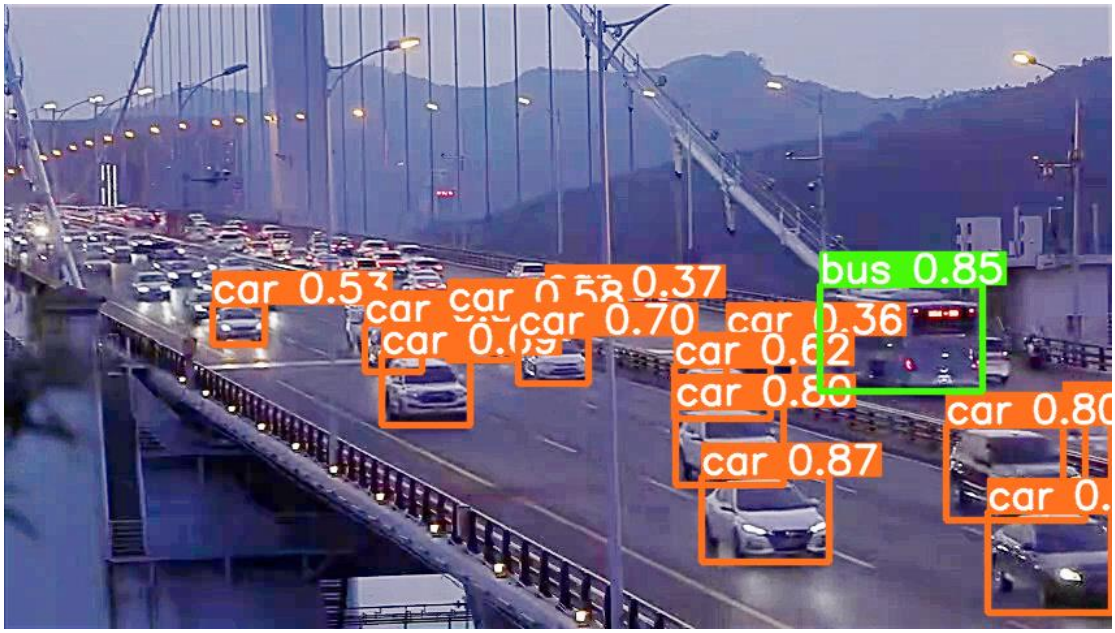


图 9 增强后图片的目标检测结果

以图 8 和图 9 为分析对象，对本文的封装算法进行性能分析。由图可知，图 8 正确检测（TP）数为 17 和误检(FP)数为 2， 漏检（FN）数为 2

可得以下数据

目标检测准确率 Percision：Percision1=TP/(TP+FP)=16/19=0.84

目标检测召回率 Recall：Recall1=TP / (TP + FN)=17/19=0.89

图 9 正确检测（TP）数为 14 和误检(FP)数为 1， 漏检（FN）数为 1

可得以下数据

目标检测准确率 Percision：Percision2=TP/(TP+FP)=14/15=0.93

目标检测召回率 Recall：Recall2=TP / (TP + FN)=14/15=0.93

很显然，经过图像增强处理后的图片的目标检测效果要优于原图

目标检测稳定性：通过对比图 7 图 9 的图像清晰度，可知经过增强处理后，输入图像的清晰度并未造成明显的损失。

视频质量：增强后的视频祥见附件中的代码目录中的 output 文件夹

5. 参考文献

[1] L. Ma, T. Ma, R. Liu, X. Fan and Z. Luo, "Toward Fast, Flexible, and Robust Low-Light Image Enhancement," 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, LA, USA, 2022, pp.

5627-5636, doi: 10.1109/CVPR52688.2022.00555.

[2] X. Guo, Y. Li and H. Ling, "LIME: Low-Light Image Enhancement via Illumination Map Estimation," in IEEE Transactions on Image Processing, vol. 26, no. 2, pp. 982-993, Feb. 2017, doi: 10.1109/TIP.2016.2639450.

[3] https://docs.ultralytics.com/

[4] mmyolo/README.md at dev ·Open-mmlab/mmyolo ·GitHub