

本系统先由已知的公开数据集进行模型训练，采用百度飞桨用户提供的公开数据集训练 YOLOv10 模型。然后将 YOLOv10 的.pt 存档权重文件转化为边缘设备通用模型.onnx 文件后再在华为 Atlas 开发板上通过 ATC 工具包转换为适配于 CANN 算子 torch 架构的.om 模型文件而来。然后基于已有的基于.pt 和.onnx 模型方法的检测程序并结合开发板特性转换开发而来。

首先是模型文件：

模型文件通过 YOLOv10s 预训练模型训练 200 轮而成：

```
#coding:utf-8
from ultralytics import YOLOv10
# 模型配置文件
model_yaml_path = "ultralytics/cfg/models/v10/yolov10s.yaml"
#数据集配置文件
data_yaml_path = 'D:/yolov10-main/datasets/insulator_detect/data.yaml'
#预训练模型
pre_model_name = 'D:/yolov10-main/yolov10s.pt'

if __name__ == '__main__':
    #加载预训练模型
    model = YOLOv10(model_yaml_path).load(pre_model_name)
    #训练模型
    results = model.train(data=data_yaml_path,
                           epochs=200,
                           batch=12,
                           name='insulator_detect')
```

图 1 训练函数

其中，训练集为我们寻找的二分类开源数据集：

```
1 train: train
2 val: valid
3 test: test
4
5 nc: 2
6 names: ['insulator_defect', 'insulator_normal']
```

图 2 数据集 yaml 文件

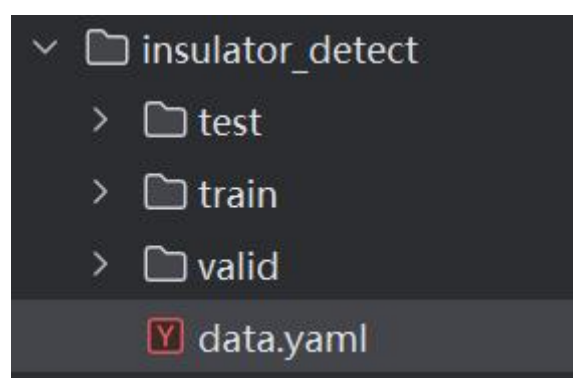
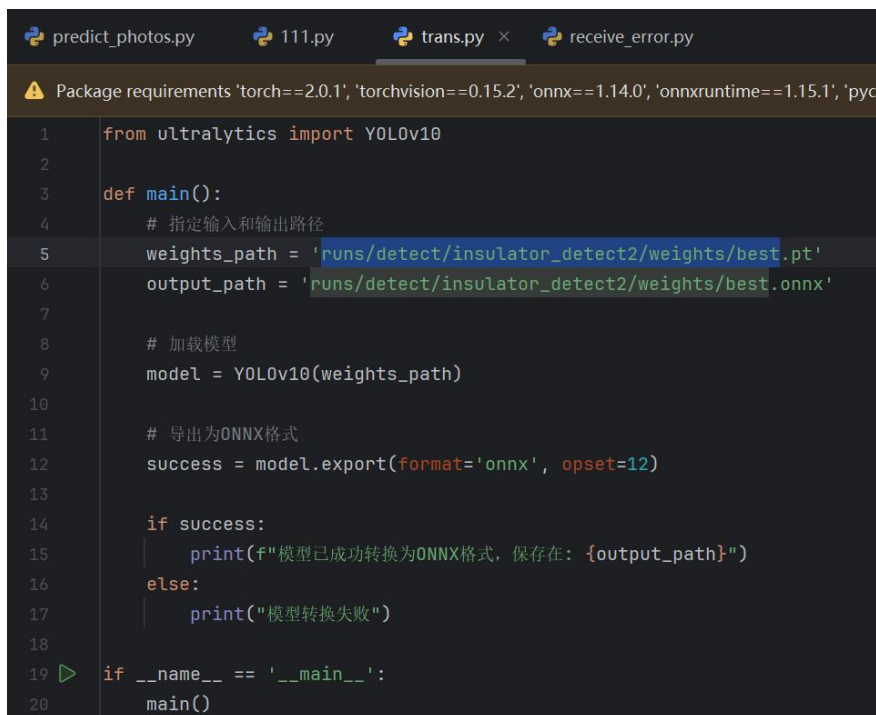


图 3 数据集目录结构

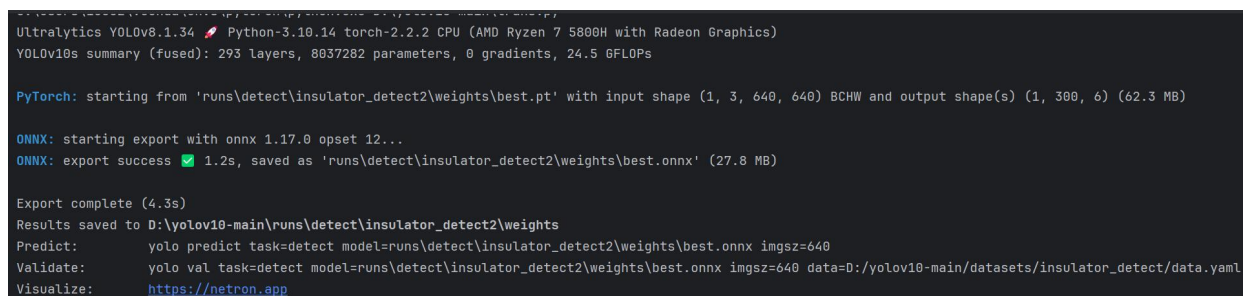
我们通过算法将我们的数据集分为了训练集、测试集、验证集三部分
训练完成之后，我们通过 YOLO 的工具包将其转化为 .onnx 文件从而让其变成 Atlas 开发板所能转换的边缘设备格式：



```
predict_photos.py 111.py trans.py × receive_error.py
⚠ Package requirements 'torch==2.0.1', 'torchvision==0.15.2', 'onnx==1.14.0', 'onnxruntime==1.15.1', 'pycocotools==2.0.0'
1 from ultralytics import YOLOv10
2
3 def main():
4     # 指定输入和输出路径
5     weights_path = 'runs/detect/insulator_detect2/weights/best.pt'
6     output_path = 'runs/detect/insulator_detect2/weights/best.onnx'
7
8     # 加载模型
9     model = YOLOv10(weights_path)
10
11    # 导出为ONNX格式
12    success = model.export(format='onnx', opset=12)
13
14    if success:
15        print(f"模型已成功转换为ONNX格式，保存在: {output_path}")
16    else:
17        print("模型转换失败")
18
19 if __name__ == '__main__':
20     main()
```

图 4 模型格式转换

转换后我们会得到一个应用于边缘设备的 best.onnx 文件：



```
Ultralytics YOLOv8.1.34 Python-3.10.14 torch-2.2.2 CPU (AMD Ryzen 7 5800H with Radeon Graphics)
YOLOv10s summary (fused): 293 layers, 8037282 parameters, 0 gradients, 24.5 GFLOPs

PyTorch: starting from 'runs\detect\insulator_detect2\weights\best.pt' with input shape (1, 3, 640, 640) BCHW and output shape(s) (1, 300, 6) (62.3 MB)

ONNX: starting export with onnx 1.17.0 opset 12...
ONNX: export success 1.2s, saved as 'runs\detect\insulator_detect2\weights\best.onnx' (27.8 MB)

Export complete (4.3s)
Results saved to D:\yolov10-main\runs\detect\insulator_detect2\weights
Predict:      yolo predict task=detect model=runs\detect\insulator_detect2\weights\best.onnx imgsz=640
Validate:     yolo val task=detect model=runs\detect\insulator_detect2\weights\best.onnx imgsz=640 data=D:/yolov10-main/datasets/insulator_detect/data.yaml
Visualize:    https://netron.app
```

图 5 转换结果

然后，我们将该 .onnx 文件传输到 Atlas 开发板上，并使用 ATC 工具，采取以下指令进行转换：（在我们实验中，我们的输出文件名称为 yolov10s_insulator.om，输入名字为 best.onnx）

```

(base) root@davinci-mini:~# ls
Ascend  Music  Videos  edge_infer  resnet50_sdk_python_sample
Desktop Pictures aicpu_kernels edge_infer_yolo edge_infer_yolo resnet50_sdk_python_sample.zip
Documents Public aicpu_package_install.info hdc_ppc
Downloads Templates configFile log
(base) root@davinci-mini:~# cd edge_infer_yolo/
(base) root@davinci-mini:~/edge_infer_yolo# ls
(base) root@davinci-mini:~/edge_infer_yolo# atc --model=best.onnx --framework=5 --output=yolov10s_in
nsulator --input_format=NCCHW --input_shape="images:1,3,640,640" --log=error --soc_version=Ascend310
B1 --input_fp16_nodes="images" --output_type=FP16
ATC start working now, please wait for a moment.
...
ATC run success, welcome to the next use.
W11001: Op [/model.23/Expand_1] does not hit the high-priority operator information library, which
might result in compromised performance.
W11001: Op [/model.23/Mod] does not hit the high-priority operator information library, which might
result in compromised performance.
W11001: Op [/model.23/Div] does not hit the high-priority operator information library, which might
result in compromised performance.
W11001: Op [/model.23/Expand_2] does not hit the high-priority operator information library, which
might result in compromised performance.
W11001: Op [/model.23/Expand] does not hit the high-priority operator information library, which mi
ght result in compromised performance.
(base) root@davinci-mini:~/edge_infer_yolo# █

```

图 6 从 .onnx 转换为 .om

转换成功后，我们便会在工作目录下获得一个 .om 文件。此文件是 Atlas 设备适配后的边缘检测文件。由于此时我们已经无法使用 YOLO 为我们提供的封装函数包了。所以，此时我们参考了 BS04 实验中的 yolov5 检测的方法，我们使用 cv2 包进行检测函数的构建。其为我们提供了使用 AtlasAPI 以及使用 cv2 进行检测和框选结果的参考。

我们的应用开发主要由两个程序组成，一个是开发板端的程序，一个是服务端的程序。

开发端板程序：

其主要功能是不循环运行，并通过监测其开放端口，判断是否接受到了正确的 TCP 报文。当接收到了正确的 TCP 报文后，其会对 testphotos 文件夹下的所有图片进行检测判断并将检测结果（出现绝缘子缺失的图片信息）进行封装，将其封装为 TCP 报文然后发送到我们服务端的指定端口。让我们的指定端口收到绝缘子问题报告信息。

接下来是对其结构的分析：

```

1  # 开发板端程序
2  import os
3  import json
4  import socket
5  import numpy as np
6  import cv2
7  import torch
8  from ais_bench.infer.interface import InferSession

```

图 7 开发板端的导入

我们的开发板端程序使用了以上的包。其中 os 进行文件的读写操作，socket 包用于 TCP 报文通信，json 包用于构建封装报文的信息格式。Numpy、torch 包进行数据处理，cv2 包进行图像读取、格式转换、识别等。Ais_bench 包是 atlas 开发板的 SDK 包，其为我们提供了 NPU 的调动与选择，提供了硬件支持。

然后是我们的主要部分，检测类及其函数，我们的检测类由五个部分组成：

```
class EdgeDetectionServer: 1 usage
    def __init__(self, host='0.0.0.0', port=12345):
        self.host = host
        self.port = port
        # 加载OM模型，使用InferSession替代onnxruntime
        self.model_path = 'yolov10s_insulator.om'
        self.session = InferSession(0, self.model_path) # 0表示设备ID
        self.DEFECT_CLASS_ID = 0
        self.CONFIDENCE_THRESHOLD = 0.5
        self.IOU_THRESHOLD = 0.45
        self.input_shape = [640, 640]
```

图 8 初始函数

其中定义了我们的端口号信息、模型信息以及我们的 NPU 信息、缺失类别属性 class_id、置信度阈值、IOU 阈值、输入图像标准尺寸等。并使用华为 Atlas 的 InferSession 接口加载 OM 模型。相较于传统的 ONNX Runtime, InferSession 专为华为昇腾 AI 处理器优化，能够充分利用 Atlas 的 NPU 加速能力，并且经过处理后我们的模型才可以开始任务。

然后是我们的预处理函数：

```
def preprocess_image(self, img_path, target_size=640): 1 usage
    # 读取图像
    img = cv2.imread(img_path)
    # 图像预处理
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, dsize=(target_size, target_size))
    img = img.astype(np.float32) / 255.0
    img = np.transpose(img, axes=(2, 0, 1)) # HWC转CHW
    img = np.ascontiguousarray(img, dtype=np.float16) # 转换为float16类型
    img = np.expand_dims(img, axis=0) # 添加batch维度
    return img, img.shape
```

图 9 预处理函数

其将我们的图像读入并进行预处理函数中，然后进行图像格式的转换，首先是图像颜色通道的转换。并对其尺寸进行了重构对色彩像素值进行归一化操作，使得其可以配模型的输入尺寸并达到更好效果。由于 cv2 的默认格式为 HWC，所以将其转化为适配于模型的 CHW 通道排列。并将其数据类型转为适配 CANN 算子的 float16 类型，从而适配 NPU。并最后增加维度，让 Ascend 平台支持对我们检测任务的批量处理。

然后是针对图像的检测函数：

```
def detect_defects(self, input_dir='testphoto/input'): 1 usage
    fault_data = {
        'header': 'insulator_error',
        'count': 0,
        'defect_details': []
    }
}
```

图 10 检测函数中的错误信息存储 JSON 格式

我们可以看到，检测程序是对指定路径的批量检测。Fault_data 字典是我们返回报文的格式。

然后是，我们的批量检测循环：

```
for filename in os.listdir(input_dir):
    if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
        img_path = os.path.join(input_dir, filename)
        try:
            # 预处理图像
            img, _ = self.preprocess_image(img_path)

            # 运行推理
            outputs = self.session.infer([img])[0]

            # 非极大值抑制后处理
            boxout = self.nms(outputs, conf_thres=self.CONFIDENCE_THRESHOLD, iou_thres=self.IOU_THRESHOLD)

            if boxout[0] is not None:
                # 这里不需要再调用.numpy(). 因为boxout[0]已经是numpy数组
                pred_all = boxout[0]

                # 收集当前图片的缺陷信息
                defect_boxes = []
                for det in pred_all:
                    if det[5] == self.DEFECT_CLASS_ID and det[4] >= self.CONFIDENCE_THRESHOLD:
                        # 获取原始图像尺寸用于坐标转换
                        original_img = cv2.imread(img_path)
                        orig_h, orig_w = original_img.shape[:2]

                        # 转换坐标到原始图像尺寸
                        x1 = int(det[0] * orig_w / 640)
                        y1 = int(det[1] * orig_h / 640)
                        x2 = int(det[2] * orig_w / 640)
                        y2 = int(det[3] * orig_h / 640)

                        defect_boxes.append({
                            'bbox': [x1, y1, x2, y2],
                            'confidence': float(det[4])
                        })
                }
```

图 11 图像检测以及结果处理


```

# 如果有检测到缺陷，添加到报告中
if defect_boxes:
    fault_data['count'] += 1
    fault_data['defect_details'].append({
        'filename': filename,
        'defect_count': len(defect_boxes),
        'defects': defect_boxes
    })
    print(f"在图片 {filename} 中检测到 {len(defect_boxes)} 个缺陷")

except Exception as e:
    print(f"处理图像 {filename} 时出错: {str(e)}")
    continue

return fault_data

```

图 12 将缺陷信息存入报告 JSON 中

在这个循环中，我们使用 `os.listdir(input_dir)` 得到了指定目录下所有图片名字的列表，然后我们使用 `for` 循环遍历每个文件名，对其检测达到了对于目录下批量图片信息检测的处理。对每个图片，我将其进行预处理并对其进行推理得到一个 `output` 结果。然后对检测结果进行 `nms` 非极大值抑制去掉低于置信度阈值和 `Iou` 阈值的选框，达到更优的检测效果。得到的结果返回到 `boxout` 中，其中存储了剩余选框的选框坐标信息以及置信度信息。然后通过我们读出的 `boxout` 数组，判断我们的图像中是否包含我们要检测的目标 (`boxout[0]` 前 4 位存储了置信度框选坐标、之后两位存储检测类别与置信度信息)。若有则进行记录，并将其框选坐标进行实际坐标的转换，存入我们的报错 `json` 数组中。并在终端中打印错误信息 (调试时可用，实际场景中其实可以删除)。

然后是 `nms` 极大值抑制函数，由于我们调取时遇到了报错，貌似不能直接使用 `atlas` 本身的 `nms` 包或者说函数，于是我们便进行了 `nms` 的自定义构建：

由于开始时，我是通过先将 `.pt` 文件转换为 `.onnx` 文件在本机上进行实验的，所以原先使用了 `torchvision` 的 `nms` 进行处理，但在转换到 `atlas` 平台后，仿佛并没法支持这个包进行运行。所以，我们便进行了此函数的自定义编写：

```

def nms(self, prediction, conf_thres=0.5, iou_thres=0.45):
    """非极大值抑制处理 - 自定义实现, 不依赖torchvision"""
    # 转换为numpy数组处理
    if isinstance(prediction, torch.Tensor):
        prediction = prediction.numpy()

    # 获取置信度大于阈值的索引
    mask = prediction[..., 4] > conf_thres
    if not np.any(mask):
        return [None]

    # 筛选出高置信度的检测框
    x = prediction[mask]

    # 按置信度排序(降序)
    indices = np.argsort(-x[:, 4])
    x = x[indices]

```

图 13 非极大值抑制处理

Nms 首先对读入的预测结果进行了筛选，第一遍是对质心的筛选，先通过置信度阈值过滤掉一部分预测结果。然后将这些监测数据存入 x 中，并将其降序排列。然后通过计算 Iou 排除掉另外部分的选择框，留下最优的一批选择框数据：

```

# 执行NMS
keep = []
while len(x) > 0:
    keep.append(x[0])
    if len(x) == 1:
        break

    # 计算IoU
    box1 = x[0, :4]
    boxes = x[1:, :4]

    # 计算交集区域
    xx1 = np.maximum(box1[0], boxes[:, 0])
    yy1 = np.maximum(box1[1], boxes[:, 1])
    xx2 = np.minimum(box1[2], boxes[:, 2])
    yy2 = np.minimum(box1[3], boxes[:, 3])

    w = np.maximum(0, xx2 - xx1)
    h = np.maximum(0, yy2 - yy1)
    inter = w * h

    # 计算并集区域
    area1 = (box1[2] - box1[0]) * (box1[3] - box1[1])
    area2 = (boxes[:, 2] - boxes[:, 0]) * (boxes[:, 3] - boxes[:, 1])
    union = area1 + area2 - inter

    # 计算IoU
    iou = inter / (union + 1e-16)

    # 保留IoU小于阈值的框
    inds = np.where(iou <= iou_thres)[0]
    x = x[inds + 1]

return [np.array(keep) if keep else None]

```

图 14 针对 Iou 进行选框去除

并将剩下的预测数据返回给我们的 detect 函数中。

最后是我们的 start_server 函数，其是启动我们 socket 服务以及进行图像检测主要函数：

```

def start_server(self): 1 usage
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, value=1)
        s.bind((self.host, self.port))
        s.listen(1)
        print(f"边缘检测服务已启动，监听端口: {self.port}")

```

图 15 启动 socket 服务

对于 socket.socket(socket.AF_INET, socket.SOCK_STREAM)

- 创建 IPv4 TCP 套接字
- AF_INET 表示使用 IPv4 地址族
- SOCK_STREAM 表示使用面向连接的 TCP 协议

首先，我们创建面向连接的 IPv4 TCP 套接字连接服务。并绑定本机地址与开放的端口，并开放连接且仅接受一台设备进行连接。

然后不断去等待接收外来的 TCP 报文：

```
while True:
    conn, addr = s.accept()
```

图 16 等待并接收返回的信号

当接收到 TCP 报文后，我们会先判断接收到的 TCP 报文内容是否是我们的启动指令：

```
with conn:
    try:
        # 接收触发信号
        trigger = conn.recv(1024).decode('utf-8')
        if trigger == 'START_DETECTION':
            print("收到检测请求，开始检测...")
            report = self.detect_defects()

            # 检查报告大小
            payload = json.dumps(report).encode('utf-8')
            if len(payload) > 5 * 1024 * 1024: # 限制5MB
                raise ValueError("生成报告过大")
```

图 17 接收报告信息，并进行大小判断

如果是启动指令，程序便会调动 detect 检测程序并接受其检测结果。然后将其结果转换成方便发送的 UTF-8 的 JSON 格式。

在检验无误且报文大小满足要求后，我们尝试向我们的服务器端发送检测结果的 TCP 报文：

```

try:
    # 分块发送数据
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as response_socket:
        response_socket.settimeout(5.0)
        print(f"尝试连接到 192.168.137.1...")
        response_socket.connect(('192.168.137.1', 12346))
        header = f"INSULATOR_REPORT:{len(payload)}|".encode()
        response_socket.sendall(header)
        response_socket.sendall(payload)
        print(f"检测完成, 已发送报告 ({len(payload)}字节)")
except ConnectionRefusedError:
    print("警告: 无法连接到接收服务器(192.168.137.1:12346). 请确保接收服务已启动")
    # 将结果保存到本地文件作为备份
    with open('detection_report.json', 'w', encoding='utf-8') as f:
        json.dump(report, f, ensure_ascii=False, indent=2)
    print(f"检测报告已保存到本地文件: detection_report.json")
except Exception as e:
    print(f"发送报告时出错: {str(e)}")
    # 将结果保存到本地文件作为备份
    with open('detection_report.json', 'w', encoding='utf-8') as f:
        json.dump(report, f, ensure_ascii=False, indent=2)
    print(f"检测报告已保存到本地文件: detection_report.json")
except Exception as e:
    print(f"处理异常: {str(e)}")

```

图 18 检测完成后, 尝试连接服务器端并返回结果报文

首先, 向指定 IP 地址的端口发送连接请求, 当服务器端同意后, 程序便会将检测结果进行 TCP 报文构建然后, 将报文发送给服务端。当连接超时后, 则会报错并将检测结果存入本地。

最后是我们的主函数:

```

if __name__ == '__main__':
    server = EdgeDetectionServer()
    server.start_server()

```

图 19 主函数

当我们的程序运行时, 主函数便会激活 start_server 函数, 并一直等待服务器端发来检测请求。

服务器端程序:

首先, 我们引入服务端程序所需要的包:

```

import sys
import socket
import json
from datetime import datetime
import os
import time # 添加time模块
from PyQt5.QtWidgets import (
    QApplication, QMainWindow, QVBoxLayout, QWidget,
    QPushButton, QLabel, QTextEdit, QScrollArea,
    QHBoxLayout, QGroupBox
)

```

图 20 导入所需的包

Sys 负责辅助构建 Qt 可视化窗口，Os 用于保存接收到的检测报告，socket 与 json 相同负责收发 TCP 报文。Datetime 负责打印日志时提供日志时间。Time 模块用于控制线程任务的休眠。

然后是我们的 Qt 接收线程程序，其由初始化模块、运行模块、保存报告模块组成，首先是初始化模块：

```

class DetectionThread(QThread):
    report_received = pyqtSignal(dict)
    status_updated = pyqtSignal(str)

    def __init__(self, port=12346):
        super().__init__()
        self.port = port
        self.running = True

```

图 21 线程函数初始化模块

首先我么进行了 Qt 数据信号的声明，方便后面进行 Qt 的信号队列处理，并进行了我们接收 TCP 报文的端口信息的初始化。然后是我们的运行模块：

```

def run(self):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, value=1)
            s.bind(('0.0.0.0', self.port))
            s.listen(5)
            self.status_updated.emit(f"绝缘子检测服务监听中，端口: {self.port}")

```

图 22 监听本地端口

运行模块中，我们首先开启对本地端口的监听。让其可以一直监听外来的信

号。在得到连接的信号后，我们便开始为时 10s 的接受等待，当超时后，我们便不再等到并继续进入循环：

```
with conn:
    try:
        # 增加接收超时设置
        conn.settimeout(10.0) # 增加超时时间

        # 安全接收数据
        raw_data = b''
        while True:
            try:
                chunk = conn.recv(4096)
                if not chunk:
                    break
                raw_data += chunk
                # 增加数据长度检查防止溢出
                if len(raw_data) > 10 * 1024 * 1024: # 限制10MB
                    raise ValueError("接收数据过大")
            except socket.timeout:
                # 接收超时，可能是数据已经全部接收
                break
```

图 23 接收数据报文，并判断大小

同时，我们也对接收到的报文进行检查防止其数据长度过长。当超时后，程序会先进行接收数据的判断：

```
if b'INSULATOR_REPORT:' in raw_data:
    try:
        header, payload = raw_data.split(sep: b'|', maxsplit: 1)
        length = int(header.split(b':')[1])

        if len(payload) == length:
            report = json.loads(payload.decode('utf-8'))
            if report.get('header') == 'insulator_error':
                self.report_received.emit(report)
                self.save_report(report)
    except (ValueError, json.JSONDecodeError) as e:
        self.status_updated.emit(f"数据解析错误: {str(e)}")
```

图 24 判断返回报文是否符合要求

如果我们的报文进行二进制字符串比对，若是符合要求，我们的报文会被 Qt 线程展示到我们的 Qt 界面中，然后将其保存到我们的本地路径中。

然后是保存报告日志的模块：

```

def save_report(self, report): 1 usage
try:
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    report_file = f"{LOG_DIR}/report_{timestamp}.json"

    os.makedirs(LOG_DIR, exist_ok=True)

    # 保存JSON报告
    with open(report_file, 'w', encoding='utf-8') as f:
        json.dump(report, f, indent=2, ensure_ascii=False)

    # 将详细信息追加到日志文件
    with open(LOG_FILE, 'a', encoding='utf-8') as log_f:
        log_f.write(f"\n===== 检测报告 {timestamp} =====\n")
        if report['count'] == 0:
            log_f.write("未检测到缺陷绝缘子\n")
        else:
            log_f.write(f"缺陷图片总数: {report['count']}\n")
            for detail in report['defect_details']:
                log_f.write(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] INFO - 文件: {detail['filename']}\n")
                log_f.write(f"缺陷数量: {detail['defect_count']}\n")
                for i, defect in enumerate(detail['defects']):
                    log_f.write(f"缺陷{i+1}: 置信度 {defect['confidence']:.2f} 位置 {defect['bbox']}\n")
            log_f.write(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}] INFO - 报告已保存至 {report_file}\n")

    self.status_updated.emit(f"报告已保存至 {report_file}")
except Exception as e:
    self.status_updated.emit(f"报告保存失败: {str(e)}")

```

图 25 存储函数

我们的日志有两份，一份是单独的每次的报告，负责直接查看每次出错时的绝缘子的具体位置。而另一份日志则是总体的没错的操作日志，其储存了我们一直以来的操作以及返回绝缘子报告(包括了时间)。

然后是我们的 Qt 窗口函数程序，首先是我们的初始化模块：


```

class MainWindow(QMainWindow): 1 usage
    def __init__(self):
        super().__init__()
        self.setWindowTitle("绝缘子缺陷检测系统")
        self.setGeometry(100, 100, 800, 600)

        # 主布局
        main_widget = QWidget()
        main_layout = QVBoxLayout()

        # 标题
        title = QLabel("绝缘子缺陷检测系统")
        title.setFont(QFont("Arial", 16, QFont.Bold))
        title.setAlignment(Qt.AlignCenter)
        main_layout.addWidget(title)

        # 控制面板
        control_group = QGroupBox("控制面板")
        control_layout = QHBoxLayout()

        self.start_btn = QPushButton("开始检测")
        self.start_btn.setStyleSheet(
            "QPushButton {background-color: #4CAF50; color: white; padding: 10px;}"
            "QPushButton:hover {background-color: #45a049;}"
        )
        self.start_btn.clicked.connect(self.start_detection)
        control_layout.addWidget(self.start_btn)

        control_group.setLayout(control_layout)
        main_layout.addWidget(control_group)

        # 结果显示区域
        result_group = QGroupBox("检测结果")
        result_layout = QVBoxLayout()

```

图 26 Qt 界面初始化-1

其构建了 Qt 可视化窗口的大致布局与一些初始参数。

```

# 结果显示区域
result_group = QGroupBox("检测结果")
result_layout = QVBoxLayout()

self.result_text = QTextEdit()
self.result_text.setReadOnly(True)
self.result_text.setStyleSheet("background-color: #f8f9fa;")

scroll = QScrollArea()
scroll.setWidgetResizable(True)
scroll.setWidget(self.result_text)

result_layout.addWidget(scroll)
result_group.setLayout(result_layout)
main_layout.addWidget(result_group)

# 状态栏
self.status_label = QLabel("准备就绪")
self.status_label.setStyleSheet("color: #666; font-style: italic;")
main_layout.addWidget(self.status_label)

main_widget.setLayout(main_layout)
self.setCentralWidget(main_widget)

# 启动检测线程
self.detection_thread = DetectionThread()
self.detection_thread.report_received.connect(self.display_report)
self.detection_thread.status_updated.connect(self.update_status)
self.detection_thread.start()

# 添加一个标志来跟踪检测状态
self.detection_in_progress = False

```

图 27 Qt 初始化-2

然后是一些布局。不过这里有一个启动检测线程的定义。其中启动检测线程就是我们上面的另一个类。我们用 Qt 线程将两个变量与线程类的两个变量进行绑定。下面的跟踪监测状态标识变量是用于进行判断是否在检测状态，防止反复发送请求检测请求出现线程冲突问题。

然后是我们发送检测请求的函数，其由我们的检测按钮触发：

```

def start_detection(self): 1 usage
    # 如果已经在检测中，则不执行新的检测
    if self.detection_in_progress:
        self.update_status("检测正在进行中，请稍候...")
        return

    try:
        self.detection_in_progress = True
        self.start_btn.setEnabled(False) # 禁用按钮防止重复点击
        self.update_status("正在发送检测请求...")

        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.settimeout(5.0) # 设置超时
            s.connect(('192.168.137.2', 12345))
            s.sendall('START_DETECTION'.encode('utf-8'))
            self.update_status("已发送检测请求，等待结果...")
    except Exception as e:
        self.update_status(f"发送检测请求失败: {str(e)}")
        self.detection_in_progress = False
        self.start_btn.setEnabled(True) # 重新启用按钮

```

图 28 检测函数

当被按钮触发后，程序会向我们开发板端指定端口发送 TCP/IP 请求报文，然后进入等待。此间按钮会被禁用，不允许其反复调用造成冲突。

然后是我们 Qt 界面的展示函数：

```

def display_report(self, report): 1 usage
try:
    self.result_text.clear()

    if report['count'] == 0:
        self.result_text.append("未检测到缺陷绝缘子")
    else:
        self.result_text.append("==== 绝缘子缺陷报告 =====")
        self.result_text.append(f"缺陷图片总数: {report['count']}\n")

        for detail in report['defect_details']:
            self.result_text.append(f"文件: {detail['filename']}")
            self.result_text.append(f"缺陷数量: {detail['defect_count']}")

            for i, defect in enumerate(detail['defects']):
                self.result_text.append(
                    f"缺陷{i+1}: 置信度 {defect['confidence']:.2f} 位置 {defect['bbox']}"
                )
            self.result_text.append("")

        # 重置检测状态, 允许再次检测
        self.detection_in_progress = False
        self.start_btn.setEnabled(True) # 重新启用按钮
        self.update_status("检测完成")
except Exception as e:
    self.update_status(f"显示报告异常: {str(e)}")
    self.detection_in_progress = False
    self.start_btn.setEnabled(True) # 重新启用按钮

```

图 29 报告展示到 Qt 上

当我们接收到报文后, Qt 线程会自动调动该程序, 并对 text 框尽心那个情况, 然后把新的绝缘子缺陷报告打印到我们的文本框中, 然后再打印完成后重启按钮, 是程序可被重新启用。以上就是服务器端的大致结构。