German International University

Faculty of Engineering

Road Surface Deterioration Detection using ITS, AI, and IoT

By

Aditya Prakash Bhagchandani

Under supervision of

Dr. Amr Talaat

January 2025

Approval Sheet

Name: Aditya Prakash Bhagchandani

Faculty: Engineering

Program: Robotics and Automation Engineering

Thesis title: Road Surface Deterioration Detection using ITS, AI, and IoT

This thesis has been approved in partial fulfillment of the degree of Bachelors of Science in Robotics and Automation Engineering awarded by the Faculty of Engineering at the German International University.

Examiners:

Dr. Amr Talaat

Date: **3rd February 2025** **Signature:** 

Declaration

I certify that this project work titled
"Road Surface Deterioration Detection Using ITS, AI, and IoT" is my own work. The work has not been presented elsewhere for assessment. The materials that have been used from other sources have been properly acknowledged/cited.

Author

Aditya Prakash Bhagchandani

Signature: 

Date: **30th January 2025**

Acknowledgements

I would like to take this opportunity to express my deep gratitude to my parents and my brother for all the support and encouragement given along the way. Their faith in me was a major boost to motivation and personal strength.

I am especially grateful to Dr. Amr Talaat for the extensive guidance and assistance he provided throughout this project. Without his wide-ranging knowledge and acumen, this piece of work would not be as it is today and may well have fallen short of reaching its goal.

Special thanks go out to my friends for friendship and support that have been sources of comfort and inspiration. Due to them, this journey certainly has been special.

Aditya Prakash Bhagchandani

Abstract

The deterioration of road surfaces, manifesting as cracks (transverse, longitudinal, alligator, etc.) and potholes, imposes a heavy burden on municipal budgets every year. Traditional manual inspection is labor-intensive and lacks standardization. Emerging technologies—such as vehicle sensor-based monitoring and laser imaging—offer some improvements but are also subject to limitations: sensor data is vehicle-dependent and needs consistent maintenance, while laser imaging is still labor-intensive and not practical for frequent surveys. As a result, computer vision-based crack detection using machine learning has become a very active area of research. However, many existing models are computationally complex, hence demanding high-end hardware. This thesis addresses this challenge by developing a lightweight yet accurate model suitable for deployment on edge devices like dashcams. This allows for real-time crack detection and reporting to local authorities for further analysis and repair. YOLOv11n-OBB has been used as the base model and it has been trained on a custom dataset which combines the RDD2022 dataset with an additional pothole dataset from Kaggle for 450 epochs. The trained model achieves a mean Average Precision (mAP50) of 76.819% for detecting the following types of cracks: alligator, transverse, and longitudinal cracks, as well as potholes. The original model runs at 9.56 FPS on a Raspberry Pi 5 8GB, while at 10% pruning gives 10 FPS with 75.81% mAP50, and at 20 pruning yields 12.5 FPS at 74.5 % mAP50. In this line, the research showed real-time, edge-based road defect detection is feasible for proactive infrastructure maintenance.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

One of the most capricious and costly expenditures faced by municipalities around the world is road surface deterioration. In the UK alone, a house of common library report made it known that 20% of carriageway maintenance budgets are assigned to immediate repairs, costs that rise astronomically when small cracks are ignored and develop into potholes[19]. The same report found an enormous backlog of repairs to potholes during 2022/23, estimated at close to £968.9 million, hence justifying the critical nature of early crack detection[19]. Potholes do not only weigh heavily on the public purse; they are equally a serious threat to safety on the roads and to vehicles. According to research done by the government of India in 2021, main reason 0.8% of the accidents occurred during the year was due to potholes out which 1481 deaths and 3103 injuries have been recorded [20].

1.2 Problem Statement

The current road inspection techniques have different drawbacks. Manual inspection, though traditional, is laborious and slow. Technology-driven approaches, such as vehicle sensor-based monitoring, provide real-time data on road health and anomalies like sudden roughness changes. However, the accuracy of these methods depends largely on regular vehicle maintenance [7]. Laser imaging, though not constrained by the limitations of visible light, is dependent on special equipment and trained personnel, which in effect makes frequent surveys impractical[22].

It is these limitations that have motivated research into computer vision-based machine learning models for automatic road damage detection. Several recent works reported very promising results with high accuracies on diverse datasets: a VGG19-based model, an average mAP of 82.79% [2], and another with the inclusion of severity analysis at an accuracy of 91.2% [8]. Highly performing models usually require big computational resources, hence are not

affordable for real-time deployment on regular vehicles. Moreover, real-time video feeds of multiple vehicles simultaneously would introduce severe data transmission bottlenecks if processed centrally. Thus, there is a strong necessity to develop lightweight models for deployment in edge devices. In this context, the paper explores a bit deeper on YOLOv4 model that was trained using RDD2022 and got a mean average precision of 63.58% for smartphone deployment[6]. This thesis, therefore, aims to develop a lightweight but accurate model to be deployed and tested on an edge device called Raspberry Pi 5 to later on be implemented in real life to dash-cam which would provide real-world road damage detection and report automatically back to the local authority via IoT connectivity.

1.3 Methodology

The aim of this thesis is to develop a lightweight model for detecting road surface deterioration in real-time within an urban environment. This model is intended for implementation in a system as illustrated in Figure:

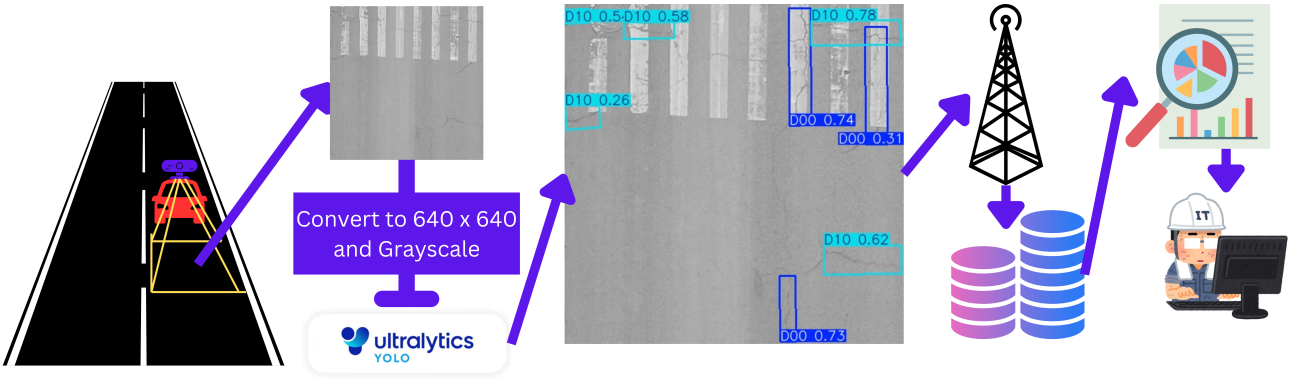


Figure 1.1: Real Time Road Surface Deterioration System

The proposed system captures frames from a car's dash-cam, processes them to meet the model's input requirements, and then feeds them into the model. Once any form of deterioration has been detected by the model, it saves the information and sends it to the database of the local municipality. It is then processed into a report that the municipal officers are able to go through and take appropriate action on it.

Considering the average city speed worldwide is 60 km/h and the an assumed camera's field of view is 2 meters, so for the system to function effectively, the model must achieve a minimum frame rate of 9 FPS (frames per second) on an edge device such as the Raspberry Pi 5. Additional tasks such as frame manipulation, saving, and sending data must also be taken into consideration which would require model to run above 9 FPS . The frame capture rate will depend

on the car's speed, provided it remains below 60 km/h and not stationary .

$$\text{Minimum FPS} = \frac{\left(\frac{\text{Distance covered in one hour in meters}}{3600}\right)}{\text{Field of view of the dash-cam in meters}} \tag{1.1}$$

To evaluate the model's accuracy, we will use the mAP50 score, which summarizes the model's performance across all classes and intersection over union thresholds.

$$\text{mAP} = \frac{1}{N}\sum_{i=1}^{N}\text{AP}_i \tag{1.2}$$

Subsequently, the model will be pruned to reduce its weight and increase FPS with minimal impact on accuracy.

1.4 Thesis overview

The rest of this thesis is organized as follows:

- Chapter 1: Introduction and Methodology of the thesis.

  This chapter introduces the thesis topic, outlines the research objectives, and details the methodologies implemented in the study.

- Chapter 2: State of the Art

  This chapter reviews various types of technologies and methods used in road surface deterioration detection and analyzes relevant research papers in the field.

- Chapter 3: Specifications

  This chapter provides a detailed description of the system, including an overview of the models, datasets, evaluation metrics, training parameters, and pruning techniques employed in the study.

- Chapter 4: Implementation, Testing & Results

  This chapter outlines the development and deployment of the proposed system, detailing the testing procedures and analyzing the results obtained from implementation and evaluation.

- Chapter 5: Conclusion & Future Work

  This chapter summarizes the key outcomes of this work and the contribution it makes. Also it discusses on the possible future work on this project and the current limitations.

Chapter 2

State of the Art

The detection and analysis of road cracks are crucial for maintaining infrastructure and ensuring road safety. Traditional methods, while effective, often involve labor-intensive manual inspections or expensive hardware solutions [15]. This chapter provides an overview of the current state of the art in road crack analysis, highlighting the limitations of existing approaches and the potential of new methodologies.

2.1 Traditional Methods

Traditional methods of road crack analysis rely heavily on manual inspections, which are labor-intensive and time-consuming [15]. Inspectors must physically examine road surfaces, documenting and assessing cracks. This process is not only slow but also prone to human error. Since documentation is not standardized, reports can vary significantly between inspectors, and inspectors may sometimes overlook certain areas. To improve efficiency and coverage, inspectors later began using cars driven at slow speeds. This approach allowed them to cover more ground while minimizing fatigue and exposure to the elements. However, it still carries the risk of human error and remains impractical for daily analysis due to traffic and other factors [15].

2.2 Hardware-Intensive Approaches

Recent advancements have introduced hardware-intensive approaches for road crack detection, such as car-mounted sensors and laser-based imaging systems [7, 22]. Laser-based imaging offers the advantage of being unaffected by lighting conditions and provides highly accurate and reproducible results, which is unlikely with traditional methods [22]. Many companies have made these systems available to governments through public-private partnerships, as they are better suited for detailed road analysis, with some even providing 3D road maps [22, 15]. However, a significant disadvantage of laser-based systems is

the cost of the hardware and the system's complexity. Although complexity has decreased over time, it still requires a specialized vehicle equipped with the hardware and system, making daily deployment costly and impractical for covering large areas quickly [22].

Vehicle sensor-based analysis, on the other hand, does not require specialized hardware. It utilizes existing vehicle sensors to detect subtle bumps and fluctuations in road roughness. These signals are processed to generate a rough estimate of the road condition, highlighting potential abnormalities [7]. However, this method faces challenges related to vehicle maintenance. The accuracy of the data depends on proper vehicle upkeep, making large-scale analysis difficult since road roughness perception varies between vehicles, and the general public may not maintain their vehicles regularly. Furthermore, the system may require recalibration after each maintenance, hindering large-scale deployment.

2.3 Using Machine Learning and Computer Vision

Machine Learning and Computer Vision algorithms are popular for automatic road surface damage detection due to their effectiveness in processing large amounts of visual data. Therefore, the mentioned approaches might represent an alternative to the traditional methodology based on manual inspection, which is too time-consuming and expensive, besides risking people.

Among these, deep learning represents a subcategory of machine learning that recently has shown quite promising results. Because of their architecture, namely the convolutional neural networks that are very good for the accurate detection and classification of road surface damages, deep learning models can thus automatically extract complex features from images. Quite a number of researchers have therefore tried different architectures of CNNs and object detection algorithms for road damage detection, with large differences in accuracy and computational efficiency.

A few of the newer models have achieved state-of-the-art accuracy for object detection tasks: two-stage detectors such as Region Convolutional Neural Networks, Fast R-CNN, and Faster R-CNN. Initially, these models produce proposals about the regions of interest and then classify and refine them. However, because of their computationally intensive nature, they may not be appropriate for real-time applications or any deployment on resource-constrained platforms. For example, although effective, Faster R-CNN may be computationally too expensive for bridge steel crack detection. In, Pham et al. [14] evaluated Faster R-CNN with Detectron2 and obtained F1 scores of 51.0% and 51.4% on test1 and test2 sets of the Global Road Damage Detection Challenge 2020. The authors mentioned the discrepancies in labeling the dataset. Although the Pham et al. visualisations produced good prediction results, their F1 scores were low.

These models have, in addition, high precision, requiring much computational power and, hence, are not suitable for real-time use on low-end devices [14].

The opposite is the case in the family of one-stage detectors such as SSD and YOLO. They are faster because both tasks of classification and bounding box regression are performed in one stage. These networks are computationally more effective and hence can be used in real-time applications. For instance, YOLOv5 achieved 59.9% mAP in road damage detection from UAV imagery [18]. YOLOv7 was also documented by another research to achieve an overall F1 score of 59% on the road damage dataset [18].

Another single-stage model reported to have high precision in detecting road damage is RetinaNet [1]. Ale et al. , for example, reported an mAP of 0.8279 with a RetinaNet detector with a VGG19 backbone but at the average time of 0.5 seconds per image. While generally one-stage detectors are faster compared to two-stage detectors, some of them might sacrifice some accuracy relative to two-stage detectors [1]. For example, while SSD and YOLO are fast, they might not be able to achieve the same level of accuracy as Faster R-CNN in complex scenarios. Few works explored lightweight models like MobileNet and SqueezeNet for real-time usage only, but these may perform not so well on such versatile datasets of road damages [8]. In particular, the works from Maeda et al. experimented SSD with Inception V2 and MobileNet, and stated that, even though they run on a smartphone, their recall on certain types of damages is low, i.e., 0.05 for D11 with SSD Inception V2 [11].These models suffer badly from the quality and quantity of the training data.

Most of the works used publicly available datasets such as RDD2018 and RDD2020 for training and testing [2, 3]. Not all datasets can be representative regarding all road conditions or damage types, thus generalization power of the trained models can be affected. Moreover, some models were trained with rather small amount of data compared with others and hence are unable to generalize well on new, previously unseen data. For example, a lightweight autoencoder-based approach reported 79.33% detection accuracy, as measured by the F1-score, but for only a small dataset of 1058 images [17]. GAN-based data augmentation methods are followed to increase the variation in the data; hence, its performance may vary. Various efforts have been made to improve the precision and effectiveness of these models.

A few works have tried to use transfer learning; they fine-tuned models that were already trained on large datasets like ImageNet, for the purpose of conducting road damage detection [3, 17]. In that respect, other domains' features are utilized in order to enhance the performance of a model when the road damage dataset is really small. Another group of techniques that has been utilized in this task incorporates an attention mechanism, normalization techniques, and multi-scale feature fusion [8, 23]. For example, Zhang et al.

proposed a baseline method with attention fusion and normalization to reduce the impact of the sparse pixel distribution of road damage, which achieved an F1-Score of 34.73% on the CNRDD dataset [23]. Furthermore, others have designed specific loss functions and modified networks to achieve higher scores [18]. Most of the deep learning models have the potential of automatic road damage detection, but model choices depend on requirements on accuracies, speeds, and available computational resources. For real-world applications, a trade-off between accuracy and computational speed has to be achieved.

Hence, this paper approach is to create a lightweight model that can be used in production for the real time system proposed mentioned in the section 1.3 by using diverse dataset and lightweight base model such as YOLO.

Chapter 3

Specifications

3.1 System overview

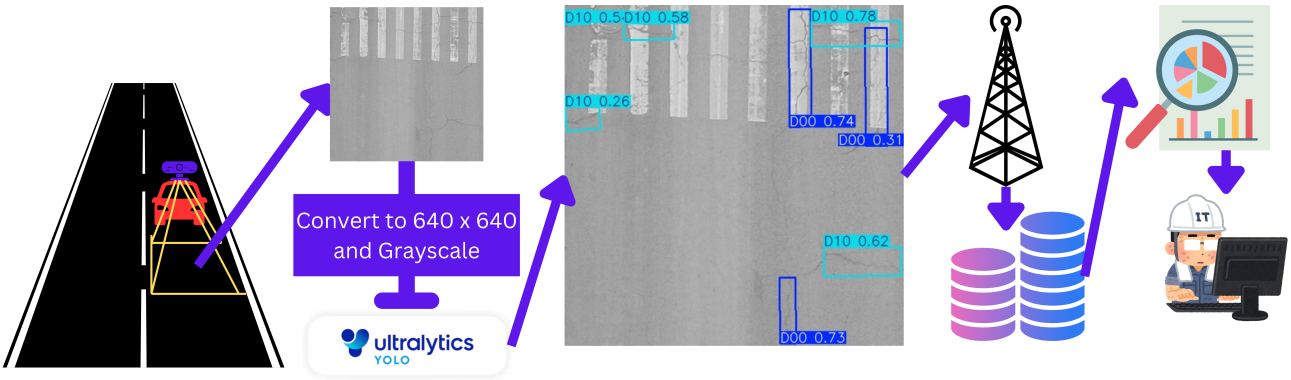


Figure 3.1: Representation of the System

The designed real-time road surface deterioration detection system incorporates computer vision, deep learning techniques, and IoT connectivity. It is optimized specifically for efficient performance over edge platforms, including the Raspberry Pi 5, for real-time processing and timely reporting of detected road abnormalities. What follows is an in-depth frame processing pipeline detail:

3.1.1 Frame Acquisition

A dashcam continuously records video frames at a predefined frame rate. The number of frames captured per second varies based on the camera's field of view (FOV) and the vehicle's speed, as determined by Equation 1.1, provided the vehicle is in motion.

3.1.2 Pre-processing of the Frame

The frames taken into the consideration for analysis then go through a pre-processing step in which the images are resized to 640 by 640 pixels which is the maximum input for an YOLO model and it is then gray-scaled as the model is trained on gray-scale images.

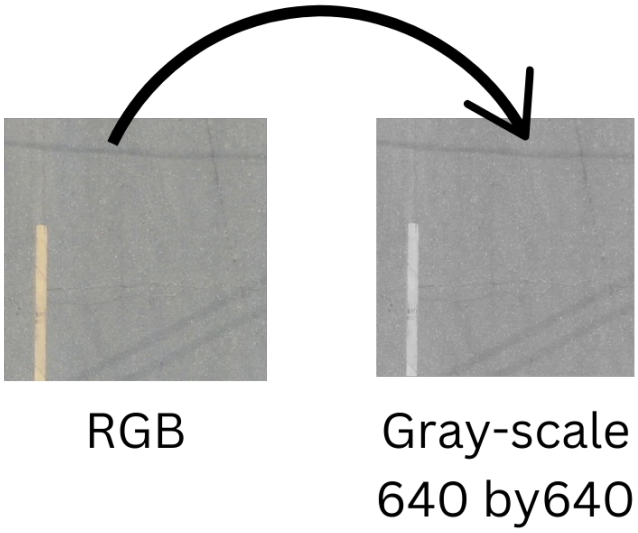


Figure 3.2: Pre-Processing of the image

3.1.3 Model Inference

Once frame is converted as per the requirement then it is given to the model to predict the following types of the cracks it is trained on:

- Longitudinal Cracks (D00)
- Transverse Cracks (D10)
- Alligator Cracks (D20)
- Potholes (D40)

After the model predicts and gives out the results with the confidence of it detecting the damage an appropriate threshold is applied to avoid false detections and Non-Maximum Suppression (NMS) is applied to remove any duplicate bounding boxes given by the model.

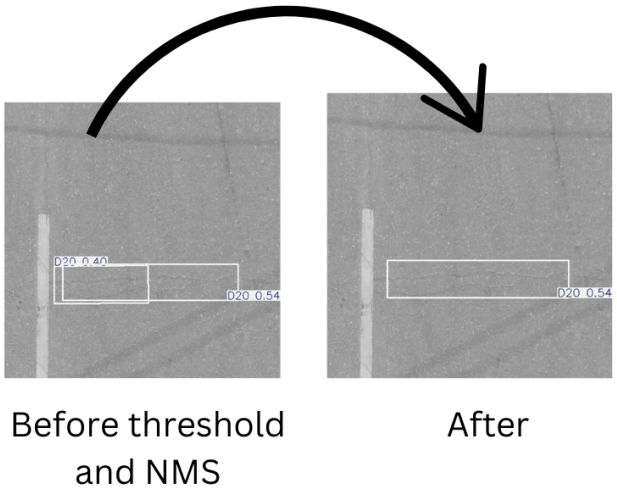


Figure 3.3: Model Predication

3.1.4 Data Storage & Transmission

The detected road defects are stored locally on the device for logging purposes. If an internet connection is available, the system transmits the detections to a cloud-based database or municipal servers via IoT connectivity.

The report includes:

- GPS coordinates (if available) for geolocation tagging.
- Timestamp for tracking when the defect was detected.
- Image of the defect for verification by authorities.

3.1.5 Actionable Insights & Reporting

The city administration is given real-time updates regarding locations of road deterioration, allowing for timely intervention and scheduling of maintenance work.

The system can even generate routine reports that analyze trends in road deterioration over a range of time-frames.

3.2 Models Overview

3.2.1 YOLOv3 Tiny

YOLOv3 Tiny is a lightweight version of the YOLOv3 model, designed for real-time object detection on resource-constrained devices. It has 13 convolutional layers and approximately 8.848 million parameters, making it faster but slightly less accurate compared to the full YOLOv3 model. It achieves a mean Average Precision (mAP) of around 35.9% on the COCO validation dataset and is well-suited for applications where speed is critical, such as embedded systems and mobile devices.[16]

3.2.2 YOLOv7

YOLOv7 is a highly efficient real-time object detection model that builds upon the advancements of previous YOLO versions. It has approximately 36.9 million parameters and 104.7 GFLOPs, making it highly efficient for a wide range of computer vision tasks. It achieves a mean Average Precision (mAP) of around 51.4% on the COCO validation dataset and is known for its balance between speed and accuracy, making it suitable for applications where both performance and efficiency are critical.[21]

3.2.3 YOLO11n

YOLO11n is the latest iteration in the Ultralytics YOLO series, offering cutting-edge accuracy, speed, and efficiency. It has approximately 2.6 million parameters and 6.5 GFLOPs, making it versatile for various computer vision tasks, including object detection, instance segmentation, and image classification. It achieves a mean Average Precision (mAP) of around 39.5% on the COCO validation dataset. YOLO11n is optimized for deployment across different environments, including edge devices and cloud platforms.[10, 9]

3.2.4 YOLO11n-OBB

YOLO11n-OBB is a variant of the YOLO11 model designed for oriented bounding box (OBB) detection. It enhances traditional object detection by detecting objects at different angles, making it suitable for applications such as aerial or satellite image analysis. YOLO11n-OBB offers high accuracy and efficiency, making it ideal for edge devices and real-time applications.[10, 9]

3.2.5 YOLO11n-Seg

YOLO11n-Seg is a variant of the YOLO11 model designed for instance segmentation. It goes beyond object detection by identifying individual objects in an image and segmenting them from the rest of the image. YOLO11n-Seg provides high accuracy and efficiency, making it suitable for applications where precise object boundaries are required.[10, 9]

3.3 Data Collection and Preprocessing

In this thesis we mainly focus on four types of main cracks as followed: Longitudinal Cracks (D00), Transverse Cracks (D10), Alligator Cracks (D20), and Potholes (D40).

3.3.1 Dataset Overview

To Train a model, a combination of two datasets was used by taking the inspiration from a GitHub repo of Ahmed Nahmad with the title RDD2022 [12].

The main dataset that is used is RDD2022 dataset from Crowd Sensing-based Road Damage Detection challenge, CRDDC'2022. It is one of the biggest dataset available for road surface deterioration. [4]. The dataset contains 11 classes: D10, D00, D20, D40, D44, D01, D11, D43, D50, Repair, Block crack. The images were collected from six countries as followed: India, Japan, Czech Republic, Norway, Unites States, and China [4]. The dataset is annotated in

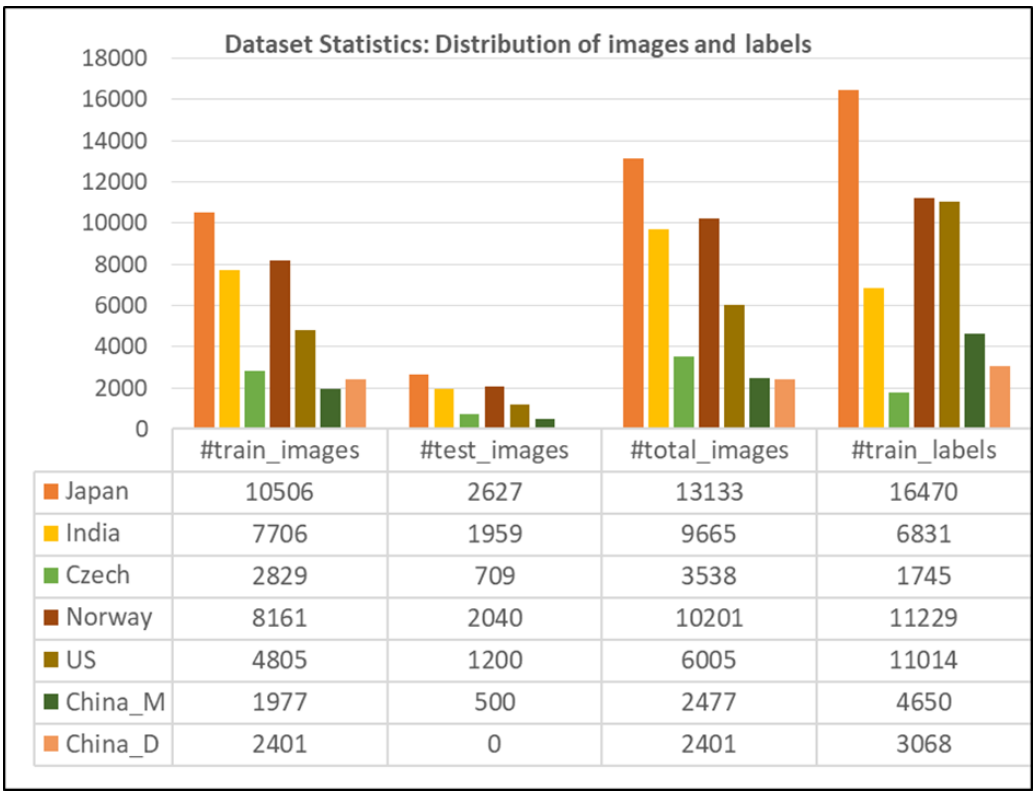
Figure 3.4: Longitudinal Cracks (D00)
Source: Engineering Discoveries

Figure 3.6: Alligator Cracks (D20)
Source: Preventive Maintenance

Figure 3.5: Transverse Cracks (D10) Source: Researchgate

Figure 3.7: Potholes (D40)
Source: DePaula Chevrolet

json format. Here is a detail chart on contents of the dataset provided by the organizers of the challenge:



	#train_images	#test_images	#total_images	#train_labels
Japan	10506	2627	13133	16470
India	7706	1959	9665	6831
Czech	2829	709	3538	1745
Norway	8161	2040	10201	11229
US	4805	1200	6005	11014
China_M	1977	500	2477	4650
China_D	2401	0	2401	3068

Figure 3.8: RDD2022 Dataset Overview Source: ORDDC

The second dataset used is from Kaggle which is combination of multiple other datasets solely focusing on potholes by the user DenisG04. The version used in thesis is 8th version where the user has re-annoted the images the annotation have been save yolo format.[5].

3.3.2 Preprocessing Steps

First, the annotations were converted to one and the same format that could be used in training; then the images were gray-scaled and resized to 640 by 640 pixels for each dimension, according to the model's input layer requirements. Then these two datasets were combined. In the case of duplicates where the same images appeared in both the RDD2022 and Kaggle datasets, the image from RDD2022 was retained since its number of crack annotations was greater. Finally, the combined dataset was split into training, validation, and test sets in a ratio of 70% for training, 20% for validation, and 10% for testing.

3.4 Performance Metrics

This paper focuses on two metrics, mean Average Precision at 50% (mAP50) for the accuracy of the model and Frames per second (FPS) which is evaluated how many frames can the model process in one second.

3.4.1 Frames per second (FPS)

The FPS metric is one of the key factors in assessing the performance of the exported model in real-time applications. Testing for FPS calculation is to be done in the test section of the dataset. Thus, the average FPS will come out to be the total number of images in that section divided by the total duration taken to process those. This metric gives a full overview of how many frames the model can process in real life with efficiency, ensuring practical applicability and effectiveness.

$$\text{FPS} = \frac{\text{Total number of images}}{\text{Total duration taken}} \tag{3.1}$$

3.4.2 mean Average Precision at 50% (mAP50)

The mean Average Precision (mAP) is a widely used metric for evaluating object detection models. The mAP50 metric specifically refers to the mean Average Precision calculated at an Intersection over Union (IoU) threshold of 50%.[13]

3.4.3 Intersection over Union (IoU)

IoU measures the overlap between a predicted bounding box and a ground truth bounding box. It is calculated as:

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}} \tag{3.2}$$

[13]

3.4.4 Precision and Recall

Precision is the ratio of true positive detections to the total number of positive detections (true positives + false positives):

$$P = \frac{TP}{TP + FP} \tag{3.3}$$

[13]

Recall is the ratio of true positive detections to the total number of ground truth instances (true positives + false negatives):

$$R = \frac{TP}{TP + FN} \tag{3.4}$$

[13]

3.4.5 Average Precision (AP)

Average Precision (AP) is calculated as the area under the precision-recall curve. It summarizes the model's precision and recall performance for a specific class:

$$AP = \int_0^1 P(R)\, dR \tag{3.5}$$

[13]

3.4.6 Mean Average Precision (mAP)

Mean Average Precision (mAP) is the average of the AP values across all classes. The mAP50 metric calculates the mAP at an IoU threshold of 50%, meaning a detection is considered correct if the IoU between the predicted and ground truth boxes is at least 50%.

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} \text{AP}_i \tag{3.6}$$

In this equation, N represents the number of classes, and AP_i is the average precision for the i-th class.[13]

3.5 Training Parameters

Careful parameter tuning was done for the training of the model. During training, the learning rate was kept constant at 0.001, while the weight decay was kept constant at 0.0005 to avoid over-fitting. The training image size for the model was 640 × 640 pixels and was resized appropriately to match the input layer. The optimizer used here is Adam, efficient for sparse gradients. Besides, momentum was set to 0.937 while the learning rate factor (lrf) was set to 0.001. Training also included validation, caching of the results, and resume if some checkpoint exists. Model saving was done every 20 epochs.

All the YOLO models initially were set to train with the minimum required training cycles to see the potential whether further training would be worth it or not. This helped select those that were most promising to be taken for extended training. Later on, promising models were further trained up to a total of 450 epochs in total.

3.6 Model Optimization Pruning Techniques

For better efficiency with maintaining performance, several pruning techniques have been applied to the YOLO model by reducing its size and inference time.

3.6.1 Threshold-Based Pruning:

Threshold: Threshold refers to the value computed from weights of the Batch-Norm layers that prune all weights less than the computed threshold value and maintain only the most important filters.
BatchNorm Layers: Normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.

3.6.2 Layer-Wise Pruning:

Convolutional Layers: Apply convolution on the input for feature extraction and reducing the spatial dimensions.

Filters: Kernels in the convolutional layers are for the detection of particular features in the input data.
BatchNorm Parameters: Weights and biases of BatchNorm layers are modified in the process of pruning.

3.6.3 Sequential Pruning:

Layers that come in a sequence such as C3k2 and SPPF for sequential layers are pruned while maintaining the model architecture.

C3k2 : A layer type particular in the YOLO model.
SPPF : A specific layer type in YOLO.

3.6.4 Distillation Loss:

Distillation loss function: Loss function used when training a smaller model, or student model, to mimic a larger one, teacher model.

Student Model: Smaller model which learns from the knowledge of a larger teacher model.
Teacher Model: Larger model, usually pre-trained, which provides knowledge to the student model.
Soft Labels: Probabilities produced by the teacher model to guide the student model.
Hard Loss: The standard cross-entropy loss between the student model outputs and the true labels.

Chapter 4

Implementation, Testing & Results

This chapter will be demonstrating the implementation and testing done during the thesis also analyzing the results.

4.1 Hardware Components

The hardware used for training and initial testing includes a laptop equipped with an RTX 3050ti with CUDA drivers enabled (laptop edition) GPU, 16GB of RAM, and an Intel i7 H 12th Gen processor (laptop edition). For edge deployment, a Raspberry Pi 5 with 8GB of RAM and 32GB of storage was utilized, powered by a 5V, 5A adapter.

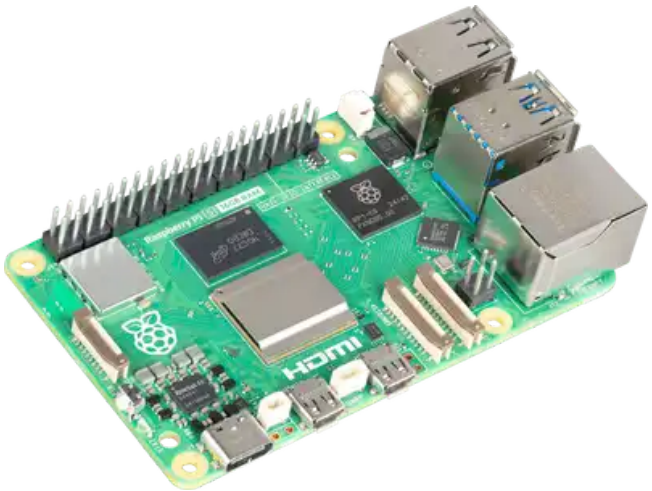


Figure 4.1: Raspberry Pi 5

4.2 Software Components

4.2.1 Operating System and Libraries

The training of models were done using Darknet and PyTorch on Windows 11.

4.2.2 Frameworks and Libraries

Darknet: Used for training models YOLO v3 Tiny and YOLO v7. Darknet is an open-source neural network framework written in C and CUDA, known for its efficiency and speed in real-time object detection tasks.
PyTorch 2.5.1: This version of PyTorch was used to train models like YOLO11n-OBB, YOLO11n, and YOLO11n-Seg. PyTorch is an open-source machine learning library developed by Facebook's AI Research lab, known for its dynamic computation graph and ease of use.

4.2.3 Libraries

- **Ultralytics**: A library that provides implementations of YOLO models, facilitating easy training and deployment.
- **OS**: A standard Python library for interacting with the operating system, used for tasks such as file and directory management.
- **OpenCV (cv2)**: An open-source computer vision library that provides tools for image processing, video capture, and analysis.
- **xml.etree.ElementTree**: A library for parsing and creating XML documents, used for handling annotation files.
- **Shutil**: A Python library for high-level file operations, such as copying and removing files.

  TQDM: A library for creating progress bars, useful for tracking the progress of long-running tasks.
- **Pathlib**: A library for object-oriented filesystem paths, providing an intuitive way to handle file paths.
- **PIL (Pillow)**: A Python Imaging Library that adds image processing capabilities to Python, used for tasks such as image resizing and format conversion.
- **Numpy**: A fundamental package for scientific computing in Python, providing support for large, multidimensional arrays and matrices, along with a collection of mathematical functions.

4.2.4 Edge Device Operating System

The Raspberry Pi 5 runs on Ubuntu 24.04 LTS, with the Ultralytics library used for deploying the final trained model.

```
cd Aditya_Thesis_Project
pip install python3-venv
python -m venv yolo_env
cd yolo_env/Scripts
activate.bat
```

Figure 4.2: Activating Environment

4.3 Language & IDE Setup

Throughout the duration of this thesis, Python 3.12 was utilized for all implementations and testing. The models were trained using the PyTorch and Ultralytics libraries, both of which are Python-based. Additionally, all dataset manipulations were performed using Python, ensuring a seamless and efficient workflow. For models based on the Darknet framework, C++ was used. As for the IDE Visual Studio code Insiders build was used.

4.4 Environment Setup

A new environment was built on the laptop (the training machine) to install the dependencies and libraries. The following code was inserted line by line into the IDE's Terminal:

Once the environment was activated, all dependencies and libraries listed in Chapter 3 were installed to their latest compatible versions.

4.5 Dataset Preparation

Both datasets mentioned in the dataset overview section in Chapter 3 were downloaded and extracted.

Firstly, a `data.yaml` file was created, containing details about the file locations and annotations, such as the classes and the number of classes.

```
train: ../train/images
val: ../val/images
test: ../test/images

nc: 4
names: ['D00', 'D10', 'D20', 'D40']
```

Figure 4.3: Data.yaml File

Next, the annotations were converted from XML format, used for TensorFlow-based models, to appropriate formats such as YOLO format in text files for Darknet models and YOLO11n. Here is an example:

```
<object-class> <x_center> <y_center> <width> <height>
0 0.6591796875 0.796875 0.064453125 0.34375
1 0.8701171875 0.5986328125 0.248046875 0.068359375
```

Figure 4.4: Example of YOLO Annotation Format

Figure 4.5: YOLO Format Representation

The YOLO annotation format contains:

- **Object class:** Represents the class according to the index value of the class array in the YAML file. For example, in Figure 4.4, 0 represents D00, and so on.
- **X_center and Y_center:** These represent the bounding box center values according to the image's pixels, as shown in Figure 4.5.
- **Width and height:** The width and height of the bounding box in pixels, also illustrated in Figure 4.5.

The code for this process is available in Appendix A.1.

As for yolo11n obb and seg, they have their own unique format in txt files and the yolo format were converted to them for these models.

YOLO OBB format:

OBB stands for Oriented Bounding Box which is different from the YOLO format as it introduces orientation to the box and it is represented like this:

```
<object-class> <x1>, <y1>, <x2> <y2> <x3> <y3> <x4> <y4>
0 0.780811 0.743961 0.782371 0.74686 0.777691 0.752174 0.776131 0.749758
```

Figure 4.6: Example of YOLO OBB Annotation Format

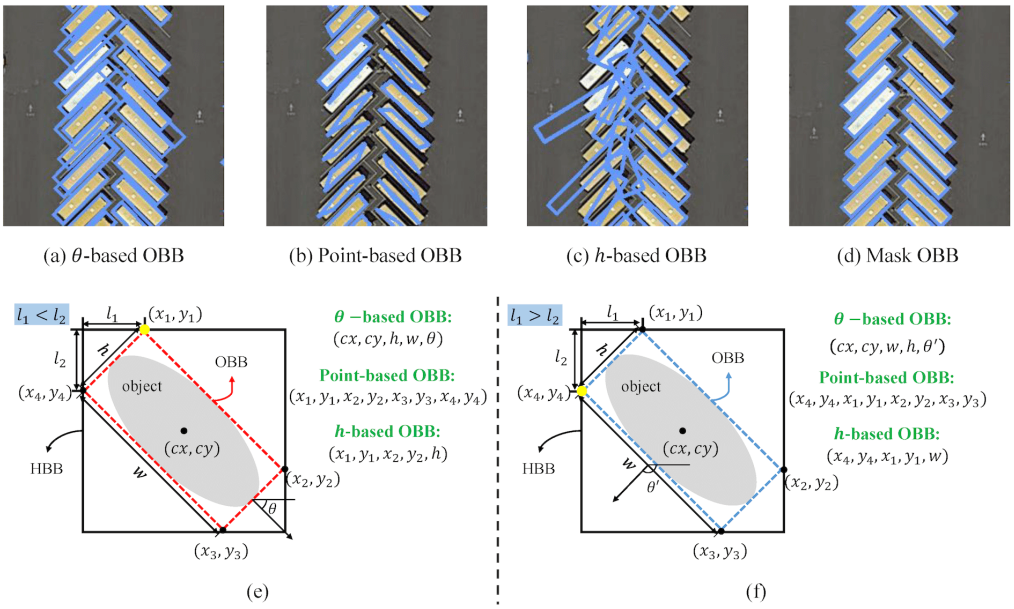


Figure 4.7: Explanation of YOLO OBB Format
Source: THU-MIG GITHUB

YOLO seg format:

Segmentation annotation format introduces segmentation masks to the bounding boxes, allowing for more precise object localization. Unlike the standard YOLO format, which uses rectangular bounding boxes, YOLO11n segmentation provides pixel-level annotations. This format is represented as follows:

```
<object-class> <x_center> <y_center> <width> <height> <segmentation-mask>
0 0.6591796875 0.796875 0.064453125 0.34375 [mask data]
```

Figure 4.8: Example of YOLO OBB Annotation Format

This can be visualized like this:

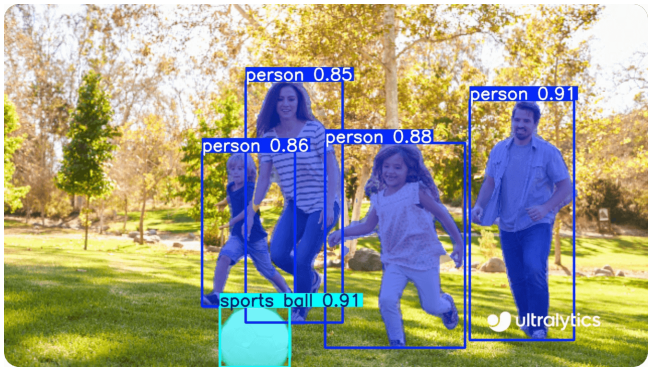


Figure 4.9: Segmentation Visual
Source: Ultralytics

The codes to convert to these format are available in the appendix at A.2 and A.3 respectively.

After getting all of the appropriate formats for the models, next the images were converted gray scale images. The code for this is available in the appendix at A.4.

Then the Dataset was split into 80% Train, 10% Valid and 10% Test using the code available in the appendix at A.5

4.6 Framework Setup

For both of the frameworks Compatible CUDA and cuDNN Drivers were install and verified.

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 561.19                 Driver Version: 561.19         CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3050 ...  WDDM  |   00000000:01:00.0 Off |                  N/A |
| N/A   44C    P0              7W /   75W |      67MiB /   4096MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

Figure 4.10: nvidia-smi Details

```
C:\Users\Aditya>nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Sep_12_02:55:00_Pacific_Daylight_Time_2024
Cuda compilation tools, release 12.6, V12.6.77
Build cuda_12.6.r12.6/compiler.34841621_0
```

Figure 4.11: nvcc Version details

4.6.1 Darknet

To set up the Darknet framework on Windows, I followed the detailed instructions provided in the Darknet GitHub repository from hank-ai. Here are the steps I took to ensure a smooth installation and configuration:

First, I installed the necessary tools and dependencies. This included Visual Studio 2022 Community Edition, CMake, and Git. I used the following commands in the command prompt to install these tools:

```
winget install Git.Git
winget install Kitware.CMake
winget install Microsoft.VisualStudio.2022.Community
```

Figure 4.12: Installing necessary tools

Next, I modified the Visual Studio installation to include support for C++ applications. I opened the Visual Studio Installer, selected "Modify," and then chose "Desktop Development with C++."

After setting up Visual Studio, I installed Microsoft VCPKG to manage the dependencies. I ran the following commands in the Developer Command Prompt for Visual Studio:

```
cd C:\
mkdir C:\src
cd C:\src
git clone https://github.com/microsoft/vcpkg
cd vcpkg
bootstrap-vcpkg.bat
.\vcpkg.exe integrate install
.\vcpkg.exe install
opencv[contrib,dnn,freetype,jpeg,openmp,png,webp,world]:x64-windows
```

Figure 4.13: VCPKG installation

Once VCPKG was set up, I cloned the Darknet repository and built the project using CMake:

```
cd C:\src
git clone https://github.com/hank-ai/darknet.git
cd darknet
mkdir build
cd build

cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_TOOLCHAIN_FILE=
C:/src/vcpkg/scripts/buildsystems/vcpkg.cmake ..

msbuild.exe /property:Platform=x64;Configuration=Release /target:Build
-maxCpuCount -verbosity:normal -detailedSummary darknet.sln

msbuild.exe /property:Platform=x64;Configuration=Release PACKAGE.vcxproj
```

Figure 4.14: Cloning Darknet Repository

Finally, I ran the NSIS installation wizard to correctly install Darknet, including the necessary libraries, include files, and DLLs. This ensured that Darknet was fully operational on my Windows system.

4.6.2 PyTorch with Ultralytics

To set up the PyTorch environment with the Ultralytics library, I followed the detailed instructions provided in the Ultralytics YOLO Docs.

First, I installed the necessary tools and dependencies. This included Python, PyTorch, and the Ultralytics library.
For PyTorch installation command was taken from the official website using the settings adjusted to the required:

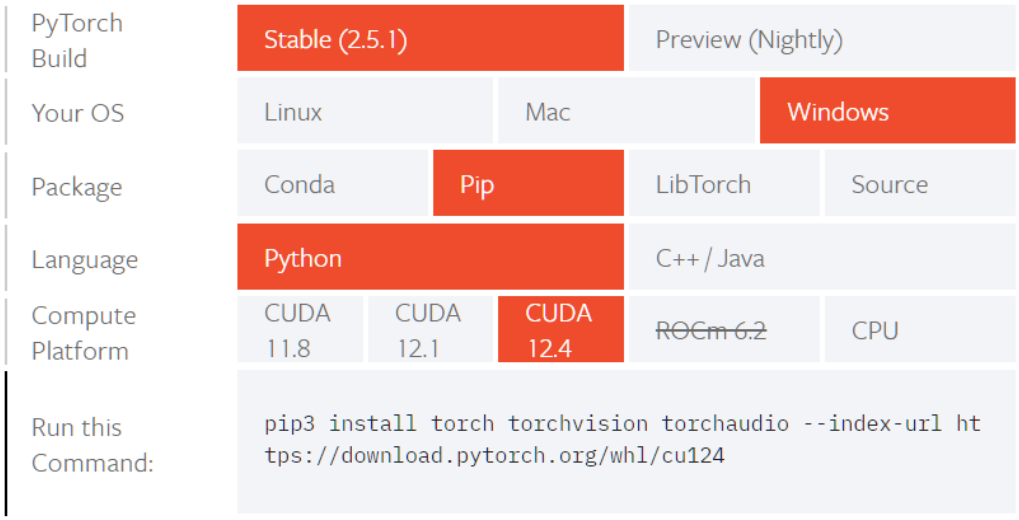


Figure 4.15: PyTorch Installation Command

As for installing Ultralytics library this comand was used

```
pip install ultralytics
```

Next, I verified the installation of PyTorch and CUDA to ensure that the GPU acceleration was functional. This was done by running the following commands:

```
python -c "import torch; print(torch.cuda.is_available())"
```

and the output was `True`.

4.7 Phase Briefing

Implementation and testing have been divided into four phases, as follows:

- Phase 1: Initial Training and Evaluation

  In this phase, all models were tested with minimal training cycles to assess which ones showed promise for further training.

- Phase 2: Final Model Training

  In this phase, the selected promising model will undergo additional training cycles until further improvement becomes insignificant relative to the time spent on training.

- Phase 3: Pruning

  This phase will focus on pruning the model, as discussed in Chapter 3, to optimize performance while maintaining accuracy with minimal impact.

- Phase 4: Benchmarking on Edge Device

  In this phase, the original and pruned models will be benchmarked on the edge device using the test dataset to evaluate their performance in real-world scenarios.

4.8 Phase 1: Initial Training and evaluation

For the initial training, all models were trained for minimum recommend cycles.

4.8.1 YOLO V3 Tiny and Yolo V7

For Darknet models, .data extension file needs to be made instead of .yaml extension in which file paths and amount of classes are inserted. Additionally, a names file is also made in which all of the names of the classes are inserted in order.The dataset has to be setup like this in order to train:

```
rdd
      |\_train
      |  \_data.names
      |   \_sample1.jpg
      |    \_sample1.txt
      |
      |\_valid
      |  \_data.names
      |   \_sample1.jpg
      |    \_sample1.txt
      |
      |\rdd.data
      |
      |\rdd_train.txt(list of all the images path
      | in the train folder)
      |
       \rdd_valid.txt(list of all the images path
       in the valid folder)
```

Then modification of the original configuration of the models needs to be done:

- Batch size and subdivision are set to 32 to make sure they run on this pc efficiently and correctly.
- The minimum recommended max batch value is 2000 multiplied by the number of classes, which in this class would be 8000

- Steps need to be set 80% and 90% of the max batch so that would be 6400 and 7200.
- Width and Height were set 640,640 pixels as said in Chapter 3.
- Classes is set to 4 as the dataset has 4 classes.
- For all of the filters in the convolutional prior to YOLO section in the configuration is set 27 based on the formula:

$$FilterValue = (numberofclasses + 5) * 3 \tag{4.1}$$

- Rest of the configuration were similar to the ones mentioned in specification chapter 3.

The following CLI commands are used to train Yolo v3 tiny and Yolo v7 with coco trained weights.

```
darknet detector -map -dont_show --verbose train rdd.data
yolov3tiny.cfg

darknet detector -map -dont_show --verbose train rdd.data
yolov7.cfg
```

After completing through the minium batches, Yolo v3 tiny gave a mAP50 score of 36.57% and yolo7 gave 41.23%.

4.9 Yolo11n, Yolo11n-OBB and Yolo11n-seg

For PyTorch models, No major modifications or changes were needed from what was described in Specifications chapter 3. These are the codes used to train the models yolo11n, yolo11n-obb, yolo11n-seg:

```
from ultralytics import YOLO  # build a new model from YAML

model = YOLO("yolo11n.yaml")
model = YOLO("yolo11n.pt")
model = YOLO("yolo11n.yaml").load("yolo11n.pt")

results = model.train(data="yolo-Dataset/data.yaml", epochs=100, imgsz=640, plots=True,
    optimizer='Adam', lr0=0.001, weight_decay=0.0005, momentum=0.937, device=0, val=True,
    lrf=0.001,resume=True)
```

Listing 4.1: Yolo11n Training code

```
from ultralytics import YOLO  # build a new model from YAML

model = YOLO("yolo11n-obb.yaml")
model = YOLO("yolo11n.pt")
model = YOLO("yolo11n-obb.yaml").load("yolo11n.pt")

results = model.train(data="yolo-obb-Dataset/data.yaml", epochs=100, imgsz=640, plots=True,
    optimizer='Adam', lr0=0.001, weight_decay=0.0005, momentum=0.937, device=0, val=True,
    lrf=0.001,resume=True)
```

Listing 4.2: Yolo11n-obb Training code

```
from ultralytics import YOLO  # build a new model from YAML

model = YOLO("yolo11n-seg.yaml")
model = YOLO("yolo11n.pt")
model = YOLO("yolo11n-seg.yaml").load("yolo11n.pt")

results = model.train(data="yolo-seg-Dataset/data.yaml", epochs=100, imgsz=640, plots=True,
    optimizer='Adam', lr0=0.001, weight_decay=0.0005, momentum=0.937, device=0, val=True,
    lrf=0.001,resume=True)
```

Listing 4.3: Yolo11n-seg Training code

After completing through the 100 epochs, Yolo11n gave a mAP50 score of 56.95%, yolo11n-obb gave 67.34% and yolo11n-seg gave 47.27%.

Here is a summary of the result of phase 1:

Model	**mAP50**
V3 Tiny	36.57%
V7	41.23%
V11	56.95%
V11 obb	67.34%
V11 seg	47.27%

Table 4.1: Phase 1 Results

After looking at the results Yolo11n-obb model was selected.

4.10 Phase 2: Final model training

4.10.1 Training

The model was trained for 450 epochs using the parameters as the previous phase to give a proper training period. Total training time was approximately 55 hours with mAP50 score of 76.819%.

Here are the Precision-Recall graph and F1 score graph after training:

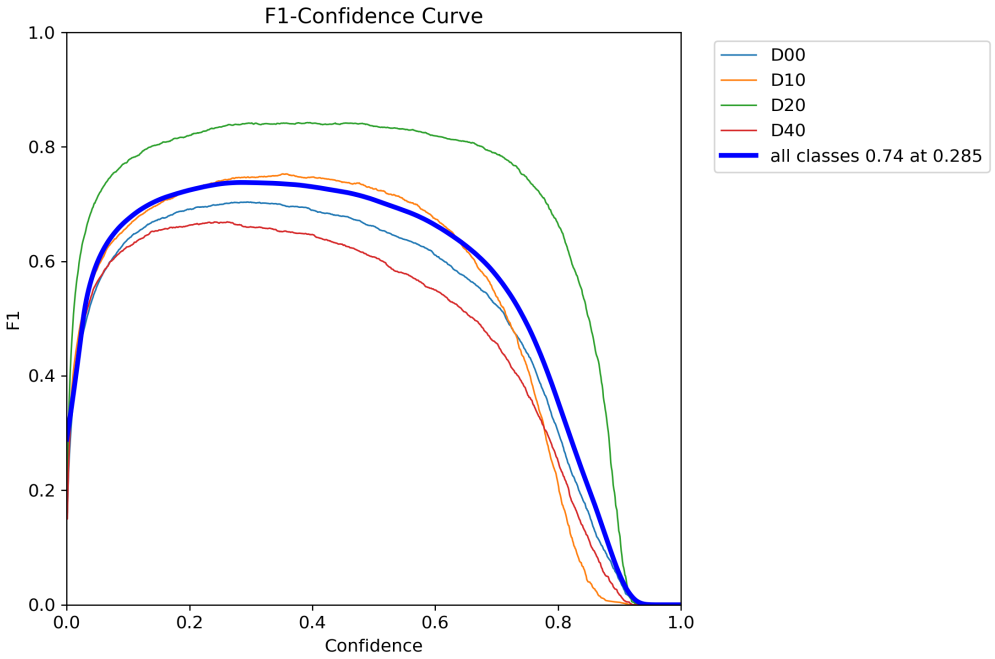


Figure 4.16: F1 Score Graph

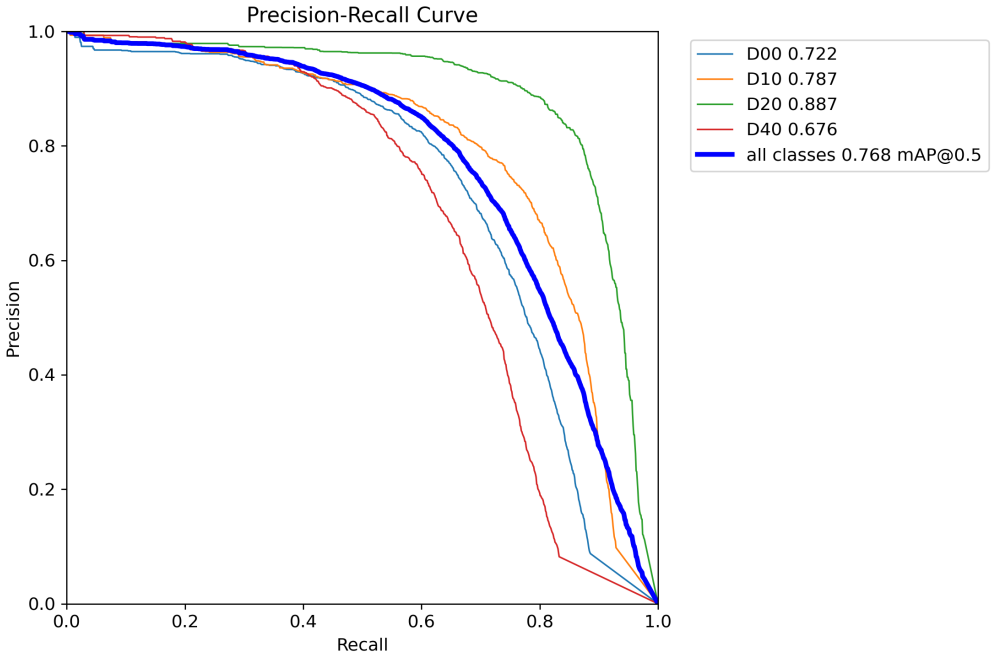


Figure 4.17: PR Graph

The figure 4.18 in the next page represents the network of the model.

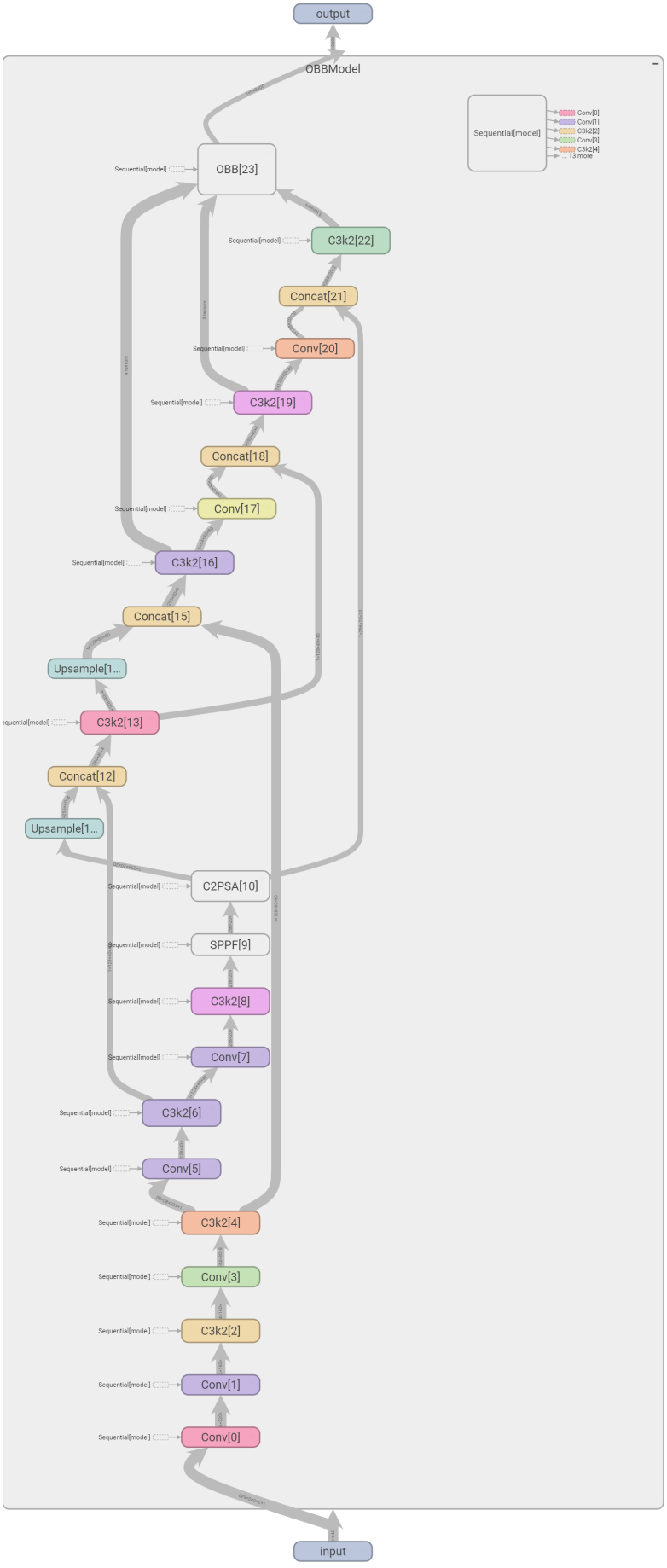


Figure 4.18: Yolo11n-OBB Model Network

4.11 Phase 3: Pruning

As Discussed in the Model Optimization Pruning Techniques section 3.6. The following code applies all of them and is modified from the code originally available at CSDN Blog by the user 数学人学python.
Implemented from the section original code from the Chinese website modified code version explanation of the code pruned at 10 and 20 show graphs model network

4.11.1 Imports and Environment Setup

```
from ultralytics import YOLO
import torch
from ultralytics.nn.modules import Bottleneck, Conv, C2f, SPPF, Detect, C3k2
from torch.nn.modules.container import Sequential
import os
import torch.nn.functional as F

# os.environ["CUDA_VISIBLE_DEVICES"] = "2"
```

The code begins by importing necessary libraries and modules. The `YOLO` class from `ultralytics` is used for the YOLO model. Various neural network modules and functionalities from `torch` are imported for model manipulation and functional operations.

4.11.2 PRUNE Class

```
class PRUNE():
    def __init__(self) -> None:
        self.threshold = None
```

The `PRUNE` class is defined to handle the pruning operations. The `__init__` method initializes the threshold to `None`.

Threshold Calculation

```
    def get_threshold(self, model, factor=0.9):
        ws = []
        bs = []
        for name, m in model.named_modules():
            if isinstance(m, torch.nn.BatchNorm2d):
                w = m.weight.abs().detach()
                b = m.bias.abs().detach()
                ws.append(w)
                bs.append(b)
                print(name, w.max().item(), w.min().item(), b.max().item(), b.min().item())
                print()
        ws = torch.cat(ws)
        self.threshold = torch.sort(ws, descending=True)[0][int(len(ws) * factor)]
```

The `get_threshold` method calculates the threshold value for pruning based on the weights of the BatchNorm layers (Section 3.7.1). It collects the weights and biases of all BatchNorm layers, sorts them, and sets the threshold to the value at a specified factor of the sorted weights.

Convolutional Layer Pruning

```
def prune_conv(self, conv1: Conv, conv2: Conv):
    gamma = conv1.bn.weight.data.detach()
    beta = conv1.bn.bias.data.detach()

    keep_idxs = []
    local_threshold = self.threshold
    while len(keep_idxs) < 8:
        keep_idxs = torch.where(gamma.abs() >= local_threshold)[0]
        local_threshold = local_threshold * 0.5
    n = len(keep_idxs)
    print(n / len(gamma) * 100)
    conv1.bn.weight.data = gamma[keep_idxs]
    conv1.bn.bias.data = beta[keep_idxs]
    conv1.bn.running_var.data = conv1.bn.running_var.data[keep_idxs]
    conv1.bn.running_mean.data = conv1.bn.running_mean.data[keep_idxs]
    conv1.bn.num_features = n
    conv1.conv.weight.data = conv1.conv.weight.data[keep_idxs]
    conv1.conv.out_channels = n

    if isinstance(conv2, list) and len(conv2) > 3 and conv2[-1]._get_name() == "Proto":
        proto = conv2.pop()
        proto.cv1.conv.in_channels = n
        proto.cv1.conv.weight.data = proto.cv1.conv.weight.data[:, keep_idxs]
    if conv1.conv.bias is not None:
        conv1.conv.bias.data = conv1.conv.bias.data[keep_idxs]

    if not isinstance(conv2, list):
        conv2 = [conv2]
    for item in conv2:
        if item is None: continue
        if isinstance(item, Conv):
            conv = item.conv
        else:
            conv = item
        if isinstance(item, Sequential):
            conv1 = item[0]
            conv = item[1].conv
            conv1.conv.in_channels = n
            conv1.conv.out_channels = n
            conv1.conv.groups = n
            conv1.conv.weight.data = conv1.conv.weight.data[keep_idxs, :]
            conv1.bn.bias.data = conv1.bn.bias.data[keep_idxs]
            conv1.bn.weight.data = conv1.bn.weight.data[keep_idxs]
            conv1.bn.running_var.data = conv1.bn.running_var.data[keep_idxs]
            conv1.bn.running_mean.data = conv1.bn.running_mean.data[keep_idxs]
            conv1.bn.num_features = n
        conv.in_channels = n
        conv.weight.data = conv.weight.data[:, keep_idxs]
```

The `prune_conv` method prunes the Convolutional layers based on the calculated threshold (Section 3.7.2). It modifies the weights, biases, and other parameters of the BatchNorm and Convolutional layers to retain only the important filters.

Sequential Pruning

```
def prune(self, m1, m2):
    if isinstance(m1, C3k2):
        m1 = m1.cv2
    if isinstance(m1, Sequential):
        m1 = m1[1]
    if not isinstance(m2, list):
        m2 = [m2]
    for i, item in enumerate(m2):
        if isinstance(item, C3k2) or isinstance(item, SPPF):
            m2[i] = item.cv1

    self.prune_conv(m1, m2)
```

The `prune` method handles sequential pruning (Section 3.7.3). It ensures that layers like `C3k2` and `SPPF` are pruned while maintaining the model architecture.

4.11.3 Distillation Loss

```
def distillation_loss(student_output, teacher_output, labels, temperature=3.0, alpha=0.5):
    soft_labels = F.softmax(teacher_output / temperature, dim=1)
    soft_loss = F.kl_div(F.log_softmax(student_output / temperature, dim=1), soft_labels,
        reduction='batchmean') * (temperature ** 2)
    hard_loss = F.cross_entropy(student_output, labels)
    return alpha * soft_loss + (1 - alpha) * hard_loss
```

The `distillation_loss` function calculates the loss for knowledge distillation (Section 3.7.4). It combines the soft loss (KL divergence) and hard loss (cross-entropy) to train the student model to mimic the teacher model.

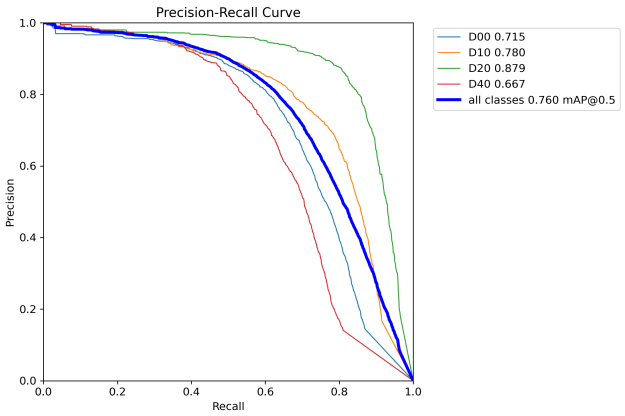
4.11.4 Pruning Process

```
def do_pruning(modelpath, savepath):
    pruning = PRUNE()

    yolo = YOLO(modelpath)
    pruning.get_threshold(yolo.model, [Ratio of the pruning needed])

    for name, m in yolo.model.named_modules():
        if isinstance(m, Bottleneck):
            pruning.prune_conv(m.cv1, m.cv2)

    seq = yolo.model.model
    for i in [3, 5, 7, 8]:
        pruning.prune(seq[i], seq[i + 1])

    detect: Detect = seq[-1]
    proto = getattr(detect, 'proto', None)
    last_inputs = [seq[16], seq[19], seq[22]]
    colasts = [seq[17], seq[20], None]
    for idx, (last_input, colast, cv2, cv3, cv4) in enumerate(zip(last_inputs, colasts,
        detect.cv2, detect.cv3, detect.cv4)):
        if idx == 0:
            pruning.prune(last_input, [colast, cv2[0], cv3[0], cv4[0], proto] if proto else
                [colast, cv2[0], cv3[0], cv4[0]])
        else:
            pruning.prune(last_input, [colast, cv2[0], cv3[0], cv4[0]])
        pruning.prune(cv2[0], cv2[1])
        pruning.prune(cv2[1], cv2[2])
        pruning.prune(cv3[0], cv3[1])
        pruning.prune(cv3[1], cv3[2])
        pruning.prune(cv4[0], cv4[1])
        pruning.prune(cv4[1], cv4[2])

    for name, p in yolo.model.named_parameters():
        p.requires_grad = True

    yolo.val(data='Balanced-Dataset/data.yaml', batch=2, device=0, workers=0)
    torch.save(yolo.ckpt, savepath)

if __name__ == "__main__":
    modelpath = "runs/obb/train8/weights/last.pt"
    savepath = "runs/obb/train8/weights/pruned_model.pt"
    do_pruning(modelpath, savepath)
```

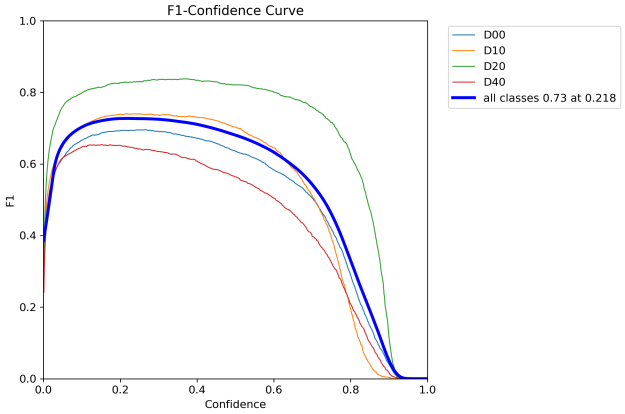
The `do_pruning` function orchestrates the pruning process. It loads the YOLO model, calculates the threshold, and applies pruning to various layers. Finally, it saves the pruned model.

4.11.5 Pruning results:

The model was pruned at 90% and 80% and here are the results:

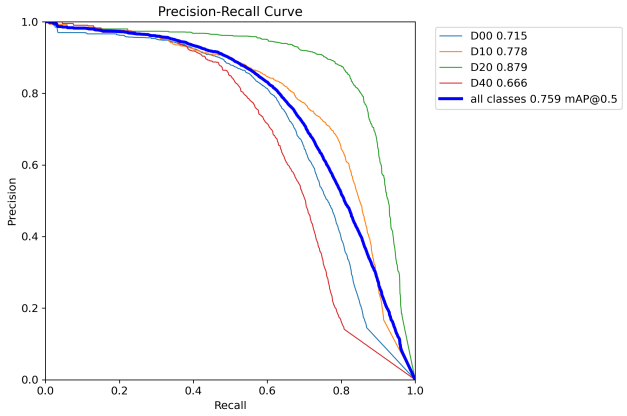


((a)) PR Curve

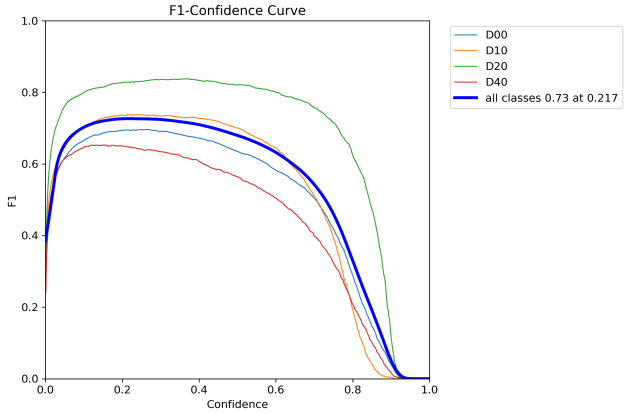


((b)) F1 Curve

Figure 4.19: At 90% Pruning Results



((a)) PR Curve



((b)) F1 Curve

Figure 4.20: At 80% Pruning Results

4.12 Phase 4: Benchmarking on Edge Device

4.12.1 Installation of Ubuntu and Dependencies

Firstly using Raspberry Pi Imager on windows, the SD card is flashed with Ubuntu 24.04 LTS OS for RPI5.

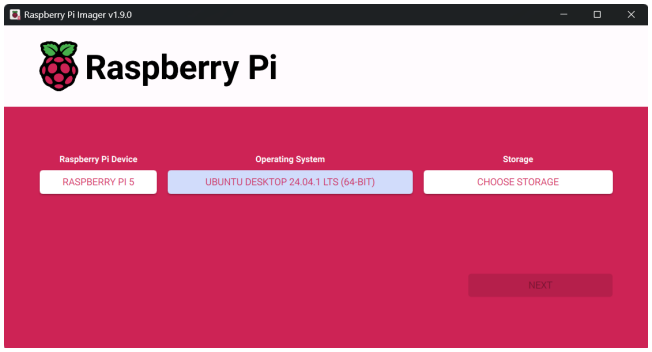


Figure 4.21: Raspberry Pi Imager

After that, Ultralytics was installed for benchmarking by following the instructions provided by Aleksandar Haber on his website.

Installing Git and Git LFS:

```
sudo apt update && sudo apt upgrade
sudo apt install git
sudo apt-get install git-lfs
```

Initializing Git LFS:

```
sudo git lfs install
git lfs install
```

Creating a workspace folder and a Python virtual environment:

```
sudo apt install python3.12-venv
cd ~
mkdir testYolo
cd testYolo
python3 -m venv env1
source env1/bin/activate
```

Installing the necessary libraries:

```
pip install setuptools
pip install git+https://github.com/ultralytics/ultra-
lytics.git@main
```

Now once this was done the test dataset was renamed to valid and and copied to Raspberry pi 5 in directory.

4.12.2 Benchmarking

The original and pruned models are benchmarked using the following command:

```
yolo benchmark model=weights/{MODEL NAME}
data='Balanced-Dataset/data.yaml' imgsz=640
```

This command is inserted the terminal after activating the enviorment and then it exports the models in various formats such as TorchScript, ONNX and OpenVINO to name a few. Once exported they are benchmarked on the test dataset and mAP50 scores and average FPS results are given.

The whole benchmarking is done over ssh to bring out the best performance possible from the device as graphical output to the display takes quite a few resources.

4.13 Results

4.13.1 Performance Metrics Overview

The performance of the YOLOv11n-OBB model was assessed based on two key metrics:

- mAP50 (Mean Average Precision at 50% IoU) – Measures model accuracy for detecting road damages.
- FPS (Frames Per Second) – Evaluates the real-time feasibility of the model on the edge device.

4.13.2 Benchmarking on Edge Device

The original and pruned models were tested on Raspberry Pi 5 using the test dataset. The results were as follows:

Format	mAP50	Interference Time (ms/im)	FPS	Size (MB)
PyTorch	76.90%	604	1.66	15.9
TorchScript	75.60%	619.71	1.61	10.7
ONNX	75.60%	244.89	4.08	10.3
OpenVINO	75.50%	111.63	8.96	10.5
PaddlePaddle	75.60%	698.57	1.43	20.7
MNN	75.60%	141.99	7.04	10.2
NCNN	75.50%	104.65	9.56	10.2

Table 4.2: Orignal Model Results

Format	mAP50	Interference Time (ms/im)	FPS	Size (MB)
PyTorch	76.20%	603.53	1.66	19.9
TorchScript	74.70%	602.11	1.66	9.8
ONNX	74.70%	221.72	4.51	9.5
OpenVINO	74.60%	103.12	9.7	9.7
PaddlePaddle	74.70%	614.84	1.63	19
MNN	74.60%	135.1	7.4	9.4
NCNN	74.60%	100.04	10	9.4

Table 4.3: Pruned at 90% Results

Format	mAP50	Interference Time (ms/im)	FPS	Size (MB)
PyTorch	76%	489.97	2.04	19.2
TorchScript	74.60%	485.25	2.06	9.1
ONNX	74.60%	181.2	5.52	8.7
OpenVINO	74.50%	79.97	12.5	8.9
PaddlePaddle	74.60%	552.02	1.81	17.5
MNN	74.60%	109.7	9.12	8.6
NCNN	74.60%	103.49	9.66	8.7

Table 4.4: Pruned at 80% Results

4.13.3 Analysis of Model Performance

Original Model Performance: The original YOLOv11n-OBB model achieved a mAP50 of 75.50% and a frame processing rate of 9.56 FPS using the NCNN format. These results indicate that the model can efficiently detect road damage while maintaining real-time processing capabilities. The relatively high accuracy suggests that the model effectively differentiates between different crack types and potholes, making it suitable for deployment but it does not have any buffer in the case of delay in pre-processing of the images. This makes the model applicable to be used during manual inspection to detect damages which might be overseen by the inspectors due to human error.

Impact of Pruning at 90%:
After applying 90% pruning, the model's FPS increased to 10 FPS, while the mAP50 slightly decreased to 74.60%. This trade-off shows that pruning can effectively improve inference speed with a negligible drop in detection accuracy. The increase in FPS ensures smoother real-time detection, making this pruned model optimal for cities with lower limit of driving speed and to handle any delays introduced by other parts of the system.

Impact of Pruning at 80%:
Further pruning at 80% significantly enhances FPS, reaching 12.5 FPS, but results in a minor accuracy reduction to 74.50%. This version of the model is best suited for the cities with higher speed limits and also handling the delays introduced by other processes of the system.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis has presented the possibility of running a light and efficient AI-powered road damage detection system on resource-restricted edge devices such as Raspberry Pi 5 using the ITS-AI-IoT architecture. This proposed solution addresses to the serious challenges of infrastructure monitoring and road maintenance, cost-effectively and scalable. In this study, the deep learning model YOLOv11n-OBB was utilized, which has already been optimized for real-time object detection tasks, exhibiting an mAP50 of 76.819% with an inference speed of 9.56 FPS and additional FPS have been gained with minium losses in accuracy. This is due to the creation of a strong dataset developed from a combination of RDD2022 and Kaggle pothole data, along with other efficient training and pruning techniques.

The deployed edge-based approach renders the capability of real-time processing and reporting road damages such as longitudinal cracks, transverse cracks, alligator cracks, and potholes. Results confirm that lightweight AI models achieve a good balance between computation efficiency and detection accuracy, hence scalable deployment in urban areas. This research work will contribute significantly to improving road safety and reducing road maintenance costs by using an automated road damage detection and reporting approach.

5.2 Future Work

Though the results look promising, several limitations are identified and some avenues for improvement are presented below. These will be the basis for future research:

- Dataset Expansion: The existing model is trained on a dataset that is representative, especially for urban road conditions. The addition of images from a variety of weather conditions, such as rain, fog, or poor illumination and low light or night-time conditions, would increase the generality and

robustness of the model.

- Night Vision and Enviro-Adaptability: Infrared camera technology or low-light imaging can be utilized for improving the performance during poor illuminating conditions. In addition, consideration for extreme weather will make the system adaptable.
- Highway and Rural Deployment: This system is developed for urban roads with a speed of approximately 60 km/hr. Highways and Rural in future: Implementation on highways and rural regions involves much increased speeds and changing topography, and hence will present new challenges.
- Severity Analysis: Implements the feature to categorize the road damages in several classes according to the severity. In this manner, the repair assets will be utilized in a much improved manner. Integration with the historic information will enable prediction of when maintenance will be necessitated and the trend.

Addressing these items, the proposed system would expand to become an overall system for road surface monitoring, delivering best value for sustainable infrastructure and citizen safety.

References

[1] L. Ale, N. Zhang, and L. Li. Road damage detection using retinanet. In *Proc. of IEEE International Conference on Big Data (Big Data)*, 2018.

[2] Laha Ale, Ning Zhang, and Longzhuang Li. Road damage detection using retinanet. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 5197–5200, 2018.

[3] Deeksha Arya, Hiroya Maeda, Sanjay Kumar Ghosh, Durga Toshniwal, Alexander Mraz, Takehiro Kashiyama, and Yoshihide Sekimoto. Deep learning-based road damage detection and classification for multiple countries. *Automation in Construction*, 122:103492, 2021.

[4] Deeksha Arya, Hiroya Maeda, Yoshihide Sekimoto, Hiroshi Omata, Sanjay Kumar Ghosh, Durga Toshniwal, Madhavendra Sharma, Van Vung Pham, Jingtao Zhong, Muneer Al-Hammadi, Mamoona Birkhez Shami, Du Nguyen, Hanglin Cheng, Jing Zhang, Alex Klein-Paste, Helge Mork, Frank Lindseth, Toshikazu Seto, Alexander Mraz, and Takehiro Kashiyama. RDD2022 - The multi-national Road Damage Dataset released through CRDDC'2022. 10 2022.

[5] DenisG04. Pothole dataset v8 for detection. `https://www.kaggle.com/datasets/denisg04/pothle-detect`, 2023.

[6] Keval Doshi and Yasin Yilmaz. Road damage detection using deep ensemble learning. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 5197–5200, 2020.

[7] Christian Gorges, Kemal Öztürk, and Robert Liebich. Impact detection using a machine learning approach and experimental road roughness classification. *Mechanical Systems and Signal Processing*, 117:738–756, 2018.

[8] Jongwoo Ha, Dongsoo Kim, and Minsoo Kim. Assessing severity of road cracks using deep learning-based segmentation and detection. *The Journal of Supercomputing*, 78:17721–17735, 2022.

[9] Glenn Jocher and Jing Qiu. Ultralytics yolo11, 2024.

[10] Rahima Khanam and Muhammad Hussain. Yolov11: An overview of the key architectural enhancements. *arXiv preprint arXiv:2410.17725*, 2024.

[11] Hiroya Maeda, Yoshihide Sekimoto, Toshikazu Seto, Takehiro Kashiyama, and Hiroshi Omata. Road damage detection and classification using deep neural networks with smartphone images. *Computer-Aided Civil and Infrastructure Engineering*, 33:1–15, 2018.

[12] Ahmed Nahmad. Modified-rdd2022-dataset. `https://github.com/nahmad2000/Modified-RDD2022-Dataset`, 2025.

[13] Rafael Padilla, Wesley L. Passos, Thadeu L. B. Dias, Sergio L. Netto, and Eduardo A. B. da Silva. A comparative analysis of object detection metrics with a companion open-source toolkit. *Electronics*, 10(3):279, 2021.

[14] V. Pham, C. Pham, and T. Dang. Road damage detection and classification with detectron2 and faster r-cnn. In *2020 IEEE International Conference on Big Data (Big Data)*, 2020.

[15] Linda M. Pierce, Sarah E. Lopez, Jose R. Medina, and Vivek Jha. *AI Applications for Automatic Pavement Condition Evaluation.* National Academies of Sciences, Engineering, and Medicine, 2024.

[16] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[17] Hussein Samma, Shahrel Azmin Suandi, Nor Azman Ismail, Sarina Sulaiman, and Lee Li Ping. Evolving pre-trained cnn using two-layers optimizer for road damage detection from drone images. *IEEE Access*, 9:158215–158226, 2021.

[18] Luís Augusto Silva, Valderi Reis Quietinho Leithardt, Vivian Félix López Batista, Gabriel Villarrubia González, and Juan Francisco De Paz Santana. Automated road damage detection using uav images and deep learning techniques. *IEEE Access*, 11:62918–62931, 2023.

[19] Iona Stewart and Michael Benson. Potholes and local road maintenance funding, 2024.

[20] MINISTRY OF ROAD TRANSPORT and HIGHWAYS TRANSPORT RESEARCH WING NEW DELHI. ROAD ACCIDENTS IN INDIA 2021. Technical report, 12 2022.

[21] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:2207.02696*, 2022.

[22] X. Yu and E. Salari. Pavement pothole detection and severity measurement using laser imaging. In *2011 IEEE International Conference on Electro/Information Technology*, pages 1–5, Mankato, MN, USA, 2011. IEEE.

[23] Hongwei Zhang, Zhaohui Wu, Yuxuan Qiu, Xiangcheng Zhai, Zichen Wang, Peng Xu, Zhenzheng Liu, Xiantong Li, and Na Jiang. A new road damage detection baseline with attention learning. *Applied Sciences*, 12:7594, 2022.

Appendix A

Appendix

A.1 Code Snippets

```
import os
import xml.etree.ElementTree as ET
from pathlib import Path
from tqdm import tqdm

class XMLToYOLOConverter:
    def __init__(self, dataset_path):
        self.dataset_path = Path(dataset_path)
        self.labels_path = self.dataset_path / 'labels'
        self.output_path = self.dataset_path / 'txt_labels'
        self.class_mapping = {}
        self.next_class_id = 0

    def get_class_id(self, class_name):
        if class_name not in self.class_mapping:
            self.class_mapping[class_name] = self.next_class_id
            self.next_class_id += 1
        return self.class_mapping[class_name]

    def convert_coordinates(self, size, box):

        dw = 1.0 / size[0]
        dh = 1.0 / size[1]

        # Extract coordinates
        xmin = float(box.find('xmin').text)
        xmax = float(box.find('xmax').text)
        ymin = float(box.find('ymin').text)
        ymax = float(box.find('ymax').text)

        # Calculate YOLO coordinates
        x_center = ((xmin + xmax) / 2.0) * dw
        y_center = ((ymin + ymax) / 2.0) * dh
        w = (xmax - xmin) * dw
        h = (ymax - ymin) * dh

        return x_center, y_center, w, h

    def convert_xml_to_yolo(self, xml_path):

        tree = ET.parse(xml_path)
        root = tree.getroot()

        # Get image size
        size = root.find('size')
        width = int(size.find('width').text)
        height = int(size.find('height').text)

        # Process each object
        yolo_lines = []
        for obj in root.findall('object'):
            class_name = obj.find('name').text
            class_id = self.get_class_id(class_name)
```

```
            # Get bounding box
            bbox = obj.find('bndbox')
            x_center, y_center, w, h = self.convert_coordinates((width, height), bbox)

            # Format YOLO line: <class_id> <x_center> <y_center> <width> <height>
            yolo_lines.append(f"{class_id} {x_center:.6f} {y_center:.6f} {w:.6f} {h:.6f}")

        return yolo_lines

    def process_dataset(self):
        # Create output directory
        self.output_path.mkdir(parents=True, exist_ok=True)

        # Get list of XML files
        xml_files = list(self.labels_path.glob('*.xml'))
        if not xml_files:
            raise ValueError("No XML files found in the labels directory!")

        print("\nConverting XML files to YOLO format...")

        # Process each XML file with progress bar
        for xml_file in tqdm(xml_files, desc="Converting files", unit="file"):
            try:
                # Convert XML to YOLO format
                yolo_lines = self.convert_xml_to_yolo(xml_file)

                # Save YOLO format to txt file
                txt_filename = xml_file.stem + '.txt'
                txt_path = self.output_path / txt_filename

                with open(txt_path, 'w') as f:
                    f.write('\n'.join(yolo_lines))

            except Exception as e:
                print(f"\nError processing {xml_file.name}: {e}")
                continue

        # Save class mapping
        self.save_class_mapping()

        return len(xml_files)

    def save_class_mapping(self):

        mapping_path = self.output_path / 'classes.txt'

        # Sort classes by their ID
        sorted_classes = sorted(self.class_mapping.items(), key=lambda x: x[1])

        with open(mapping_path, 'w') as f:
            for class_name, class_id in sorted_classes:
                f.write(f"{class_name}: {class_id}\n")

def main():
    # Get dataset path from user
    dataset_path = input("Enter the path to your dataset directory: ")

    try:
        converter = XMLToYOLOConverter(dataset_path)
        num_files = converter.process_dataset()

        print("\nConversion Summary:")
        print("="*50)
        print(f"Total files processed: {num_files}")
        print(f"Number of classes found: {len(converter.class_mapping)}")
        print("\nClass Mapping:")
        for class_name, class_id in sorted(converter.class_mapping.items(), key=lambda x:
            x[1]):
            print(f"  {class_name}: {class_id}")
        print(f"\nOutput files saved to: {converter.output_path}")
        print("\nConversion completed successfully!")

    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
```

```
    main()
```

Listing A.1: converting from xml to txt format

```
import os
from tqdm import tqdm

def convert_to_obb(yolo_label):
    # Assuming the YOLO label format is: class_index x_center y_center
        width height
    class_index, x_center, y_center, width, height = map(float,
        yolo_label.split())

    # Calculate the coordinates of the four corners of the bounding box
    x1 = x_center - width / 2
    y1 = y_center - height / 2
    x2 = x_center + width / 2
    y2 = y_center - height / 2
    x3 = x_center + width / 2
    y3 = y_center + height / 2
    x4 = x_center - width / 2
    y4 = y_center + height / 2

    return f"{int(class_index)} {x1} {y1} {x2} {y2} {x3} {y3} {x4} {y4}"

def convert_labels_to_obb(dataset_folder):
    labels_folder = os.path.join(dataset_folder, 'labels')
    obb_labels_folder = os.path.join(dataset_folder, 'obb_labels')

    # Create the obb_labels folder if it doesn't exist
    os.makedirs(obb_labels_folder, exist_ok=True)

    label_files = [f for f in os.listdir(labels_folder) if
        f.endswith('.txt')]

    for label_file in tqdm(label_files, desc="Converting labels"):
        with open(os.path.join(labels_folder, label_file), 'r') as f:
            yolo_labels = f.readlines()

        obb_labels = [convert_to_obb(label.strip()) for label in
            yolo_labels]

        with open(os.path.join(obb_labels_folder, label_file), 'w') as f:
            f.write('\n'.join(obb_labels))

    print(f"Converted labels saved in {obb_labels_folder}")

# Example usage
dataset_folder = 'Balanced-Dataset/val'
convert_labels_to_obb(dataset_folder)
```

Listing A.2: converting from yolo to yolo obb format

```
import os
import yaml
from tqdm import tqdm

def obb_to_seg(obb_file, seg_file):
    with open(obb_file, 'r') as f:
        lines = f.readlines()

    with open(seg_file, 'w') as f:
        for line in lines:
            parts = line.strip().split()
```

```
            class_index = parts[0]
            coords = parts[1:]
            # Convert OBB to SEG format
            seg_coords = ' '.join(coords)
            f.write(f"{class_index} {seg_coords}\n")

def process_annotations(input_dir, output_dir):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    files = [f for f in os.listdir(input_dir) if f.endswith('.txt')]
    for file_name in tqdm(files, desc='Converting'):
        obb_file = os.path.join(input_dir, file_name)
        seg_file = os.path.join(output_dir, file_name)
        obb_to_seg(obb_file, seg_file)

def process_yaml(data_yaml):
    with open(data_yaml, 'r') as f:
        data = yaml.safe_load(f)

    # Directories for train, val, and test
    base_dir = os.path.dirname(data_yaml)
    train_dir = os.path.join(base_dir, 'train', 'labels')
    val_dir = os.path.join(base_dir, 'val', 'labels')
    test_dir = os.path.join(base_dir, 'test', 'labels')

    output_dirs = {
        'train': os.path.join(base_dir, 'seg_labels', 'train'),
        'val': os.path.join(base_dir, 'seg_labels', 'val'),
        'test': os.path.join(base_dir, 'seg_labels', 'test')
    }

    # Process annotations for train, val, and test
    process_annotations(train_dir, output_dirs['train'])
    process_annotations(val_dir, output_dirs['val'])
    process_annotations(test_dir, output_dirs['test'])

# Example usage
data_yaml = 'Balanced-Dataset/data.yaml'
process_yaml(data_yaml)
```

Listing A.3: converting from obb to seg format

```
import os
import cv2
from tqdm import tqdm

# Paths
base_path = './Balanced-Dataset'

# Directories to process
subsets = ['train', 'val', 'test']

# Function to convert images to grayscale
def convert_to_grayscale(image_path):
    image = cv2.imread(image_path)
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    cv2.imwrite(image_path, gray_image)

# Process images
for subset in subsets:
    image_dir = os.path.join(base_path, subset, 'images')
    image_files = [f for f in os.listdir(image_dir) if f.endswith('.jpg')]

    for image_file in tqdm(image_files, desc=f"Converting {subset} images to grayscale"):
        image_path = os.path.join(image_dir, image_file)
        convert_to_grayscale(image_path)

print("All images have been converted to grayscale!")
```

Listing A.4: Converted Images in Gray-scale Images

```
import os
import shutil
import random
from collections import defaultdict
```

```
from tqdm import tqdm

# Paths
data_path = "E:\Aditya_Thesis_Project_Data_Backup\combined dataset - Copy"
image_path = os.path.join(data_path, 'images')
label_path = os.path.join(data_path, 'labels')
output_path = './Balanced-Dataset'
train_ratio = 0.8
val_ratio = 0.1
test_ratio = 0.1

# Create directories
os.makedirs(os.path.join(output_path, 'train/images'), exist_ok=True)
os.makedirs(os.path.join(output_path, 'train/labels'), exist_ok=True)
os.makedirs(os.path.join(output_path, 'val/images'), exist_ok=True)
os.makedirs(os.path.join(output_path, 'val/labels'), exist_ok=True)
os.makedirs(os.path.join(output_path, 'test/images'), exist_ok=True)
os.makedirs(os.path.join(output_path, 'test/labels'), exist_ok=True)

# Read all labels and categorize by class
label_files = [f for f in os.listdir(label_path) if f.endswith('.txt')]
class_distribution = defaultdict(list)

for label_file in tqdm(label_files, desc="Reading labels"):
    with open(os.path.join(label_path, label_file), 'r') as file:
        lines = file.readlines()
        if not lines:  # Blank images
            class_distribution['blank'].append(label_file)
        for line in lines:
            class_id = line.split()[0]
            class_distribution[class_id].append(label_file)

# Get the minimum number of objects across all classes
min_objects = min(len(files) for files in class_distribution.values() if files != 'blank')

# Balance the dataset by oversampling/undersampling
balanced_files = []
for class_id, files in tqdm(class_distribution.items(), desc="Balancing dataset"):
    if class_id == 'blank':
        continue
    if len(files) > min_objects:
        balanced_files.extend(random.sample(files, min_objects))
    else:
        balanced_files.extend(files + random.choices(files, k=min_objects - len(files)))

# Include blank images
balanced_files.extend(class_distribution['blank'])

# Split the dataset
random.shuffle(balanced_files)
num_files = len(balanced_files)
train_files = balanced_files[:int(train_ratio * num_files)]
val_files = balanced_files[int(train_ratio * num_files):int((train_ratio + val_ratio) *
    num_files)]
test_files = balanced_files[int((train_ratio + val_ratio) * num_files):]

def copy_files(file_list, subset):
    for file in tqdm(file_list, desc=f"Copying files to {subset}"):
        image_file = file.replace('.txt', '.jpg')
        shutil.copy(os.path.join(image_path, image_file), os.path.join(output_path,
            f'{subset}/images', image_file))
        shutil.copy(os.path.join(label_path, file), os.path.join(output_path,
            f'{subset}/labels', file))

copy_files(train_files, 'train')
copy_files(val_files, 'val')
copy_files(test_files, 'test')

print("Dataset balanced and split successfully!")
```

Listing A.5: Splitting the dataset