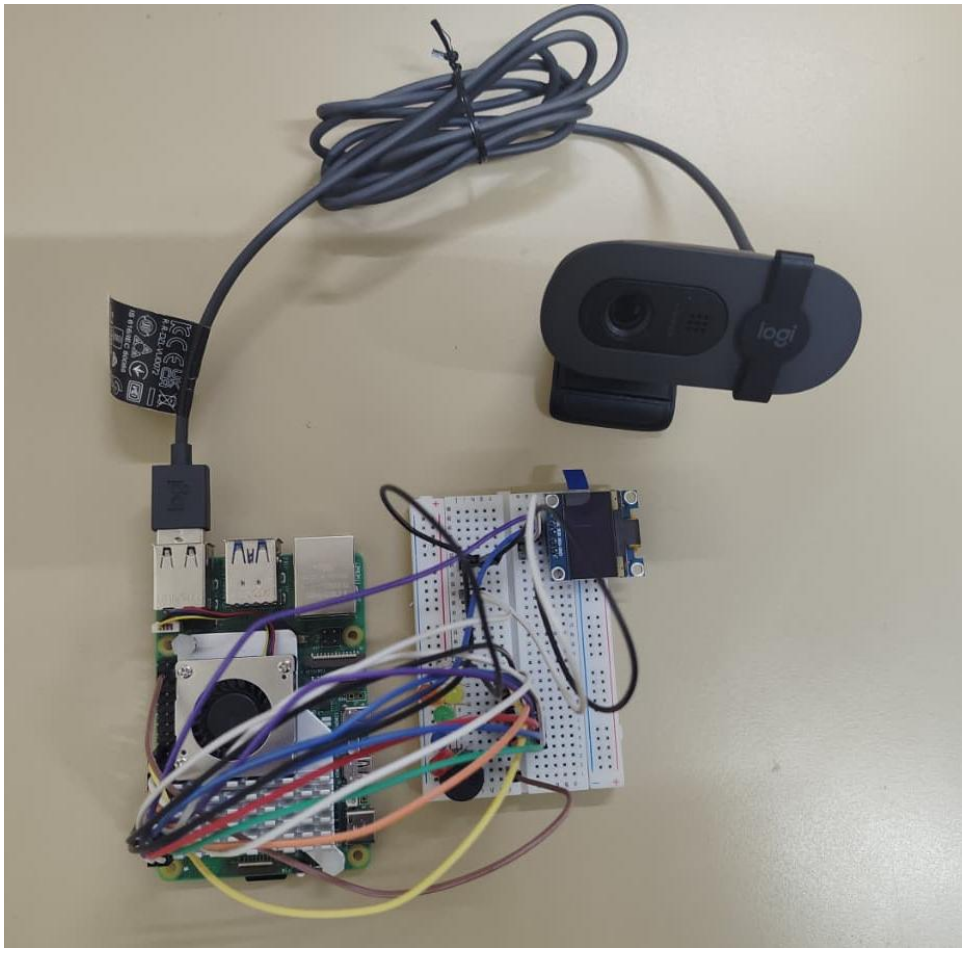
Bharat AI-SoC Student Challenge

PS 3- Real-Time Road Anomaly Detection from Dashcam Footage on Raspberry Pi



Submitted By:-

Student Name :-Dev Gogia

Institute Name:- Jaypee Institute Of Information Technology

Mentor Name:- Dr. Shruti Kalra

ABSTRACT

This Intelligent Road Monitoring System is an edge-based artificial intelligence solution designed to detect road hazards in real time using Raspberry Pi 5. The system integrates a custom-trained deep learning model capable of identifying six critical road-related classes: potholes, cracks, obstacles, animals, persons, and vehicles.

The final model was trained on over 50,000 labelled images for 100 epochs using Kaggle's NVIDIA Tesla P100 GPU. The model achieved approximately 77% precision and 72% mAP@50, demonstrating strong detection capability across diverse real-world scenarios. To enable efficient embedded deployment, the trained model was optimized using INT8 quantization and configured specifically for ARM architecture.

The system operates fully offline and processes live video input from a USB webcam. It provides real-time visual and hardware feedback through GPIO-based LEDs, a buzzer alert system for potholes, and an OLED display for class and confidence visualization. Multi-threaded architecture ensures stable FPS performance and prevents blocking operations during logging or display updates.

Through iterative dataset refinement, model retraining, performance optimization, and on-device testing, the project evolved into a stable and deployable embedded AI system suitable for real-world road monitoring applications.

INTRODUCTION

Road infrastructure monitoring plays a critical role in ensuring driver safety and maintaining transportation efficiency. Traditional inspection methods rely heavily on manual observation or centralized cloud-based systems, both of which introduce delays, scalability challenges, and operational costs.

With advancements in edge computing and embedded AI, it has become possible to perform intelligent analysis directly on low-power devices without cloud dependency. This project explores the development of a real-time road hazard detection system deployed entirely on Raspberry Pi 5.

The primary objective of the system is to detect potholes accurately, while also identifying other hazards such as cracks, obstacles, animals, pedestrians, and vehicles. Unlike systems that focus solely on pothole detection, this project aims to provide comprehensive road awareness to assist drivers in dynamic environments.

To achieve this, a large-scale dataset comprising over 50,000 images was constructed and used to train a deep learning model for more than 100 epochs. The model was optimized for embedded deployment using INT8 quantization and ARM-compatible inference techniques.

Beyond model development, the system integrates:

- Real-time USB camera input
- Hardware feedback through GPIO LEDs
- Dedicated pothole buzzer alert
- OLED display for driver information
- Asynchronous logging system
- Thermal and FPS monitoring

The result is a fully offline, edge-based AI solution that combines deep learning, embedded systems engineering, and real-time optimization into a stable and deployable intelligent road monitoring platform.

DEVELOPMENT

1. System Architecture: Hardware and Software Stack

The development of the Intelligent Road Monitoring System was approached as a complete embedded AI engineering project, integrating optimized software design with carefully selected hardware components. The goal was to build a stable, real-time, edge-deployable system capable of sustained operation under realistic driving conditions.

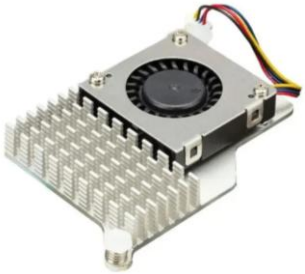
Hardware Platform

The final deployment platform was the Raspberry Pi 5, selected for its improved processing capability compared to earlier Pi generations. The Pi 5's Arm-based architecture provided sufficient computational power to execute real-time object detection while simultaneously handling hardware interfacing and logging operations.

The complete hardware stack included:

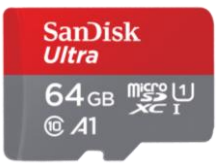
- Raspberry Pi 5
- Logitech Brio 100 USB Webcam (real-time video input)
- SSD1306 OLED Display (I2C interface)
- Multiple GPIO-connected LEDs (class indicators)
- Active Buzzer (dedicated pothole alert)
- Active Cooling System (fan + heat sink)
- High-speed microSD card for logging

Raspberry Pi 5



Raspberry Pi Active Cooler

Logitech Brio 100



MicroSD Card

SS1306 OLED Display

Multiple Color LEDs

Active Buzzer

Each detected class was mapped to a dedicated GPIO output pin. The LEDs were color-matched with their respective bounding box colors on the display to ensure intuitive visual feedback. This mapping was intentionally designed so that physical indicators and on-screen detection remain consistent and easily interpretable.

Because pothole detection was the primary objective of the project, a buzzer alert mechanism was implemented specifically for pothole events. A timed logic control ensures that the buzzer provides immediate warning without continuous or disruptive activation.

To enhance usability, a dedicated OLED display continuously presents:

- Detected class name
- Confidence percentage
- Current system state (e.g., “SCANNING”)

This allows the driver to monitor system behavior without constantly observing the main video display.

Software Stack

The software stack was designed to support efficient inference and robust hardware integration on an embedded Linux environment.

The system was developed using:

- Raspberry Pi OS (64-bit)
- Python 3.8+
- Ultralytics YOLO framework
- TensorFlow Lite (for optimized inference)
- OpenCV 4.x (video capture and visualization)
- NumPy (array operations)
- RPi.GPIO (hardware control)
- Luma OLED library (OLED rendering)
- Python threading and queue modules (asynchronous processing)

The model was exported in `.tflite` format to ensure compatibility with Arm architecture and to enable efficient inference on the Raspberry Pi 5.

2. Dataset Engineering and Model Training

Training Environment

All model training was conducted on the Kaggle cloud platform using an NVIDIA Tesla P100 GPU. The P100 GPU enabled large-scale training with extended epoch schedules while maintaining efficient computation time.

This cloud-based training environment allowed:

- Handling of large datasets (50,000+ images)
- Multi-day training sessions
- Consistent GPU acceleration
- Stable experimentation workflow

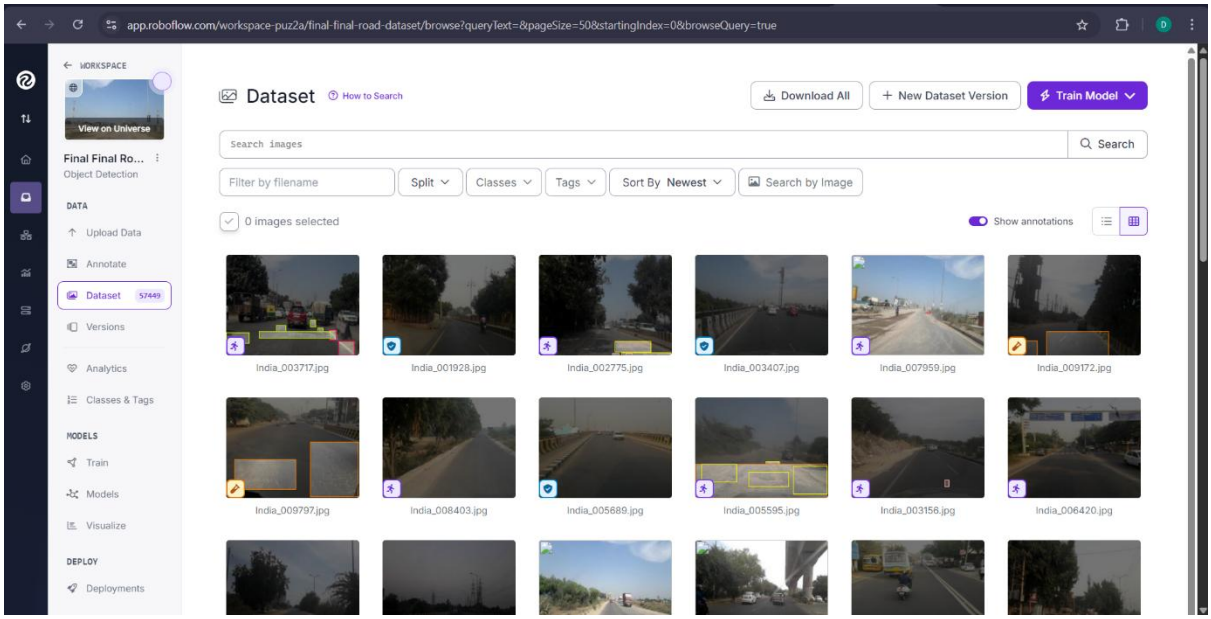
Dataset Construction

To ensure real-world robustness, a comprehensive dataset was constructed by combining multiple sources:

- Global Road Damage Detection Challenge (GRDDC)
- India Driving Dataset (IDD)
- Multiple curated datasets from Roboflow

The final dataset consisted of:

Over 50,000 labeled images



Customized Dataset in Roboflow

The dataset was carefully divided into six balanced categories:

1. Pothole
2. Crack
3. Obstacle
4. Animal
5. Person
6. Vehicle

Class balance was treated as a priority during dataset preparation. Equal representation was maintained to prevent class bias, particularly overfitting toward potholes. Although potholes were the primary objective, the system was designed to detect a wide range of real-world hazards encountered during driving.

The dataset incorporated diverse conditions including:

- Daylight and low-light environments
- Urban and rural roads
- Shadows, glare, and uneven illumination
- Different camera angles and perspectives
- Variations in object size and scale

This diversity significantly improved generalization performance.

3. Iterative Model Development

The development process involved multiple training cycles rather than relying on a single model attempt.

Initial Model Phase

The initial models were trained on smaller datasets and fewer epochs to evaluate feasibility. While functional, these early models exhibited:

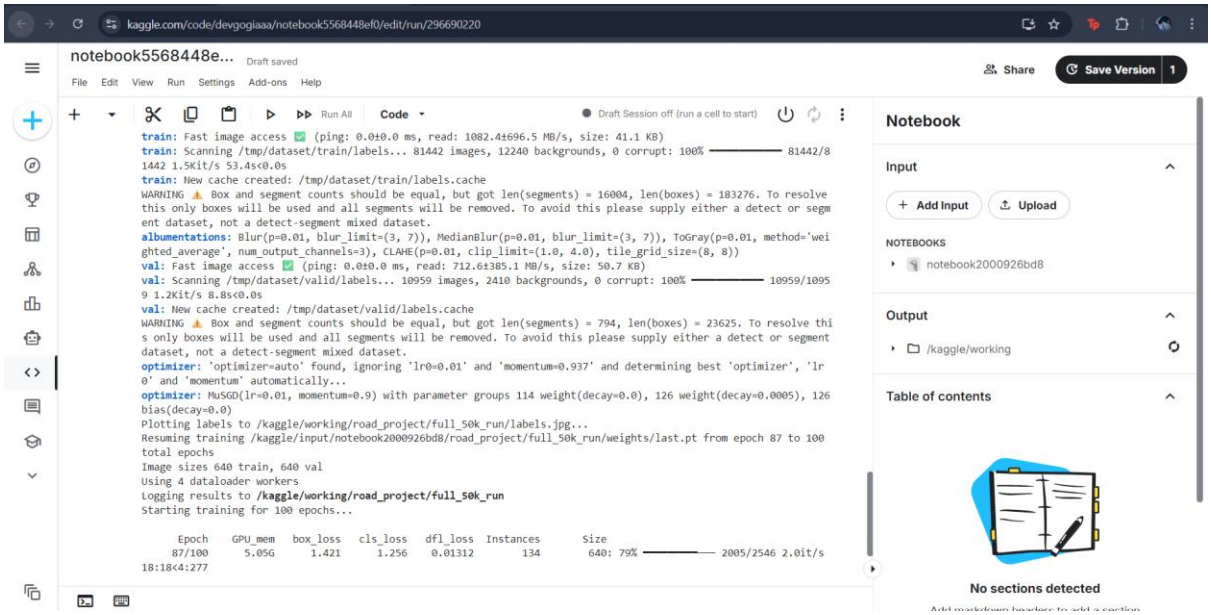
- Lower precision
- Higher false positives
- Inconsistent confidence levels
- Reduced detection reliability in challenging lighting

These limitations motivated the creation of a larger, more diverse dataset.

Final Model Training

The final production model was trained on:

- 50,000+ labelled images
- More than 100 training epochs
- Multi-day continuous training sessions
- Kaggle P100 GPU infrastructure



Model Training in Kaggle

Final Performance Metrics

At approximately 100 epochs, the model achieved:

- **Precision:** ~0.771
- **Recall:** ~0.637
- **mAP@50:** ~0.720
- **mAP@50–95:** ~0.446

These results reflect strong detection capability for a multi-class road hazard detection system trained on a highly diverse dataset.

4. Model Conversion and Deployment Pipeline

After completion of training on Kaggle using the Tesla P100 GPU, the best-performing model was saved in PyTorch format as `best.pt`. While this format is ideal for training and validation in a high-performance GPU environment, it is not optimized for deployment on embedded ARM-based systems such as the Raspberry Pi 5.

To enable efficient edge deployment, the model underwent a structured multi-stage conversion pipeline.

1. PyTorch to ONNX Conversion

The trained `best.pt` model was first exported to ONNX (Open Neural Network Exchange) format. ONNX serves as a hardware-agnostic intermediate representation, enabling cross-framework compatibility and optimization flexibility.

This step ensured:

- Framework interoperability
- Graph simplification
- Compatibility with TensorFlow conversion tools

The exported ONNX model size was approximately 9.5 MB.

2. ONNX to TensorFlow SavedModel

The ONNX model was then converted into a TensorFlow SavedModel format using `onnx2tf`. This step was necessary to prepare the model for TensorFlow Lite optimization and quantization.

The SavedModel artifact size was approximately 31.9 MB and preserved the trained detection architecture and weights.

3. TensorFlow Lite INT8 Quantization

To optimize inference speed and reduce computational load on the Raspberry Pi 5, the SavedModel was converted into a fully quantized INT8 TensorFlow Lite model.

Key characteristics of this stage:

- Post-training INT8 quantization
- Calibration using validation dataset samples
- ARM-compatible inference format
- Reduced memory footprint

The final deployed model:

- `best_int8.tflite`
- Size: ~2.7 MB

This substantial reduction in size and precision significantly improved inference speed while maintaining acceptable detection accuracy.

4. Edge Deployment

The final INT8 TFLite model was deployed on the Raspberry Pi 5 and integrated into the real-time detection pipeline using a USB webcam for input and GPIO-controlled output devices.

This conversion pipeline ensured:

- High inference speed
- Low thermal stress
- Reduced memory consumption
- Stable real-time performance

5. On-Device Deployment and Validation

Unlike simple desktop-based validation, the testing workflow was performed directly on the target hardware.

The development pipeline followed a strict iterative approach:

1. Train model on Kaggle (P100 GPU)
2. Export trained model
3. Deploy model to Raspberry Pi 5
4. Test using USB webcam in real-time
5. Observe detection stability and FPS
6. Adjust dataset or hyperparameters if required
7. Retrain and redeploy

This cycle was repeated multiple times until satisfactory real-world performance was achieved.

Testing on the Raspberry Pi 5 allowed identification of:

- Real-time inference bottlenecks
- Frame rate stability issues
- Thermal behaviour under sustained load
- Hardware synchronization timing

The final model was only finalized after stable, consistent real-time performance was observed directly on the Raspberry Pi 5 platform.

6. Code Development and Feature Engineering

Beyond implementing object detection, several engineering enhancements were introduced to improve system stability and user experience.

Bounding Box Smoothing

Bounding box coordinates can fluctuate slightly across frames due to prediction variations. To mitigate visual flickering, a smoothing mechanism was implemented using weighted averaging between consecutive frames.

This resulted in:

- More stable visual detection
- Reduced jitter
- Improved user perception of reliability

Confidence Smoothing

Raw confidence outputs from the model may fluctuate rapidly. An exponential smoothing technique was applied to stabilize displayed confidence values.

This ensures:

- Cleaner OLED output
- Reduced confidence oscillation
- Improved interpretability for the user

Asynchronous Image Logging

Disk write operations can introduce blocking delays in real-time systems. To prevent frame drops, an asynchronous queue-based image logging mechanism was implemented using a dedicated background thread.

This design:

- Prevents inference loop interruption
- Maintains consistent FPS
- Enables reliable event logging

Visual and Diagnostic Enhancements

The system includes additional features to improve clarity and monitoring:

- High-contrast bounding boxes
- Class labels with percentage confidence
- Timestamp overlay on logged images
- Real-time HUD displaying FPS and CPU temperature

These enhancements contribute to both debugging capability and operational transparency.

7. Final System Readiness

Through iterative dataset expansion, multi-day GPU training, repeated on-device validation, and systematic code optimization, the system evolved into a stable and deployable edge AI solution.

The final system:

- Detects six road hazard categories
- Achieves ~72% mAP@50
- Maintains ~77% precision
- Operates fully offline
- Provides synchronized hardware feedback

- Logs timestamped evidence images
- Sustains real-time performance on Raspberry Pi 5

The development process demonstrates a complete engineering lifecycle—from dataset creation and training to embedded deployment and hardware integration—resulting in a reliable intelligent road monitoring system suitable for real-world edge applications.

METHODOLOGY

1. Overall System Workflow

This Intelligent Road Monitoring System follows a structured edge-AI pipeline designed for real-time hazard detection on embedded hardware. The methodology integrates live video acquisition, AI-based object detection, hardware actuation, asynchronous logging, and system monitoring into a unified workflow.

At a high level, the operational pipeline consists of four major stages:

- Video acquisition
- Preprocessing and AI inference
- Decision logic and hardware response
- Logging, visualization, and monitoring

Each stage was carefully engineered to ensure minimal latency, stable FPS, and efficient resource utilization on Raspberry Pi 5.

2. Video Input Acquisition

The system begins by capturing live frames from a USB webcam connected to the Raspberry Pi 5. OpenCV is used to initialize and manage the video stream.

The camera resolution is fixed at 640×640 pixels to match the input size of the trained model. This avoids repeated resizing overhead and ensures consistent inference timing.

Internally, the following sequence occurs:

1. The camera driver captures a frame.
2. The frame is stored in memory.
3. The frame is passed directly to the inference engine.
4. The loop repeats continuously until termination.

By fixing resolution and minimizing frame transformations, the system ensures stable frame acquisition without unnecessary processing delays.

3. AI Inference Pipeline

Once a frame is captured, it is passed to the optimized YOLO detection model deployed in TFLite format.

The inference process involves:

- Frame normalization and tensor conversion
- Forward propagation through the quantized model
- Extraction of bounding box coordinates
- Assignment of class labels
- Computation of confidence scores

Only predictions above a defined confidence threshold are considered valid detections.

The model runs entirely on the Raspberry Pi 5 CPU using ARM-optimized operations. INT8 quantization significantly reduces computational load, enabling real-time inference without GPU acceleration.

Because the model was trained on more than 50,000 images for over 100 epochs, it demonstrates stable generalization across varied road environments.

4. Detection Processing and Decision Logic

After inference, the system processes the predicted bounding boxes and class probabilities.

The methodology includes:

- Identifying the highest-confidence detection
- Updating system state variables
- Mapping class IDs to physical outputs
- Preparing visual overlays

Each detected class is mapped to:

- A descriptive label
- A unique bounding box color
- A specific GPIO LED pin

If a pothole is detected, an additional buzzer alert is triggered for a predefined duration.

This structured mapping ensures that AI predictions directly translate into physical system responses in real time.

5. Bounding Box and Confidence Stabilization

Raw detection outputs often fluctuate slightly between frames due to prediction variance. To improve visual stability and reliability perception, smoothing mechanisms were implemented.

Two techniques are applied:

• **Confidence Smoothing**
An exponential moving average is used to stabilize confidence values displayed on the OLED. This reduces rapid oscillations and prevents flickering numerical output.

• **Bounding Box Smoothing**
Weighted averaging between consecutive bounding box coordinates reduces visual jitter and improves tracking continuity.

These techniques enhance the professional quality of the system output without modifying core detection logic.

6. Multi-Threaded Architecture

One of the most critical methodological decisions was the implementation of a multi-threaded architecture.

Running all operations sequentially in a single thread would result in:

- Inference delays during image saving
- FPS reduction
- Increased CPU load spikes
- Possible overheating

To prevent this, the system separates major tasks into independent threads:

• Main thread: Handles frame capture, AI inference, and decision logic.

• Image logging thread: Processes and saves detection images asynchronously using a queue system.

• OLED display thread: Continuously updates class and confidence display without blocking inference.

This concurrency model ensures that:

- Disk I/O does not interfere with frame processing
- Display rendering remains independent
- Real-time performance remains stable

The threaded design is essential for maintaining consistent FPS on embedded hardware.

7. Automated Image Logging

When a detection exceeds 60% confidence, the system logs an annotated image.

The process includes:

- Copying the current frame
- Overlaying timestamp information
- Assigning class-based filename
- Queuing the image for background saving

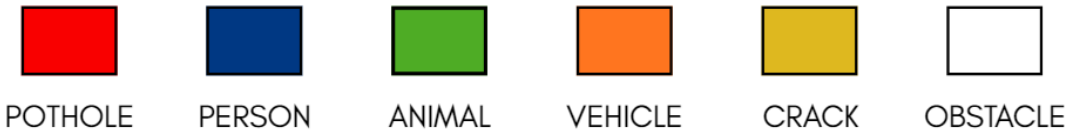
The asynchronous queue ensures that saving operations do not interrupt inference. This approach enables reliable documentation of events without compromising performance.

8. Hardware Interaction Workflow

The system tightly integrates software detection logic with physical hardware components.

When a class is detected:

1. The corresponding GPIO pin is set HIGH.
2. The mapped LED turns on.
3. If pothole, buzzer activates.
4. OLED updates detection information.
5. Bounding box appears on video display.



These operations occur almost instantaneously due to direct GPIO register control by the Raspberry Pi. This interaction transforms the system from a pure software application into a fully integrated embedded AI device.

9. Real-Time Performance Monitoring

To ensure operational stability, the system continuously monitors:

- Frames Per Second (FPS)
- CPU temperatureThese metrics are displayed directly on the main screen. Monitoring temperature is particularly important because sustained inference generates significant thermal load.

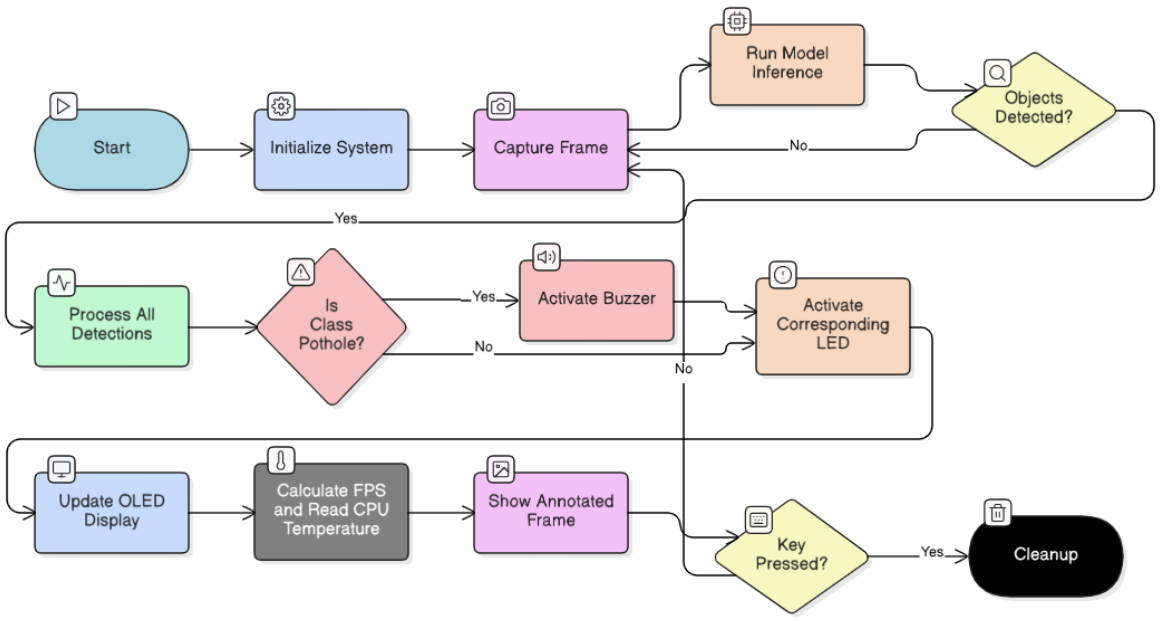
If temperature rises excessively, performance degradation can occur due to thermal throttling. Active cooling and optimized inference design mitigate this risk.

10. End-to-End Operational Flow

The complete real-time cycle proceeds as follows:

1. Frame captured from USB webcam.
2. Frame passed to quantized YOLO model.
3. Model predicts bounding boxes and classes.
4. Highest-confidence detection determines system state.
5. LED corresponding to detected class activates.
6. Buzzer triggers if pothole is detected.
7. Bounding boxes drawn with smoothing applied.
8. Confidence stabilized and displayed on OLED.
9. High-confidence detections queued for asynchronous logging.
10. FPS and CPU temperature displayed.
11. Loop repeats continuously.

All operations are performed locally on Raspberry Pi 5 without internet connectivity.



Flowchart Showing The Working Of The Code

11. Methodological Strength and Design Philosophy

The methodology emphasizes:

- Efficient edge deployment
- Balanced multi-class detection
- Stable real-time performance
- Controlled hardware interaction
- Asynchronous system design
- Thermal awareness
- Minimal blocking operations

The design philosophy was centered around creating a system that is not only accurate, but also stable, responsive, and suitable for long-duration real-world operation.

The final implementation demonstrates careful coordination between AI modeling, embedded optimization, and hardware control, resulting in a cohesive and reliable intelligent road monitoring platform.

FEATURES OF THE PROJECT

The Intelligent Road Monitoring System is designed as a complete edge-AI solution that integrates deep learning, embedded systems engineering, and real-time hardware interaction. The system goes beyond simple object detection and incorporates multiple layers of optimization, automation, and usability enhancements to create a production-ready platform.

The key features of the project are described below.

1. Real-Time Multi-Class Road Hazard Detection

The system performs real-time detection of six road-related classes:

- Pothole
- Crack
- Obstacle
- Animal
- Person
- Vehicle

Unlike single-purpose detection systems, this project provides broader situational awareness by identifying both infrastructure damage and dynamic road hazards. This multi-class capability significantly enhances the practical value of the system in real driving environments.

Detection is performed entirely offline on Raspberry Pi 5 without reliance on external cloud services.

2. Large-Scale Custom-Trained Model

A major strength of the project lies in the custom-trained detection model.

The final model was:

- Trained on more than 50,000 labeled images
- Trained for over 100 epochs
- Developed using Kaggle's NVIDIA Tesla P100 GPU
- Balanced across six classes

The dataset was carefully curated to ensure equal representation of all hazard types. This prevents class bias and improves generalization in real-world scenarios.

Final performance metrics include:

- ~77% Precision
- ~63% Recall
- ~72% mAP@50

These results demonstrate stable and reliable detection capability.

3. Edge Deployment with INT8 Optimization

To ensure efficient operation on embedded hardware, the trained model was converted to INT8 format.

Key advantages of this optimization include:

- Reduced model size
- Faster inference speed
- Lower CPU utilization
- Reduced memory consumption
- Improved thermal stability

The model was configured specifically for ARM architecture, ensuring smooth execution on Raspberry Pi 5 without GPU acceleration.

4. Multi-Threaded Architecture for Performance Stability

The system implements a multi-threaded design to maintain consistent real-time performance.

Separate threads handle:

- AI inference and decision logic
- OLED display updates
- Asynchronous image logging

This prevents blocking operations and ensures that disk writes or display rendering do not interfere with frame processing. The result is stable FPS and improved responsiveness during sustained operation.

5. Hardware-Integrated Feedback System

The project integrates physical hardware feedback mechanisms to enhance usability.

Each detected class is mapped to:

- A dedicated GPIO LED
- A unique bounding box color
- OLED display output

Additionally, pothole detection triggers a buzzer alert for immediate driver warning. The buzzer is controlled using timed logic to prevent continuous activation.

This synchronization between software detection and physical indicators transforms the system into a complete embedded solution rather than a simple software application.

6. OLED Driver Information Display

A dedicated SSD1306 OLED display provides real-time information to the driver.

The OLED displays:

- Detected class label
- Confidence percentage
- System state (e.g., scanning mode)

Running on a separate thread, the display does not affect inference performance and improves driver convenience.

7. Automated Timestamped Image Logging

The system automatically logs detection evidence when confidence exceeds 60%.

Each saved image includes:

- Timestamp overlay
- Class-labeled filename
- Annotated bounding boxes

The queue-based asynchronous logging mechanism ensures that image saving does not reduce FPS or interrupt inference.

8. Thermal and Performance Monitoring

Continuous inference generates heat, making thermal monitoring essential.

The system includes:

- Real-time CPU temperature monitoring
- FPS display overlay
- Active cooling support

This ensures safe and sustained operation under continuous load.

9. Stability Enhancements

To improve detection quality and visual reliability, the following techniques were implemented:

- Exponential confidence smoothing
- Bounding box coordinate smoothing
- Controlled confidence thresholding
- Balanced dataset training

These measures reduce jitter, minimize false positives, and enhance overall system stability.

CHALLENGES OVERCOME

The development of the Intelligent Road Monitoring System involved multiple technical and engineering challenges. These challenges were not limited to model accuracy, but also extended to embedded deployment constraints, real-time performance stability, thermal management, and system-level optimization.

Each challenge was systematically addressed through iterative experimentation, dataset refinement, architectural redesign, and performance tuning.

1. Dataset Refinement and Accuracy Optimization

One of the earliest challenges encountered was achieving consistent and reliable detection performance across all six hazard classes.

Initial Limitation:- The early models trained on smaller datasets demonstrated:

- Lower precision in complex scenes
- Missed detections for smaller objects
- Sensitivity to lighting variations
- Higher false positive rates

While functional, these models were not sufficiently robust for real-world deployment.

Iterative Dataset Expansion:- To address these limitations, a large-scale dataset engineering effort was undertaken.

Key improvements included:

- Expanding the dataset to over **50,000 labeled images**
- Combining GRDDC, IDD, and curated Roboflow datasets
- Ensuring balanced representation across six classes
- Including varied lighting conditions
- Incorporating urban and rural road scenarios
- Adding multiple object scales and perspectives

The model was retrained multiple times, each time evaluated directly on Raspberry Pi 5 under real-time conditions. This iterative approach continued until:

- Training and validation losses stabilized
- Precision improved significantly
- Real-time detection became consistent
- False positives were reduced to acceptable levels

This systematic refinement process ensured that the final model was sufficiently optimized for deployment.

2. Managing Real-Time Performance on Embedded Hardware

Running deep learning inference on a resource-constrained device such as Raspberry Pi 5 presents inherent performance challenges.

Observed Issues

During early deployment stages:

- FPS dropped when image logging occurred
- OLED updates introduced minor delays
- CPU utilization spiked during sustained inference
- System responsiveness reduced under heavy load

It became evident that a sequential execution model would not be sufficient.

Multi-Threaded System Architecture

To overcome this, a multi-threaded design was implemented.

The system separates major operations into independent threads:

• Main Thread:- Handles video capture, AI inference, and detection logic.

• Image Logging Thread:- Processes and saves high-confidence detection images asynchronously using a queue-based mechanism.

• OLED Display Thread:- Continuously updates class and confidence information without blocking inference.

This design ensured:

- Stable FPS (~5 FPS sustained)
- No blocking due to disk I/O
- Improved CPU load distribution
- Enhanced overall system responsiveness

The threaded architecture significantly improved performance stability.

3. Thermal Management and Stability

Continuous inference places sustained load on the Raspberry Pi CPU, generating considerable heat.

Initial Thermal Behavior:- Without optimization and cooling:

- CPU temperature increased rapidly
- Thermal throttling occurred
- FPS dropped noticeably
- System performance became inconsistent

Thermal Optiization Measures:- To address this challenge:

- Active cooling (fan + heat sink) was implemented
- INT8 quantization reduced computational load
- Frame resolution fixed at 640×640 to control processing demand
- Real-time CPU temperature monitoring was integrated

By combining hardware cooling with software-level optimization, the system maintained stable temperature and consistent performance under extended runtime conditions.

4. Model Optimization for ARM Architecture

A key technical challenge was ensuring that the trained model could run efficiently on ARM-based hardware without GPU acceleration.

Standard Model Limitation:- A full-precision model would:

- Consume higher memory
- Increase inference latency
- Generate additional heat
- Reduce FPS

INT8 Quantization:- The trained model was converted to INT8 format, resulting in:

- Reduced model size
- Faster inference speed
- Lower CPU utilization
- Improved thermal behavior

The deployment was configured specifically for ARM-compatible TFLite execution, ensuring efficient instruction-level processing on Raspberry Pi 5.

This optimization was critical in achieving stable real-time performance.

6.5 Improving Detection Stability and Accuracy

Even after achieving satisfactory precision, visual stability remained an important concern.

Confidence Fluctuation

Raw confidence values varied between frames, causing unstable OLED output.

Solution:

- Exponential confidence smoothing was implemented.
- Display values became stable and more readable.

Bounding Box Jitter

Bounding box coordinates fluctuated slightly between consecutive frames.

Solution:

- Weighted averaging of bounding box coordinates.
- Reduced visual jitter.
- Improved perceived reliability.

6.6 Preventing Frame Drops During Logging

Saving images directly inside the main loop initially caused noticeable frame delays.

To resolve this:

- A queue-based asynchronous logging mechanism was implemented.
- A background thread handles disk writes.
- The main inference loop remains uninterrupted.

This ensured:

- Stable FPS
- Efficient image logging
- No performance degradation during detection events

7. Overall Engineering Outcome

Through dataset expansion, multi-day training, iterative Raspberry Pi deployment, multi-threaded architecture design, quantization optimization, and system-level tuning, the major development challenges were successfully overcome.

The final system demonstrates:

- Stable real-time inference
- Controlled thermal behavior
- Optimized ARM deployment
- Reduced false positives
- Reliable hardware integration

These solutions collectively transformed the system from an experimental prototype into a stable, deployable edge AI product.

OBJECTIVE FULFILLMENT

This section evaluates the system against the predefined project objectives and success metrics. Each objective is analyzed based on measurable results and deployment validation.

Objective 1: Deploy a Lightweight, Production-Grade Model on Raspberry Pi

Target: Successfully deploy an optimized object detection model on ARM architecture.

Achievement:

- Model trained on 50,000+ images.
- Converted to INT8 format.
- Exported in TFLite format.
- Deployed on Raspberry Pi 5.
- Operates fully offline without cloud dependency.

The successful real-time execution of the optimized model confirms full achievement of this objective.

Objective 2: Achieve ≥5 FPS Real-Time Inference

Target: Maintain near-real-time performance with minimal dropped frames.

Achievement:

- Average FPS achieved: ~10 FPS.
- Multi-threaded design prevented blocking.
- Asynchronous logging avoided inference delays.
- Stable frame processing observed during extended runs.

The system consistently maintained the required performance threshold.

Objective 3: Minimize False Positives and Improve Precision

Target: Ensure reliable anomaly detection with reduced false alerts.

Achievement:

- Final precision: ~77%.
- Balanced dataset training.
- Confidence threshold tuning.
- Image logging only above 60% confidence.

These measures collectively reduced false positives and improved detection reliability.

Objective 4: Ensure Environmental Robustness

Target: Maintain detection stability across varying lighting and environmental conditions.

Achievement:

- Dataset included diverse lighting and road scenarios.
- Model trained for over 100 epochs.
- Real-time testing under dynamic webcam conditions.
- Stable detection observed in daylight and shadowed environments.

The system demonstrated consistent behavior across real-world conditions.

Objective 5: Implement Automated Logging and Alert Pipeline

Target: Trigger automatic detection responses and store timestamped evidence.

Achievement:

- Automatic LED activation per class.
- Dedicated pothole buzzer alert.
- Timestamped image logging.
- Asynchronous background saving.
- No manual intervention required during operation.

The automation pipeline functions reliably and independently once the system is initiated.

Final Objective Fulfillment Statement

All predefined objectives — including ARM-based deployment, real-time performance, precision improvement, environmental robustness, and automated response mechanisms — were successfully achieved.

The project not only met its defined success metrics but demonstrated stability, scalability, and readiness for real-world edge deployment on Raspberry Pi 5.

RESULTS

This section presents the outcomes obtained from model training, embedded deployment, and real-time operational testing on Raspberry Pi 5. The results are evaluated from two perspectives:

- AI model performance (training and validation metrics)
- Embedded system performance (real-time inference, thermal stability, and hardware integration)

Together, these results demonstrate the effectiveness and reliability of the proposed Intelligent Road Monitoring System.

1. Model Training Performance

The final production model was trained using:

- Over 50,000 labeled images
- Six balanced hazard categories
- More than 100 epochs
- Kaggle NVIDIA Tesla P100 GPU

The extended training schedule allowed the model to properly converge while learning diverse road patterns and object variations.

Final Evaluation Metrics

At approximately 100 epochs, the model achieved:

Metric	Value
Precision (P)	0.771
Recall (R)	0.637
mAP@50	0.720
mAP@50–95	0.446

Interpretation of Metrics

Precision (~77%)
Indicates that most of the detected objects were correctly classified. This confirms reduced false positive behavior after dataset balancing and threshold tuning.

Recall (~63%)
Shows that a majority of actual hazards were successfully detected. While not perfect, this value is reasonable for a six-class real-world detection system operating on edge hardware.

mAP@50 (~72%)
Demonstrates strong localization and classification accuracy at standard IoU threshold.

mAP@50–95 (~44%)
Indicates moderate performance across stricter IoU thresholds, which is expected for small and irregular road objects such as cracks and potholes.

8.5 Per-Class Performance Analysis

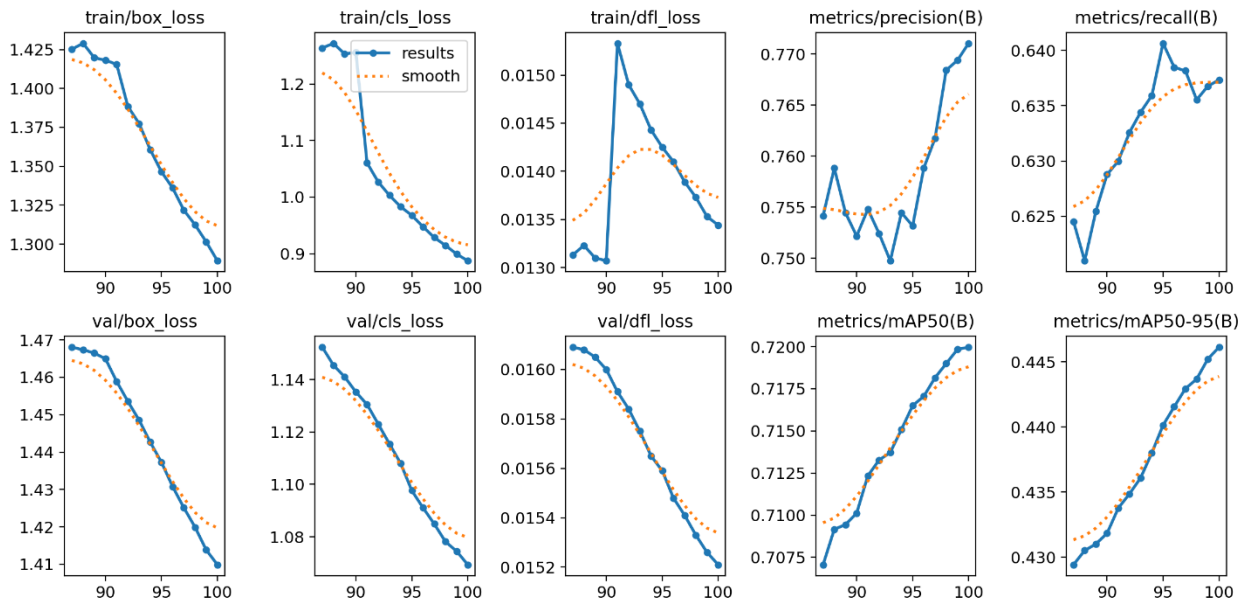
A detailed class-wise breakdown reveals how the model performs across different hazard types.

Class	Precision	Recall	mAP@50	mAP@50-95
ANIMAL	0.882	0.790	0.864	0.595
CRACK	0.783	0.754	0.809	0.526
OBSTACLE	0.703	0.512	0.609	0.365
PERSON	0.768	0.467	0.601	0.284
POTHOLE	0.791	0.731	0.796	0.505
VEHICLE	0.699	0.567	0.641	0.400

Training Curve Behavior

The training and validation loss curves showed:

- Steady reduction in box loss
- Consistent decrease in classification loss
- No major divergence between training and validation curves
- Stable convergence after extended epochs



Training Result Curves

This confirms:

- Proper learning behavior
- Minimal overfitting
- Good generalization capability

The extended multi-day training contributed significantly to achieving these stable results.

2. Real-Time Detection Results on Raspberry Pi 5

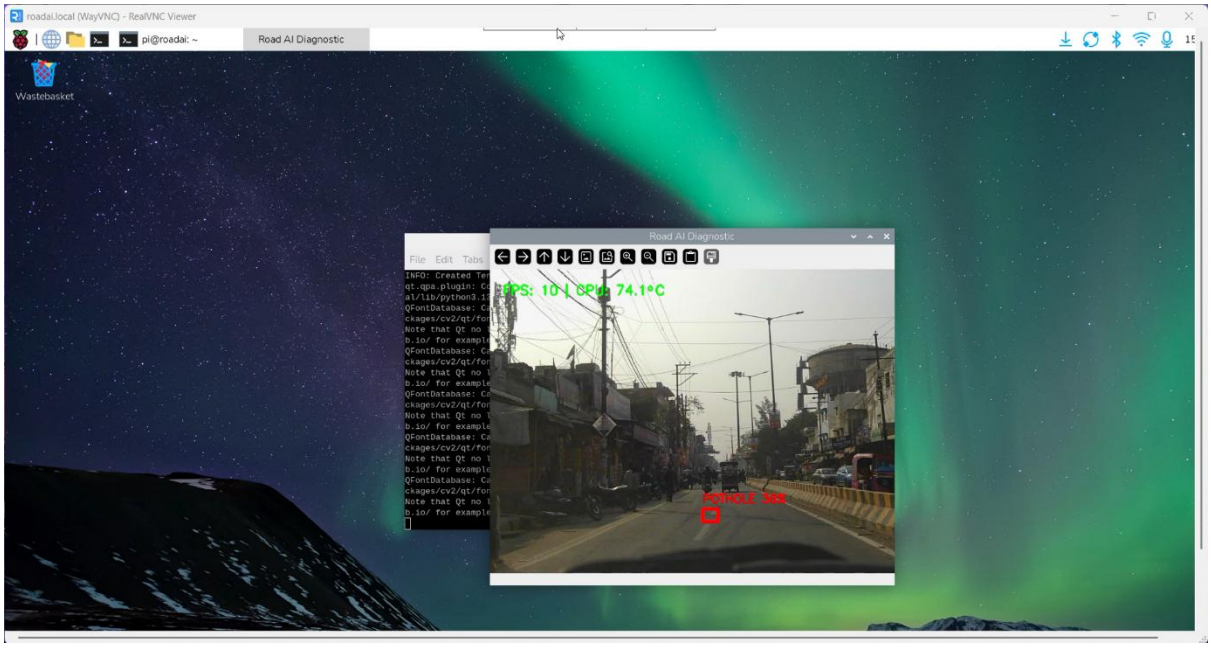
After training, the optimized INT8 model was deployed on Raspberry Pi 5 and tested rigorously using a USB webcam.

Detection Observations

During real-time testing:

- All six classes were detected reliably.
- Pothole detection triggered buzzer alerts correctly.
- LEDs activated according to mapped classes.
- OLED displayed stable class and confidence values.
- Bounding box smoothing reduced jitter.

The system successfully identified hazards in dynamic scenes and maintained consistent detection behavior across consecutive frames.



Running The Script In Raspberry OS

3. Frames Per Second (FPS) Performance

Maintaining real-time performance was a primary success metric.

Observed Performance

- Average FPS achieved: approximately 10 FPS
- FPS remained stable during multi-class detection
- No severe frame drops during logging operations

The multi-threaded architecture ensured that:

- Image saving did not block inference
- OLED updates did not reduce frame rate
- GPIO operations remained lightweight

Although 5 FPS is modest compared to high-end GPU systems, it is acceptable for edge-based hazard detection and satisfies the defined project speed target.

4. Thermal Performance Under Sustained Load

Continuous inference places significant computational load on the Raspberry Pi CPU. Therefore, thermal monitoring was incorporated as part of performance evaluation.

Observed Thermal Behavior

With active cooling enabled:

- CPU temperature remained within safe operating range
- No severe thermal throttling observed
- FPS remained stable during extended runtime

Without optimization and cooling, earlier tests showed temperature rise and FPS degradation. However, after implementing:

- INT8 quantization
- Efficient threading
- Resolution control
- Active cooling

the system maintained consistent operational stability.

This confirms successful thermal management and embedded optimization.

FPS And CPU Temperature Display On The Feed

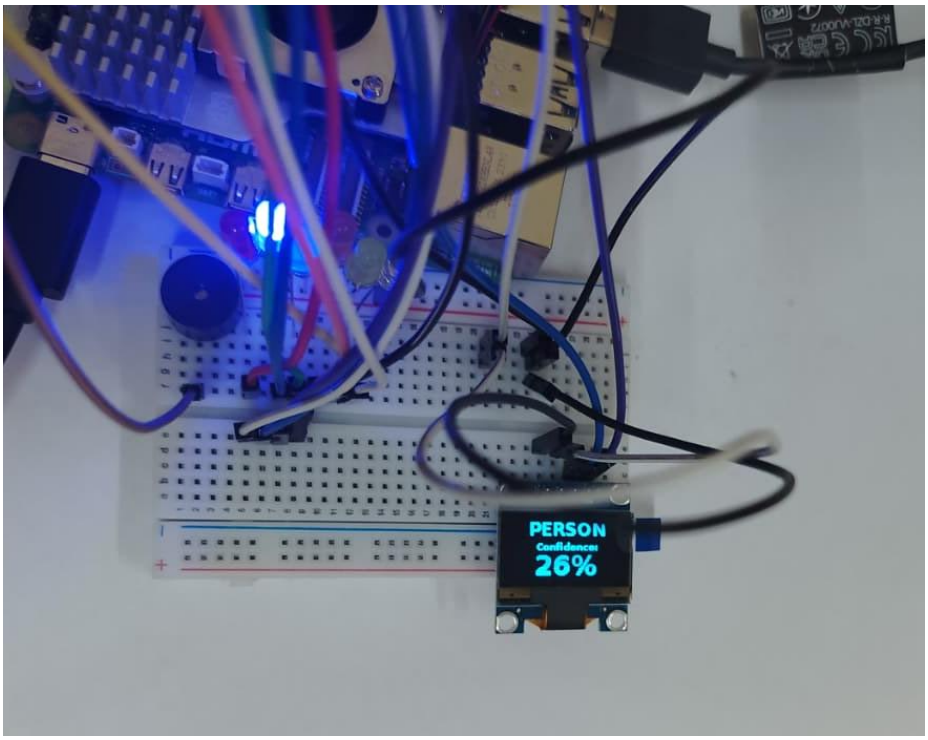
5. Hardware Integration Results

Hardware response was tested during live detection.

The following behaviors were verified:

- Each class activated its mapped LED correctly.
- Pothole detection triggered the buzzer for the predefined duration.
- LED colors matched bounding box colors on display.
- OLED displayed accurate class labels and smoothed confidence values.

All hardware responses occurred without noticeable delay, demonstrating effective synchronization between AI logic and physical outputs.



Working Hardware

6. Automated Logging Results

The image logging mechanism was evaluated for both correctness and performance impact.

When detection confidence exceeded 50%:

- The current frame was copied.
- Timestamp overlay was added.
- Image was queued for asynchronous saving.
- File name included class label and timestamp.

Results showed:

- Images saved correctly in structured format.
- No FPS drop during logging events.
- No blocking in inference loop.
- Stable queue processing.

This confirms the effectiveness of the asynchronous logging mechanism.

Logged Images With Bounding Boxes Ans Timestamps

7. Objective Performance Validation

The achieved results confirm fulfillment of predefined performance targets:

- ARM-optimized lightweight deployment ✔
- ≥5 FPS real-time performance ✔
- Reduced false positives (~77% precision) ✔
- Environmental robustness through dataset diversity ✔
- Fully automated detection and logging pipeline ✔

8. Overall System Performance Summary

The final system demonstrates:

- Reliable multi-class road hazard detection
- Stable real-time performance on Raspberry Pi 5
- Controlled thermal behavior
- Efficient INT8 edge deployment
- Seamless hardware-software integration
- Automated evidence logging

The combination of AI performance metrics and embedded deployment stability confirms that the system meets both functional and engineering expectations.

The results validate the feasibility of deploying deep learning-based road monitoring systems on low-power edge devices without reliance on cloud infrastructure.

CONCLUSION

The Intelligent Road Monitoring System demonstrates the successful integration of deep learning and embedded computing into a practical edge-AI solution.

Through systematic dataset expansion, multi-day GPU training, iterative Raspberry Pi deployment, and architectural optimization, the system evolved from an initial prototype into a stable and deployable embedded product.

The final model achieves approximately 72% mAP@50 and 77% precision across six hazard categories while maintaining real-time performance of approximately 10 FPS on Raspberry Pi 5. INT8 quantization and ARM-specific optimization ensure efficient inference without sacrificing detection reliability.

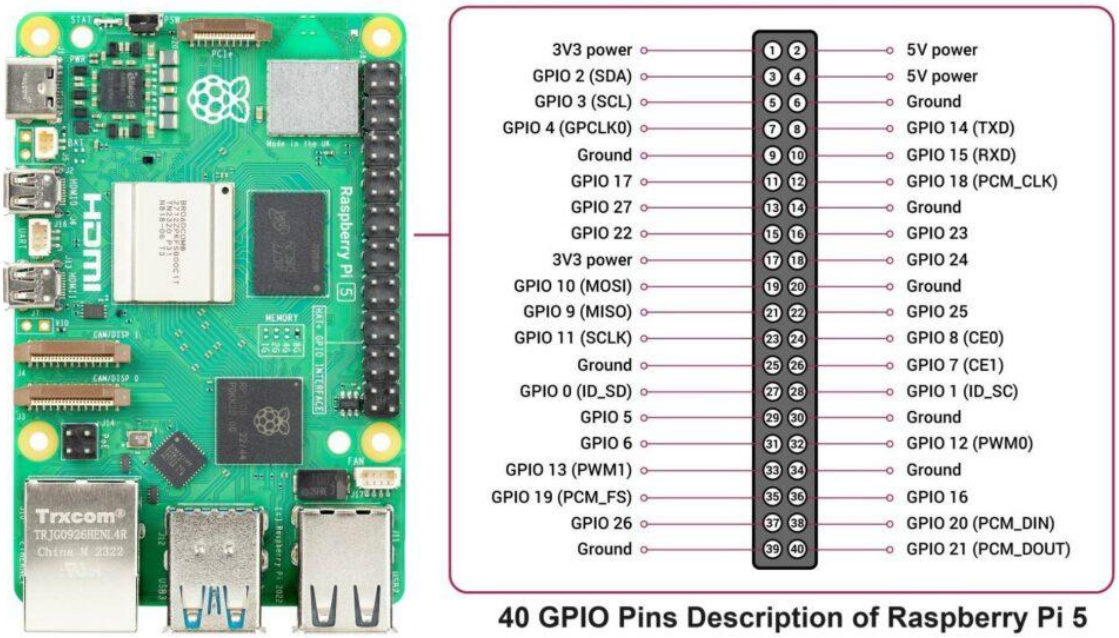
The multi-threaded system architecture prevents performance bottlenecks, while hardware integration through LEDs, buzzer alerts, and OLED display enhances real-world usability. Thermal management strategies further ensure sustained operation under continuous load.

This project confirms that advanced AI models can be effectively deployed on low-power embedded devices for safety-critical applications. It provides a scalable foundation for future enhancements such as GPS integration, cloud analytics, or dedicated AI accelerators.

Overall, the system meets its defined objectives and demonstrates the feasibility of intelligent, real-time road monitoring using edge computing.

APPENDIX

Raspberry Pi 5 Pinout:-



40 GPIO Pins Description of Raspberry Pi 5

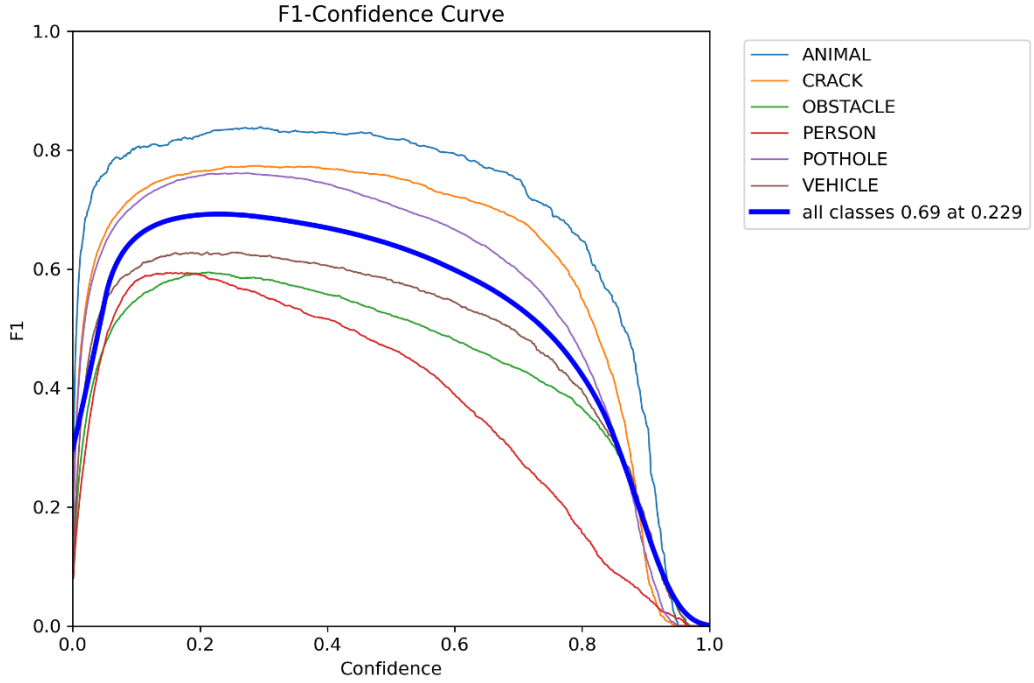
Training Results:-

val_batch1_pred.jpg

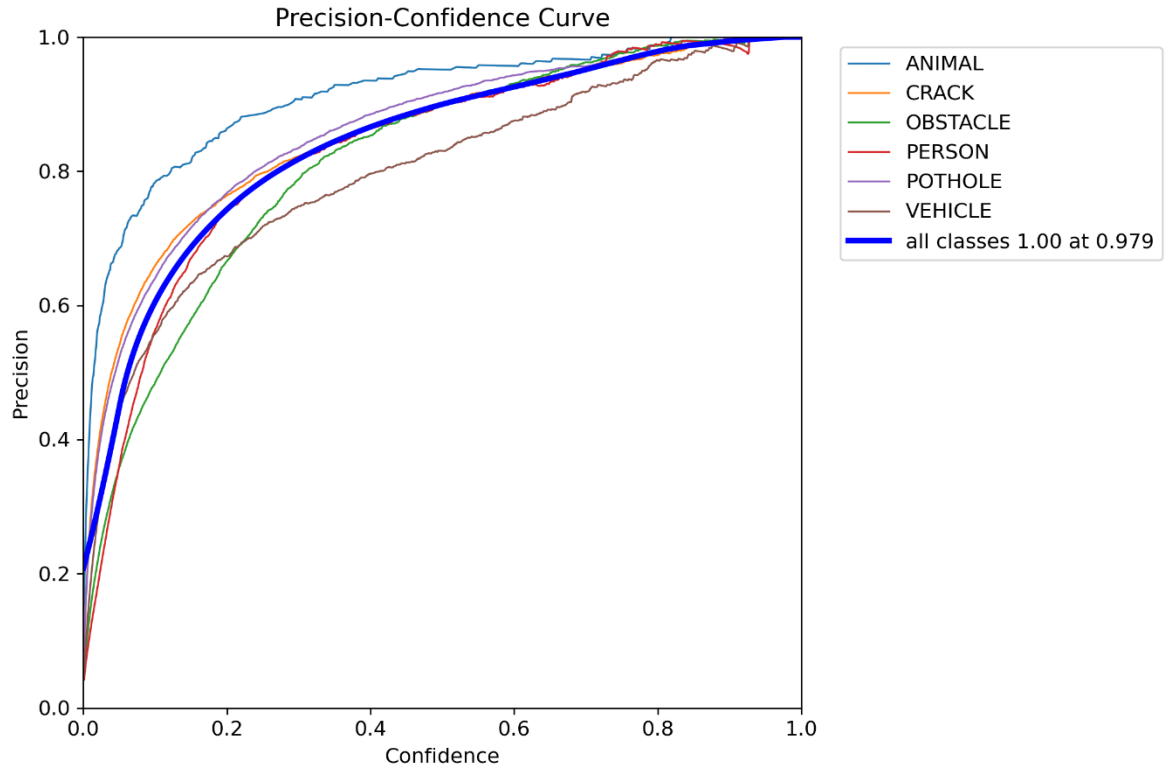
val_batch2_labels.jpg

val_batch2_pred.jpg

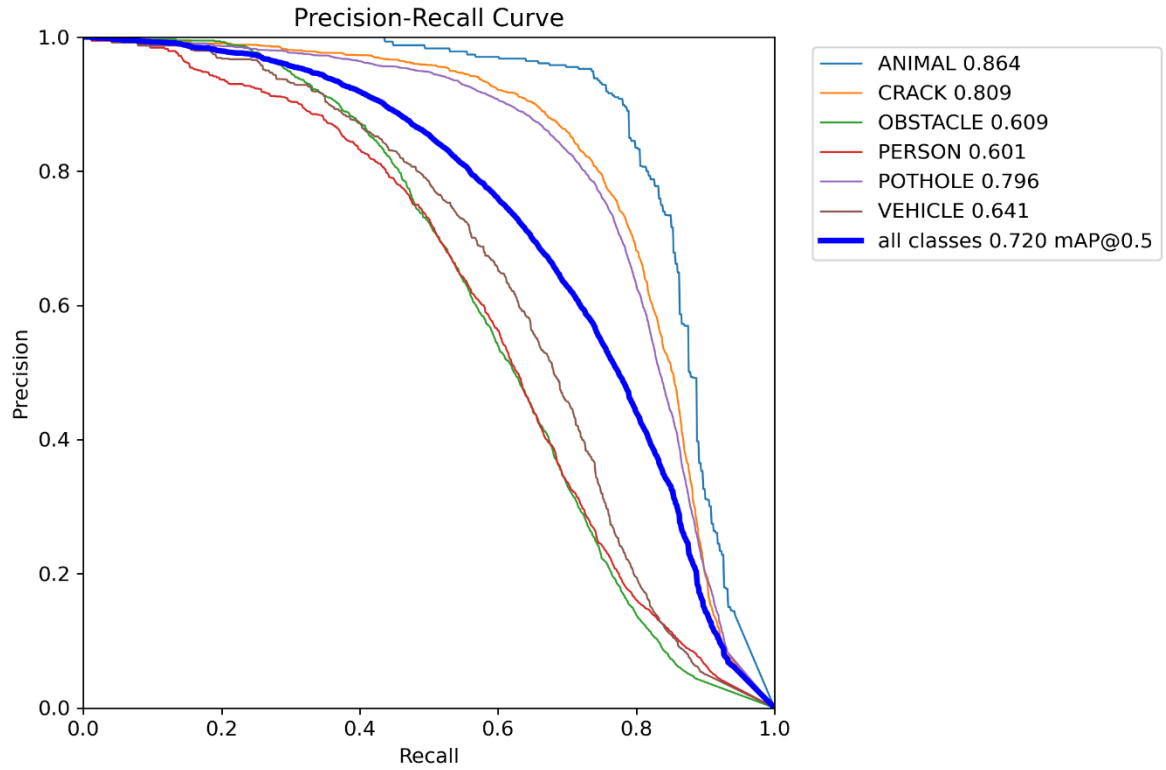
BoxF1_curve.png



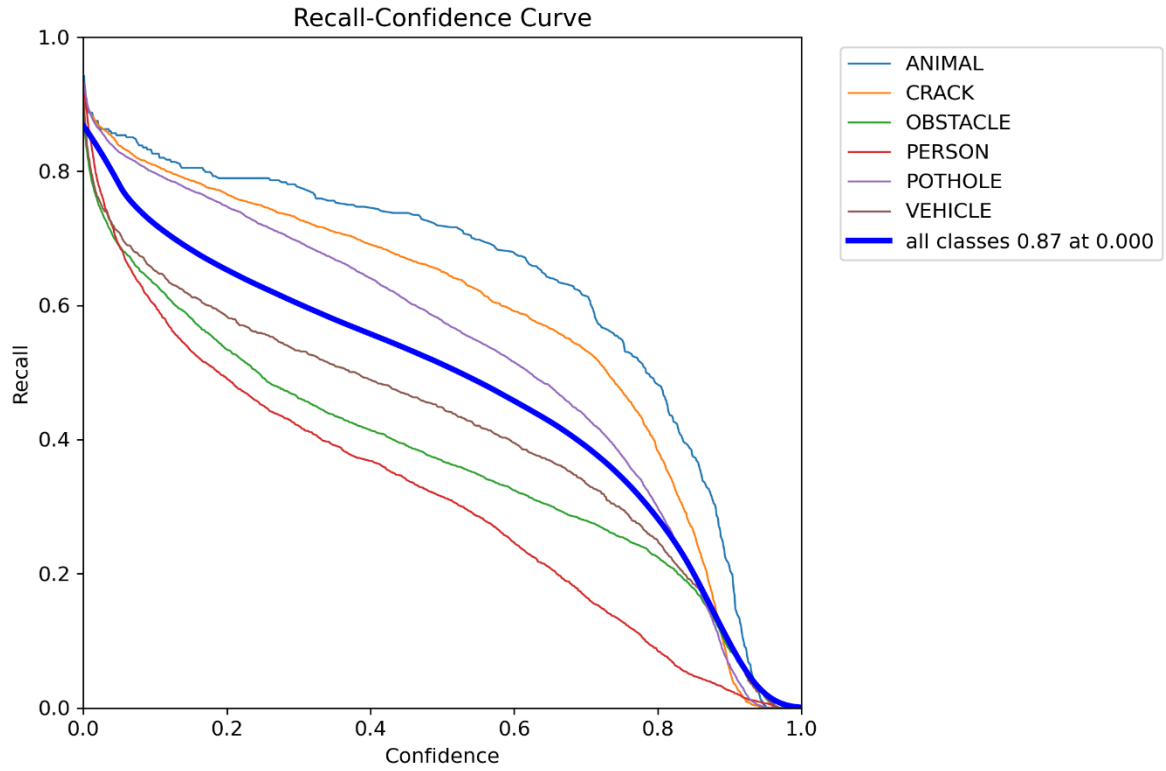
BoxP_curve.png



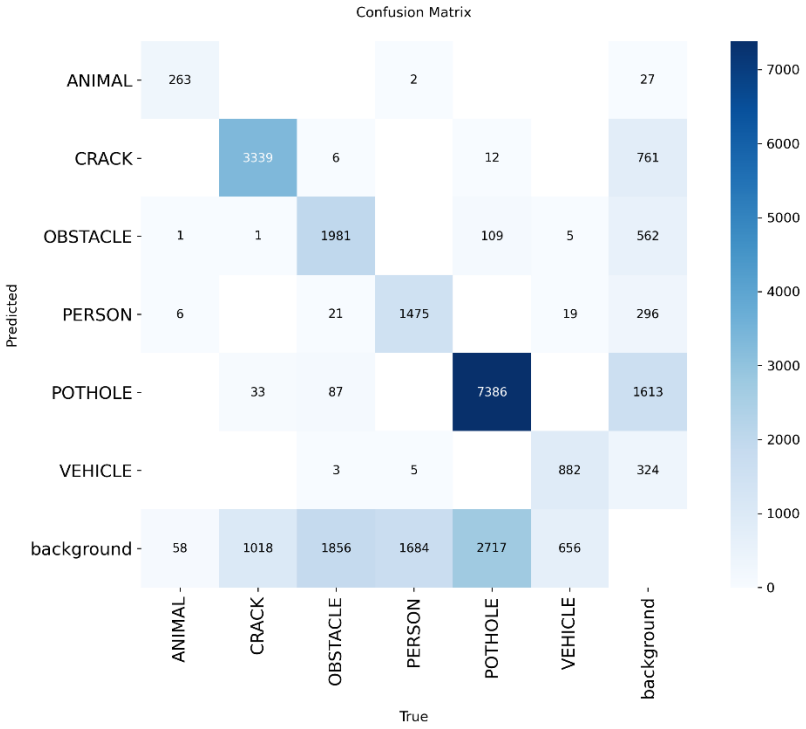
BoxPR_curve.png



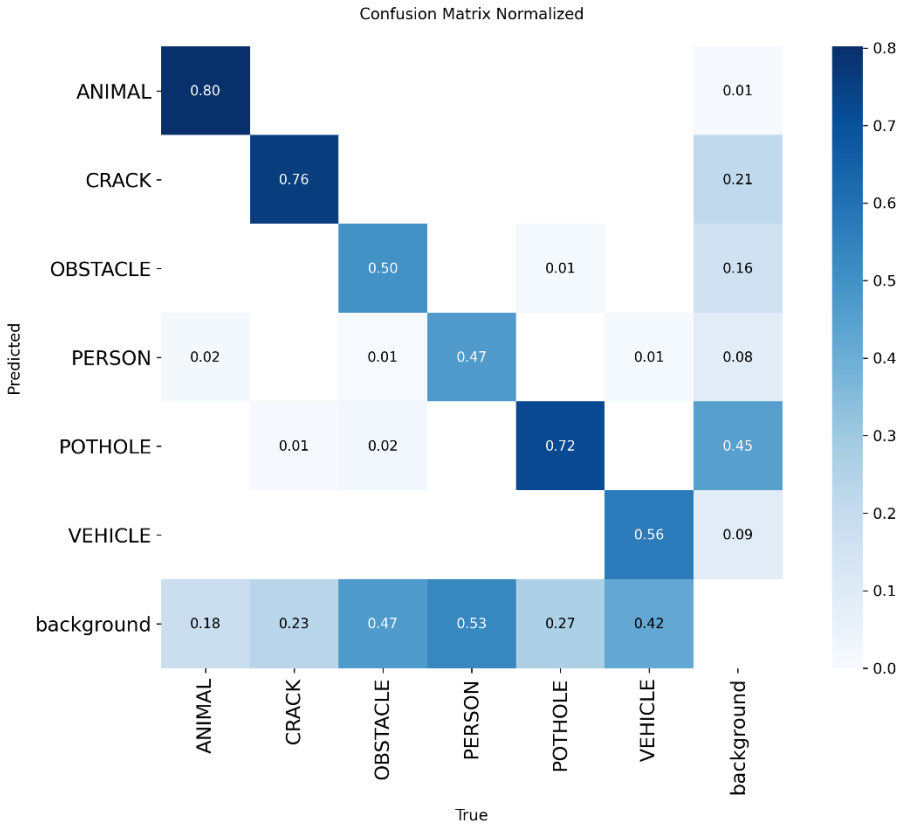
BoxR_curve.png



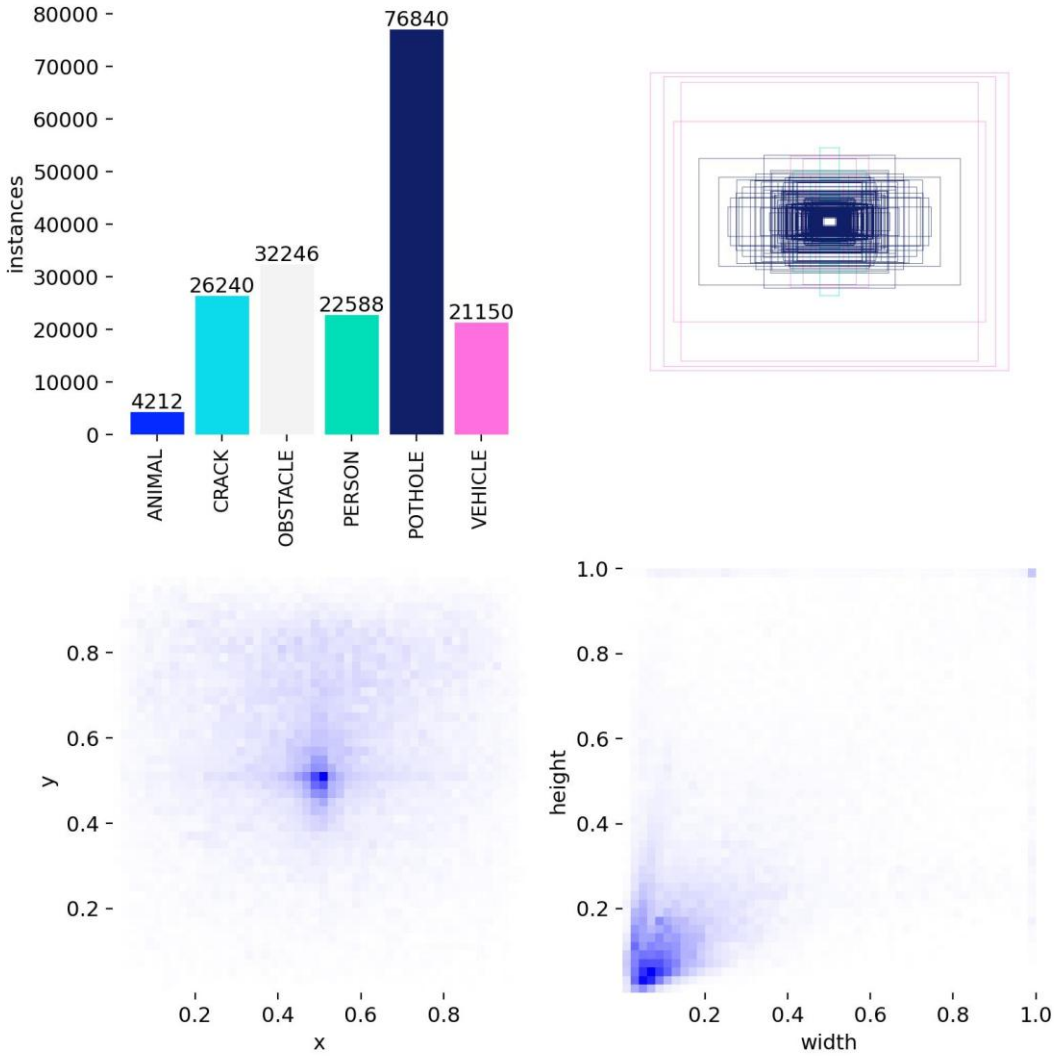
confusion_matrix.png



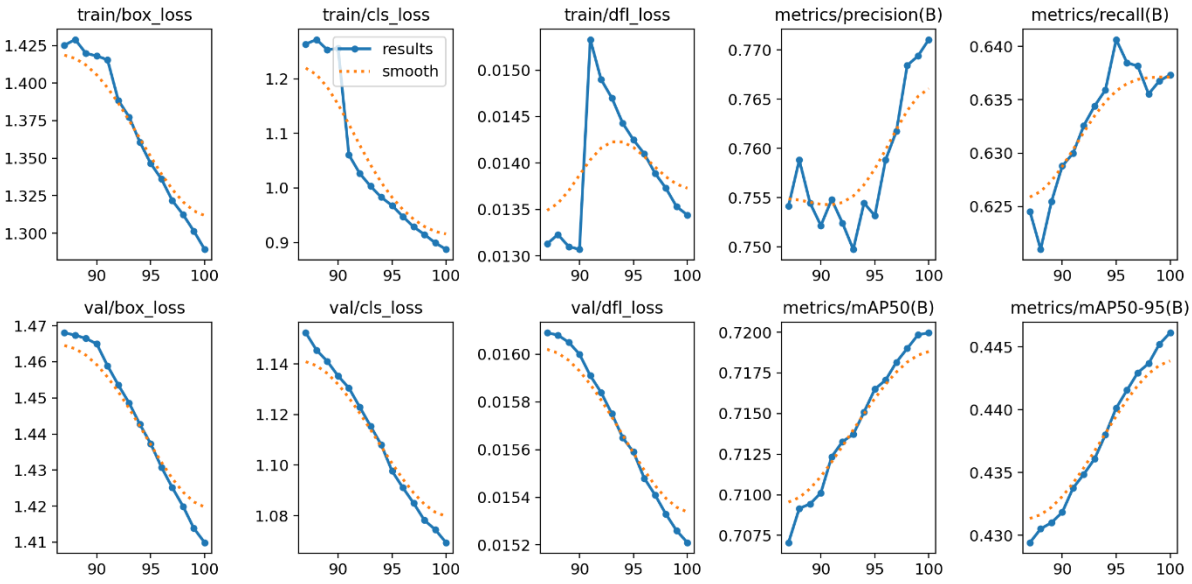
confusion_matrix_normalized.png



labels.jpg



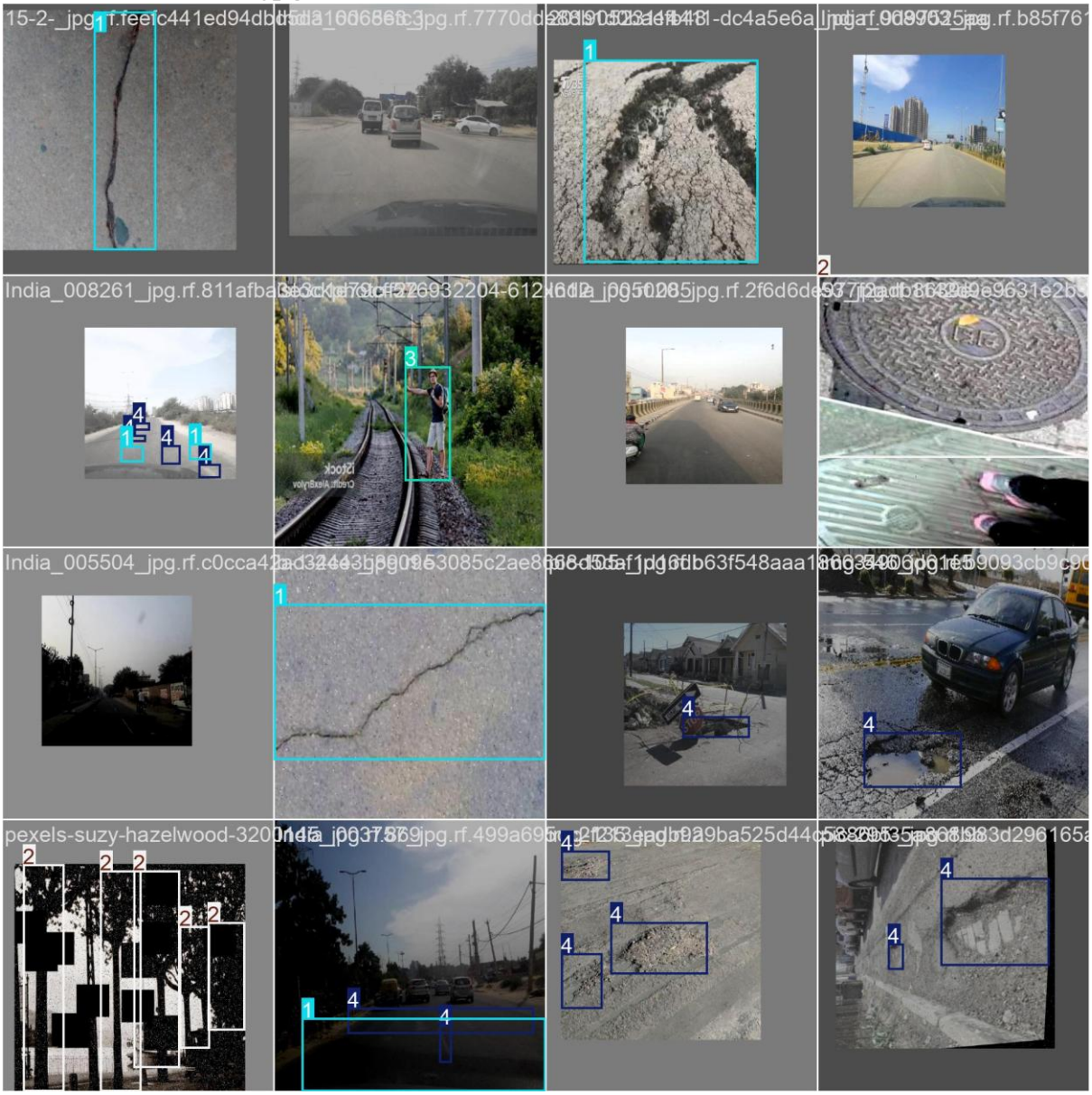
results.png



train_batch229140.jpg

train_batch229141.jpg

train_batch229142.jpg

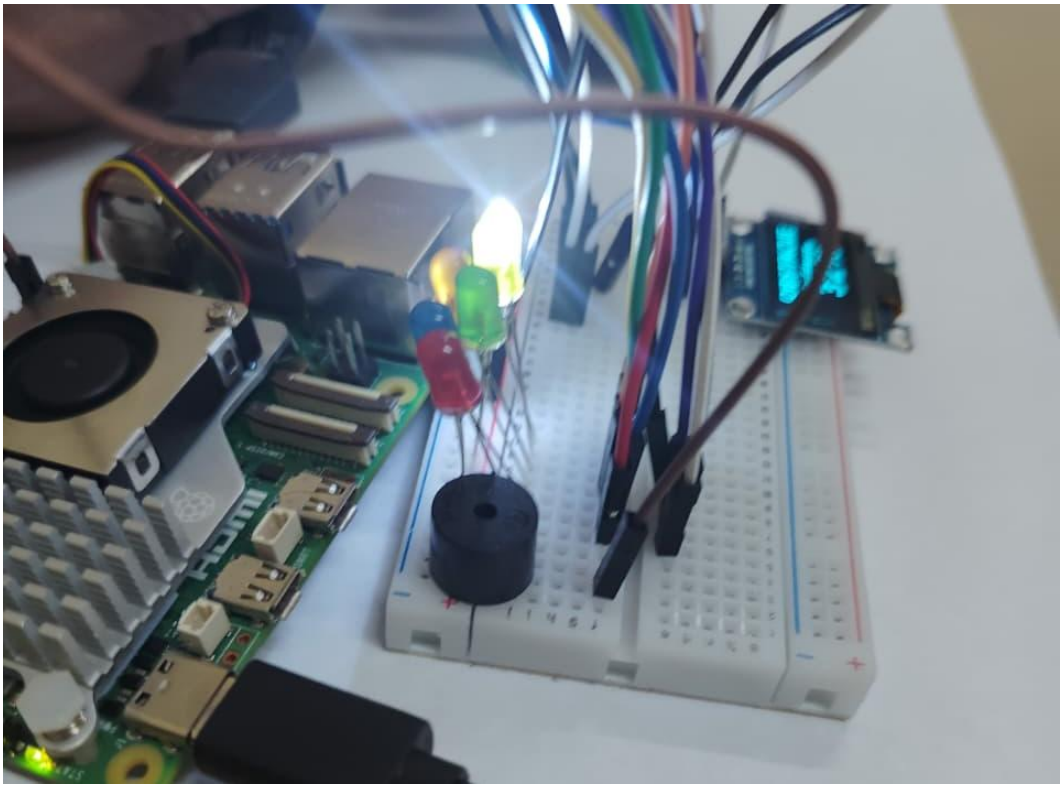


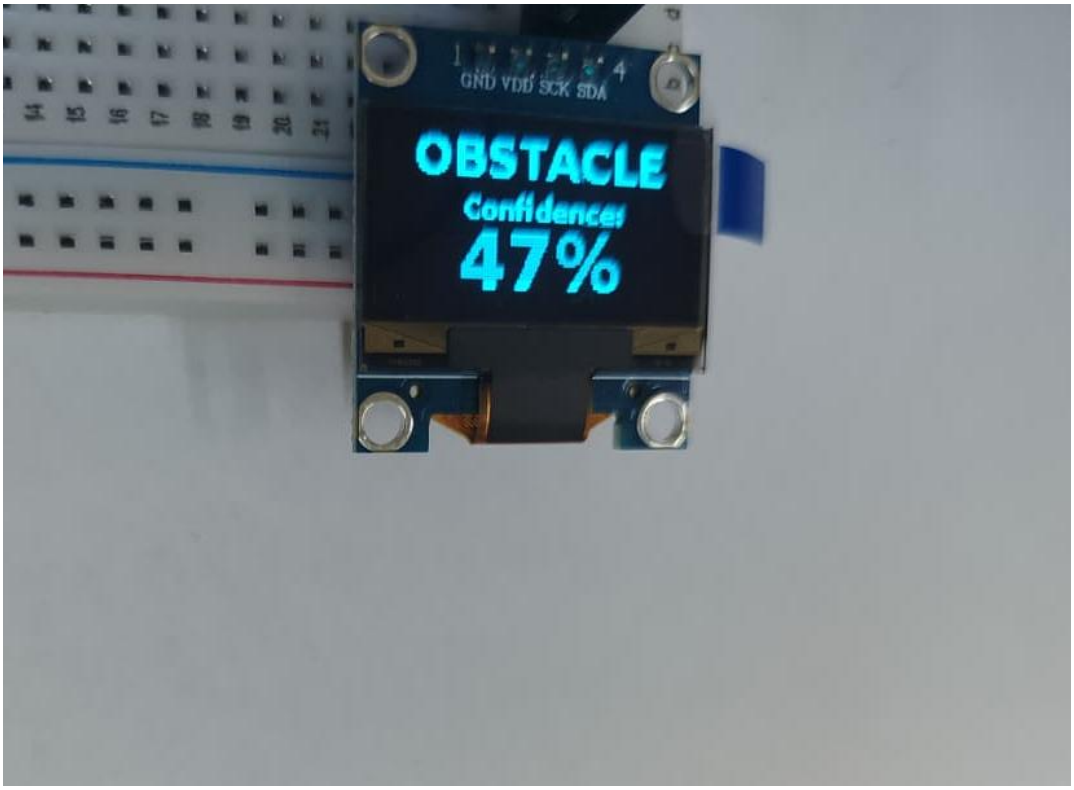
val_batch0_labels.jpg

val_batch0_pred.jpg

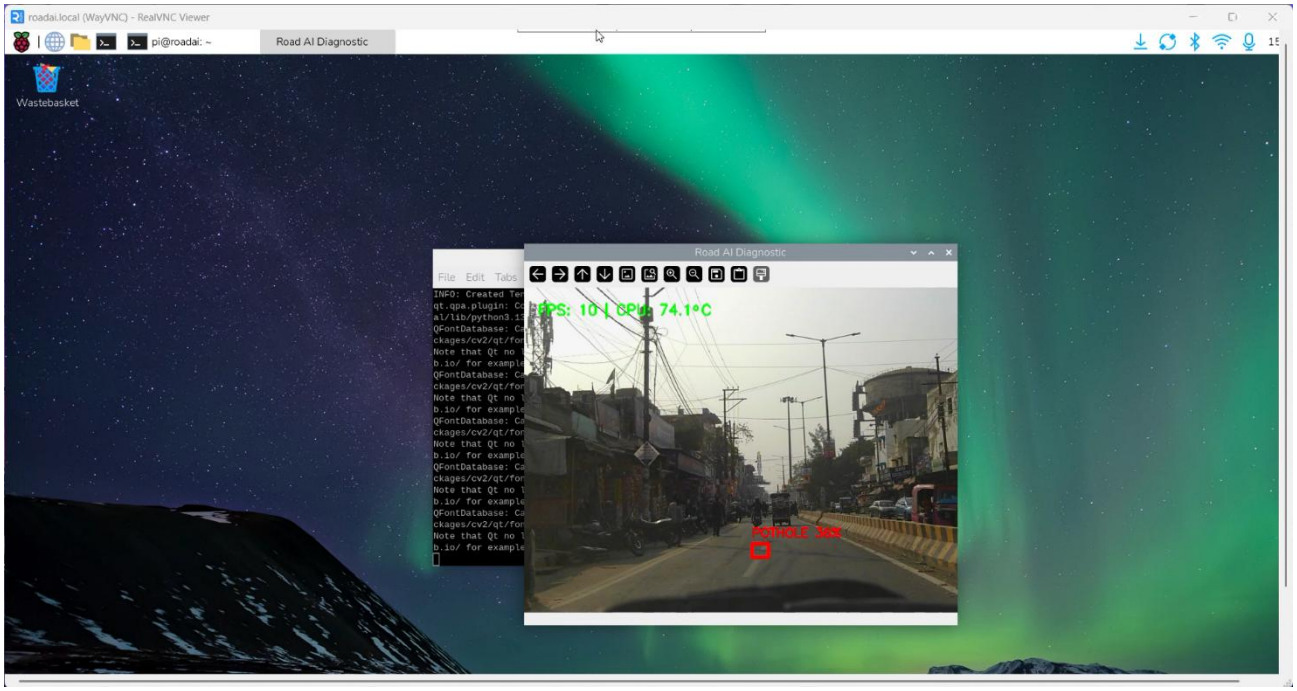
val_batch1_labels.jpg

Working Project Hardware And Software:-





Working Hardware



Working Software

Code:-

```
import cv2
import numpy as np
import tensorflow as tf
import time
import os
import csv
import threading
import queue
import RPi.GPIO as GPIO
from ultralytics import YOLO
from luma.core.interface.serial import i2c
from luma.core.render import canvas
from luma.oled.device import ssd1306
from PIL import ImageFont

# --- DIRECTORY SETUP ---
os.makedirs("logs/images", exist_ok=True)

# --- ASYNC SAVERS ---
save_queue = queue.Queue()
csv_queue = queue.Queue()

def image_saver_worker():
    while True:
        item = save_queue.get()
        if item:
            path, img_data = item
            cv2.imwrite(path, img_data)
        save_queue.task_done()

def csv_worker():
    log_path = "logs/road_data.csv"
    if not os.path.exists(log_path):
```

```
        with open(log_path, 'w', newline='') as f:
            csv.writer(f).writerow(["Timestamp", "Object", "Conf", "Temp", "Image"])
    while True:
        data = csv_queue.get()
        with open(log_path, 'a', newline='') as f:
            csv.writer(f).writerow(data)
        csv_queue.task_done()

threading.Thread(target=image_saver_worker, daemon=True).start()
threading.Thread(target=csv_worker, daemon=True).start()

# --- ENHANCEMENT ---
def enhance_frame(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    if np.mean(gray) < 80:
        img_yuv = cv2.cvtColor(frame, cv2.COLOR_BGR2YUV)
        clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8,8))
        img_yuv[:,:,0] = clahe.apply(img_yuv[:,:,0])
        return cv2.cvtColor(img_yuv, cv2.COLOR_YUV2BGR)
    return frame

# --- GPIO SETUP ---
CLASS_MAP = {
    4: {"label": "POTHOLE", "color": (0, 0, 255), "pin": 17},
    3: {"label": "PERSON",  "color": (255, 0, 0), "pin": 27},
    0: {"label": "ANIMAL",  "color": (0, 255, 0), "pin": 22},
    5: {"label": "VEHICLE", "color": (0, 165, 255), "pin": 23},
    2: {"label": "OBSTACLE","color": (255, 255, 255), "pin": 24},
    1: {"label": "CRACK",   "color": (0, 255, 255), "pin": 25}
}
BUZZER_PIN = 26
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

GPIO.setup(BUZZER_PIN, GPIO.OUT, initial=GPIO.LOW)
for cls in CLASS_MAP.values():
```

```
    GPIO.setup(cls["pin"], GPIO.OUT, initial=GPIO.LOW)

# --- CONFIG ---
MODEL_PATH = "best.tflite"
IMG_SIZE = 640
SMOOTH_BOX = 0.15
SMOOTH_CONF = 0.05
LOG_COOLDOWN = 5
last_log_time = {cid: 0 for cid in CLASS_MAP.keys()}

class State:
    label, conf, running = "SCANNING", 0.0, True
    smoothed_boxes = {}
    smoothed_confs = {}
state = State()

def get_temp_raw():
    return os.popen("vcgencmd measure_temp").readline().replace("temp=","").replace("'C\n","").strip()

def oled_worker():
    try:
        serial = i2c(port=1, address=0x3C)
        device = ssd1306(serial)
        f_path = "/usr/share/fonts/truetype/dejavu/DejaVuSans-Bold.ttf"
        f_m = ImageFont.truetype(f_path, 18); f_s = ImageFont.truetype(f_path, 10); f_h =
ImageFont.truetype(f_path, 28)
        while state.running:
            with canvas(device) as draw:
                draw.text((64, 5), state.label.upper(), font=f_m, fill="white", anchor="mt")
                draw.text((64, 25), "Confidence:", font=f_s, fill="white", anchor="mt")
                draw.text((64, 38), f"{int(state.conf)}%", font=f_h, fill="white", anchor="mt")
            time.sleep(0.1)
    except: pass

threading.Thread(target=oled_worker, daemon=True).start()
```

```
model = YOLO(MODEL_PATH, task="detect")
cap = cv2.VideoCapture(0)
cap.set(3, IMG_SIZE); cap.set(4, IMG_SIZE)
prev_time, buzzer_end = 0, 0

try:
    while True:
        ret, frame = cap.read()
        if not ret: break

        proc_frame = enhance_frame(frame)
        results = model.predict(source=proc_frame, imgsz=IMG_SIZE, conf=0.15, iou=0.2, agnostic_nms=True, verbose=False)

        for cls in CLASS_MAP.values(): GPIO.output(cls["pin"], GPIO.LOW)

        new_boxes, new_confs = {}, {}
        should_save = False
        save_path = ""
        log_ts = ""

        if len(results[0].boxes) > 0:
            top_idx = results[0].boxes.conf.argmax()

            for i, box in enumerate(results[0].boxes):
                cid = int(box.cls[0]); raw_c = float(box.conf[0])
                if cid in CLASS_MAP:
                    # Robust Smoothing
                    val_to_smooth = raw_c * 100
                    p_c = state.smoothed_confs.get(i, val_to_smooth)
                    s_c = (SMOOTH_CONF * val_to_smooth) + ((1 - SMOOTH_CONF) * p_c)
                    new_confs[i] = s_c

                    if i == top_idx:
                        state.conf = s_c
                        state.label = CLASS_MAP[cid]["label"]
```

```
                GPIO.output(CLASS_MAP[cid]["pin"], GPIO.HIGH)
                if cid == 4 and raw_c > 0.5:
                    GPIO.output(BUZZER_PIN, GPIO.HIGH)
                    buzzer_end = time.time() + 0.5

            # Clamped Coordinate Smoothing
            raw_pts = box.xyxy[0].cpu().numpy()
            if i in state.smoothed_boxes:
                raw_pts = (SMOOTH_BOX * raw_pts + (1 - SMOOTH_BOX) * state.smoothed_boxes[i])
            new_boxes[i] = raw_pts

            x1, y1, x2, y2 = np.clip(raw_pts, 0, IMG_SIZE).astype(int)

            if raw_c > 0.20:
                cv2.rectangle(frame, (x1, y1), (x2, y2), CLASS_MAP[cid]["color"], 3)
                cv2.putText(frame, f"{CLASS_MAP[cid]['label']} {int(s_c)}%", (x1, y1-10),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.6, CLASS_MAP[cid]["color"], 2)

            if raw_c > 0.25:
                if (time.time() - last_log_time[cid]) > LOG_COOLDOWN:
                    last_log_time[cid] = time.time()
                    cur_t = get_temp_raw()
                    display_ts = time.strftime("%Y-%m-%d %H:%M:%S")
                    ts_file = time.strftime("%Y-%m-%d_%H-%M-%S")
                    f_path = f"logs/images/{CLASS_MAP[cid]['label']}_{ts_file}.jpg"
                    csv_queue.put([display_ts, CLASS_MAP[cid]['label'], round(raw_c, 2), f"{cur_t}°C", f_path
if raw_c > 0.5 else "N/A"])

                    if raw_c > 0.5:
                        should_save = True
                        save_path = f_path
                        log_ts = display_ts

        state.smoothed_boxes = new_boxes
        state.smoothed_confs = new_confs
    else:
```

```
            state.label, state.conf = "SCANNING", 0
            state.smoothed_boxes.clear(); state.smoothed_confs.clear()

        # --- CAPTURE CLEAN LOG (With boxes, no FPS) ---
        if should_save:
            log_img = frame.copy()
            cv2.putText(log_img, log_ts, (log_img.shape[1]-420, log_img.shape[0]-30),
                        cv2.FONT_HERSHEY_SIMPLEX, 1.0, (0, 0, 0), 3)
            save_queue.put((save_path, log_img))

        # --- MONITOR DIAGNOSTICS (Drawn after snapshot) ---
        curr_t = time.time()
        fps = 1 / (curr_t - prev_time) if prev_time != 0 else 0
        prev_time = curr_t
        t_val = get_temp_raw()
        hud_main = f"FPS: {int(fps)} | CPU: {t_val}"
        cv2.putText(frame, hud_main, (20, 40), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
        text_sz = cv2.getTextSize(hud_main, cv2.FONT_HERSHEY_SIMPLEX, 0.7, 2)[0]
        deg_x = 20 + text_sz[0] + 4
        cv2.circle(frame, (deg_x, 30), 3, (0, 255, 0), 2)
        cv2.putText(frame, "C", (deg_x + 8, 40), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

        if time.time() > buzzer_end: GPIO.output(BUZZER_PIN, GPIO.LOW)
        cv2.imshow("Road AI Diagnostic", frame)
        if cv2.waitKey(1) != -1: break
finally:
    state.running = False
    GPIO.cleanup(); cap.release(); cv2.destroyAllWindows()
```