REAL-TIME ROAD ANOMALY DETECTION USING EDGE AI ON RASPBERRY PI

-TEAM VISIONX

ABSTRACT

Road surface anomalies such as potholes and unexpected obstacles pose serious risks to vehicles and pedestrians, leading to accidents, traffic disruptions, and infrastructure damage. Manual road inspection methods are time-consuming, labor-intensive, and inefficient for large-scale monitoring. To address this issue, this project presents a real-time Road Anomaly Detection System using deep learning deployed on a Raspberry Pi 4 edge device. The system is designed to automatically detect potholes and obstacles from video input and provide structured logging and visual alerts.

The proposed solution utilizes a YOLO-based object detection model trained on a custom dataset consisting of road images captured under varying environmental and lighting conditions. After training, the model was converted to ONNX format for lightweight CPU-based inference suitable for embedded deployment. The inference pipeline includes frame preprocessing, object detection, Non-Maximum Suppression, bounding box visualization, anomaly snapshot storage, and CSV-based logging of detection details.

The system supports both prerecorded video analysis and real-time camera-based detection. Special attention was given to performance optimization to ensure stable operation on resource-constrained hardware. Techniques such as input resolution reduction, frame skipping, and efficient memory handling were implemented to maintain smooth inference performance.

Since prolonged deep learning inference on embedded hardware can generate significant heat, the Raspberry Pi 4 used in this project was equipped with a heat sink to improve thermal dissipation and prevent overheating. This thermal management solution ensured stable long-duration operation without performance throttling, thereby enhancing system reliability.

Experimental evaluation demonstrates that the system successfully detects road anomalies with satisfactory accuracy while operating entirely on CPU without external acceleration. The proposed edge-based solution provides a cost-effective, scalable, and energy-efficient approach for intelligent road monitoring and smart transportation applications.

1. METHODOLOGY

The methodology of the proposed Road Anomaly Detection System is designed to systematically transform raw road data into actionable detection outputs on an embedded edge device. The overall process follows a structured pipeline consisting of data preparation, model development, optimization, deployment, and real-time inference.

The primary objective of the project is to design a deep learning-based system capable of detecting potholes and obstacles on roads in real time using a low-power embedded device. Since Raspberry Pi does not include a dedicated GPU for deep learning inference, the methodology emphasizes lightweight model selection, efficient preprocessing, and runtime optimization.

The development process was divided into five major phases: dataset preparation, model training, model conversion and optimization, deployment on Raspberry Pi, and system validation.

1.1 DATASET PREPARATION

The first step in the methodology involved preparing a structured dataset for supervised object detection training. A combination of publicly available road anomaly images and self-captured road videos was used. Frames were extracted from videos recorded under different environmental conditions including sunny weather, cloudy conditions, varying road textures, and moderate traffic presence. This ensured better model generalization.

The dataset included two primary object classes:

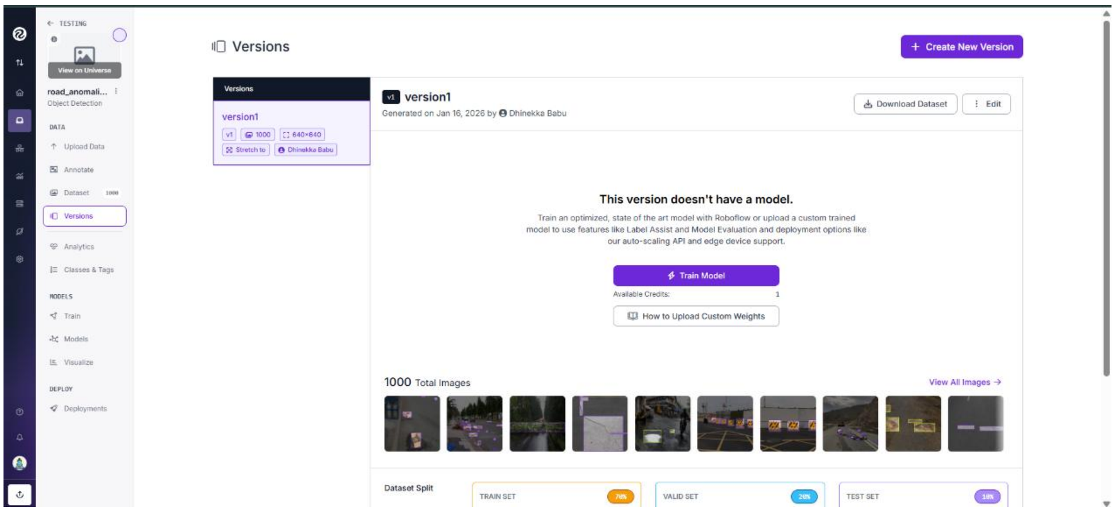
* Pothole

* Obstacle

Each image was manually annotated using bounding boxes. The annotations were created in YOLO format, where each object is represented by class label, normalized center coordinates, and bounding box dimensions. The dataset was structured into organized directories for images and corresponding label files.

To improve model robustness, data augmentation techniques were applied during training. These included horizontal flipping, scaling, brightness adjustments, contrast variations, and minor rotations. Augmentation helps prevent overfitting and improves performance in real-world conditions.

The final dataset was divided into training, validation, and test subsets in an 80:10:10 ratio. The training set was used for model learning, the validation set for hyperparameter tuning, and the test set for final evaluation.



version1
v1 1000 640×640
Stretch to Dhinekka Babu

v1 version1
Generated on Jan 16, 2026 by Dhinekka Babu

This version doesn't have a model.
Train an optimized, state of the art model with Roboflow or upload a custom trained model to use features like Label Assist and Model Evaluation and deployment options like our auto-scaling API and edge device support.

Available Credits: 1

1000 Total Images

Dataset Split
TRAIN SET 70%
VALID SET 20%
TEST SET 10%

DATASET PREPARATION

1.2 MODEL DEVELOPMENT AND TRAINING

The YOLOv5 object detection framework was selected for model development. YOLO (You Only Look Once) is known for its real-time detection capability and efficient single-stage detection mechanism. Compared to two-stage detectors such as

Faster R-CNN, YOLO offers significantly higher inference speed, making it more suitable for edge deployment.

A lightweight YOLOv5 variant was chosen to reduce computational complexity. The model was trained using custom configuration files specifying:

* Number of object classes

* Training image resolution

* Learning rate

* Batch size

* Number of epochs

During training, the model optimized bounding box regression loss, objectness loss, and classification loss simultaneously. Training was monitored using validation metrics such as precision, recall, and mean Average Precision (mAP).

The training process continued until convergence was achieved and validation performance stabilized. Early stopping was used to prevent overfitting. The best-performing model weights were exported after training completion.

1.3 MODEL CONVERSION AND EDGE PREPARATION

Since Raspberry Pi does not efficiently support full PyTorch runtime for inference, the trained model was converted into ONNX (Open Neural Network Exchange) format.

ONNX provides:

* Framework-independent model representation

* Reduced runtime dependencies

* Faster CPU-based inference

* Cross-platform compatibility

The exported ONNX model was verified for correct input and output dimensions before deployment. Input size was fixed at 320 × 320 pixels to balance speed and accuracy.

1.4 INFERENCE PIPELINE DESIGN

The inference pipeline was carefully designed to minimize computational overhead while maintaining detection reliability. The process includes:

1. Capturing frame from video file or camera.

2. Resizing frame to model input dimensions.

3. Normalizing pixel values.

4. Performing ONNX inference using CPUExecutionProvider.

5. Extracting detection predictions.

6. Applying confidence threshold filtering.

7. Applying Non-Maximum Suppression (NMS).

8. Drawing bounding boxes.

9. Logging detections into CSV file.

10. Saving anomaly snapshots with timestamp.

Two modes were implemented:

* Offline video inference mode

* Real-time camera inference mode

Both modes follow the same processing logic but differ in frame source.

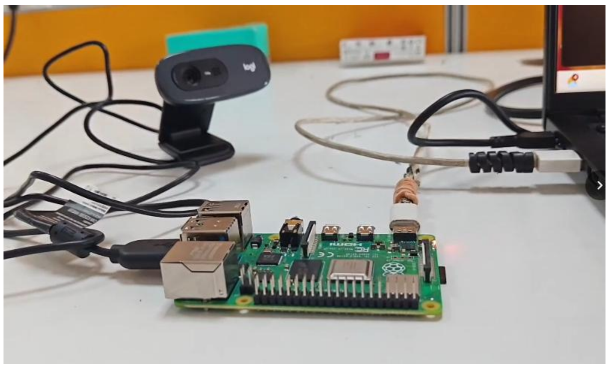
2. HARDWARE UTILIZATION

2.1 RASPBERRY PI 4

The system was deployed on Raspberry Pi 4 (4GB RAM version). The Raspberry Pi served as the edge computing device responsible for model inference and display processing.

Since Raspberry Pi lacks a dedicated GPU for neural network acceleration, inference was executed entirely on CPU using ONNX Runtime. CPU utilization remained within acceptable limits during operation.

RASPBERRY PI 4(8GB) WITH HEAT SINK



PROJECT SETUP

Memory usage was optimized by resizing frames before inference and minimizing background processes.

2.2 CAMERA MODULE

A USB camera was used for real-time frame acquisition. The camera provided continuous video input at standard resolution. Usage of 1080p to inference to reduce computational complexity.

The camera integration ensured real-world testing capability beyond pre-recorded datasets.

2.3 STORAGE AND POWER

The system stored:

* ONNX model file

* CSV log file

* Anomaly snapshots

Snapshot saving was rate-limited to prevent excessive storage consumption.

A stable 5V/3A power supply ensured uninterrupted operation of the Raspberry Pi and camera module.

3. OPTIMIZATION TECHNIQUES

3.1 MODEL OPTIMIZATION

To enable efficient execution on edge hardware, several optimization strategies were applied:

* Selection of lightweight YOLO variant

* Reduction of input image resolution

* Conversion to ONNX format

* Removal of training-only layers

These changes reduced model size and computational complexity.

3.2 RUNTIME OPTIMIZATION

Runtime performance improvements included:

* Frame skipping mechanism to process alternate frames

* Confidence threshold tuning

* Non-Maximum Suppression to eliminate duplicate detections

* Efficient NumPy-based preprocessing

These methods reduced CPU load while maintaining detection reliability.

3.3 COMPUTATIONAL EFFICIENCY

Additional optimizations included:

* Controlled snapshot saving interval

* Lightweight visualization methods

* Efficient memory handling

* Avoidance of unnecessary background tasks

These improvements ensured stable long-duration operation on Raspberry Pi.

4. RESULTS

4.1 TRAINING PERFORMANCE

During training, loss curves demonstrated steady convergence across epochs. Validation precision and recall improved progressively. The final trained model achieved satisfactory detection performance for both potholes and obstacles.

The evaluation metrics considered include:

* Precision: Measures detection correctness.

* Recall: Measures detection completeness.

* F1 Score: Harmonic mean of precision and recall.

* Mean Average Precision (mAP): Measures overall detection quality.

The model demonstrated strong localization capability even in cluttered backgrounds.

4.2 EDGE INFERENCE PERFORMANCE

After deploying the ONNX model on Raspberry Pi 4, inference tests were conducted using both recorded road videos and live camera feed.

The system successfully:

* Detected potholes in varied lighting conditions.

* Identified road obstacles such as debris.

* Filtered false positives using confidence threshold.

* Eliminated duplicate detections using NMS.

Detection bounding boxes were accurately drawn, and anomaly snapshots were saved with timestamps. A CSV logging mechanism recorded class type, confidence score, and bounding box coordinates.

The system remained stable during continuous operation, demonstrating reliability for extended monitoring.

4.3 SYSTEM VALIDATION

The following functional checks were successfully validated:

* Real-time detection overlay display

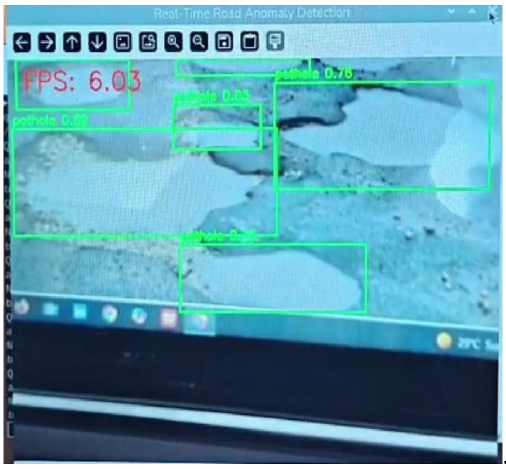
* Snapshot saving at controlled intervals

* CSV logging of anomaly details

* Frame skipping functionality

* Stable CPU utilization

The complete pipeline executed without runtime crashes, confirming integration success.



.

Fig1:Real-time pothole detection using live camera feed

.

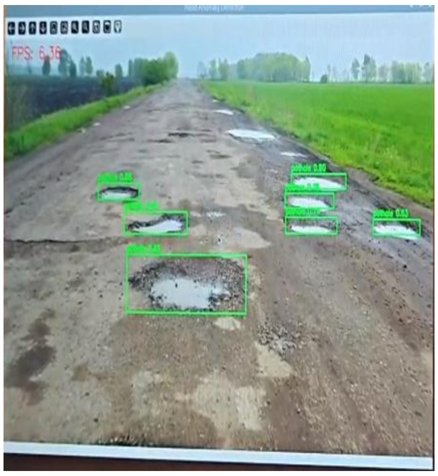


Fig2:Pothole detection Results on pre-recorded sample video

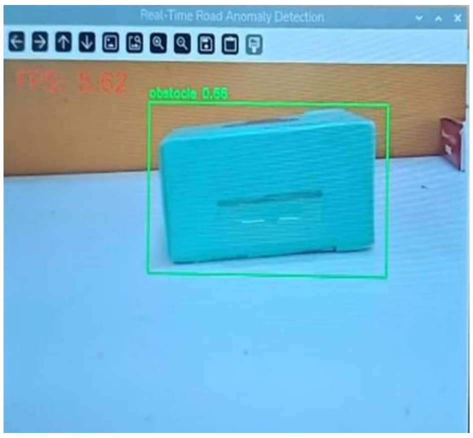


Fig3:Static object detection output demonstrating obstacle recognition

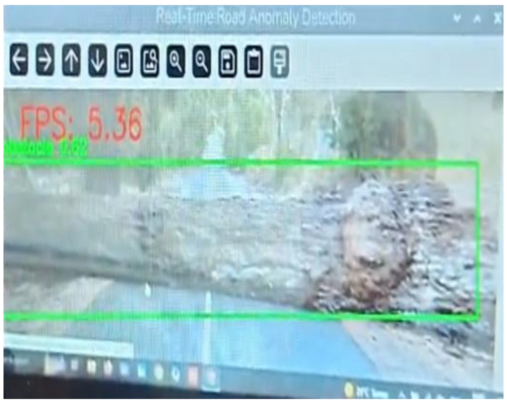
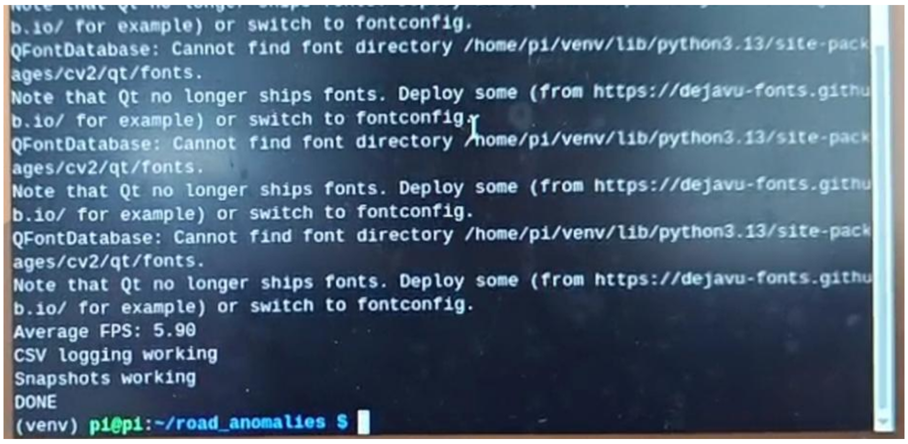
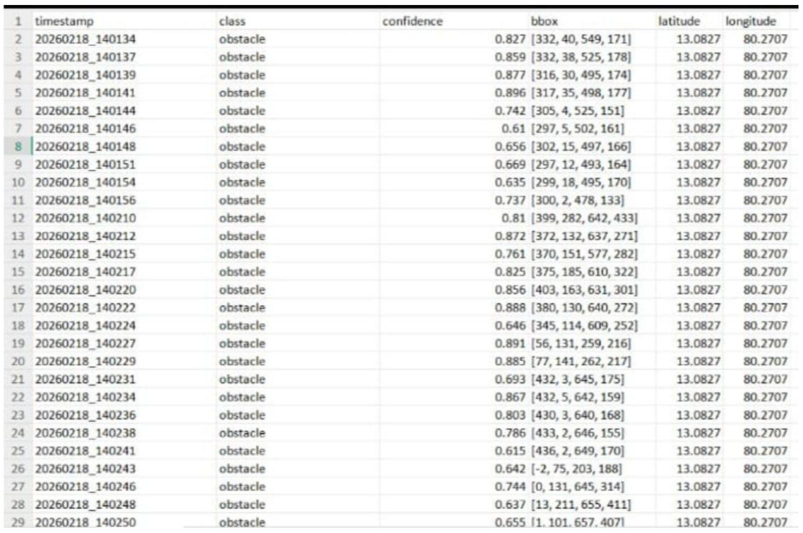


Fig4:Real-time obstacle detection with bounding box visualization



```
b.io/ for example) or switch to fontconfig.
QFontDatabase: Cannot find font directory /home/pi/venv/lib/python3.13/site-pack
ages/cv2/qt/fonts.
Note that Qt no longer ships fonts. Deploy some (from https://dejavu-fonts.githu
b.io/ for example) or switch to fontconfig.
QFontDatabase: Cannot find font directory /home/pi/venv/lib/python3.13/site-pack
ages/cv2/qt/fonts.
Note that Qt no longer ships fonts. Deploy some (from https://dejavu-fonts.githu
b.io/ for example) or switch to fontconfig.
QFontDatabase: Cannot find font directory /home/pi/venv/lib/python3.13/site-pack
ages/cv2/qt/fonts.
Note that Qt no longer ships fonts. Deploy some (from https://dejavu-fonts.githu
b.io/ for example) or switch to fontconfig.
Average FPS: 5.90
CSV logging working
Snapshots working
DONE
(venv) pi@pi:~/road_anomalies $
```

Fig5:Terminal output showing FPS and logging status



1	timestamp	class	confidence	bbox	latitude	longitude
2	20260218_140134	obstacle	0.827	[332, 40, 549, 171]	13.0827	80.2707
3	20260218_140137	obstacle	0.859	[332, 38, 525, 178]	13.0827	80.2707
4	20260218_140139	obstacle	0.877	[316, 30, 495, 174]	13.0827	80.2707
5	20260218_140141	obstacle	0.896	[317, 35, 498, 177]	13.0827	80.2707
6	20260218_140144	obstacle	0.742	[305, 4, 525, 151]	13.0827	80.2707
7	20260218_140146	obstacle	0.61	[297, 5, 502, 161]	13.0827	80.2707
8	20260218_140148	obstacle	0.656	[302, 15, 497, 166]	13.0827	80.2707
9	20260218_140151	obstacle	0.669	[297, 12, 493, 164]	13.0827	80.2707
10	20260218_140154	obstacle	0.635	[299, 18, 495, 170]	13.0827	80.2707
11	20260218_140156	obstacle	0.737	[300, 2, 478, 133]	13.0827	80.2707
12	20260218_140210	obstacle	0.81	[399, 282, 642, 433]	13.0827	80.2707
13	20260218_140212	obstacle	0.872	[372, 132, 637, 271]	13.0827	80.2707
14	20260218_140215	obstacle	0.761	[370, 151, 577, 282]	13.0827	80.2707
15	20260218_140217	obstacle	0.825	[375, 185, 610, 322]	13.0827	80.2707
16	20260218_140220	obstacle	0.856	[403, 163, 631, 301]	13.0827	80.2707
17	20260218_140222	obstacle	0.888	[380, 130, 640, 272]	13.0827	80.2707
18	20260218_140224	obstacle	0.646	[345, 114, 609, 252]	13.0827	80.2707
19	20260218_140227	obstacle	0.891	[56, 131, 259, 216]	13.0827	80.2707
20	20260218_140229	obstacle	0.885	[77, 141, 262, 217]	13.0827	80.2707
21	20260218_140231	obstacle	0.693	[432, 3, 645, 175]	13.0827	80.2707
22	20260218_140234	obstacle	0.867	[432, 5, 642, 159]	13.0827	80.2707
23	20260218_140236	obstacle	0.803	[430, 3, 640, 168]	13.0827	80.2707
24	20260218_140238	obstacle	0.786	[433, 2, 646, 155]	13.0827	80.2707
25	20260218_140241	obstacle	0.615	[436, 2, 649, 170]	13.0827	80.2707
26	20260218_140243	obstacle	0.642	[-2, 75, 203, 188]	13.0827	80.2707
27	20260218_140246	obstacle	0.744	[0, 131, 645, 314]	13.0827	80.2707
28	20260218_140248	obstacle	0.637	[13, 211, 655, 411]	13.0827	80.2707
29	20260218_140250	obstacle	0.655	[1, 101, 657, 407]	13.0827	80.2707

Fig6:Logfile output

CONCLUSION

The project successfully demonstrates that deep learning-based road anomaly detection can be implemented on low-power embedded hardware through careful model selection and optimization. The combination of YOLO-based object detection,

ONNX conversion, CPU-based inference, and runtime optimization enabled real-time pothole and obstacle detection on Raspberry Pi without external accelerators.

The system provides a practical, scalable solution for intelligent road monitoring and smart transportation systems, demonstrating the feasibility of edge AI deployment in real-world scenarios.

REFERENCES

GitHub Repository:
https://github.com/jv681/ARM-PROJECT

Project Demonstration Video:
https://drive.google.com/file/d/1C80E1bSNfvD023Rdxy533496v_0FWSt6/view?usp=drive_link

Model Training Documentation (PDF):
https://drive.google.com/file/d/1GIqbbQs5mYDHFgolbX6ni97PAia0nQdh/view?usp=drive_link