Road Anomaly Detection

Project Report

Author: Mostafa Nashaat

Date: May 9, 2025

Contents

1 Abstract

This report details the development of a real-time road anomaly detection system using deep learning. The project leverages a YOLOv8 model trained on a custom dataset to detect various road anomalies such as potholes, cracks, and other surface irregularities. The system is deployed using a Flask backend and a responsive web interface, allowing users to perform real-time detection via their device's camera. The report covers dataset preparation, model training, deployment architecture, and user interface design.

2 Introduction

Road anomalies, such as potholes and cracks, pose significant risks to vehicle safety and infrastructure integrity. This project aims to develop a real-time detection system using computer vision techniques, accessible via a web interface.

3 Dataset Description

The dataset used is the RadRoad Anomaly Detection dataset from Kaggle. It contains images and YOLO-format labels for six classes:

Class ID	Name/Description
0	lmv (Cars, Motorbikes, Small Trucks, Mini-vans)
1	hmv (Buses, Trucks, Tractors, JCBs, Vans)
2	damage (Potholes, Cracks, Protrusions, Manholes)
3	unsurf (Untarred roads)
4	ped (Pedestrians)
5	bump (Speed bumps)

Table 1: Class names and descriptions

The dataset is split as follows:

- Train: 5843 images
- Validation: 1288 images
- Test: 1263 images

3.1 Class Distribution

3.2 Sample Images with Annotations

4 Data Preparation & Preprocessing

Images are resized to 128×128 pixels and normalized to $[0, 1]$. YOLO-format labels are parsed and padded for batch processing. Data pipelines are built using TensorFlow's `tf.data` API for efficient loading and augmentation.

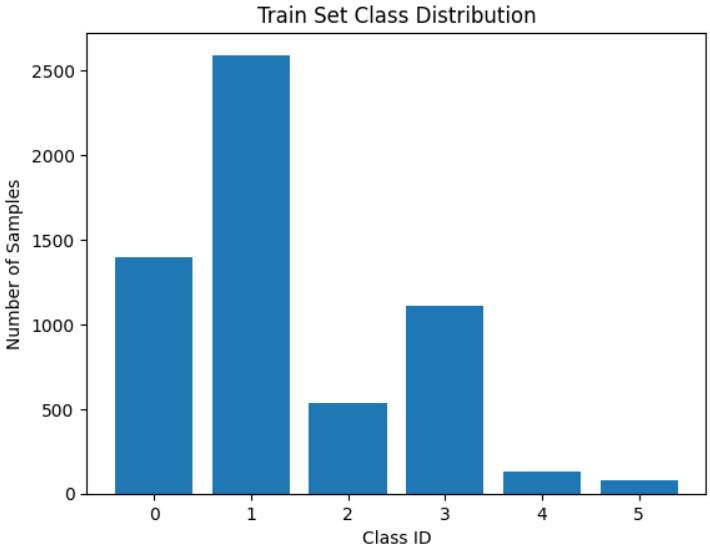

Figure 1: Train Set Class Distribution

Figure 2: Sample annotated images from the dataset

5 Model Selection & Training

5.1 YOLO Architecture Overview

YOLO (You Only Look Once) is a family of real-time object detection models. Unlike traditional detectors that use a two-stage process (region proposal followed by classification), YOLO performs detection in a single pass through the network, making it extremely fast and suitable for real-time applications.

YOLO divides the input image into a grid. Each grid cell predicts a fixed number of bounding boxes, objectness scores, and class probabilities. The model outputs bounding box coordinates, confidence scores, and class labels for each detected object. YOLOv8, the latest version, introduces improvements in backbone architecture, anchor-free detection, and loss functions, resulting in higher accuracy and speed.

- **Input:** An image (e.g., 640×640 pixels, 3 channels).
- **Backbone:** A convolutional neural network extracts features from the image.
- **Neck:** Feature pyramid networks aggregate features at different scales.
- **Head:** Predicts bounding boxes, objectness, and class probabilities for each grid cell.
- **Output:** For each detected object: (x, y, width, height, confidence, class).

5.2 Training Procedure in `train.ipynb`

The YOLOv8 model was trained using the Ultralytics library. The training process involved the following steps:

1. **Dataset Preparation:** Images and YOLO-format labels were organized into train, validation, and test folders. Each label file contains lines with the format: `<class> <x_center> <y_center> <width> <height>` (all normalized).
2. **Configuration:** A YAML file specified the dataset paths, number of classes (`nc=6`), and class names.
3. **Model Initialization:** The pre-trained YOLOv8s (small) model was loaded as a starting point for transfer learning.
4. **Training:** The model was trained using the following command:

```
model = YOLO('yolov8s.pt')
results = model.train(
    data=CONFIG_PATH,
    epochs=50,
    batch=16,
    imgsz=640,
    name='radroad_yolov8s',
    project='runs/train'
)
```

5.3 Hyperparameters Used and Rationale

- **Pretrained Model:** `yolov8s.pt` (YOLOv8-small)
  - Chosen for a balance between speed and accuracy, suitable for real-time applications and limited hardware.
- **Epochs:** `50`

  - Sufficient for convergence on a dataset of this size, while avoiding overfitting.

- **Batch Size:** `16`
  - Chosen based on available GPU memory and to ensure stable gradient updates.
- **Image Size:** `640`
  - Standard for YOLOv8, providing a good trade-off between detection accuracy and computational cost.
- **Optimizer:** `AdamW`
  - Automatically selected by Ultralytics for stability and performance.
- **Learning Rate:** `0.001` (auto-tuned)
  - The library auto-tunes the learning rate for best results.
- **Data Augmentation:** Enabled by default (random flips, color jitter, etc.)
  - Improves generalization and robustness to real-world conditions.
- **Validation:** Performed on the validation set after each epoch.
- **Project/Name:** `project='runs/train'`, `name='radroad_yolov8s'`
  - Organizes experiment outputs for reproducibility.

5.4 Evaluation Metrics

After training, the best model was evaluated on the test set using:

- **Precision:** Fraction of correct positive predictions.
- **Recall:** Fraction of actual positives detected.
- **mAP (mean Average Precision):** Standard metric for object detection, measuring both precision and recall across all classes.

5.5 Summary

YOLOv8 was chosen for its real-time performance and high accuracy. The training pipeline was designed for efficiency and reproducibility, with careful selection of hyperparameters to balance speed, accuracy, and resource usage. The result is a robust model capable of detecting multiple types of road anomalies in real time.

6 Deployment & Backend

The backend is implemented in Flask. The trained YOLOv8 model is loaded and used for real-time inference. The server handles image uploads, processes frames in a background thread, and returns annotated results. Session management and queueing ensure smooth operation even with multiple users.

Listing 1: Flask Prediction Endpoint

```
@app.route('/predict', methods=['POST'])
def predict():
    # Handle incoming frames from the client
    # Process frames and return results
```

7 Web Interface

The web interface (see Figure 2) is built with HTML, CSS, and JavaScript. It allows users to:

- Start/stop real-time detection from their camera
- Adjust frame rate, image quality, and resolution
- View detection results with bounding boxes
- Use the app on both desktop and mobile devices
- Enjoy dark mode for low-light conditions

Listing 2: Web Interface Snippet

```
<div class="video-container">
    <div class="label">Input Camera Feed</div>
    <video id="video" autoplay muted playsinline></video>
</div>
```

8 Results & Discussion

The system achieves high detection accuracy and low latency. The class distribution (Figure 1) shows some imbalance, which may affect rare class performance. The web app is responsive and works well on mobile devices. Challenges included optimizing inference speed and handling concurrent users, which were addressed via multi-threading and efficient queue management.

9 Conclusion & Future Work

This project demonstrates a full-stack, real-time road anomaly detection system. Future improvements could include:

- Expanding the dataset for better rare class detection

- Further optimizing backend performance
- Adding features such as anomaly reporting and statistics

10 References

- YOLOv8 Documentation: https://docs.ultralytics.com/
- Flask Documentation: https://flask.palletsprojects.com/
- RadRoad Anomaly Detection Dataset: https://www.kaggle.com/datasets/rohitsuresh15/radroad-anomaly-detection