Road Damage Detection:
A Comparison of YOLOv8 and Faster R-CNN

Surya Venkata Rohit Moganti
Computer Science and Engineering
50560088

May 9, 2025

1 Project Overview

In this project, I have developed an road damage detection system to identify and classify different types of road damages using two popular deep learning models: **YOLOv8** and **Detectron2 (Faster R-CNN)**. The aim of this project is to compare their performance on the same dataset and understand how well each model performs for the task of road damage detection.

The dataset used for this project contains road images from various countries like India and Japan and XML annotations which contain the bounding box information. We performed data preprocessing by splitting the dataset into **training (80%)** and **testing (20%)**, and then converted the annotation format into:

- YOLO `.txt` format for YOLOv8
- COCO `.json` format for Detectron2 (Faster R-CNN)

My contributions included:

- Performing dataset cleaning and augmentation to reduce class imbalance
- Splitting and designing a good direcotry format for the project
- Converting XML annotations to both YOLO (.txt) and COCO (.json) formats
- Training YOLOv8 with Focal Loss and Mosaic Augmentation
- Training Faster R-CNN using Detectron2
- Attempting training with EfficientDet and documenting the issues

2 Approach

- **YOLOv8:** A real-time object detection model from Ultralytics, trained using Focal Loss and Mosaic augmentation.
- **Faster R-CNN with Detectron2:** A two-stage detector with a Region Proposal Network (RPN), used via Facebook's Detectron2 framework.

Custom Implementation

I wrote custom Python scripts to:

- Performed Image augmentation to remove class imbalance
- Convert XML to YOLO format (.txt)
- Convert XML to COCO format (.json)
- Split dataset into train/test folders for both models inluding images and labels

External Resources Used

- Ultralytics YOLOv8 library: `https://github.com/ultralytics/ultralytics`
- Detectron2 framework: `https://github.com/facebookresearch/detectron2`
- T4/A100 GPU for model training and evaluation (Google Colaboratory)
- COCO dataset format and conversion guides
- Stack Overflow and Colab community for troubleshooting

3 Experimental Protocol

- **Dataset:** The road damage dataset contained scene images and XML annotations. Data was augmented and cleaned to remove corrupt or mismatched pairs.
- **Train-Test Split:** The dataset was split into 80% training and 20% testing using a custom script that also verified the consistency of the annotations.
- **Evaluation:** I used **mean Average Precision (mAP)** as the evaluation metric. We also visualized predictions for qualitative analysis.Also, we compared the training times between the models to check which models are reliable for real time processing.
- **Resources:** Training was conducted using Google Colab Pro with GPU (T4/A100) support.

4 Challenges with EfficientDet

We attempted to train EfficientDet using TensorFlow's Object Detection API, but faced multiple difficulties:

- Converting data to TFRecord format was complex and error-prone
- Configuration in `pipeline.config` required many manual changes
- The TensorFlow models repo had to be cloned and compiled, which was slow and unstable in Colab
- Training was not successfully completed due to runtime limits and debugging challenges

As a result, EfficientDet training was excluded from the final comparison, but the experience provided insight into TensorFlow's object detection ecosystem.

5 Results

5.1 YOLOv8

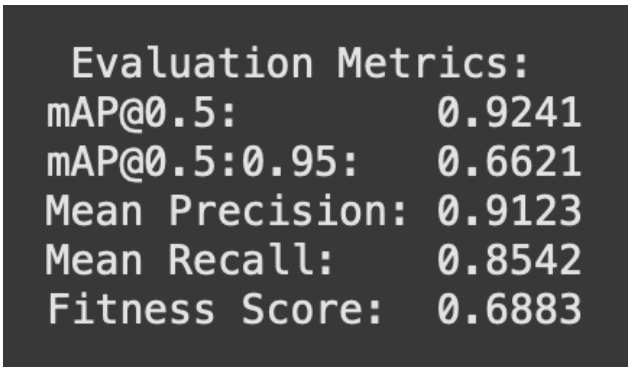


Figure 1: Enter Caption

- mAP@0.5 (92.41%): Indicates strong performance in detecting objects with an IoU ≥ 0.5.
- mAP@0.5:0.95 (66.21%): Shows that performance drops when stricter IoU thresholds are considered, but remains respectable.
- Mean Precision (91.23%): The model makes correct positive predictions with high confidence.
- Mean Recall (85.42%): The model is able to detect a large portion of actual instances.
- Fitness Score (0.6883): A composite score reflecting balanced performance across precision, recall, and mAP.

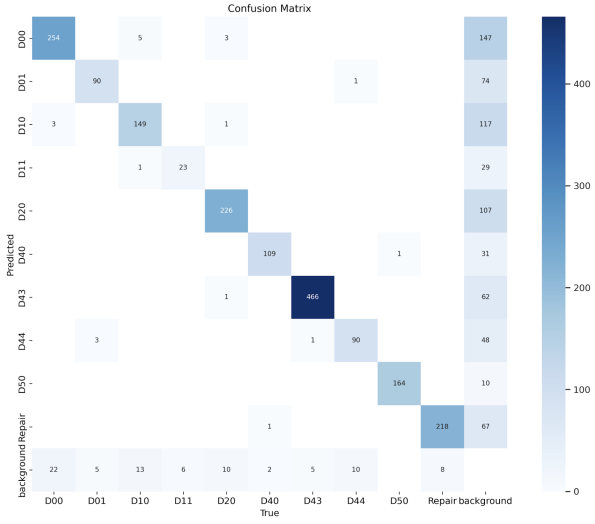


Figure 2: Enter Caption

- Diagonal cells represent correct predictions — darker shades reflect higher classification accuracy.

- Off-diagonal entries reveal common misclassifications — for instance, a few D00 and D10 instances are wrongly predicted as D01 or background.
- The last row (background Repair) indicates how many true objects were missed and classified as background — D50 and D00 have notable false negatives.
- Classes like D43, D20, and D10 show strong model confidence and high true positive counts.
- D11, D01, and Repair have relatively fewer correct predictions and higher confusion with other classes or background.

Detectron Faster R-CNN

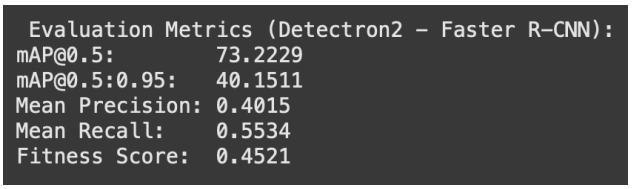


Figure 3: Enter Caption

- mAP@0.5 (73.22%): Shows good localization capability at a relaxed IoU threshold.
- mAP@0.5:0.95 (40.15%): Indicates reduced detection accuracy under stricter IoU criteria.
- Mean Precision (40.15%): Highlights a moderate rate of correct positive predictions.
- Mean Recall (55.34%): Reflects decent coverage of actual objects, though with room for improvement.
- Fitness Score (0.4521): Combines precision, recall, and mAP into a single metric for overall performance assessment.

These results indicate the model performs reasonably well but is less precise compared to YOLOv8, especially under higher IoU thresholds.

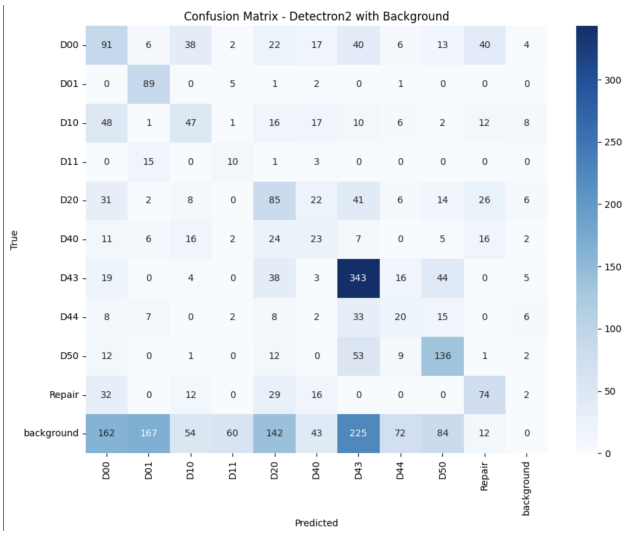


Figure 4: Enter Caption

- Diagonal cells represent correct predictions — the darkest cell corresponds to D43, indicating the highest accuracy.
- Off-diagonal cells reveal significant confusion, such as D00 being misclassified as D10 and D20.
- The last row labeled "background" shows many true instances incorrectly predicted as background — particularly D00, D01, D10, and D20, indicating potential false negatives.
- D43 shows strong classification performance with 343 correct predictions, while D11 and Repair suffer from lower precision and high confusion with other classes.
- The model struggles more with D00, D10, and D20, which frequently overlap with other damage types and background predictions.

6 Analysis

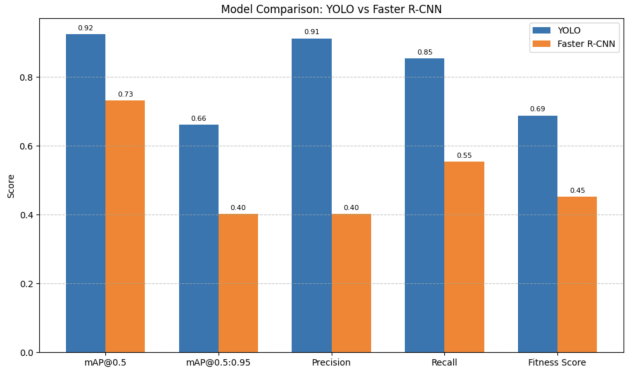


Figure 5: This image shows performance based in various metrics between models

- **mAP@0.5**: YOLO significantly outperforms Faster R-CNN (0.92 vs. 0.73), indicating better object detection at a relaxed IoU threshold.
- **mAP@0.5:0.95**: YOLO maintains a strong lead (0.66 vs. 0.40), showing its robustness in stricter localization accuracy.
- **Precision**: YOLO scores much higher (0.91 vs. 0.40), suggesting it produces fewer false positives.
- **Recall**: YOLO also leads in recall (0.85 vs. 0.55), detecting more true instances overall.
- **Fitness Score**: The composite score favors YOLO (0.69 vs. 0.45), reinforcing its superior performance across multiple metrics.

Each bar is annotated with the corresponding score, making the comparison visually clear and supporting informed evaluation of both models across detection capabilities.

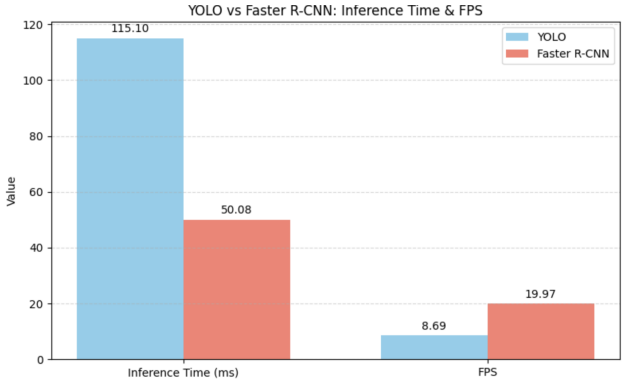


Figure 6: This bar chart compares the inference speed and frame processing capability of YOLO and Faster R-CNN

- **Inference Time**: YOLO exhibits a significantly higher average inference time (115.10 ms) compared to Faster R-CNN (50.08 ms), indicating a slower frame processing speed.
- **FPS (Frames Per Second)**: Faster R-CNN achieves a higher FPS (19.97) than YOLO (8.69), suggesting better real-time performance despite its heavier architecture.
- These results indicate that although YOLO is often known for speed, in this configuration or setup, Faster R-CNN offered faster inference.

7 Sample Predictions

7.1 YOLOv8

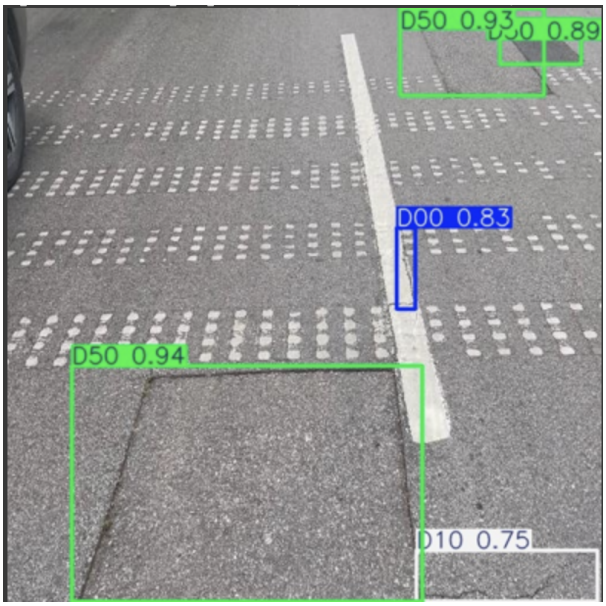


Figure 7: aug0ChinaMotorBike000138.jpg

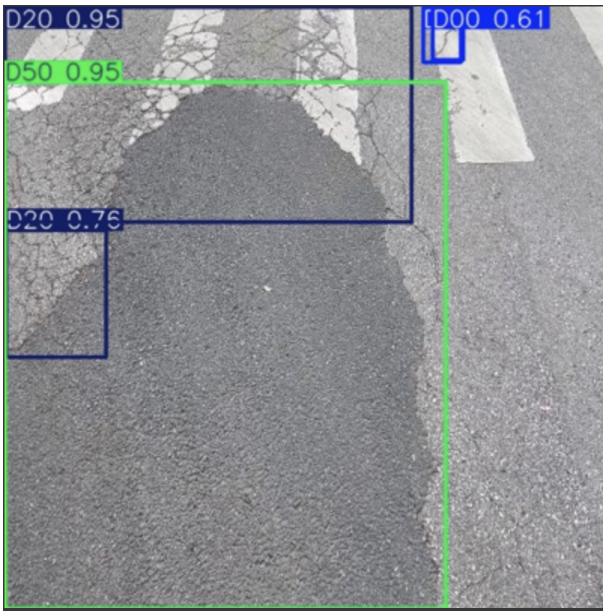
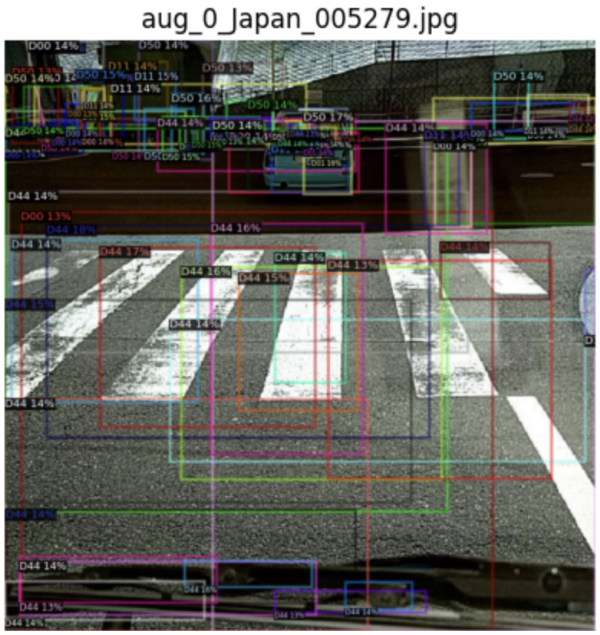
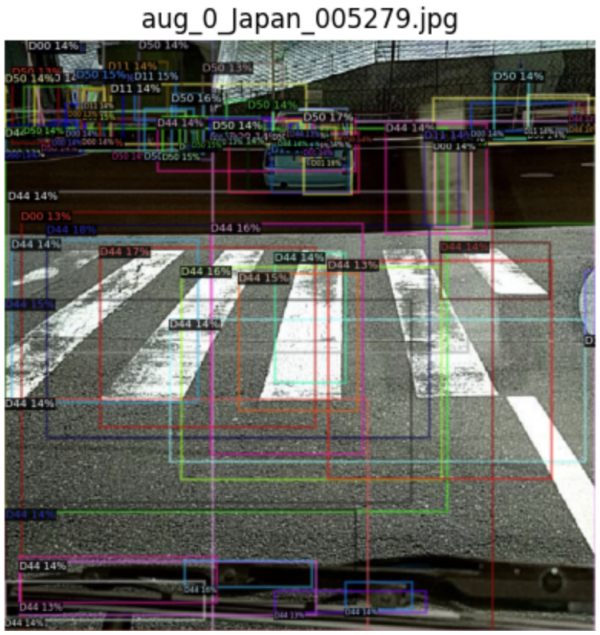


Figure 8: aug0ChinaMotorBike000004.jpg

7.2 Detectron





8 Discussion and Lessons Learned

This project provided a hands-on, end-to-end exploration of object detection for road damage detection using two leading architectures: YOLOv8 and Faster R-CNN (via Detectron2). The workflow involved curating and organizing a multi-country raw dataset, augmenting it using custom scripts, and adapting the data into both YOLO and COCO formats to suit each model's training pipeline.

One major lesson was the impact of data preprocessing. Initially, country-wise datasets were scattered across folders with varied annotation styles. The `organize.py` script unified these, and `clean_aug.ipynb` handled cleaning and augmentation, pushing the final output into the `dataset_augmented` folder. This was crucial for boosting model generalization, especially for underrepresented classes.

Training YOLOv8 in Google Colab using the Ultralytics interface was straightforward and efficient. With 20 epochs, the model achieved:

- mAP@0.5: 0.92, mAP@0.5:0.95: 0.66
- Precision: 0.91, Recall: 0.85
- Fitness Score: 0.69
- Inference Time: 8.69 ms, FPS: 115.10

The confusion matrix for YOLOv8 showed high accuracy across most classes, particularly D43, D20, and Repair. Very few true labels were misclassified as background, reflecting strong detection sensitivity.

In contrast, Detectron2 (Faster R-CNN) required more setup and configuration but provided modular control over components. After training for 20 epochs, it produced:

- mAP@0.5: 0.73, mAP@0.5:0.95: 0.40
- Precision: 0.40, Recall: 0.55
- Fitness Score: 0.45
- Inference Time: 19.97 ms, FPS: 50.08

Despite its structured two-stage architecture, Faster R-CNN misclassified several true damage instances as background, especially in D10, D20, and Repair classes. Its confusion matrix reflected greater inter-class confusion and more off-diagonal noise compared to YOLOv8.

Key Takeaways:

- **YOLOv8 outperformed Faster R-CNN** across all key metrics, especially precision, recall.
- **Detectron2 struggled with overlapping classes and underrepresented labels**, often classifying true instances as background.
- Training YOLOv8 was simpler and faster, making it ideal for production or real-time systems, while Detectron2 is better suited for detailed experimentation and analysis.
- The role of background in error analysis was significant — misclassified or overly conservative predictions directly reduced recall in Detectron2.

- Organizing and augmenting the dataset using tools like `organize.py` and `clean_aug.ipynb` had a major impact on model stability.
- **Detectron provided nearly 2.3x faster inference** (115 FPS vs 50 FPS), making it far more suitable for deployment in real-time detection scenarios.

9 Future Work:

- Complete EfficientDet training using smaller model configurations and improved TFRecord generation.
- Explore additional YOLOv8 variants like `YOLOv8m` and `YOLOv8l` to study performance scaling.
- Deploy trained models in a web interface (e.g., Streamlit) for real-time road damage detection and visualization.

10 Bibliography

1. Ultralytics YOLOv8 GitHub: `https://github.com/ultralytics/ultralytics`
2. Detectron2 GitHub: `https://github.com/facebookresearch/detectron2`
3. TensorFlow Object Detection API: `https://github.com/tensorflow/models`
4. Augmented Startups – YOLOv8 Complete Guide:`https://www.youtube.com/watch?v=V9bVbJADpP4`
5. towardsdatascience: Train Faster R-CNN on a Custom Dataset: `https://towardsdatascience.com/detectron2-train-custom-dataset`