Real-Time Scrap Classifier & Robotic Pick Simulation

Name: Mallikarjun Reddy Bardipuram

Submission For: Computer Vision Engineer Intern Assignment, AlfaStack

Date: October 17, 2025

Problem Understanding

Start by clearly stating the project's objective as you understood it.

- The primary goal was to design and build a proof-of-concept simulation of an AI-powered industrial scrap sorting system.
- The system needed to perform three core tasks in real-time:
    1. **Detect** various types of scrap items from a video feed.
    2. **Classify** each item into a specific material category.(Plastic, metal, Biodegradable, cardboard, glass, paper)
    3. **Generate a pick-point** (center coordinate) for each detected item, simulating a target for a robotic arm.
- The solution had to be built using open-source tools like YOLOv8, OpenCV, and Streamlit, as specified in the assignment.

Step-by-Step Breakdown (My Approach)

This is the narrative of your project. Describe your workflow from start to finish.

Phase 1: Data Exploration & Analysis

- My first step was to research and find a suitable public dataset for garbage object detection.
- I began with a large, popular dataset from Kaggle. Upon initial analysis, I discovered a **severe class imbalance**. For example, one class ('BIODEGRADABLE') comprised over 72% of the total instances, while critical minority classes like 'PAPER' had less than 0.2%.
- This analysis was crucial because training on such a skewed dataset would result in a model heavily biased towards the majority class, making it useless for reliably sorting different types of scrap.

Phase 2: Strategic Data Curation

- Recognizing the critical data quality issue, I decided to find a better dataset. My search led me to the **"Garbage Detection – 6 Waste Categories"** dataset on Kaggle.
- While this dataset also exhibited a moderate imbalance, it was far more manageable. Most importantly, every key class (GLASS, METAL, CARDBOARD, etc.) had a sufficient number of instances (hundreds or thousands) for a model to learn from.
- This decision to prioritize data quality and balance over sheer data size was the most critical step in ensuring the project's success.

Phase 3: Model Training

- I chose a lightweight **YOLOv8n** model for its excellent balance of speed and accuracy, making it ideal for real-time applications.
- The model was fine-tuned on the curated dataset using Google Colab to leverage GPU acceleration, which reduced training time from hours to minutes and allowed for a higher number of training epochs (50-75).
- The final trained model (Best.pt) demonstrated strong performance on the validation set, proving its ability to accurately detect and classify the target scrap categories.

Phase 4: Real-Time Application Development

- I developed the final application as an interactive web dashboard using **Streamlit**.
- The dashboard features a **live webcam feed** to simulate a conveyor belt. The YOLOv8 model processes this feed in real-time to overlay bounding boxes and class labels.
- For each detection, a "pick-point" crosshair is drawn at the center of the bounding box.
- A **statistics panel** on the side of the dashboard displays a live, running count of each detected scrap category.
- Finally, I added a feature in the sidebar to allow a user to **upload a single image** to test the model's performance on static files.

Key Decisions

Isolate and explain the most important strategic choices you made.

- **Decision 1: Switching Datasets to Prioritize Data Quality.** My most important decision was to abandon the initial, severely imbalanced dataset. Instead of attempting to use complex techniques to mitigate the imbalance, I concluded that starting with a higher-quality, better-balanced dataset would yield a fundamentally better model. This reflects an engineering principle that a good model cannot be built on bad data.
- **Decision 2: Implementing a Thread-Safe Queue for Dashboard Performance.** The initial Streamlit dashboard froze because the model inference was blocking the video stream. My key technical decision was to re-architect the application to use a thread-safe queue. This decoupled the heavy computation of the AI model from the user interface rendering, resulting in a smooth and responsive real-time experience.

Challenges and Learnings

This is where you show deep understanding by reflecting on the problems you solved.

- **Challenge: Severe Class Imbalance.** I learned firsthand how class imbalance can make a dataset practically unusable. A model trained on it would have high "accuracy" by simply guessing the majority class, but it would fail at its real-world task of differentiation. This solidified my understanding of the importance of thorough data analysis before training.
- **Challenge: Real-Time Performance Bottlenecks.** I learned that running deep learning models in real-time involves more than just a fast model. I faced a significant performance issue with the Streamlit dashboard freezing. The solution taught me about the practical challenges of multi-threading in web applications and how to solve them by decoupling processes with a queue, a common pattern in software engineering.
- **Learning: The "Domain Gap".** I observed that the model's performance on my webcam was sensitive to lighting and background. This illustrates the "domain gap" between the varied, real-world images in the training set and the specific conditions of my live demo. It taught me the importance of controlling the environment during testing or augmenting a dataset to be more robust to different conditions.

How You Would Optimize for Edge Deployment

This section shows you're thinking about real-world application, a key requirement from the prompt.

- For deployment on a low-power edge device like a Jetson Nano, several optimizations would be critical:

- **Model Quantization:** I would convert the trained model from its 32-bit floating-point format (FP32) to a more efficient 8-bit integer format (INT8) using a tool like **NVIDIA's TensorRT**. This dramatically reduces the model's size and increases inference speed with a minimal drop in accuracy.
- **Hardware Acceleration:** I would leverage hardware-specific libraries, such as NVIDIA's DeepStream SDK, to ensure the entire vision pipeline (decoding, resizing, inference, and rendering) is accelerated on the GPU, minimizing CPU usage.
- **Frame Rate Optimization:** Instead of running the model on every single frame from the camera, I could implement a "frame skipping" or "throttling" logic. For a slow-moving conveyor belt, running inference on every 3rd or 5th frame might be sufficient, significantly reducing the computational load.