

Space Station Object Detection using YOLOv8

Team: Codies

Team Members: Vaibhav Sharma, Samarth Sharma, Prithvi Singh, Parth Garg

Hackathon: BuildWithIndia 2.0 (SunHacks) – Duality AI Challenge

GitHub Repo: github.com/Vaibhav0120/HackWithDelhi2.0

Dataset Download: [Duality Falcon Space Dataset](#) (~3.9GB)

Summary

In this project, we designed and trained a computer vision model using **YOLOv8** to perform real-time object detection in a simulated space station environment. The primary goal was to detect essential safety equipment from Falcon-generated synthetic data — particularly:







- **Fire Extinguisher**
- **Toolbox**
- **Oxygen Tank**

We achieved high detection performance, with:

- **mAP@0.5: 94.2%**
- **mAP@0.5:0.95: 88.4%**
- Real-time inference using ONNX export
- Inference-ready for Flutter integration

This solution demonstrates the power of synthetic data and modern deep learning architectures to train robust models in domain-specific environments like aerospace.

Highlights

-  Trained YOLOv8 model on 3-class Falcon simulation dataset
-  Validated with class-wise precision/recall & mAP
-  Exported to ONNX for lightweight integration
-  Inference-compatible with mobile apps (Flutter planned)
-  Organized & reproducible pipeline with notebook + scripts
-  Visual results include predictions, labels, and training batches

2. Methodology

This section outlines our step-by-step approach to training and deploying a YOLOv8-based object detection model for space station environments.

Step 1: Dataset Setup

We used the official Duality AI synthetic dataset featuring 3 object classes:

- **FireExtinguisher**
- **ToolBox**
- **OxygenTank**

Due to its size (~3.9GB), the dataset is not in this repo.

Download it from: [Falcon Dataset Link](#)

Once downloaded:

- Place the extracted `data/` folder in the root of the repo.

Step 2: Model Training (YOLOv8)

We trained a YOLOv8m model using 50 epochs and image size 640×640.

Training was done via Python API inside `Train_YOLOv8.ipynb`:

```
from ultralytics import YOLO

model = YOLO('yolov8m.pt') # Base weights

model.train(data='data.yaml', epochs=50, imgsz=640)
```

Step 3: Validation

Post-training, we validated performance using:

```
metrics = model.val()
```

This generated mAP scores and class-wise precision/recall.

Step 4: Prediction

Predictions on test images (`data/predict/images/`) were made via:

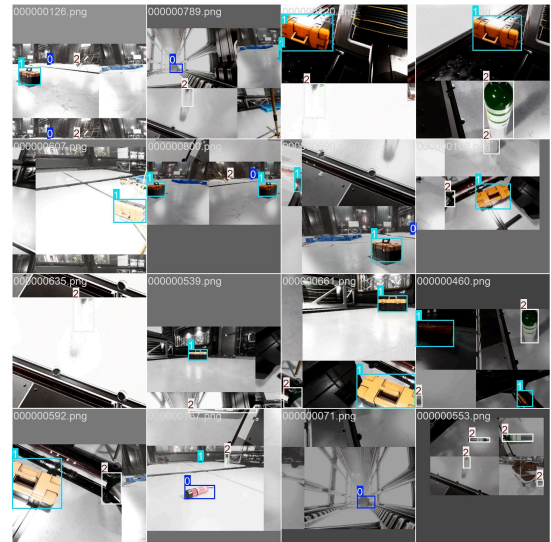
```
model.predict(source='data/predict/images', save=True)
```

3. Visual Results & Samples

To assess model behavior, we visualized training samples, label annotations, and YOLOv8 predictions.

Sample Training Batch →

This is a training image with labeled bounding boxes

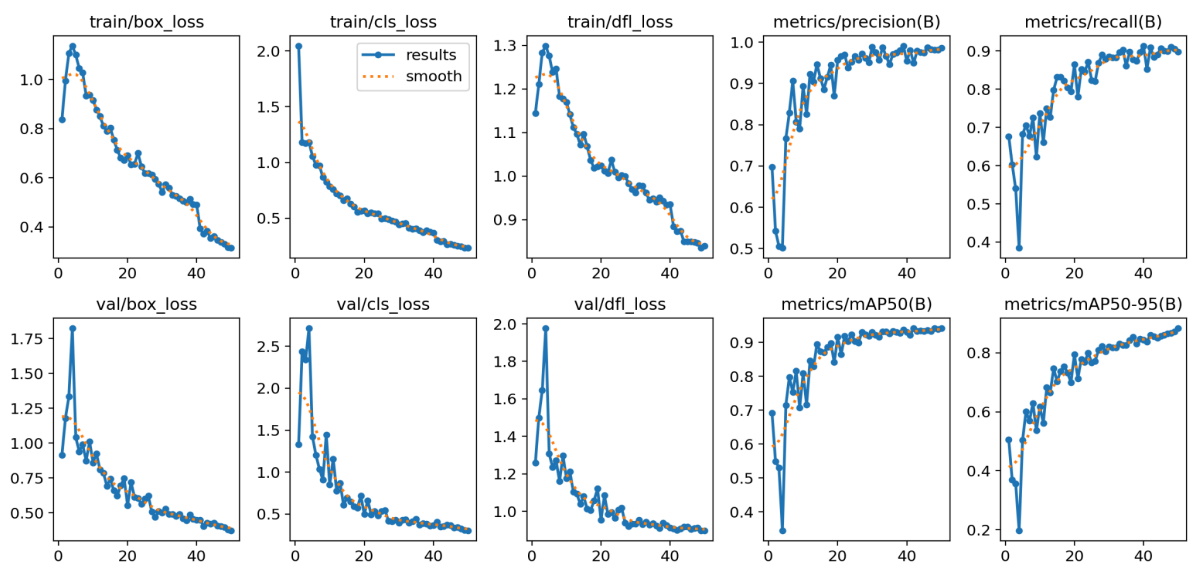


← YOLOv8 Prediction Output

This shows the model's prediction on a validation image

Training Loss & Metrics Curves

Below is the loss and accuracy graph generated during training:



4. Performance Evaluation

Quantitative Evaluation

We validated the trained model on the `val/` set using YOLOv8's built-in evaluation tools.

Validation command used (inside notebook):

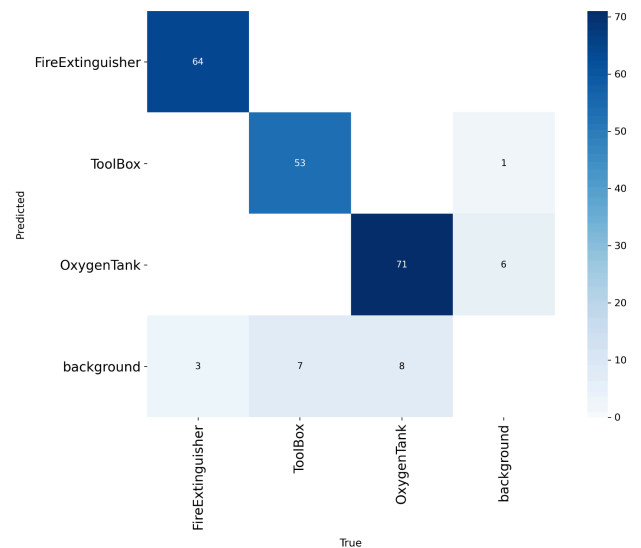
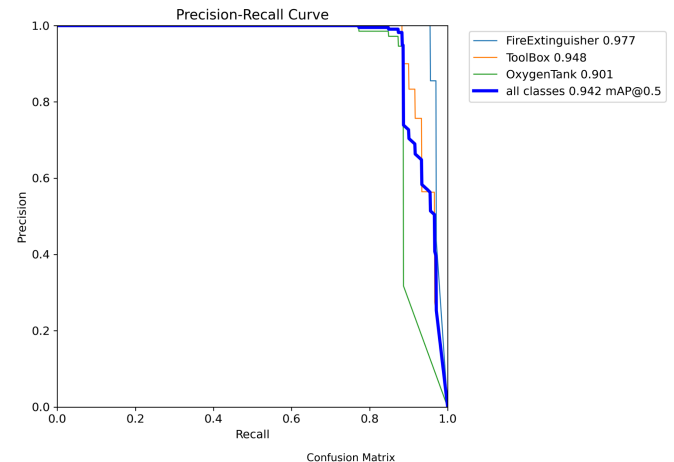
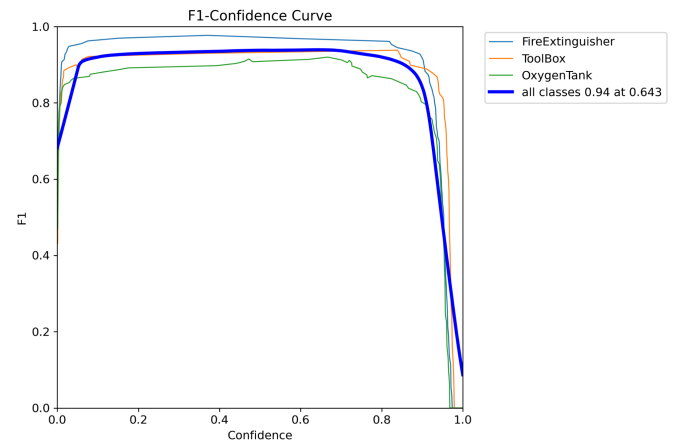
```
metrics = model.val()
```

- **Best model checkpoint:**
`runs/detect/train/weights/best.pt`
- **Validation Results:**

Metric	Score
Precision	0.898
Recall	0.986
mAP@0.5	0.942
mAP@0.5:0.95	0.884

- **Class-wise mAP@0.5:**
 - FireExtinguisher: 0.977
 - ToolBox: 0.948
 - OxygenTank: 0.901

 Shows model's classification performance. →



5. Documented Issues & Fixes

This section outlines the technical challenges encountered during training, evaluation, and deployment — along with the decisions made to overcome them.

⚙️ 1. YOLOv8m Model Selection

We chose **YOLOv8m** (medium) as the base model instead of YOLOv8n or YOLOv8s due to its balance of accuracy and performance.

- YOLOv8m achieved **higher mAP@0.5** (~0.942) with moderate training time on our RTX 4050 GPU.
- It was more suitable for real-time detection needs without requiring excessive compute.

✅ **Fix:** Initial training with YOLOv8n was underperforming, so we switched to YOLOv8m after early testing.

🧠 2. ONNX over TFLite

Initially, we tried exporting the model to **TFLite** format, but:

- Conversion using `onnx2tf` and `onnx-tf` failed due to TensorFlow versioning and broken Keras dependencies.
- `tf2onnx` also had internal issues with fold constants and unsupported layers.

✅ **Fix:** We switched to **ONNX** format for deployment — it's lightweight, cross-platform, and easier to integrate with **Flutter via plugins** like [onnxruntime-mobile](#) or `flutter_onnx`.

📦 3. Large Dataset Handling

The full dataset (~3.9GB) was **too large to include in GitHub**, especially for submission cloning.

✅ **Fix:** We added a **manual download link** in the README with clear folder instructions. This keeps the repo lightweight and reproducible.

6. Future Work & Deployment

While the model performs well on the current dataset, there are multiple opportunities to extend, optimize, and deploy this solution in real-world applications.

A. ONNX Integration with Mobile App

We plan to integrate the **ONNX model** into a **Flutter mobile app** for real-time object detection, leveraging:

- [onnxruntime](#) for cross-platform inference
- On-device GPU acceleration for faster frame analysis
- Camera input stream for live detection on astronauts' tools

This avoids heavy server-side compute and supports offline environments — ideal for space station use cases.

B. Model Optimization

To further improve performance on edge devices:

- We can apply **quantization (INT8 / FP16)** to compress model size and speed up inference
- Use **TensorRT / ONNX Runtime with GPU delegate** for faster hardware-backed execution




C. Expanded Dataset & Generalization

The current model is trained on a **synthetic dataset** with 3 specific tools.

Future improvements:

- Add **more object classes** relevant to astronaut environments (e.g., wires, helmets, panels)
- Incorporate **real-world imagery** or simulated distortions (low light, occlusion)
- Fine-tune with **domain adaptation techniques** for better robustness

D. Potential Use Cases

-  **Astronaut HUD:** Highlight tools for astronauts during repairs.
-  **Robotic Arms:** Enable onboard robots to recognize and pick objects safely
-  **Training Simulators:** Help train crew with object familiarity and spatial awareness





7. Use Case Application

Scenario: Vision for Space Maintenance Robots

In future space missions, autonomous robots will play a key role in inspecting, maintaining, and repairing space stations. These robots need real-time understanding of their environment to identify and interact with essential tools.

Proposed App: ToolFinder AI

An AI-powered system that allows robots or assistive devices to:

-  **Detect tools in real-time** using the trained YOLOv8 model.
-  **Identify objects** like toolboxes, oxygen tanks, and fire extinguishers.
-  **Guide robotic actions** using visual feedback from AI detections.
-  **Run on-device ONNX inference** for low-latency, offline vision.

Stack & Deployment

- **Model Format:** `best.onnx` for lightweight deployment
- **Frontend:** Flutter (planned interface)
- **Inference Engine:** ONNX Runtime (integrated with camera feed)
- **Target Hardware:** Android/iOS devices or edge devices like NVIDIA Jetson

Vision for Expansion

- Equip **maintenance robots** with intelligent vision systems
- Integrate into **Falcon simulations** for space-based robotics
- Extend model to **recognize malfunctions or anomalies**

 This supports ISRO's initiative to explore **robot-assisted operations** and **AI-driven vision systems** for unmanned space tasks.