Pallet Detection & Segmentation for Edge Devices in Manufacturing/Warehousing

Objective

This project aims to develop a real-time pallet detection and segmentation application, designed specifically for deployment in manufacturing or warehousing environments. The solution will be optimized for edge devices like the NVIDIA Jetson AGX Orin, ensuring real-time performance suitable for mobile robotics applications.

Prerequisites:

Anaconda
Pytorch
TensorRT
Ultralytics
OpenCV
ROS2 Foxy

1. Dataset Acquisition and Preparation

Dataset Recommendation

For this project, I used an open-source dataset containing images of pallets in various scenarios. The dataset can be accessed through the provided link: Pallets Dataset.

Data Annotation

The dataset was uploaded to **Roboflow** for annotation. I utilized **Roboflow Annotate**'s Smart Polygon feature, powered by Meta AI's **Segment Anything Model 2 (SAM 2)**. This allowed for zero-shot segmentation labeling, enhancing the labeling process by creating polygon annotations quickly, accurately, and with minimal effort.

Once labeled, the dataset was downloaded in the **YOLOv11** format, which is ideal for training object detection models due to its compatibility with various machine learning frameworks and its efficiency in real-time object detection.

---

2. Data Preparation

Dataset Organization

After the dataset was annotated, I organized it into **three sets**:

- **Training Set**: The main dataset used for model training.
- **Validation Set**: A separate set used to evaluate the model during training.
- **Test Set**: Used to assess the model's performance after training.

3. Model Training

Model Training Configuration

For training the pallet detection and segmentation model, the **Ultralytics YOLO** framework was used. The following script was executed to train the model on the **pallet dataset** with a series of augmentation techniques applied to improve the model’s robustness:

```
from ultralytics import YOLO


if __name__ == "__main__":


    # Load pre-trained model
    model = YOLO("/home/santosh/pallet-detection/trained_model.pt")


    # Train the model with specified parameters
    model.train(
        data="/home/santosh/pallet-detection/pallet_data/data.yaml",
        epochs=100,  # Number of epochs to train
        imgsz=640,  # Image size for training
        device=[0, 1],  # Use GPUs 0 and 1 for training
        batch=8,  # Batch size
        # Augmentation settings
        hsv_h=0.015,  # HSV-Hue augmentation
        hsv_s=0.7,    # HSV-Saturation augmentation
        hsv_v=0.4,    # HSV-Value augmentation
```

```
    degrees=0.2,  # Image rotation range
    translate=0.1,  # Image translation range
    scale=0.0,    # Image scaling range
    shear=0.0,    # Shear range
    perspective=0.0,  # Perspective distortion
    flipud=0.5,   # Flip image up-down probability
    fliplr=0.5,   # Flip image left-right probability
    mosaic=0.3,   # Mosaic augmentation probability
    mixup=0.3,    # Mixup augmentation probability
    copy_paste=0.3,  # Copy-paste segment probability
    bgr=0.7,      # BGR color augmentation
)
```

Key Parameters in the Training Script:

- **Epochs**: The model was trained for **100 epochs** to ensure that it converged well and learned from the full augmented dataset.
- **Batch Size**: A batch size of **8** was chosen to balance training speed and memory usage.
- **Device**: The model training was distributed across **two GPUs** ([0, 1]) to leverage parallel computing for faster training.
- **Image Size**: The images were resized to **640x640** for training, ensuring compatibility with the YOLO model architecture.
- **Augmentation Techniques**: Various augmentation techniques were applied to simulate real-world conditions and improve model robustness. These included:
    - **HSV Augmentation** (Hue, Saturation, Value) to introduce color variations.
    - **Rotation** and **Translation** to simulate changes in object orientation and position.
    - **Flipping** (up-down and left-right) to introduce mirror variations.
    - **Mosaic** and **Mixup** augmentations to simulate the presence of multiple objects and complex scenes.
    - **Copy-Paste** to randomly paste object segments into other parts of the image for variation.

Why These Hyperparameters?

- **Augmentation Techniques**: The combination of these augmentations allows the model to better handle variations in the real-world environment, such as changes in lighting, object position, and orientation.

- **Multiple GPUs**: Using multiple GPUs (devices 0 and 1) allows the training process to scale efficiently, reducing the time needed for model convergence.
- **Batch Size**: The batch size of 8 strikes a balance between GPU memory usage and training speed, ensuring the model can train effectively on the available hardware.

Training Evolution

The model was trained in two stages:

1. **Initial Training**: A model was trained for **50 epochs** without any augmentation techniques to establish a baseline for performance.
2. **Augmented Training**: The model was then retrained for **100 epochs** using the configuration detailed above. The augmentation techniques were intended to refine the model by exposing it to more diverse training examples, ultimately improving its ability to generalize to new data.

Evaluation Overview

The model was evaluated on **52 images** containing **264 instances** of ground and pallet objects. Key metrics for both bounding box detection and segmentation are presented below.

Metrics Summary

- **Box Precision (P)**: Indicates how accurately the model identifies relevant objects without false positives.
- **Box Recall (R)**: Measures the model's ability to detect all relevant instances.
- **mAP50**: Mean Average Precision at IoU threshold of 0.5, representing detection and segmentation performance.
- **mAP50-95**: Mean Average Precision over multiple IoU thresholds (0.5 to 0.95) for a more comprehensive performance measure.

Results

Metric	Box Precision	Box Recall	Box mAP50	Box mAP50-95	Mask Precision	Mask Recall	Mask mAP50	Mask mAP50-95
Overall	0.67	0.709	0.725	0.572	0.662	0.701	0.709	0.548
Ground	0.809	0.902	0.929	0.829	0.794	0.885	0.917	0.832
Pallets	0.53	0.517	0.52	0.315	0.53	0.517	0.502	0.264

Inference Performance

- **Preprocessing Time**: 1.9 ms per image
- **Inference Time**: 22.5 ms per image
- **Postprocessing Time**: 3.7 ms per image

Detailed Analysis

Overall Performance

The model achieves a mean Average Precision (mAP) of:

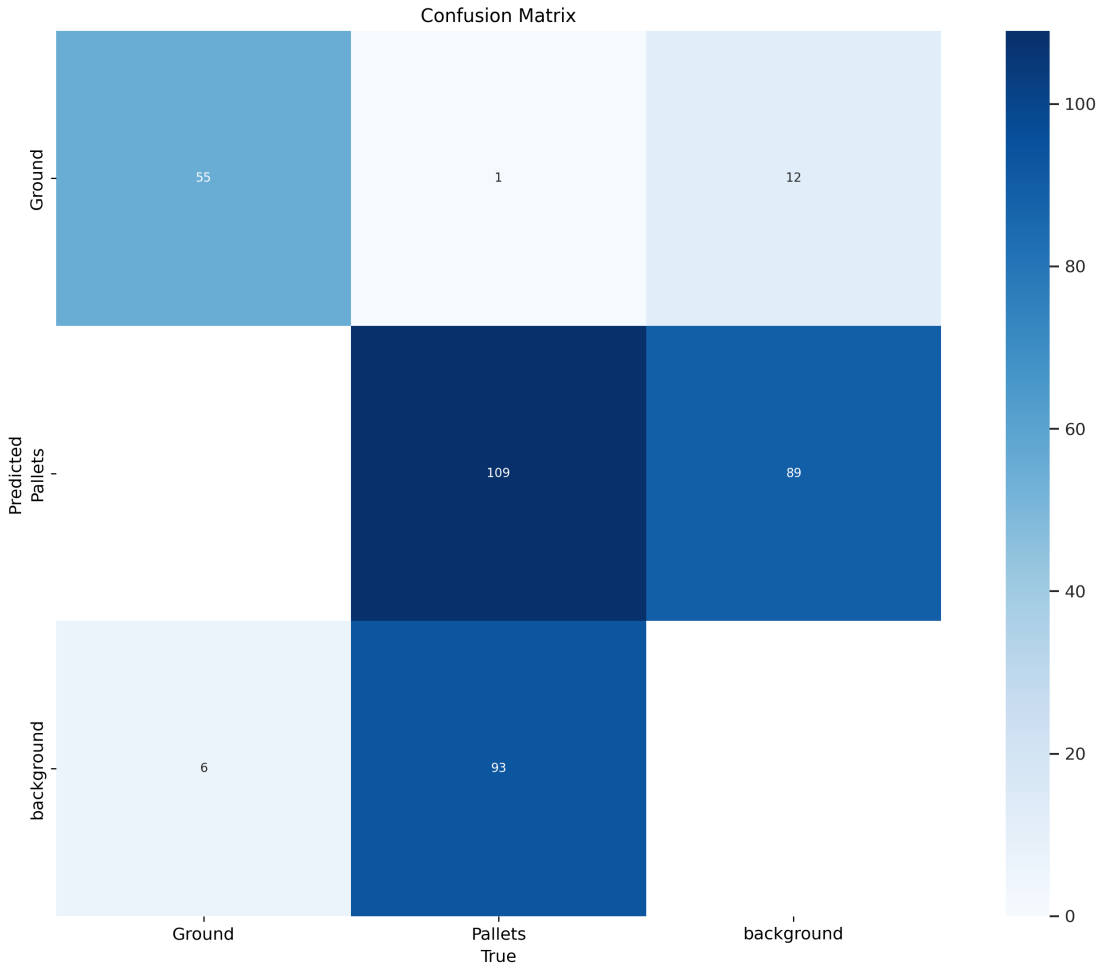
- **Box mAP50**: 0.725
- **Box mAP50-95**: 0.572
- **Mask mAP50**: 0.709
- **Mask mAP50-95**: 0.548

These scores reflect the model's overall effectiveness at both bounding box detection and segmentation. The segmentation model demonstrated strong performance, with slightly better results for the ground class compared to the pallet class, suggesting that the model more accurately distinguishes ground than pallets in complex scenes.

Class-Specific Performance

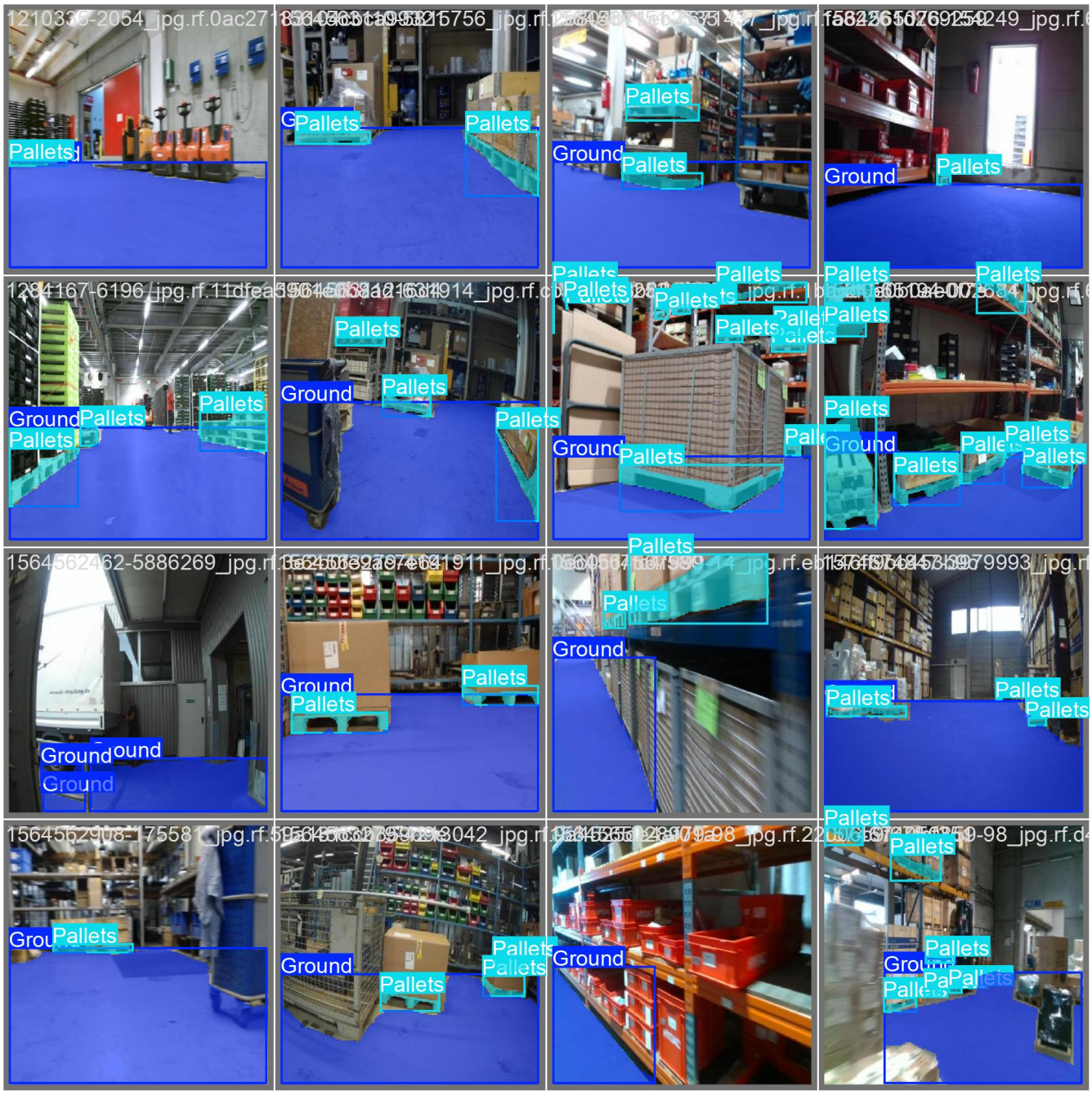
1. **Ground**
    - The ground class performed the best across all metrics. It achieved a high **Box mAP50** of 0.929 and **Mask mAP50** of 0.917, indicating excellent precision and recall in detecting ground areas.
    - The model's **Mask mAP50-95** for the ground was also high at 0.832, demonstrating that the model can reliably segment the ground at various IoU thresholds.
2. **Pallets**
    - The pallet class had lower precision and recall, with a **Box mAP50** of 0.52 and a **Mask mAP50** of 0.502.
    - The **Mask mAP50-95** for pallets was 0.264, indicating that pallet segmentation may be less consistent, particularly at higher IoU thresholds.
    - These results suggest that while the model can generally detect pallets, additional training data or fine-tuning may be needed to improve accuracy, especially under varying lighting or occlusion.

I decided to stick with just the dataset provided and didn't add any other datasets from the internet. I think that using a wider variety of data would help the model improve, especially for recognizing pallets in different situations. Also, adding some synthetic data could make the model a bit stronger by teaching it to handle scenarios that are hard to capture in real life. With these additions, I believe the model could perform even better, especially for real-time use in busy environments.

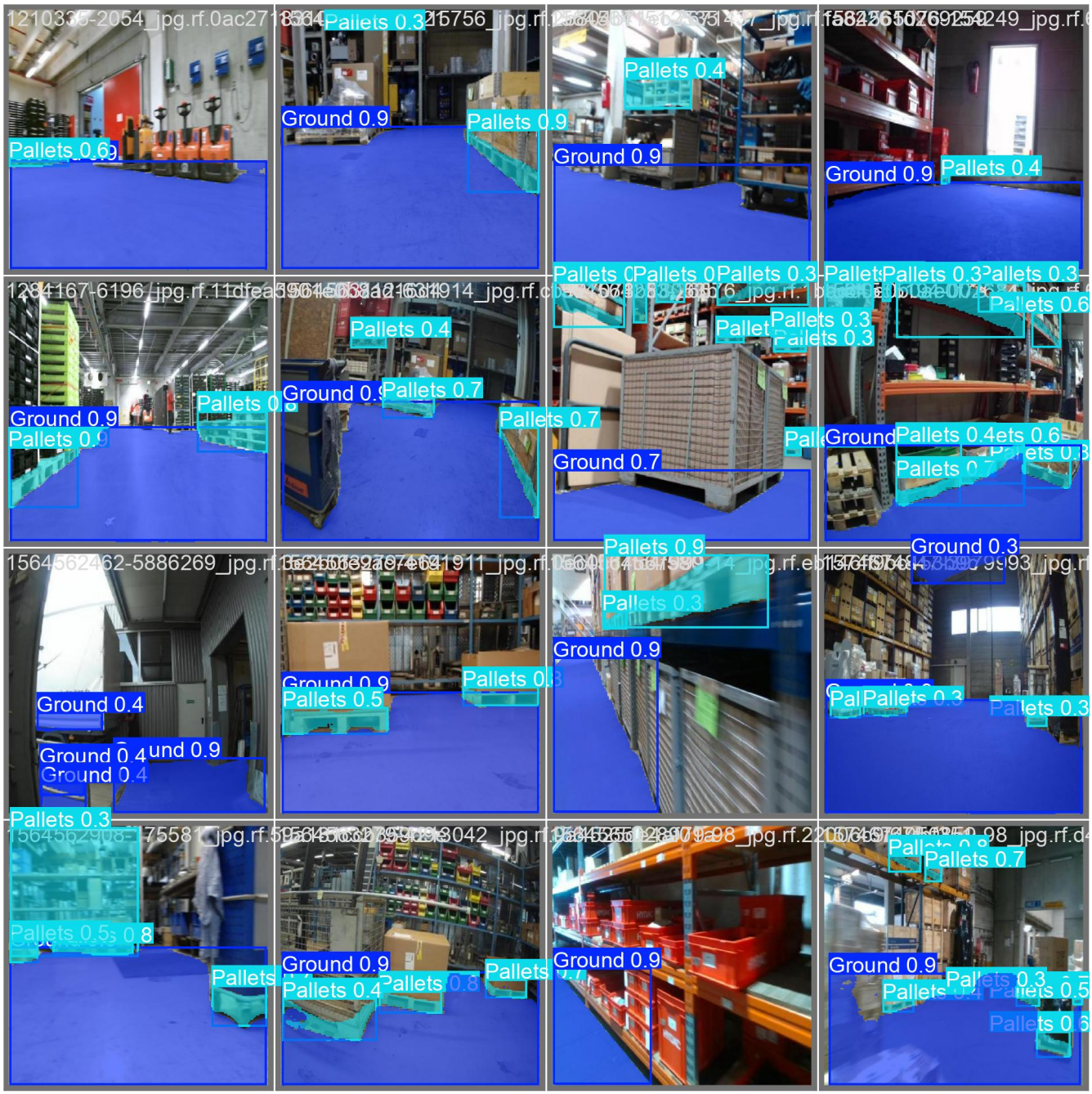


Some of the pallets are detected as background because when I was labeling the pallets, I ignored some of the far off pallets in the image which were not clear. But still the model predicted pallets as background, this can be improved by labeling with a lot more accuracy, instead of using Smart AI labeling option.
And also model predicted true pallets as background, proving we need more dataset and fine-tuning of the model to improve this detection.

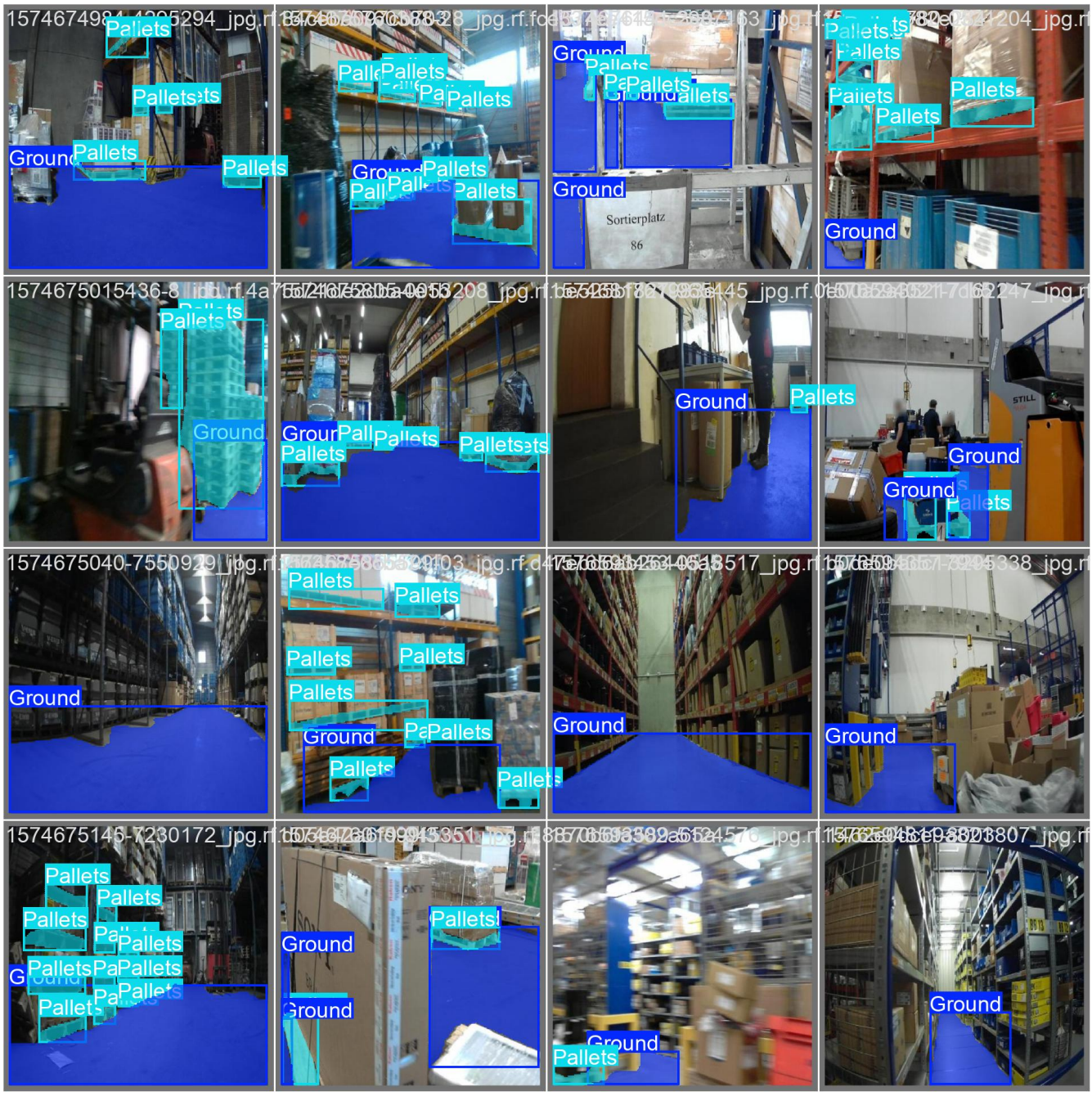
Validation Labels batch0:



Validation Predictions batch0:



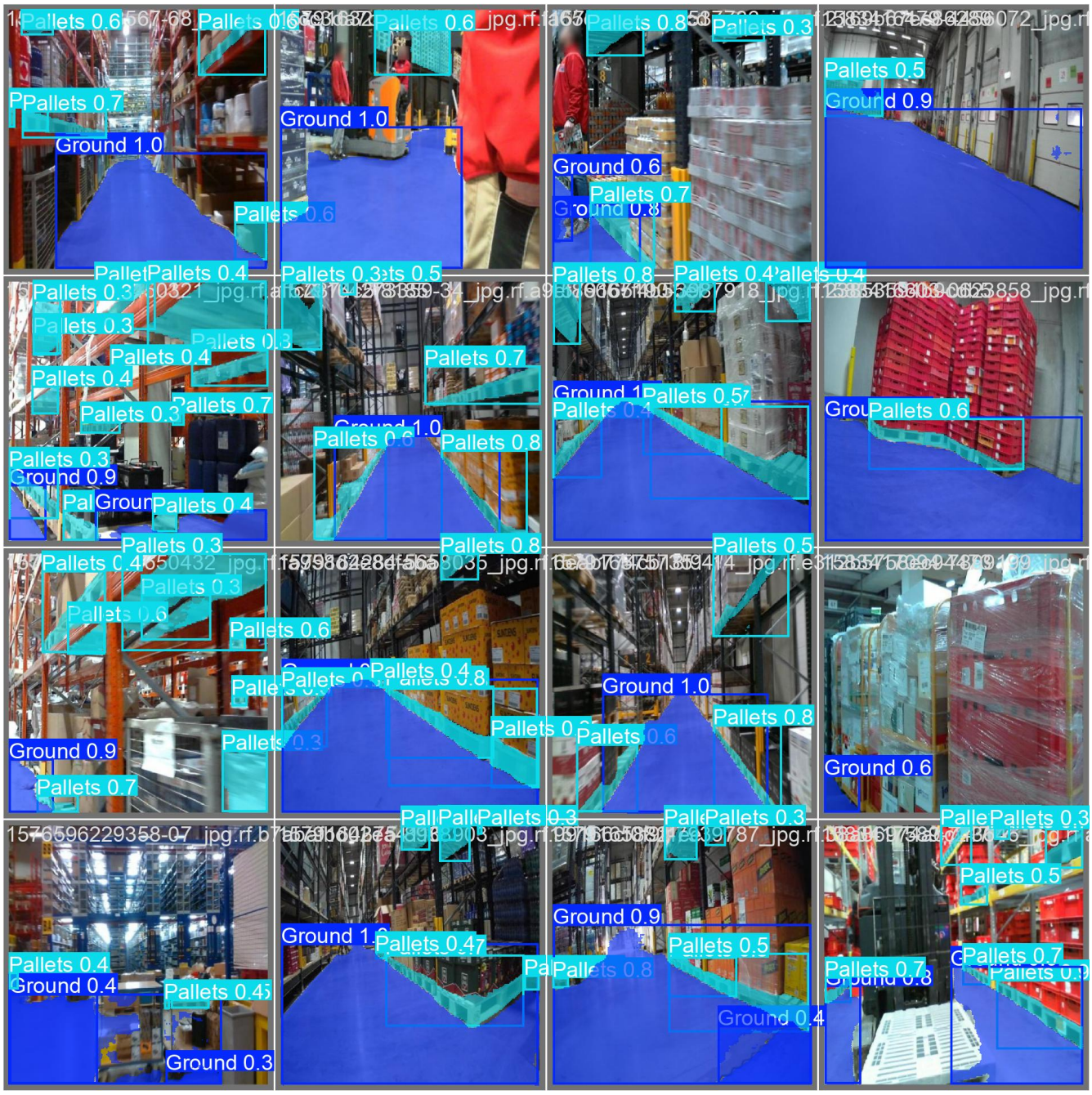
Validation Labels batch1:



Validation Predictions batch1:

Validation Labels batch2:

Validation Predictions batch2:



Exporting the model to TensorRT:

export YOLOv11 model using TensorRT with INT8 quantization. This process involves post-training quantization (PTQ) and calibration:

```
from ultralytics import YOLO

# Load the YOLO model
model = YOLO("/content/best.pt")

# Export to TensorRT format with desired options
model.export(format="engine",int8=True, data="/content/drive/MyDrive/Pallets
2.v1i.yolov11/data.yaml")
```

Developing a ROS2 node:
Prerequisites:

Anaconda
Pytorch
TensorRT
Ultralytics
OpenCV
ROS2 Foxy

Create the conda environment using this file : environment.yaml
conda env create -f environment.yaml -n enter_your_environment_name

conda activate enter_your_environment_name

Structure of yolobot workspace:
Yolobot

- build
- install
- log
- src
  - yolobot_recognition
    - launch
    - scritps
      - **yolov11_ros2_pt.py**
  - yolov8_msgs

Building the workspace:

- cd ~/yolobot

- colcon build

If “colcon build” is giving the error try these commands:

- PYTHON_EXECUTABLE=$(which python)
- colcon build --allow-overriding yolov8_msgs --cmake-args -DPYTHON_EXECUTABLE=$PYTHON_EXECUTABLE

Once you build your workspace successfully

- ros2 launch yolobot_recognition launch_yolov8.launch.py

Once you run this command, it will start publishing **inference_result** topic, you can visualize this in **rviz2 by clicking on “Add” button -- > “By topic” – > dropdown inference_result and click on Image.**

Now you are visualizing the detection and segmentation results real-time !!!

Important:-
yolov11_ros2_pt.py

```
#!/usr/bin/env python3

from ultralytics import YOLO
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

from yolov8_msgs.msg import InferenceResult
from yolov8_msgs.msg import Yolov8Inference

bridge = CvBridge()

class Camera_subscriber(Node):

    def __init__(self):
        super().__init__('camera_subscriber')

        self.model = YOLO('/home/vikram/ros2_ws/src/yolobot_recognition/scripts/best.pt')
            – > Change this model path to your own...

        self.yolov8_inference = Yolov8Inference()

        self.subscription = self.create_subscription(
```

```
            Image,
            'camera/image_raw',          – > Change this to your Camera image topic like
                                                '/zed2i/zed_node/left/image_rect_color'
            self.camera_callback,
            10)
        self.subscription

        self.yolov8_pub = self.create_publisher(Yolov8Inference, "/Yolov8_Inference", 1)
        self.img_pub = self.create_publisher(Image, "/inference_result", 1)

    def camera_callback(self, data):

        img = bridge.imgmsg_to_cv2(data, "bgr8")
        results = self.model(img)

        self.yolov8_inference.header.frame_id = "inference"
        self.yolov8_inference.header.stamp = camera_subscriber.get_clock().now().to_msg()

        for r in results:
            boxes = r.boxes
            for box in boxes:
                self.inference_result = InferenceResult()
                b = box.xyxy[0].to('cpu').detach().numpy().copy()  # get box coordinates in (top, left, bottom,
right) format
                c = box.cls
                self.inference_result.class_name = self.model.names[int(c)]
                self.inference_result.top = int(b[0])
                self.inference_result.left = int(b[1])
                self.inference_result.bottom = int(b[2])
                self.inference_result.right = int(b[3])
                self.yolov8_inference.yolov8_inference.append(self.inference_result)

            #camera_subscriber.get_logger().info(f"{self.yolov8_inference}")

        annotated_frame = results[0].plot()
        img_msg = bridge.cv2_to_imgmsg(annotated_frame)

        self.img_pub.publish(img_msg)
        self.yolov8_pub.publish(self.yolov8_inference)
        self.yolov8_inference.yolov8_inference.clear()

if __name__ == '__main__':
    rclpy.init(args=None)
    camera_subscriber = Camera_subscriber()
    rclpy.spin(camera_subscriber)
    rclpy.shutdown()
```

Name: Vikram Shahapur
Email: vikramvs1306@gmail.com