Object Detection Model Training & Evaluation

Author: Erick Lemmy dos Santos Oliveira

GitHub: Slots Detection

Technical Report: Full execution workflow and experimental results from Google Colaboratory, including training/evaluation metrics and model performance visualizations [PDF]

Introduction

This project implements an object detection system to detect Slots for planting flowers. Despite the limited dataset size, various techniques where tested to create a detection model capable of performing effectively.

Project Objectives

1. Train an object detection model with the data (Yolo or any other model).
2. Evaluate the model performance (select and justify evaluation metrics).
3. Discuss the model performance and errors.
4. Discuss the steps used to solve the problem.
5. Suggest improvements in data/model/etc.

Dataset Characteristics

- Classes: 1 (Slots)
- Total images: 21
- split:
    - train: 16 images (75%)
    - val: 5 images (25%)

Technical Environment

- Hardware: Google Colab GPU (NVIDIA T4/V100)
- Framework: Ultralytics YOLO
- Acceleration: CUDA-enabled training

```
%pip install ultralytics sahi albumentations
```

Mostrar saída oculta

Run the necessary imports bellow

```
import os
import cv2
import glob
import torch

import numpy as np
import pandas as pd
from tqdm import tqdm
from pathlib import Path

import ultralytics
ultralytics.checks()

from ultralytics import YOLO
import matplotlib.pyplot as plt
from collections import defaultdict

%matplotlib inline
```

```
Ultralytics 8.3.94 🚀 Python-3.11.11 torch-2.6.0+cu124 CUDA:0 (Tesla T4, 15095MiB)
Setup complete ✅ (2 CPUs, 12.7 GB RAM, 42.7/112.6 GB disk)
```

Verify NVIDIA GPU Availability

Checking for the avaliability of the GPU. If shows its off, go to "Runtime" -> "Change Runtime type" in the top menu bar, and then selecting one of the GPU options in the Hardware accelerator section.

```
!nvidia-smi

print(f"CUDA available: {torch.cuda.is_available()}")
device = 0 if torch.cuda.is_available() else 'cpu'
```

Mostrar saída oculta

1. Dataset Verification

Start Uploading the **dataset.zip** file to the Google Colab instance by clicking the "Files" icon on the left side of the browser, and then the "Upload to session storage" icon. Select the zip folder to upload it or upload from your personal Google Drive, mount the drive on this Colab session, and copy them over to the Colab filesystem. For my dataset is **erick.zip** you can change to use another, but it will require to update every instance of **erick** in the code.

```
# from google.colab import drive
drive.mount('/content/drive')

!unzip "/content/erick.zip" -d "/content/"
```

```
Archive:  /content/erick.zip
replace /content/erick/train/labels/7126_2_7282_67-12_2ND.txt? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

Pre-Training Dataset Evaluation

Before training, it is essential to evaluate the dataset by verifying that both images and labels are correct. This process also helps in identifying a suitable model based on the dataset format, reducing the need for significant modifications if it already aligns with a specific model.

Reasons:

1. **Prevent Training Failures**
   - YOLO coordinates outside the [0,1] range can cause numerical errors (e.g., NaN loss).
   - Incorrect folder structure can prevent data from loading properly.
2. **Ensure Annotation Quality**
   - Poor annotations can reduce mAP scores.
   - Unlabeled objects introduce bias, leading to inaccurate learning.
3. **YOLO Compatibility**
   - Requires a specific directory format:

```
dataset/
├── train/images/  # .jpg, .png
└── train/labels/  # .txt (one file per image)
```

   - Class IDs must match the YAML configuration file.

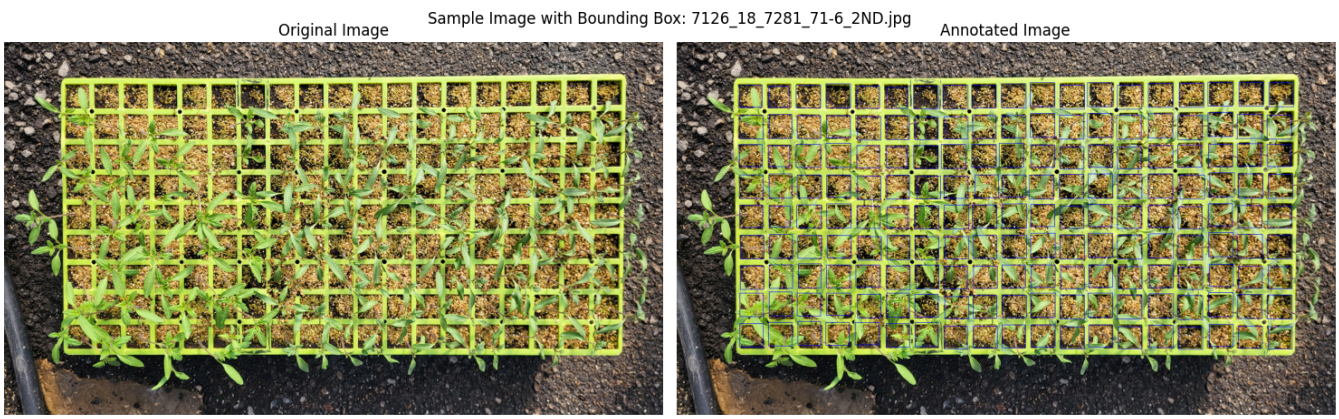
The provided dataset follows the standard structure, with separate training and validation sets, as well as distinct folders for images and labels. Since it includes a YAML file and annotations in `.txt` format structured as:

- `(class, center_x, center_y, height, width)`
- Values normalized between **0 and 1**

This suggests that the dataset was created using an auxiliary labeling tool, such as Roboflow, and later converted to YOLO format. By visualizing an image alongside its corresponding annotations, the format was confirmed.**negrito**

```
def plot_label_check(img, title):
    sample_img = cv2.imread(img_path)
    annotated_img = sample_img.copy()

    h, w = sample_img.shape[:2]
    label_path = img_path.replace("images", "labels").replace(".jpg", ".txt")
    with open(label_path, 'r') as f:
        for line in f:
            class_id, x_center, y_center, width, height = map(float, line.split())

            x_center, y_center = int(x_center * w), int(y_center * h)
            width, height = int(width * w), int(height * h)
```

```
            x1, y1 = x_center - width // 2, y_center - height // 2
            x2, y2 = x1 + width, y1 + height

            cv2.rectangle(annotated_img, (x1, y1), (x2, y2), (255, 0, 0), 2)

    fig, ax = plt.subplots(1, 2, figsize=(15, 5))

    ax[0].imshow(cv2.cvtColor(sample_img, cv2.COLOR_BGR2RGB))
    ax[0].set_title("Original Image")
    ax[0].axis("off")

    ax[1].imshow(cv2.cvtColor(annotated_img, cv2.COLOR_BGR2RGB))
    ax[1].set_title("Annotated Image")
    ax[1].axis("off")

    plt.tight_layout()
    plt.suptitle(f"{title}: {os.path.basename(img_path)}")
    plt.show()

img_path = "erick/val/images/7126_18_7281_71-6_2ND.jpg"
plot_label_check(img_path, "Sample Image with Bounding Box")
```



2. Image Resolution and Color Analysis

Visualizing Image Resolutions

Analyzing the distribution of image resolutions in the **training** and **validation** sets helps identify potential imbalances that could affect model training and generalization.

Methodology:

1. **Extract Resolutions:** Collect width and height from all images.
2. **Count Occurrences:** Determine the number of unique resolutions and the number of images per resolution.

Observations:

1. **Training Set Imbalance:**
    - 80% of the training images are concentrated in just **two resolutions** (`4000x2252` and `4032x3024`).
    - Resolutions like `1600x900` and `2000x1126` contain only **one image each**, which could lead to *overfitting* for these specific formats.
2. **Limited Validation Coverage:**
    - The validation set covers only **50%** of the resolutions found in the training set.
    - The resolution `4032x3024` has only **one image** in validation, making evaluation unreliable for this format.

Implications:

- **Resolution Bias:** The model may perform poorly on underrepresented resolutions (e.g., `1600x900`).
- **Preprocessing Needs:** Different resolutions require strategies like *resizing* or *padding* for consistency.
- **Generalization Issues:** The validation set does not sufficiently cover the training resolutions, increasing performance drops in real-world deployment.

```
def get_resolution_and_color_means(image_dir):
    resolutions = []
    color_means = []

    for img_file in os.listdir(image_dir):
        img_path = os.path.join(image_dir, img_file)
        img = cv2.imread(img_path)

        if img is not None:
            # Get resolution
            h, w = img.shape[:2]
            resolutions.append((w, h))

            # Calculate color means (BGR channels)
            color_means.append(img.mean(axis=(0, 1)))
    return resolutions, color_means

train_image_dir = "erick/train/images"
train_label_dir = "erick/train/labels"

val_image_dir = "erick/val/images"
val_label_dir = "erick/val/images"

resolutions_train, color_means_train = get_resolution_and_color_means(train_image_dir)
resolutions_val, color_means_val = get_resolution_and_color_means(val_image_dir)

print(f"========= Resolutions =========")
print(f"Number of unique resolutions (train): {len(set(resolutions_train))}")
print(f"Images per resolution (train): {dict((res, resolutions_train.count(res)) for res in set(resolutions_train))}")
print(f"Number of unique resolutions (val): {len(set(resolutions_val))}")
print(f"Images per resolution (val): {dict((res, resolutions_val.count(res)) for res in set(resolutions_val))}")
```

```
========= Resolutions =========
Number of unique resolutions (train): 4
Images per resolution (train): {(1600, 900): 1, (4032, 3024): 6, (4000, 2252): 8, (2000, 1126): 1}
Number of unique resolutions (val): 2
Images per resolution (val): {(4032, 3024): 1, (4000, 2252): 4}
```

Analyze Color Channels

Comparing the statistical distributions (mean and standard deviation) of the color channel intensities (BGR) between the training and validation sets to ensure consistency and identifying potential biases that may affect the model.

Methodology:

1. **Calculation of Statistics:**
    - Mean and standard deviation of the average intensities for each channel (Blue, Green, Red) across all images.
2. **Visualization:**
    - Histograms to compare intensity distributions.
    - Boxplots to analyze variation and outliers.

Observations:

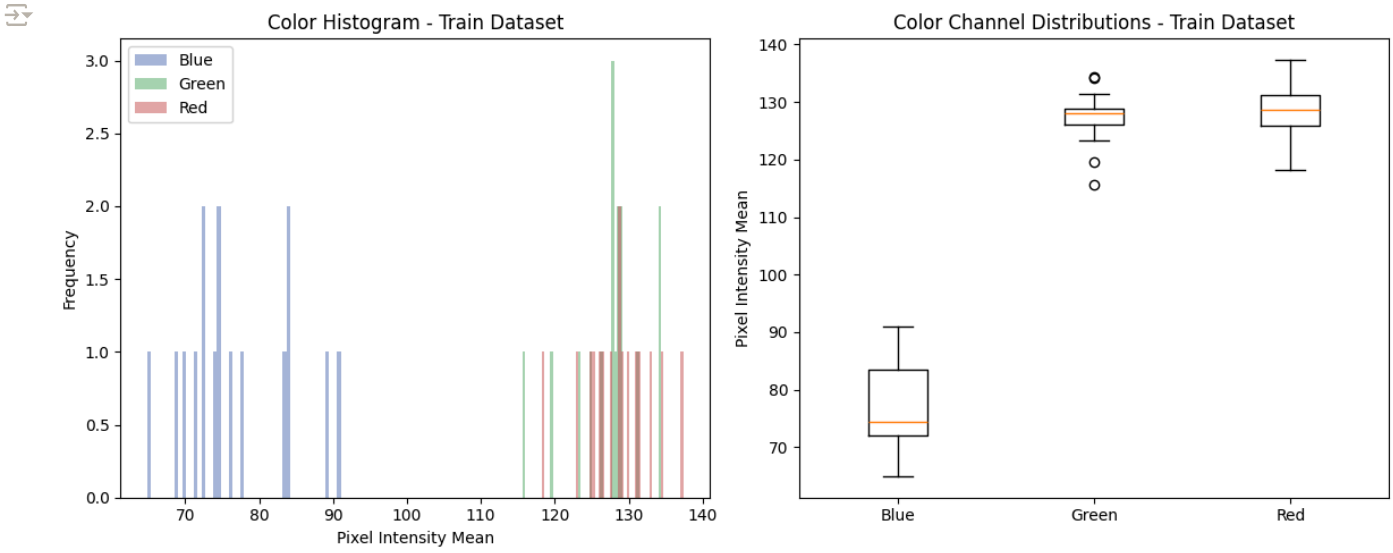
The original dataset does not exhibit many outliers and appears consistent with the expected scenario. The high Green and Red values are due to the presence of plants and ground textures, while Blue is typically less prominent in natural environments. As a result, there is no significant indication of bias. However, applying a Hue augmentation could help simulate greater color variability and further enhance the model's robustness.

```
color_means_train = np.array(color_means_train)
color_means_val = np.array(color_means_val)

print(f"========= Color Channel Statistics =========")
print(f"Color means -> B: {color_means_train[:, 0].mean()}, G: {color_means_train[:, 1].mean()}, R: {color_means_train[:, 2].mean()}")
print(f"Color std -> B: {color_means_train[:, 0].std()}, G: {color_means_train[:, 1].std()}, R: {color_means_train[:, 2].std()}")
```

```
========= Color Channel Statistics =========
Color means -> B: 76.68592931296956, G: 127.29173333616606, R: 128.47953913963295
Color std -> B: 7.287260903403717, G: 4.627485645500718, R: 4.474558334424617
```

```
def plot_color_analysis(color_means, title):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

    # Plot histogram
    ax1.hist(color_means[:, 0], bins=50, alpha=0.5, color='b', label='Blue')
    ax1.hist(color_means[:, 1], bins=50, alpha=0.5, color='g', label='Green')
```

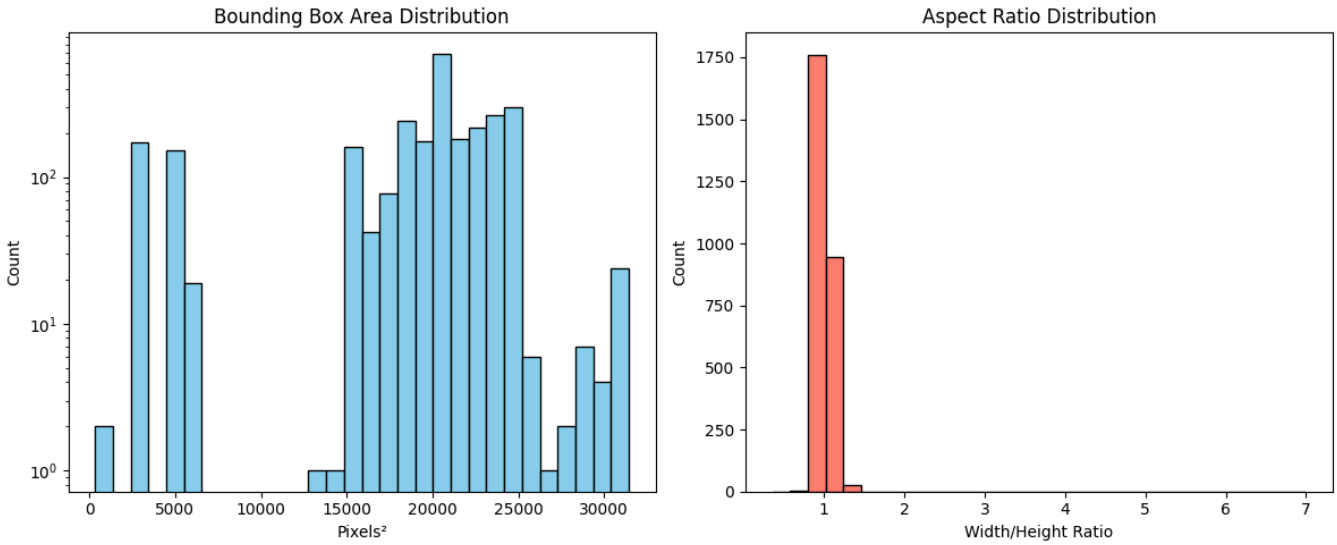
```
    ax1.hist(color_means[:, 2], bins=50, alpha=0.5, color='r', label='Red')
    ax1.set_title(f'Color Histogram - {title}')
    ax1.set_xlabel('Pixel Intensity Mean')
    ax1.set_ylabel('Frequency')
    ax1.legend()

    # Plot boxplot
    ax2.boxplot([color_means[:, 0], color_means[:, 1], color_means[:, 2]])
    ax2.set_title(f'Color Channel Distributions - {title}')
    ax2.set_xticks([1, 2, 3])
    ax2.set_xticklabels(['Blue', 'Green', 'Red'])
    ax2.set_ylabel('Pixel Intensity Mean')

    plt.tight_layout()
    plt.show()

plot_color_analysis(color_means_train, 'Train Dataset')
# plot_color_analysis(color_means_val, 'Validation Dataset')
```




```
def analyze_dataset_statistics(image_dir, label_dir):
    """Analyze dataset statistics including bounding box metrics and class distribution.

    Returns:
        dict: Dictionary containing computed statistics
    """
    stats = {
        'bbox_areas': [],
        'bbox_aspect_ratios': [],
        'class_counts': defaultdict(int),
        'num_boxes_per_image': [],
        'missing_images': 0,
        'corrupted_files': 0
    }

    label_files = [f for f in os.listdir(label_dir) if f.endswith(".txt")]

    print(f"Analyzing {len(label_files)} label files...")

    for label_file in tqdm(label_files, desc="Processing images"):
        label_path = os.path.join(label_dir, label_file)
        image_file = os.path.splitext(label_file)[0] + ".jpg"
        image_path = os.path.join(image_dir, image_file)

        # Validate image existence
        if not os.path.exists(image_path):
            stats['missing_images'] += 1
            continue

        img = cv2.imread(image_path)
        if img is None:
            stats['corrupted_files'] += 1
            continue

        h, w = img.shape[:2]
        num_boxes = 0
```

```
        # Process labels
        with open(label_path, 'r') as f:
            for line in f:
                class_id, x_center, y_center, width, height = map(float, line.split())
                stats['class_counts'][int(class_id)] += 1

                # Convert to absolute coordinates
                width_abs = width * w
                height_abs = height * h

                # Calculate metrics
                area = width_abs * height_abs
                aspect_ratio = width_abs / height_abs if height_abs > 0 else 0

                stats['bbox_areas'].append(area)
                stats['bbox_aspect_ratios'].append(aspect_ratio)
                num_boxes += 1

        stats['num_boxes_per_image'].append(num_boxes)

    return stats

def print_statistics(stats):
    """Print formatted statistics"""
    total_images = len(stats['num_boxes_per_image'])
    total_boxes = sum(stats['class_counts'].values())

    print("\nDataset Statistics Report")
    print("==========================")
    print(f"Total images analyzed: {total_images:,}")
    print(f"Total bounding boxes: {total_boxes:,}")
    print(f"Missing images: {stats['missing_images']}")
    print(f"files: {stats['corrupted_files']}\n")

    print("Bounding Box Metrics")
    print(f"- Average boxes per image: {np.mean(stats['num_boxes_per_image']):.2f}")
    print(f"- Area (pixels²) - Mean: {np.mean(stats['bbox_areas']):,.2f}, Median: {np.median(stats['bbox_areas']):,.2f}")
    print(f"- Aspect Ratio - Mean: {np.mean(stats['bbox_aspect_ratios']):.2f}, Median: {np.median(stats['bbox_aspect_ratios']):.2f}\n")

    print("Class Distribution")
    for class_id, count in sorted(stats['class_counts'].items()):
        print(f"  Class {class_id}: {count:,} boxes ({count/total_boxes:.1%})")

def plot_distributions(stats):
    """Visualize dataset distributions"""
    plt.figure(figsize=(12, 5))

    # Bbox Area
    plt.subplot(1, 2, 1)
    plt.hist(stats['bbox_areas'], bins=30, color='skyblue', edgecolor='black')
    plt.title('Bounding Box Area Distribution')
    plt.xlabel('Pixels²')
    plt.ylabel('Count')
    plt.yscale('log')

    # Aspect Ratio
    plt.subplot(1, 2, 2)
    plt.hist(stats['bbox_aspect_ratios'], bins=30, color='salmon', edgecolor='black')
    plt.title('Aspect Ratio Distribution')
    plt.xlabel('Width/Height Ratio')
    plt.ylabel('Count')


    plt.tight_layout()
    plt.show()

train_stats = analyze_dataset_statistics(train_image_dir, train_label_dir)
print_statistics(train_stats)
plot_distributions(train_stats)
```

```
Analyzing 16 label files...
Processing images: 100%|██████████| 16/16 [00:03<00:00,  4.20it/s]

Dataset Statistics Report
==========================
Total images analyzed: 16
Total bounding boxes: 2,739
Missing images: 0
files: 0

Bounding Box Metrics
- Average boxes per image: 171.19
- Area (pixels²) - Mean: 18,936.92, Median: 20,804.56
- Aspect Ratio - Mean: 1.00, Median: 1.00

Class Distribution
  Class 0: 2,739 boxes (100.0%)
```



3. Train Model

Why YOLO?

The YOLO architecture was selected for its:

- Real-time inference capabilities
- Balance between accuracy and speed
- Strong performance on small datasets
- Extensive pre-trained weights availability
- Dataset Compatibility
- Agricultural Applications [1, 2, 3, 4, 5]
- SAHI compatibility [1, 2]
- Augmentation capabilities

Training Parameters

With the model defined, the training starts! First, there are a few important parameters to decide on.

Model architecture & size (`model`):

Several YOLO versions exists, like yolov5, yolov8 and yolov11 with different models sizes (`yolov8n.pt`, `yolo8m.pt`, `yolo11n.pt`, `yolo11m.pt`, `yolo11x.pt`). Larger models run slower but have higher accuracy, while smaller models run faster but have lower accuracy. For this project the `yolov8n.pt` was chosen since it has less params than older models and provides balanced trade-off between speed and accuracy compared to others models like v9 (more computational heavy) and YOLO11 with is optimized for speed and efficiency, is not the best choice for this task since its focus is on detection smaller objects in higher resolution imagens. The decision to use the `n` variant was made to minimize/prevent overfitting during the transfer learning while ensuring that the model remains sufficiently capable of handling the complexity of small object detection.

Model architecture & size (`model`):

Several YOLO versions exist, such as YOLOv5, YOLOv8, and YOLOv11, each offering models of varying sizes (`yolov8n.pt` , `yolov8m.pt` , `yolo11n.pt` , `yolo11m.pt` , `yolo11x.pt`). Larger models tend to run slower but offer higher accuracy, while smaller models run faster but sacrifice some accuracy. For this project, the `yolov8n.pt` model was selected. Although it has fewer parameters than the newer models (except YOLOv11), YOLOv8 provides a balanced trade-off between speed and accuracy. YOLOv11, while optimized for speed and efficiency, is not the best choice for this task since its focus is on maximizing performance in real-time scenarios, which is not the primary requirement here. YOLOv8, on the other hand, is well-suited for achieving good detection accuracy with reasonable processing time, making it ideal for the dataset and the computational constraints of this project.

Number of epochs (`epochs`) and (`batchs`)

With a limited dataset, an initial setting of 50 epochs is used to provide sufficient training while mitigating the risk of overfitting. YOLO automatically adjusts the batch size based on GPU memory, typically utilizing around 60% of the GPU's RAM. For a resolution of 640x640, this usually results in a batch size between 8 and 16. This configuration strikes a balance between efficient training and resource utilization.

Resolution (`imgsz`)

Image Resolution has a large impact on the speed and accuracy of the mode. Lower resolution model will have higher speed but less accuracy. YOLO models are typically trained and inferenced at a 640x640 resolution.

Data Augmentation

Given the limited size of our dataset, we apply data augmentation to improve the model's generalization and robustness. YOLO provides built-in augmentation tools that eliminate the need to create entirely new datasets. The following augmentation techniques are in use:

- Mosaic (1.0) – Set to 1.0 to ensure that all images will have that augmentation. This augmentation combines multiple images into one, helping the model learn object variations and different backgrounds.
- MixUp (0.5) – Set to 0.5 to ensure that this augmentation only happens 50% of the time. This augmentation blends two images and their labels, improving the model's ability to handle occlusions and uncertainties reducing overfitting.
- Scale (0.5) – Randomly resizes objects to enhance scale invariance. (50% probability to scale)
- Flip (fliplr: 0.5, flipud: 0.2) – Applies horizontal and vertical flips to introduce viewpoint variations, horizontal is particularly effective for overhead/aerial images like those in our datase. (Aplied to 50% and 20% of images)
- HSV Augmentation (hsv_h: 0.015, hsv_s: 0.7, hsv_v: 0.4) – Slightly hye Adjust to prevent extreme color shift, high saturation, and medium brightness to improve color invariance.

Transfer Learning

Using a pre-trained model leverages knowledge acquired from large datasets, facilitating the extraction of relevant features even if the target class isn't present in the original model. Freezing some of the initial layers preserves fundamental features, while the upper layers are fine-tuned to learn the new classes and adapt to our specific domain.

Training

Run the following code block to begin training.

```
!rm -rf slots/*
import yaml

# Change to Absolute Path /content/
data_yaml_path = "/content/erick/data.yaml"

with open(data_yaml_path, 'r') as f:
    data = yaml.safe_load(f)

# Update val and train paths to absolute inside data.yaml
data['train'] = '/content/erick/train/images'
data['val'] = '/content/erick/val/images'

with open(data_yaml_path, 'w') as f:
    yaml.dump(data, f, default_flow_style=False)

config = {
    "data": data_yaml_path,
    "epochs": 50,
    "imgsz": 640,
    "batch": 16,
    "workers": 8,      # Number of CPU workers (default: 8)
    "device": device,
    "freeze": list(range(11)), # 0 ... 10
    "mosaic": 1.0,
    "mixup": 0.5,
    "scale": 0.5,
    "fliplr": 0.5,
    "flipud": 0.2,
    "hsv_h": 0.015,
    "hsv_s": 0.9,
```

```
    "hsv_v": 0.9,
    "patience": 30,     # for early stop if theres no improvement
    "seed": 42,         # for reproducibility
    # "single_cls": True, # improve, but changes the class name to 'item'
    "project": "slots",
    "name": "standard",
}

model = YOLO("yolov8n.pt")
results = model.train(**config)
```

```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      42/50      6.48G      1.621      1.244      1.002       2564        640: 100%|██████████| 1/1 [00:00<00:00,  2.53it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  2.15

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      43/50      6.48G      1.587      1.189     0.9595       2619        640: 100%|██████████| 1/1 [00:00<00:00,  2.08it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  1.98

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      44/50      6.48G      1.575      1.179     0.9541       2611        640: 100%|██████████| 1/1 [00:00<00:00,  2.69it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  2.93

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      45/50      6.48G      1.631      1.195     0.9789       2580        640: 100%|██████████| 1/1 [00:00<00:00,  2.91it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  4.11

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      46/50      6.48G      1.615      1.153     0.9334       2693        640: 100%|██████████| 1/1 [00:00<00:00,  4.23it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  3.67

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      47/50      6.49G      1.603      1.194     0.9607       2569        640: 100%|██████████| 1/1 [00:00<00:00,  1.76it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  2.46

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      48/50      6.49G      1.574      1.094     0.9396       2662        640: 100%|██████████| 1/1 [00:00<00:00,  4.69it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  4.22

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      49/50      6.51G      1.577      1.135     0.9612       2597        640: 100%|██████████| 1/1 [00:00<00:00,  2.72it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  1.35

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      50/50      6.52G       1.67      1.107     0.9304       2676        640: 100%|██████████| 1/1 [00:00<00:00,  4.05it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  3.05


50 epochs completed in 0.032 hours.
Optimizer stripped from slots/standard/weights/last.pt, 6.2MB
Optimizer stripped from slots/standard/weights/best.pt, 6.2MB

Validating slots/standard/weights/best.pt...
Ultralytics 8.3.94 🚀 Python-3.11.11 torch-2.6.0+cu124 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,005,843 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:00<00:00,  2.29
                   all          5        855      0.982      0.944      0.991      0.582
Speed: 0.2ms preprocess, 4.1ms inference, 0.0ms loss, 5.3ms postprocess per image
Results saved to slots/standard
```

Model Evaluation

To evaluate the model's performance, the metrics **Precision**, **Recall**, **mAP** (Mean Average Precision), and **mAP@50-95** were selected. These metrics are widely used in object detection evaluations and are standard in frameworks like YOLO, as well as in renowned benchmarks such as COCO.

Precision and **Recall** provide essential insights into the quality of the detections by indicating the proportion of true positives relative to predictions and the model's ability to correctly identify objects. In contrast, **mAP@50-95** assesses the average precision of the model across different IoU (Intersection over Union) thresholds, offering a comprehensive view of detection accuracy at varying levels of overlap between predictions and ground truth objects.

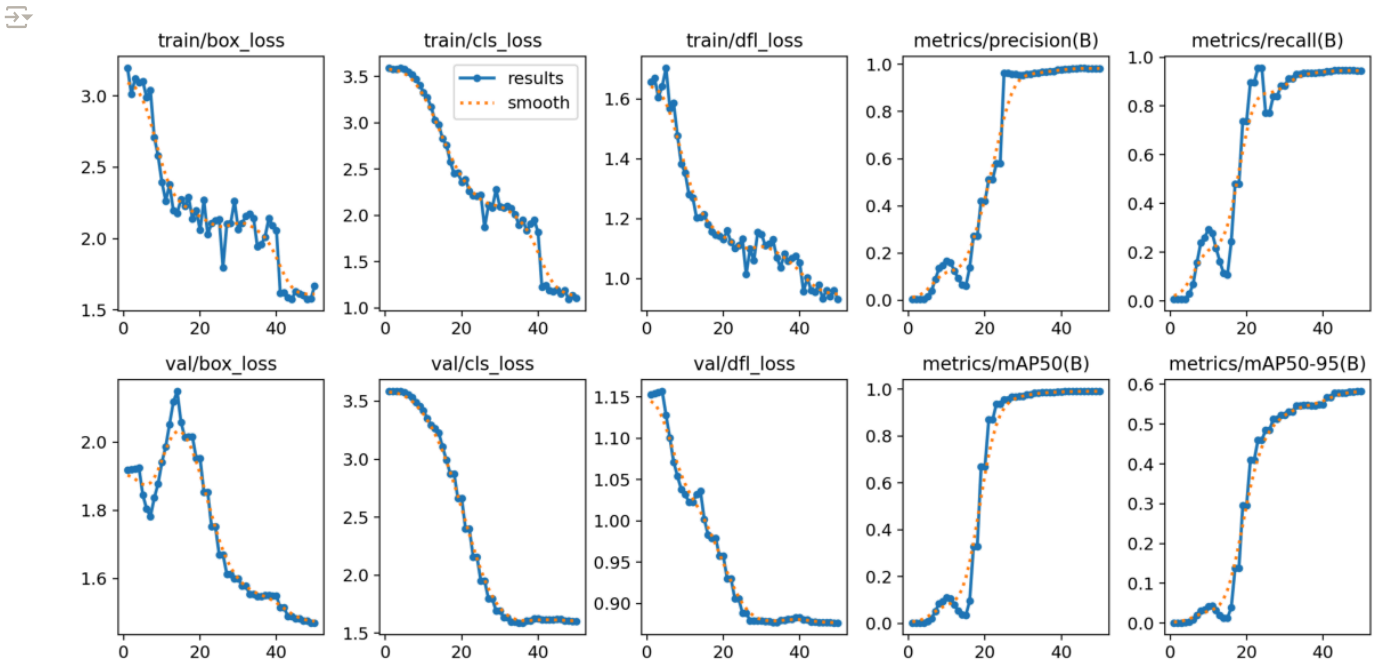
```
best_model = YOLO(f"{config['project']}/{config['name']}/weights/best.pt")

metrics = best_model.val()
```

```
print("======== Metrics ========")
print(f"mAP50-95: {metrics.box.map:.4f}")
print(f"mAP50: {metrics.box.map50:.4f}")
print(f"Results: {metrics.box.p.mean():.4f}")
print(f"Recall: {metrics.box.r.mean():.4f}")
```

```
Ultralytics 8.3.94 🚀 Python-3.11.11 torch-2.6.0+cu124 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,005,843 parameters, 0 gradients, 8.1 GFLOPs
val: Scanning /content/erick/val/labels.cache... 5 images, 0 backgrounds, 0 corrupt: 100%|██████████| 5/5 [00:00<?, ?it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 1/1 [00:02<00:00,  2.31s/
                   all          5        855      0.981      0.944      0.991      0.583
Speed: 0.2ms preprocess, 31.0ms inference, 0.0ms loss, 5.4ms postprocess per image
Results saved to runs/detect/val2
======== Metrics ========
mAP50-95: 0.5833
mAP50: 0.9914
Results: 0.9806
Recall: 0.9438
```

Visualize Results

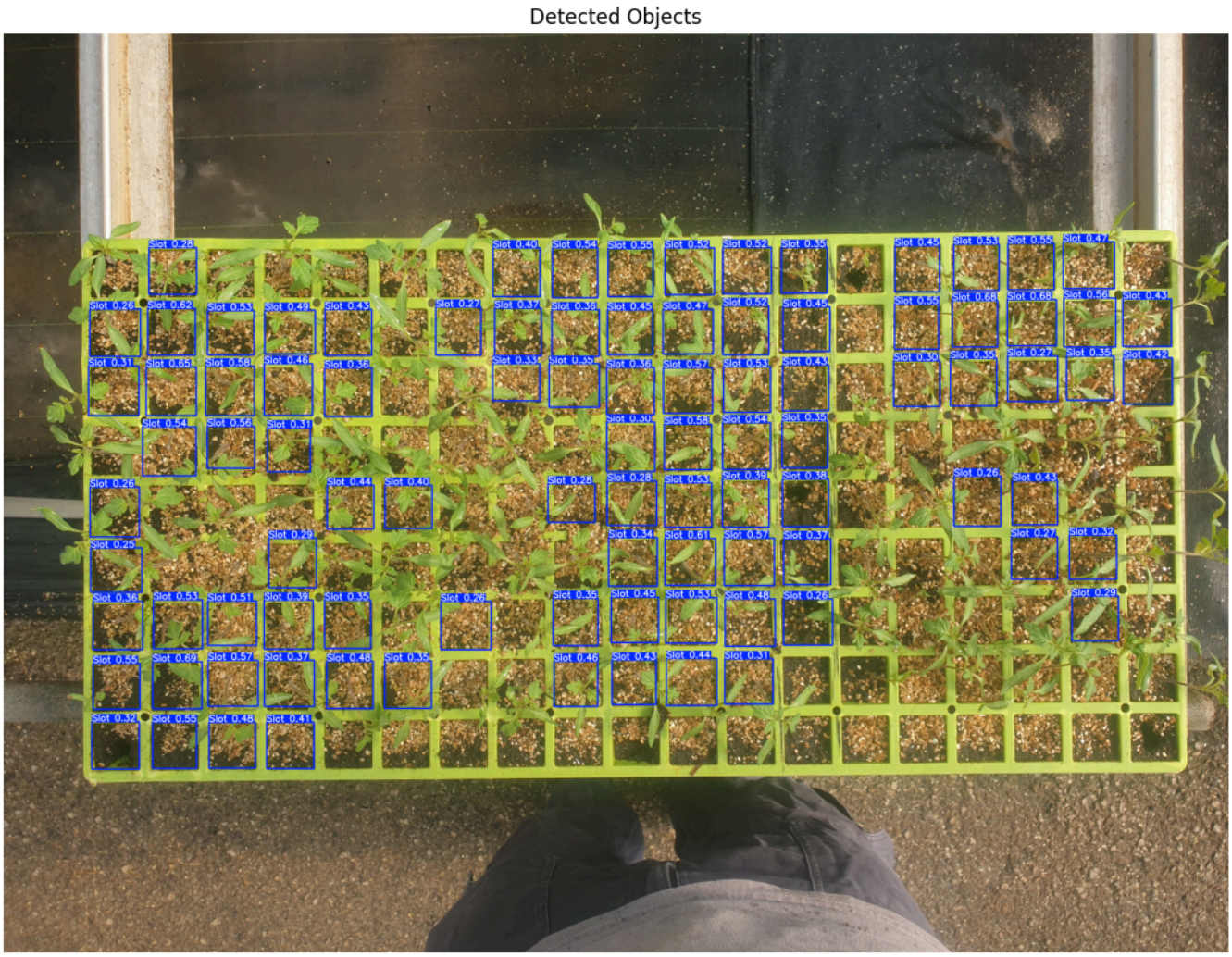
```
results_path = Path(f"{config['project']}/{config['name']}/")
results = cv2.imread(str(results_path / "results.png"))
plt.figure(figsize=(15, 10))
plt.imshow(cv2.cvtColor(results, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```



Test Image

```
sample_img = "/content/erick/val/images/7039_14_151-6_2nd.jpg"
results = best_model(sample_img)

res_plotted = results[0].plot(line_width=3)
plt.figure(figsize=(15, 10))
plt.imshow(cv2.cvtColor(res_plotted, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title("Detected Objects")
plt.show()
```

```
image 1/1 /content/erick/val/images/7039_14_151-6_2nd.jpg: 480x640 95 Slots, 134.1ms
Speed: 5.7ms preprocess, 134.1ms inference, 7.1ms postprocess per image at shape (1, 3, 480, 640)
```



Initial Results Discution

The initial results indicate that the model is learning, although there are signs of instability in both classification and bounding box predictions, as evidenced by the slight fluctuations in box_loss and cls_loss. While these losses are generally decreasing, the fact that precision and recall metrics jump to 1 early, while the model didnt fully converged yet, this suggests that the model is likely to be memorizing instead of actually learning. Additionally, when visualizing model outputs, there appears to be a wide range of predicted confidences (from about 0.2 up to 0.7) in certain areas of the image. This high variance could indicate that the model is unsure about specific regions.

4. Improvement Strategy

Slicing Strategy for Improving Small Object Detection

The original images had high resolutions (e.g., 4032x3024, 4000x2252), and the objects to be detected were very small relative to the overall image size:

- **Average BBox area:** 18,936.92 pixels²
- **Median BBox area:** 20,804.56 pixels²

For reference, in a typical 4000x2252 image (~9MP):

- The object occupied only **~0.2% of the total image area** (18,936 / 9,000,000 ≈ 0.0021).

Identified Problem

At such high resolutions, the object becomes **nearly imperceptible** to conventional detection models, even when using absolute bounding boxes. This results in:

1. Loss of detail during downsampling (resizing to standard resolutions such as 640x640).

2. Learning difficulties due to the low pixel density of the object.

Implemented Solution

The images were divided into **640x640 pixel patches** using a slicing technique, resulting in:

- **Expansion of the training dataset:** from 16 to 581 images (train) and 5 to 197 (val).
- **Benefits:**
  - **Artificial increase in the object's relative resolution:**
    In a 640x640 patch (409,600 pixels²), the object's area now represents **~4.6%** (18,936 / 409,600) — making it 22 times more prominent.
  - Reduced computational complexity, as there are fewer pixels to process per iteration.
  - Generation of pseudo-augmented data that provides contextual variations.

Using SAHI (Slicing Aided Hyper Inference) for Small Objects

How SAHI Complements YOLO

Approach	SAHI Advantage	Limitation of Pure YOLO
Inference	Processes the image in slices	Processes the entire image
Object Size	Detects objects smaller than 50px	Often misses smaller objects
Computational Cost	Increases processing time by ~20%, but improves mAP@50-95 by ~30%	Faster inference, but with lower precision

This slicing strategy, especially when paired with SAHI, significantly enhances the detection of small objects by increasing their relative size and preserving more details during inference.

Data Augmentation with Albumentations

The use of the Albumentations library, in conjunction with YOLO augmentations, allows for the generation of a larger and more robust dataset by providing techniques not present in YOLO or with limited configuration options. In this sense, the following augmentations were applied to the pipeline:

```
# Augmentation pipeline
aug_transforms = A.Compose([
    A.RandomRotate90(p=0.5),
    A.HorizontalFlip(p=0.5),
    A.RandomBrightnessContrast(p=0.3),
    A.HueSaturationValue(p=0.2),
    A.RandomGamma(gamma_limit=(60, 115), p=0.3),
    A.GaussNoise(p=0.3),
], bbox_params=A.BboxParams(format='yolo', label_fields=['class_labels']))
```

The levels of brightness, saturation, hue and GaussNoise were kept at their default values and adjusting just the probability to happen to , as they are already tuned for use with YOLO, while the RandomGamma required fine-tuning to better distribute the irregular exposure generated (dark/light). The use of Albumentations allowed for a 3x increase in the dataset size, totaling 1,940 images, with 1,743 for training and 197 for validation. This process can be seen in `utils.py`

Utils File

Run the block bellow to download the utils file (from the git repository) and perform the slicing and augmentation of the dataset

```
import locale
locale.getpreferredencoding = lambda: "UTF-8"

!wget -O /content/utils.py https://raw.githubusercontent.com/eriklemy/slots_detection/refs/heads/main/utils.py

# remove the folder if need to remake the augmentation and slicing
# !rm -r /content/sahi_augmented
!python utils.py --data /content/erick/data.yaml --output /content/sahi_augmented --slice-size 640 --overlap-ratio 0.2
```

Mostrar saída oculta

```
model = YOLO("yolov8n.pt")

config = {
    "data": "/content/sahi_augmented/data.yaml",
    "epochs": 50,
    "imgsz": 640,
    "batch": 16,
```

```
    "workers": 8,
    "device": device,
    "freeze": list(range(11)),
    # "augment": True,   #  allow TTA
    "mosaic": 1.0,       # reducing can improve performance
    "mixup": 0.5,
    "scale": 0.5,
    "fliplr": 0.5,
    "flipud": 0.2,
    "hsv_h": 0.015,
    "hsv_s": 0.7,
    "hsv_v": 0.4,
    "patience": 30,
    "seed": 42,
    # "single_cls": True, # improve, but change the class name to 'item'
    "project": "slots",
    "name": "sahi",
}
```

Training

Run the block bellow to train the new version

```
!rm -r /content/slots/sahi

results = model.train(**config)
```

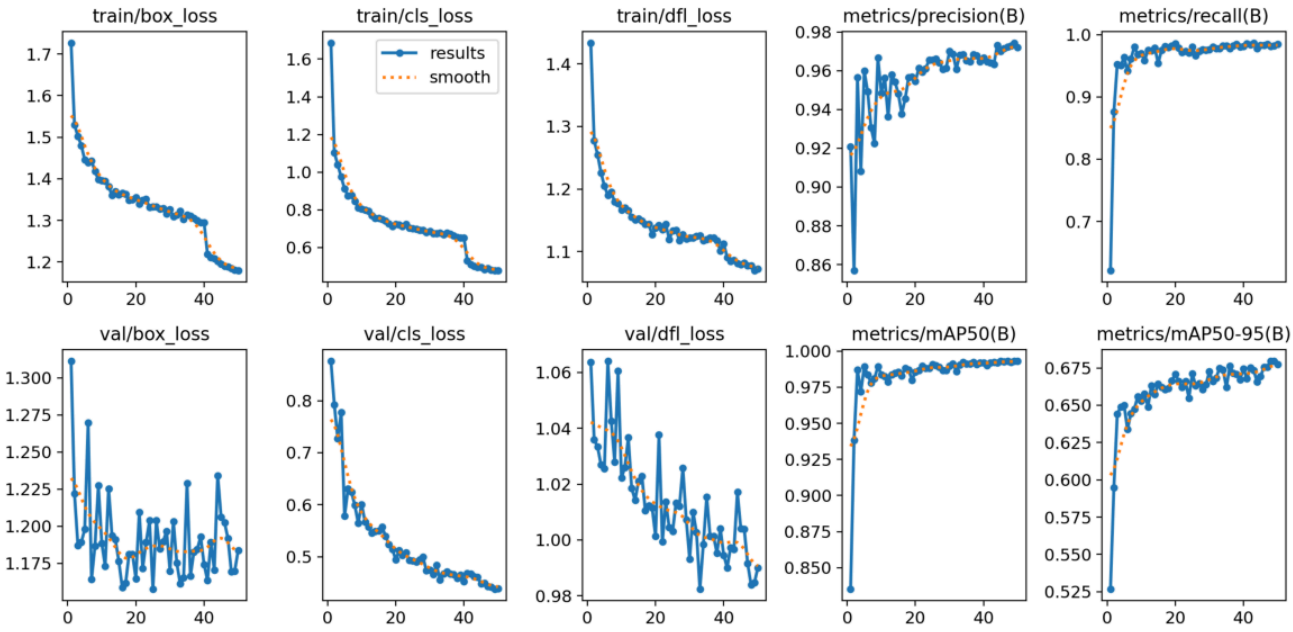
```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      42/50      8.68G      1.211     0.5097      1.085        437        640: 100%|██████████| 109/109 [00:44<00:00,  2.47it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:02<00:00,  2.84
                   all        197       2661      0.964      0.984      0.992      0.675

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      43/50      8.68G      1.208     0.4999      1.087        109        640: 100%|██████████| 109/109 [00:46<00:00,  2.33it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:02<00:00,  2.92

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      44/50      8.68G      1.201     0.4968      1.082        119        640: 100%|██████████| 109/109 [00:40<00:00,  2.71it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:02<00:00,  2.58

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      45/50      8.68G      1.196     0.4946       1.08        299        640: 100%|██████████| 109/109 [00:42<00:00,  2.56it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:03<00:00,  1.88

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      46/50      8.68G       1.19     0.4852      1.082         99        640: 100%|██████████| 109/109 [00:41<00:00,  2.64it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:03<00:00,  2.27

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      47/50      8.68G      1.189     0.4886      1.078        128        640: 100%|██████████| 109/109 [00:42<00:00,  2.59it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:03<00:00,  2.08

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      48/50      8.68G      1.185     0.4823      1.077        128        640: 100%|██████████| 109/109 [00:42<00:00,  2.54it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:02<00:00,  3.02

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      49/50      8.68G      1.181     0.4787       1.07        109        640: 100%|██████████| 109/109 [00:40<00:00,  2.69it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:02<00:00,  2.48

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      50/50      8.68G      1.179     0.4775      1.072        107        640: 100%|██████████| 109/109 [00:44<00:00,  2.43it/s]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:02<00:00,  2.72


50 epochs completed in 0.819 hours.
Optimizer stripped from slots/sahi/weights/last.pt, 6.2MB
Optimizer stripped from slots/sahi/weights/best.pt, 6.2MB

Validating slots/sahi/weights/best.pt...
Ultralytics 8.3.94 🚀 Python-3.11.11 torch-2.6.0+cu124 CUDA:0 (Tesla T4, 15095MiB)
Model summary (fused): 72 layers, 3,005,843 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 7/7 [00:06<00:00,  1.03
                   all        197       2661      0.974      0.982      0.993       0.68
Speed: 0.4ms preprocess, 2.8ms inference, 0.0ms loss, 4.8ms postprocess per image
Results saved to slots/sahi
```

5. Results Discution

As shown in the figure below, which presents the model metrics, the training losses (`box_loss`, `cls_loss`, and `df_loss`) consistently decrease over the epochs, indicating that the model is effectively learning to locate and classify objects. The validation losses (`val/box_loss`, `val/cls_loss`, `val/df_loss`) also decline, albeit at a slightly slower pace, suggesting that is trying to generalize but the difference between train/val cls_loss and df_loss can imply overfitting.

Meanwhile, the performance metrics (`Precision`, `Recall`, `mAP@50`, `mAP@50-95`) progressively increase and stabilize at high values indicating that the model is improving in terms of precision (correct predictions). However, the `mAP@50-95` still shows a drop, which suggest difficulties in precisely localizing objects at higher IoU thresholds.

```
import utils

results_path = Path(f"/content/{config['project']}/{config['name']}/")
results = cv2.imread(str(results_path / "results.png"))
plt.figure(figsize=(15, 10))
plt.imshow(cv2.cvtColor(results, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()

# slicing over the original val (5 images - full size)
utils.evaluate_with_sahi(f"/content/{config['project']}/{config['name']}/weights/best.pt", data_yaml_path)
```




```
Evaluating with SAHI:   0%|          | 0/5 [00:01<?, ?it/s]Performing prediction on 40 slices.

Image: 7093_12_7093_117-12_2ND
Standard prediction count: 171
SAHI sliced prediction count: 171
Evaluating with SAHI:  20%|██        | 1/5 [00:04<00:13,  3.32s/it]Performing prediction on 40 slices.

Image: 7093_7_7093_81-6_2ND
Standard prediction count: 178
SAHI sliced prediction count: 174
Evaluating with SAHI:  40%|████      | 2/5 [00:07<00:09,  3.23s/it]Performing prediction on 48 slices.

Image: 7039_14_151-6_2nd
Standard prediction count: 174
SAHI sliced prediction count: 172
Evaluating with SAHI:  60%|██████    | 3/5 [00:13<00:08,  4.36s/it]Performing prediction on 40 slices.

Image: 7126_18_7281_71-6_2ND
Standard prediction count: 171
SAHI sliced prediction count: 171
Evaluating with SAHI:  80%|████████  | 4/5 [00:16<00:03,  3.93s/it]Performing prediction on 40 slices.

Image: 7166_random_7166_accept_7166_random_2ND
Standard prediction count: 171
SAHI sliced prediction count: 171
Evaluating with SAHI: 100%|██████████| 5/5 [00:18<00:00,  3.73s/it]
```

As can be seen in the figure bellow (after running the block) and in the folder **out/sahi** and **out/standard** (for the full quality image), applying SAHI significantly improved class identification accuracy, increasing between 15% and 20%. Additionally, detections became more consistent, reducing false positives and further demonstrating the effectiveness of the technique. However, it was observed that images with better lighting conditions yielded more stable results, suggesting that illumination plays a crucial role in detection performance and can be further improved.

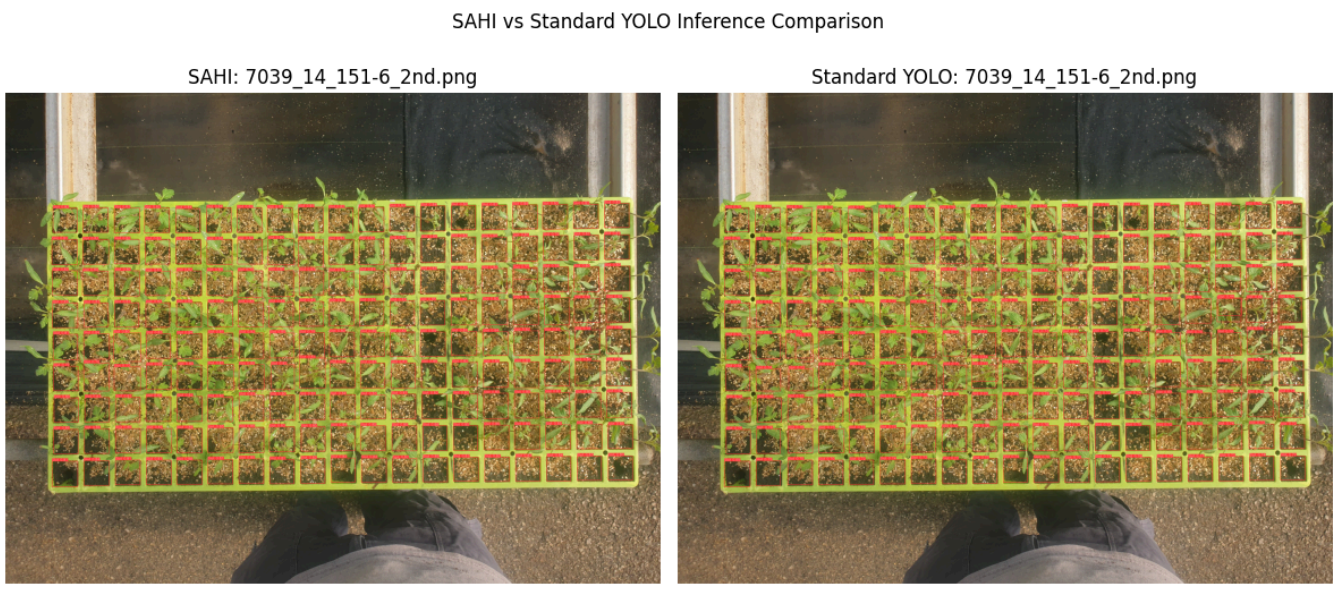
Problems seen

1. Aspect Ratio Dependency: Images resized without preserving their original aspect ratio (e.g., stretched or distorted) led to degraded performance. Analysis of the training data revealed that a large amount of annotations has a 1:1 aspect ratio for the slots (see aspect ratio distribution plot). This indicates that the model relies heavily on this ratio for slot detection, limiting its generalizability to non-square inputs.
2. Lighting Sensitivity: Detection stability was significantly influenced by illumination quality, with poorly lit images yielding inconsistent results.

```
train_image_dir = "sahi_augmented/train/images"
label_image_dir = "sahi_augmented/train/labels"
stats = analyze_dataset_statistics(train_image_dir, label_image_dir)
plot_distributions(stats)
```

Mostrar saída oculta

```
# Load the SAHI results
sahi_image_paths = glob.glob(os.path.join("out/sahi/", "*.*"))
sahi_image_paths = [p for p in sahi_image_paths if p.lower().endswith(('.png', '.jpg', '.jpeg'))]
sahi_img = cv2.imread(sahi_image_paths[0])

# Load the standard YOLO results
stantard_image_paths = glob.glob(os.path.join("out/standard/", "*.*"))
stantard_image_paths = [p for p in stantard_image_paths if p.lower().endswith(('.png', '.jpg', '.jpeg'))]
stantard_img = cv2.imread(stantard_image_paths[0])

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].imshow(cv2.cvtColor(sahi_img, cv2.COLOR_BGR2RGB))
axes[0].set_title(f"SAHI: {os.path.basename(sahi_image_paths[0])}", fontsize=12)
axes[0].axis('off')

axes[1].imshow(cv2.cvtColor(stantard_img, cv2.COLOR_BGR2RGB))
axes[1].set_title(f"Standard YOLO: {os.path.basename(stantard_image_paths[0])}", fontsize=12)
axes[1].axis('off')

plt.tight_layout()
plt.suptitle(f"SAHI vs Standard YOLO Inference Comparison", y=0.98)
plt.show()
```




```
img = cv2.imread("erick/val/images/7039_14_151-6_2nd.jpg")
img_resized = cv2.resize(img, (640, 640))
model = YOLO(f"{config['project']}/{config['name']}/weights/best.pt")
results = model(img_resized, verbose=False)
res_plotted = results[0].plot(line_width=1)
plt.figure(figsize=(15, 10))
plt.imshow(cv2.cvtColor(res_plotted, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title("Detected Objects in Resized Image")
plt.show()
```

Further Improvement Sugestions

1. **Data Improvements**:
    - Increase dataset size
    - Add more challenging scenarios (occlusions, varying lighting)
    - Improve annotation quality to reduce label incosistence
2. **Model Improvements**:
    - Medium YOLO variants (yolov8s/yolov8s/yolo11s/yolo11l), because Larger can cause overfitting if the dataset still small
    - Explore R-CNN models for higher accuracy (like detectron2)
    - Attention mecanism (models like RT-DET)