

Model Training for Lemon Defect Detection Using YOLO

1. Introduction

In this project, the model training task focuses on developing an accurate and real-time object detection system capable of identifying and classifying lemons into various conditions and defect types. Leveraging the YOLO11 architecture, the training process integrates carefully prepared datasets, robust data augmentation strategies, transfer learning, and systematic evaluation to ensure optimal performance in agricultural quality inspection.

YOLO (You Only Look Once) is a family of state-of-the-art object detection algorithms designed for high-speed, real-time applications. Unlike traditional two-stage detectors that first generate region proposals and then classify them, YOLO performs detection in a single forward pass of the neural network, simultaneously predicting bounding boxes and class probabilities. This unified approach significantly improves inference speed while maintaining competitive accuracy. The latest YOLO11 version enhances feature extraction through an optimized backbone, improved spatial pyramid pooling, and advanced feature fusion, making it particularly suitable for agricultural defect detection where both real-time performance and detection precision are crucial.

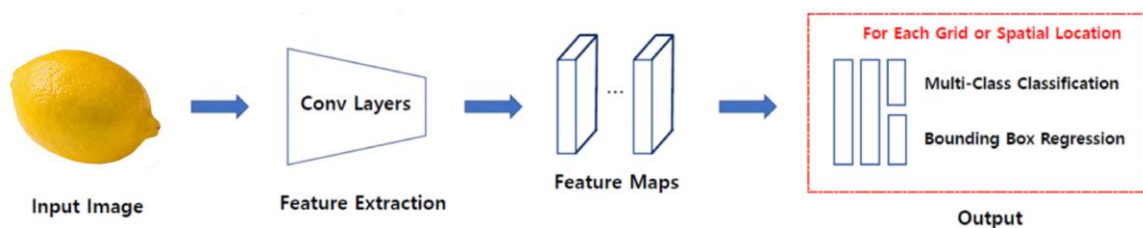


Figure 1. Overview of YOLO Object Detection Workflow

2. Dataset Preparation

2.1 Image Collection

All data were collected under a single, controlled scenario to minimize domain shift. The imaging setup used fixed lighting and view geometry: an LED light provided consistent fill illumination; the camera and lens position were kept constant (same angle, distance, and focal settings). Lemons rolled along a linear sliding rail, and we recorded videos of the rolling process.

From each video, still frames were extracted with FFmpeg and then uploaded to Roboflow for annotation. A typical command (adjust the rate as needed) was:

```
ffmpeg -i input.mp4 -r 3 output_frames/frame_%05d.jpg
```

- -r 3 samples ~3 frames per second to reduce near-duplicates.
- Filenames carry the video ID and frame index for traceability.
- We avoided extracting frames that were too similar to keep the dataset efficient.

2.2 Class Definitions

This project uses five classes arranged in two annotation layers:

- Fruit-level (one box per lemon, mutually exclusive):
GoodLemon, NotRipeLemon, DefectiveLemon
- Lesion-level (zero, one, or multiple boxes per lemon):
Bruised, Rotten

```
nc: 5
```

```
names: ['Bruised', 'DefectiveLemon', 'GoodLemon', 'NotRipeLemon', 'Rotten']
```

Class descriptions

- GoodLemon: Fruit appears healthy; no visible defects or decay, normal/yellow color.
- NotRipeLemon: Fruit is predominantly green or clearly unripe; otherwise defect-free.
- DefectiveLemon: Fruit exhibits any structural/appearance defect (e.g., cracks, cuts, holes, crushed/deformed) or visible lesions (bruises/rot).
- Bruised (lesion-level): Localized pressure damage/bruise region without active decay.
- Rotten (lesion-level): Localized decay/rot/mold region.

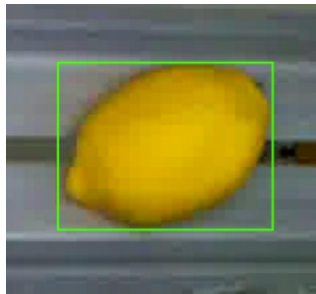


Figure 2. Example of GoodLemon annotation (fruit-level bounding box in green)

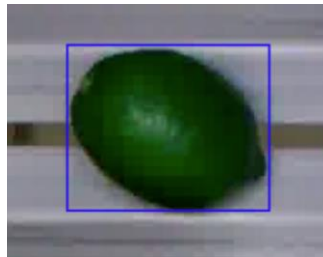


Figure 3. Example of NotRipeLemon annotation (fruit-level bounding box in blue)

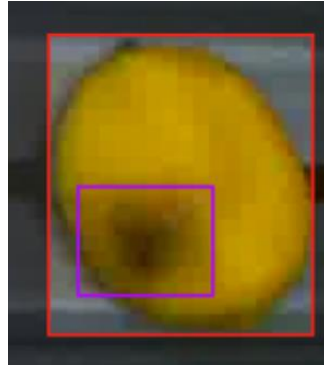


Figure 4. Example of DefectiveLemon with Bruised lesion annotation (red fruit-level box, purple lesion box)

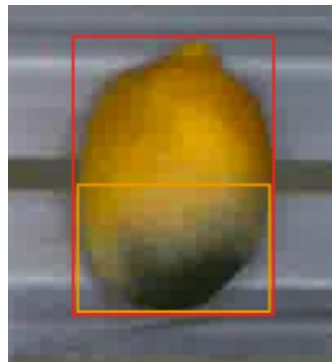


Figure 5. Example of DefectiveLemon with Rotten lesion annotation (red fruit-level box, orange lesion box)

Labeling policy

1. Fruit-level box:

For every visible lemon, draw one bounding box around the entire fruit and assign exactly one of:

GoodLemon, NotRipeLemon, or DefectiveLemon.

2. Lesion-level boxes:

If the fruit has bruises or rot, draw additional tight boxes around each lesion area and label them as Bruised or Rotten.

Multiple lesions → multiple lesion boxes are allowed.

3. Decision rule for fruit-level class (priority):

- If Rotten or Bruised is present \rightarrow fruit-level = DefectiveLemon.
- Else if clearly unripe \rightarrow fruit-level = NotRipeLemon.
- Else \rightarrow fruit-level = GoodLemon.

4. Overlap:

Overlap between the fruit-level box and lesion-level boxes is expected (fruit box contains lesion boxes).

5. Ambiguity:

If lesion is too small/uncertain, keep the fruit-level decision; you may skip the lesion box to avoid noisy labels.

6. One lemon, one fruit-level box:

Do not assign both GoodLemon and NotRipeLemon to the same lemon; fruit-level classes are mutually exclusive.

Lesion-level classes (Bruised, Rotten) are not exclusive with fruit-level and may be multiple per fruit.

Practical tip: keep lesion boxes tight; avoid “merging” distinct lesions into one big box unless they are indistinguishable.

2.3 Data Annotation

Annotations were created in Roboflow following the two-layer scheme described in Section 2.2.

- Fruit-level boxes: one per lemon, labeled as GoodLemon, NotRipeLemon, or DefectiveLemon.
- Lesion-level boxes: zero or more per lemon, labeled as Bruised or Rotten.

Roboflow workflow

- Create a Roboflow project with 5 classes:
['Bruised','DefectiveLemon','GoodLemon','NotRipeLemon','Rotten'].
- Upload FFmpeg-extracted frames; keep file names with video_id + frame_id for traceability.
- Use box annotations (not polygons) and keep a consistent color legend across classes.
- Export in YOLO format (class x_center y_center width height, normalized to image size).

Box quality & QA checks

- Fruit box: cover the whole lemon; tight but do not cut the peel boundary.
- Lesion box: tight around the visible lesion; do not “merge” distinct lesions into one big box.
- Visibility: skip fruit/lesion if < ~60% is visible or heavily truncated by the image border.
- Blur/Occlusion: discard frames with extreme motion blur or severe occlusion.
- Spot checks: review 5–10% of new labels per batch in Roboflow’s preview; verify class counts and bounding-box tightness before training.
- Consistency note: a fruit cannot be both GoodLemon and NotRipeLemon; fruit-level classes are mutually exclusive per instance.

Minimal data.yaml structure:

```
train: ../train/images
```

```
val: ../valid/images
test: ../test/images

nc: 5

names: ['Bruised', 'DefectiveLemon', 'GoodLemon', 'NotRipeLemon', 'Rotten']
```

2.4 Data Splitting

We use a 70/20/10 split for train/val/test with two constraints:

1. Grouped by video to prevent leakage from near-duplicate frames. Frames from the same video id must not appear in multiple splits.
2. Stratified by fruit-level classes so each split has a representative proportion of GoodLemon / NotRipeLemon / DefectiveLemon. (Lesion classes correlate with DefectiveLemon.)

If lesion counts are extremely imbalanced, consider oversampling lesion-heavy videos during training rather than forcing a brittle split.

2.5 Data Augmentation

To preserve the controlled lighting domain (fixed LED illumination and camera geometry) while improving geometric robustness, we only apply:

- Random horizontal and vertical flips
- Random rotations and scaling

We do not apply color jitter, noise, perspective, mosaic, mixup, or copy-paste to avoid domain drift away from the controlled setup.

3. Model Training

3.1 Environment Setup

Training was conducted using the Ultralytics YOLO11 framework in Python 3.10 with PyTorch backend.

Hardware: NVIDIA GPU with CUDA support (e.g., RTX 4070), 8 GB VRAM.

Key libraries: ultralytics, opencv-python, numpy, pandas.

3.2 Training Configuration

The YOLO11n variant was selected for a balance between speed and accuracy. The dataset was loaded using the data.yaml file prepared in Section 2.3.

- Main hyperparameters:
- Image size: 640×640
- Batch size: 16
- Epochs: 100
- Optimizer: AdamW
- Initial learning rate (lr0): 0.01
- Augmentations: only random flips, rotations, and scaling (see Section 2.5)
- Early stopping: patience = 10 epochs without improvement in validation loss.

3.3 Training Workflow

1. Initialize Model

Load COCO-pretrained YOLO11n weights to leverage transfer learning.

2. Train

Fit the model on the training set with validation at the end of each epoch.

3. Evaluate

Monitor mAP@0.5, mAP@0.5:0.95, Precision, Recall on the validation set.

4. Save Best Model

Automatically save the checkpoint with the highest validation mAP.

5. Test on Hold-out Set

After training, evaluate on the test set to confirm generalization.

6. Export for Deployment (optional)

Convert the model to ONNX or other deployment formats if needed.

3.4 Training Script (Ultralytics API)

```
from ultralytics import YOLO

# Step 1: Load pretrained YOLO11n model
model = YOLO('yolo11n.pt')

# Step 2: Train
model.train(
    data='data.yaml',
    imgsz=640,
    epochs=150,
    batch=16,
    optimizer='AdamW',
    lr0=0.01,
```

```
degrees=10.0,  
scale=0.5,  
fliplr=0.50,  
flipud=0.15,  
hsv_h=0.0, hsv_s=0.0, hsv_v=0.0,  
mosaic=0.0, mixup=0.0, copy_paste=0.0,  
patience=20,  
device=0  
)  
  
# Step 3: Evaluate on validation set  
metrics = model.val()  
  
# Step 4: Export for deployment (optional)  
model.export(format='onnx')
```

4. Model Validation

4.1 Validation Approach

After training, the best-performing model checkpoint (highest mAP on the validation set) was evaluated to quantify detection and classification performance. Validation was conducted using the hold-out validation set defined in Section 2.4, ensuring no data leakage from the training process.

We used the Ultralytics YOLO11 built-in evaluation pipeline, which measures:

- mAP@0.5 — Mean Average Precision at IoU threshold 0.5, representing detection accuracy with moderately strict localization.
- mAP@0.5:0.95 — Mean Average Precision averaged over IoU thresholds from 0.5 to 0.95 in 0.05 steps, providing a more comprehensive view of localization performance.
- Precision (P) — Proportion of predicted boxes that are correct.
- Recall (R) — Proportion of ground truth objects detected by the model.
- Confusion Matrix — To visualize per-class prediction performance.
- PR Curves — Precision–Recall trade-off for each class.

4.2 Fruit-level and Lesion-level Evaluation

Since our annotation scheme contains two layers (fruit-level and lesion-level), we validated performance separately:

- Fruit-level classes (GoodLemon, NotRipeLemon, DefectiveLemon) were evaluated for correct fruit classification and bounding box localization.
- Lesion-level classes (Bruised, Rotten) were evaluated for precise localization of defective areas.

This two-stage evaluation allows us to verify:

1. Whether the model correctly identifies the overall fruit condition.
2. Whether the model precisely localizes and labels specific defects.