# Fruit-based Object Detection Using YOLOv8

CS 463 Computer Vision: Final Individual Project
Ruvarashe Divine Ruvimbo Sadya
December 14, 2024

## 1. Introduction

### 1.1 Background

Fruit detection is a task in which computer vision principles can be applied as it involves identifying and locating fruits in images or frames which is useful in automated sorting. The task is useful across different industries but primarily in agriculture, food quality control, and inventory management **[1], [2], [3], [4]**.

Some such applications include:

- **Agriculture**: crop monitoring, yield estimation, ripeness assessment.
- **Food processing**: quality grading and waste reduction.
- **Inventory/supply chain**: stock control and optimization.
- **Retail**: freshness information and automated checkout.
- **Research**: ecological and botanical data collection.
- **Environment**: wildlife food-source monitoring.
- **Health**: dietary tracking.

Overall, fruit detection boosts efficiency, reduces waste, and supports data-driven decisions across industries.

In recent times, many automated fruit-picking systems have emerged but production costs are high [7]. Manual picking is labour-intensive, and requires skilled individuals. These challenges highlight the importance of dependable, image-based classification techniques powered by computer vision models.

Computer vision models have become central to modern object detection. Modern architectures like Faster R-CNN, SSD, and YOLO have shaped today's real-time detection landscape. While Faster R-CNN is popular for its high accuracy through a two-stage pipeline, SSD and YOLO offer faster inference in a single shot pipeline.

Recent studies have shown that YOLO-based models are particularly effective in agricultural settings [8]. Building on this foundation, this project aimed at applying the newer YOLO model YOLOv8 for multi-class fruit detection, that is, classification and localisation.

The YOLOv8 model is highly efficient at detecting objects because of its advanced training methodology, which involves the integration of knowledge distillation and pseudo-labeling techniques [5], [6]. With this in mind, the main aim of the project was to answer the question: Can transfer learning with YOLOv8 achieve robust fruit detection (>50% mAP@0.5)?

### 1.2. Model Choice: YOLOv8n

I chose to use the YOLOv8 object detection model for this project because it has a good balance between speed and accuracy compared to other architectures. Although two-stage detectors, like Faster R-CNN, can achieve higher precision, they are slower and more computationally expensive. While SSD would have also been a good choice because of its fast speed, it struggles with tiny and densely packed objects because it relies on fixed anchor boxes and lower-resolution feature maps. YOLOv8 reduces these trade-offs because of its anchor-free design, feature fusion using the FPN-PAN neck, and efficient C2f modules. Therefore it is a good choice for lightweight and accurate fruit detection under the project's computational constraints.

## 2. Methodology

### 2.1 Data

#### 2.1.1 Dataset Overview

For this project, I used the Fruit Detection Dataset obtained from Kaggle [9]. The dataset contains a total of 8,479 colored images annotated in YOLOv8

format, covering six fruit classes: Apple, Banana, Grapes, Orange, Pineapple, and Watermelon. The images are pre-split into 3 sets: training (7108 images), validation (914 images), test (457), thereby giving a 84-10-6% split across the groups.

All images in the dataset were first pre-processed by correcting their EXIF orientation (Exchangeable image file format - a protocol to store various meta-information about the images taken by digital cameras) and resizing them to 640 × 640 pixels. This ensures that every image has the same orientation and size, which helps the model train more consistently and reduces variation caused by camera differences.

Each original image also has three augmented versions created to increase dataset diversity. According to the dataset information provided, only one augmentation type is explicitly mentioned: 50% probability horizontal flip (including bounding box adjustment). This flip mirrors the fruit in the image and updates the bounding boxes so that labels remain accurate.

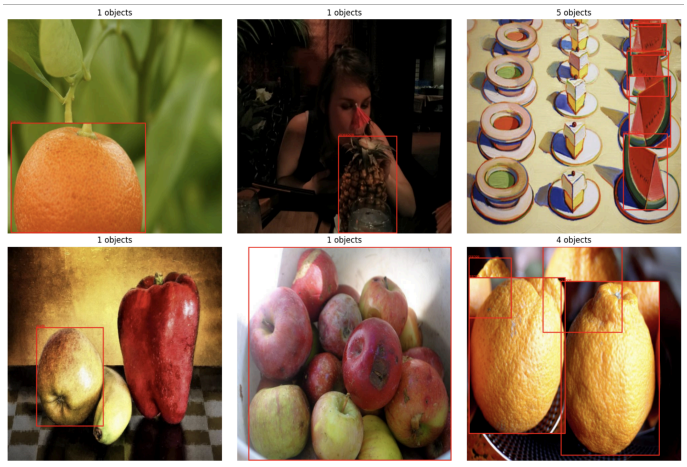


Figure 2.1: Sample images from the dataset with labels surrounding objects

### 2.1.2 Image Labels

Each image in the dataset has an accompanying label text file which shows the coordinates of the bounding boxes for a corresponding class. The coordinates of the bounding box are given in YOLO (CXCYWH) format.

The YOLO (You Only Look Once) format, specifically the CXCYWH representation, is a widely adopted standard in object detection for describing the location and size of a bounding box. This format was prominently introduced in the YOLOv3 paper and is used in many subsequent versions like v8.

The CXCYWH format describes the bounding box using four normalized floating-point values:

- *center_x* : normalized x-coordinate of the center of the bounding box.
- *center_y* : normalized y-coordinate of the center of the bounding box.
- *width* : normalized width of the bounding box.
- *height* : normalized height of the bounding box.

[*center_x*, *center_y*, *box_width*, *box_height*]

In the YOLO format, all coordinates and dimensions are normalized to a value between 0.0 and 1.0 relative to the image size. This normalization is calculated using the following formulas:

$$x_{normalized} = \frac{x_{absolute}}{image\_width} \quad (1)$$

$$y_{normalized} = \frac{y_{absolute}}{image\_height} \quad (2)$$

*x_absolute* and *y_absolute* represent the actual coordinates. *image_width* and *image_height* are the total pixel dimensions of the image. This normalized format is beneficial because it makes the bounding box coordinates independent of the image resolution, which is ideal for training deep learning models. No matter if an image is 100 pixels wide or 1000 pixels wide, the normalized numbers still describe the same position and size relative to that image. So for instance, if the center of a fruit is at x = 0.016, that means it's 1.6% of the way across the image from the left. If y = 0.5, that means it's right in the middle vertically. If width = 0.2, then the box is 20% of the image's total width. Lastly is height = 0.1, then the box is 10% of the image's total height.

Given these are the 6 classes are apple (0), banana (1), grapes (2), orange (3), pineapple (4), and

watermelon (5), then, the following image shows 3 apples and their normalised bounding box coordinates:

000d9c59687b509b_jpg.rf.c338360b3dfe1953a5d266ee861d7e79.txt (12...

```
0 0.1890625 0.18984375 0.378125 0.37890625
0 0.5765625 0.621875 0.5921875 0.4859375
0 0.36640625 0.509375 0.0390625 0.05234375
```

Figure 2.2: Sample label text file from the dataset showing YOLO CXCYWH format

As such an important part of this project was creating functions to parse the labels from the text files and convert the coordinates to their absolute values in order to draw exact bounding boxes.

### 2.1.3 Image Characteristics

Table 2.1: Class distribution of objects across dataset

| Set | Apple | Banana | Grape | Orange | Pineapple | Watermelon |
|---|---|---|---|---|---|---|
| TRAIN | 18.9% | 9.3% | 18.8% | 43.5% | 4.3% | 5.2% |
| VALID | 17.3% | 12.1% | 25.1% | 34.1% | 4.8% | 6.7% |
| TEST | 25.8% | 10.6% | 22.4% | 31.2% | 5.3% | 4.6% |

Table 2.1 shows the distribution of the number of objects of each class across the different splits of the data. As can be seen, there is class imbalance in the dataset with Oranges dominating the training set at 43.5% objects while pineapples are the least frequent with 4.3% of objects, out of a total number of 32,061 labeled instances. This gives an overall imbalance ratio between the biggest and smallest class, a ratio of 10:1, which poses a big threat to the training.

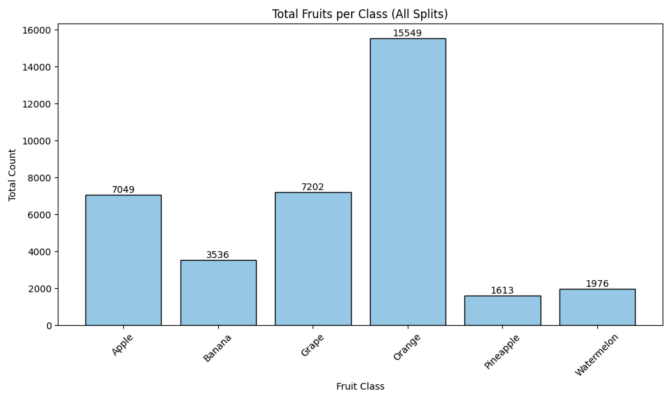


Figure 2.3: Total Number of Objects of each class

Figure 2.4 shows that the mean number of objects per image in 4.51 and the median is too, and very few outliers that there is no significant need to filter those out. This is somewhat representative of multi-object scenes like in grocery/retail scenarios.

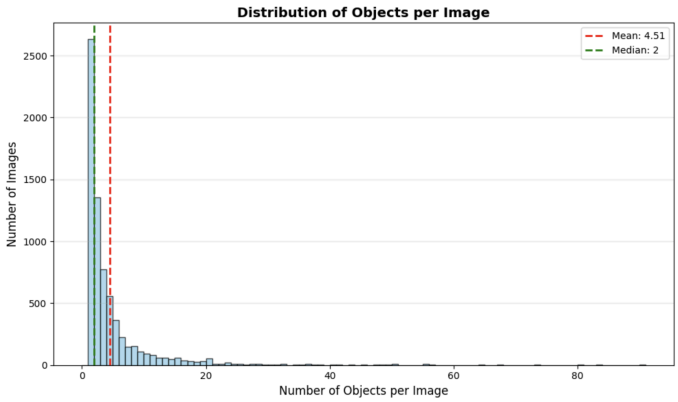


Figure 2.4: Distribution of Objects Per Image

The size of the bounding boxes ranges from <0.01% to 100% of the image area, and 39% of objects classified as tiny (<1%). Class-specific patterns show that large fruits like watermelons, pineapples, and bananas occupy substantial portions of images, while smaller fruits such as oranges, grapes, and apples are much smaller. This meant I couldn't just filter out tiny images or oranges because they may be harder to learn and need more data points.

Box plot insights (Figure 2.5) also show the size distribution of the fruits in the images.

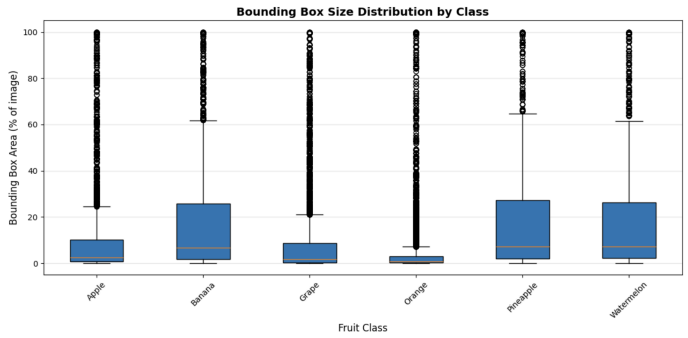


Figure 2.5: Distribution of Bounding Box Sizes

### 2.1.4 Data Curation

Ultralytics and other research encourages that background images, images with no objects related to the task at hand in them, are added to a dataset to reduce False Positives (FP). recommend about 0-10% background images to help reduce FPs (COCO has 1000 background images for reference, 1% of the total). No labels are required for background images.

Based on this information, I added 800 COCO images without the 6 classes of fruit central to the project as well as carrots and broccoli with similar shapes and textures to bananas and grapes. This essentially helps reduce false positive rate by teaching the model what "nothing" looks like and address implicit class imbalance. As such the final training size was 7,908 images (7,108 + 800 background).

### 2.1.5 Data Preprocessing

While color normalization can improve detection in scenarios where fruits blend with their environment—such as green citrus against leaves—other studies show it can reduce accuracy when fruit classes rely on subtle hue differences [10]. Because the dataset contains fruits with distinct colour identities, I intentionally avoided aggressive normalization to preserve these class-specific cues.

The Kaggle dataset came pre-augmented with horizontal flips at 50% probability. Then during training, YOLOv8 applied its own on-the-fly augmentations: mosaic augmentation (mixing four images together, though this gets disabled in the last 10 epochs to let the model fine-tune on real scenes), HSV color jittering to simulate different lighting conditions, random scaling and translation to handle size variation, and low-probability blur and CLAHE from Albumentations to mimic different camera qualities.

For handling the class imbalance, I deliberately chose not to do oversampling because on early training tests, it showed that oranges performed poorly as they have a much smaller size and ambiguous nature among other scenes while pineapples performed well because they are larger. I also trusted YOLOv8's built-in focal loss mechanism, which was specifically designed to handle class imbalance by down-weighting easy examples and focusing learning on hard cases during training.

## 2.2 Model

### 2.2.1 YOLOv8 Architecture

YOLOv8 is one of the latest versions in the YOLO (You-Only-Look-Once) object detection models, released in 2023 by Ultralytics. This version improves on the architecture of YOLOv5 with enhancements in model accuracy, speed and usability for real-time object detection [12]. When it was released, it was considered the state-of-the-art (SOTA) detection algorithm of that year. Like earlier versions, YOLOv8, still handles both object localization and classification in one-shot – one single end-to-end neural network framework

As shown in Figure 2.6, the YOLOv8 model can be broken down into 3 main parts: the backbone, neck and the head, with each playing a different role in the pipeline [12]. This default is therefore the architecture applied to this project.

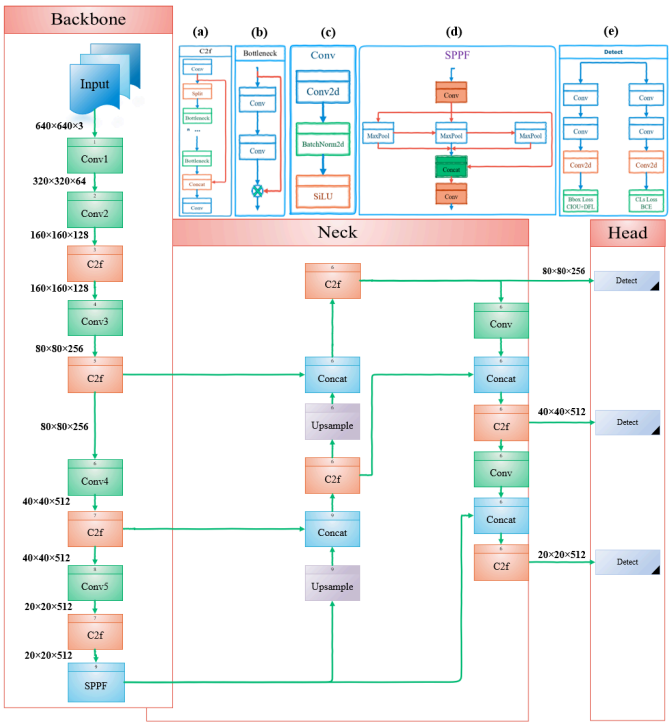


Figure 2.6: YOLOv8 Architecture [13].

**The Backbone:** The backbone is responsible for extracting features from the input image. YOLOv8 uses a

CNN-based backbone inspired by CSPDarknet, similar to earlier YOLO versions. As the image passes through the backbone, it learns features at different levels, starting from simple patterns like edges and textures and moving up to more complex shapes and object-level information. To keep the model lightweight and efficient, YOLOv8 replaces the C3 module used in YOLOv5 with a newer C2f module, which improves gradient flow and feature reuse while reducing computational cost. The backbone is built using a combination of Conv, C2f, and SPPF (Spatial Pyramid Pooling Fast) modules. The SPPF layer appears near the end of the backbone and helps the model capture information at multiple spatial scales without adding much extra computation.

**The Neck:** Once features are extracted, they are passed to the neck, whose job is to combine and refine features from different stages of the backbone. YOLOv8's neck follows the FPN–PAN (Feature Pyramid Network and Path Aggregation Network) design. This structure allows information to flow both from high-level layers to low-level layers and vice versa. In simple terms, this helps the model detect small objects using fine details and large objects using more abstract features. By merging shallow and deep feature maps through upsampling and downsampling paths, the neck improves multi-scale feature fusion and overall detection performance.

**The Head:** The final stage is the head, which produces the actual detection results. The head predicts the object class, bounding box location, and confidence score for each detection. Unlike earlier YOLO versions that relied on predefined anchor boxes, YOLOv8 uses an anchor-free, decoupled head. This means classification and bounding box regression are handled by separate branches, reducing conflicts between the two tasks and improving training stability. Bounding box regression is optimized using Distribution Focal Loss (DFL) together with CIoU loss, while classification uses binary cross-entropy loss (BCE). This design allows YOLOv8 to adapt better to objects with different shapes and sizes and improves both accuracy and convergence speed.

### 1.2.2 YOLOv8 Loss

YOLOv8 trains the object detector using a combined loss function made up of three parts: classification loss, bounding box regression loss, and distribution focal loss (DFL). Each part focuses on a different aspect of detection—identifying the correct object class, placing the bounding box accurately, and refining box boundaries.

**Classification Loss (BCE):** For classification, YOLOv8 uses Binary Cross-Entropy (BCE) loss, which measures how close the predicted class probabilities are to the true labels. BCE penalizes the model when it assigns low confidence to the correct class or high confidence to an incorrect one. BCE is calculated as:

$$\mathcal{L}_{cls} = -\left[y \log(p) + (1-y) \log(1-p)\right] \tag{3}$$

where $y$ is the ground-truth label and $\boldsymbol{p}$ is the predicted probability. BCE is applied independently to each class, making it well suited for multi-class object detection.

**Bounding Box Regression Loss (CIoU):** To learn accurate bounding box locations, YOLOv8 uses Complete Intersection over Union (CIoU) loss. Standard IoU only measures how much the predicted box overlaps with the ground-truth box, which becomes ineffective when the boxes do not overlap. CIoU improves on this by also considering how far apart the box centers are. CIoU loss is defined as:

$$\mathcal{L}_{box} = 1 - IoU + \frac{\rho^2(b_{gt}, b_{pred})}{c^2} \tag{4}$$

Here, $\boldsymbol{\rho}$ represents the Euclidean distance between the centers of the ground-truth box $\boldsymbol{b\_gt}$ and the predicted box $\boldsymbol{b\_pred}$ while $\boldsymbol{c}$ is the diagonal length of the smallest enclosing box. By penalizing both poor overlap and large center distance, CIoU helps the model converge faster and improves localization accuracy.

**Distribution Focal Loss (DFL):** YOLOv8 further improves localization by using Distribution Focal Loss (DFL). Instead of predicting each bounding box coordinate as a single value, DFL models it as a probability distribution over a set of discrete bins. This allows the model to represent uncertainty and learn more precise boundaries, which is especially useful for small, overlapping, or blurry objects. DFL is computed using a weighted cross-entropy loss:

$$\mathcal{L}_{DFL} = \sum_i [-y_i \log(p_i)] \quad (5)$$

where ***p_i*** is the predicted probability for each bin and ***y_i*** represents the target distribution. This approach improves fine-grained box alignment and stabilizes training.

**Total Loss:** The final loss used to train YOLOv8 is a weighted sum of all three components:

$$\mathcal{L}_{total} = \lambda_{box}\mathcal{L}_{box} + \lambda_{cls}\mathcal{L}_{cls} + \lambda_{DFL}\mathcal{L}_{DFL} \quad (6)$$

The weighting terms control the relative importance of localization, classification, and distribution learning. This balanced design allows YOLOv8 to achieve strong detection accuracy while maintaining real-time performance.

### 1.2.3 Range of Models

YOLOv8 is designed specifically for object detection and includes several variants that differ in accuracy and computational cost. Larger models such as YOLOv8s, YOLOv8m, and YOLOv8l achieve higher accuracy but require significantly more parameters, memory, and training time, making them less suitable for constrained environments. Given the computational limitations of this project, including restricted GPU availability and the need for efficient experimentation, YOLOv8n (nano) was selected as the baseline model due to its lightweight design (3.2 million parameters) and fast inference speed. Despite its smaller capacity, YOLOv8n has demonstrated competitive performance in prior object detection studies and offers a strong balance between accuracy and efficiency [11].

| Model | size (pixels) | mAP$^{val}$ 50-95 | Speed CPU ONNX (ms) | Speed A100 TensorRT (ms) | params (M) |
|---|---|---|---|---|---|
| YOLOv8n | 640 | 37.3 | 80.4 | 0.99 | 3.2 |
| YOLOv8s | 640 | 44.9 | 128.4 | 1.20 | 11.2 |
| YOLOv8m | 640 | 50.2 | 234.7 | 1.83 | 25.9 |
| YOLOv8l | 640 | 52.9 | 375.2 | 2.39 | 43.7 |
| YOLOv8x | 640 | 53.9 | 479.1 | 3.53 | 68.2 |

Figure 2.7: Overview of the YOLOv8 variants [11].

## 2.3 Hyperparameters

### 2.3.1 Training and Hyperparameter Configuration

I trained the model using a staged fine-tuning strategy based on the YOLOv8n architecture. In total, I ran four training experiments: one baseline model and another model of three progressively fine-tuned phases. This setup follows standard transfer learning practice, where a pre-trained detector is first adapted to a new dataset and then gradually refined to improve task-specific performance while reducing the risk of overfitting.

### 2.3.2 Baseline Model Training

I first trained a baseline model to establish a reference point for later fine-tuning. I initialized the model using the pre-trained yolov8n.pt weights provided by Ultralytics. These weights are trained on the COCO dataset and capture general visual features that transfer well to many object detection tasks.

I trained the baseline for 30 epochs, which was sufficient for initial convergence on the dataset. I also used a batch size of 16 for stability while staying within my GPU memory limits. I largely followed Ultralytics default settings, using the auto optimizer (AdamW with default momentum and weight decay) and an initial learning rate of 0.01, as these provide stable performance without extensive manual tuning. I also enabled Mixup (0.1) to improve generalization by increasing training data diversity. To avoid unnecessary training once validation performance stopped improving, I applied early stopping with a patience of 5 epochs.

For all cases, I chose Adam (AdamW) over SGD because it provides faster and more stable convergence when fine-tuning deep networks on small or imbalanced datasets. Adam adapts the learning rate individually for each parameter using gradient statistics, which is particularly beneficial in object detection models like YOLOv8 where gradients vary significantly across layers. In contrast, SGD relies on a single global learning rate and typically requires careful scheduling and longer training to converge effectively, which can be inefficient for small datasets. Prior studies consistently report that Adam and its variants outperform SGD during

fine-tuning, while SGD is more advantageous for large-scale training from scratch once optimal hyperparameters are known [14].

Table 2.10: Hyperparameters used for the baseline model

| Hyperparameter | Value |
|---|---|
| Epochs | 30 |
| Batch Size | 16 |
| Initial Learning Rate (lr0) | 0.01 |
| Optimizer | auto |
| Input Image Size (imgsz) | 640 |
| Patience | 5 |
| Mixup | 0.1 |

### 2.3.3 Fine-Tuned Model Strategy

I fine-tuned the network in three successive phases, gradually moving from a frozen backbone to full end-to-end training. This phased approach allowed the detection head to adapt to the new dataset first, before progressively updating deeper feature representations.

To address GPU memory constraints during fine-tuning, I reduced the batch size to 8 for all three phases. I also explicitly selected AdamW as the optimizer to ensure consistent optimization behavior across runs. In addition, I increased the classification loss weight (cls) to 0.6, as earlier evaluations showed that recall was the primary limiting factor, and this adjustment helped prioritize correct object detection. The table below details the hyperparameters used

Table 2.9: Hyperparameters used for the fine-tuned model

| Hyperparameter | Phase 1 | Phase 2 | Phase 3 |
|---|---|---|---|
| Epochs | 15 | 20 | 15 |
| Frozen Layers (freeze) | 10 | 5 | 0 |
| Initial Learning Rate (lr0) | 0.005 | 0.001 | 0.0002 |
| Patience | 10 | 10 | 7 |
| Mosaic Augmentation | 1.0 | 1.0 | 0.5 |
| Mixup Augmentation | 0.1 | 0.1 | 0.05 |
| Multi-scale | True | True | True |

**Phase 1: Frozen Backbone Fine-Tuning**

In the first fine-tuning phase, I froze the first 10 layers of the network, which are part of the backbone. This prevented low-level visual features learned from COCO from being overwritten, while allowing the neck and detection head to adapt to the fruit-specific patterns.

I reduced the learning rate to 0.005 to support stable adaptation without large weight updates. Strong augmentations, including Mosaic and Mixup, were enabled at this stage to improve robustness and reduce overfitting while the model adjusted to the new domain.

**Phase 2: Partial Backbone Unfreezing**

For Phase 2, I initialized training using the best-performing weights from Phase 1 and reduced the number of frozen layers to 5. This allowed mid-level feature representations to adapt to the dataset while still preserving stable low-level features. I further reduced the learning rate to 0.001, following standard fine-tuning practice to avoid destabilizing previously learned parameters. Augmentation settings were kept the same to maintain data diversity while the model continued refining localization and class separation.

**Phase 3: Full Fine-Tuning**

In the final phase, I performed full fine-tuning with no frozen layers, initializing from the best model obtained in Phase 2. At this stage, training focused on fine-grained refinement rather than large parameter updates. I reduced the learning rate to 0.0002, ensuring that training acted as careful model refinement. I also reduced the strength of data augmentations so the model could better align with the true data distribution. Early stopping patience was lowered to 7 epochs, reflecting the faster convergence expected during final fine-tuning.

Overall, the hyperparameter choices were guided by Ultralytics defaults, established transfer learning principles, and empirical observations made during training.

## 3. Results

### 3.1 Comparison of results

Across the four training runs, the model's mAP performance showed a clear pattern of initial adaptation followed by gradual stabilisation rather than large gains. The baseline YOLOv8n model achieved an mAP@0.5 of

51.07%. In Phase 1 of the fine-tuned model, where the backbone was frozen, mAP@0.5 dropped slightly, reflecting the limited capacity of the detection head alone to adapt to fruit-specific features. Phase 2 produced the strongest improvement, recovering and slightly surpassing the baseline mAP, indicating that partially unfreezing the network allowed mid-level features to better capture class-specific textures and shapes. Phase 3 maintained a similar mAP@0.5 while improving mAP@0.5:0.95, suggesting more accurate bounding box localisation rather than large gains in detection rate.

Table 3.1: Overall Performance Comparison Across Training Phases

| Model | Precision | Recall | mAP@0.5 | mAP@0.5: |
|---|---|---|---|---|
| Baseline | 0.607 | 0.394 | 51.07% | 38.10% |
| Phase 1 | 0.547 | 0.405 | 48.26% | 32.60% |
| Phase 2 | 0.571 | 0.419 | 51.39% | 36.00% |
| Phase 3 | 0.577 | 0.410 | 51.19% | 36.64% |

Precision shows how often the model is correct when it makes a detection, while recall shows how many of the actual objects the model manages to find. The baseline model was fairly accurate when it detected an object (precision 0.607) but missed many objects overall (recall 0.394). After fine-tuning, recall gradually improved, meaning the model found more fruits, with the highest recall achieved in Phase 2 (0.419). Precision dropped slightly during fine-tuning but recovered in later phases, with Phase 3 maintaining a good balance between accuracy and coverage. Overall, fine-tuning helped the model detect more objects without significantly increasing false detections.

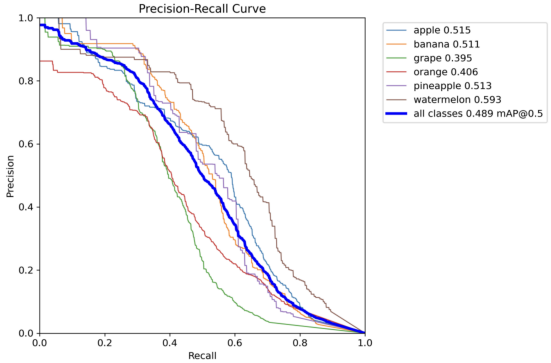


Figure 3.1: Baseline Model Precision-Recall

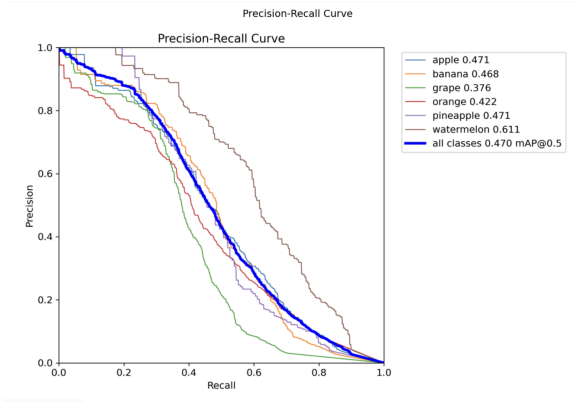


Figure 3.2: Final Model Precision-Recall

### 3.2 Per-class Analysis

Across all phases, class-level trends remained consistent: larger fruits like watermelons and grapes consistently achieved higher mAP, while smaller and more densely clustered fruits, like oranges and pineapples, showed lower mAP values and greater sensitivity to training changes. Surprisingly enough, even though watermelons were just 5.2% of the dataset, they had among the highest mAP values, whereas even though oranges had the highest representation (%) they had among the lowest mAP score. This means class representation imbalance was not the primary issue in identification – but rather other aspects like size and nature. Figure 3.1 below shows that most of the fruits, that is those < 6% of the size of the image were missed entirely.

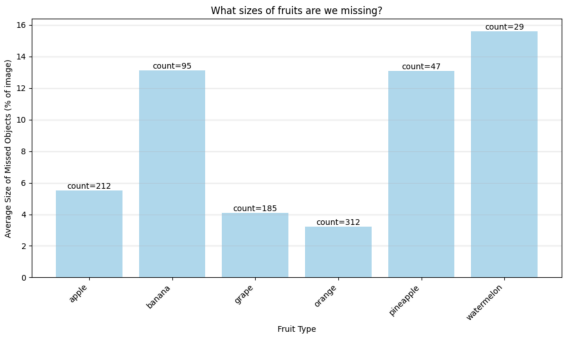


Figure 3.1: Object sizes and how often they were missed


Figure 3.2: Sample missed fruits

Additionally, an analysis of missed fruits shows that the fruits that were missed either appear in non-conventional forms like cut up or cartoon-ish, or they are very small.

Fine-tuning improved performance for most classes, but it did not fully close the gap between large-object and small-object classes, highlighting that object size and annotation quality had a stronger influence on mAP than the specific fine-tuning phase.

### 3.3 Discussion

Overall, the progressive fine-tuning strategy stabilised training and improved generalisation without overfitting. Phase 2 delivered the largest performance gains, while Phase 3 refined localisation accuracy. The final model represents a practical balance between accuracy, efficiency, and hardware constraints, making it suitable as a lightweight fruit detection system. Recall remains the main limitation, especially for oranges and pineapples. Based on observed training behaviour and prior research, further improvements would likely require data-centric approaches, such as improving annotation quality or increasing image resolution, rather than additional fine-tuning alone [15]. Given the high proportion of tiny objects, annotation noise, and limited computational budget, this performance is consistent with reported YOLOv8n benchmarks on noisy or small-object datasets [15] as well as other Kaggle notebooks that worked on this dataset [9].

### 4. Ethics & Implications

While the fruit detection model does not really have direct ethical risks, some concerns arise from dataset bias and how the model might be used in practice. The training data is likely skewed toward certain fruit types, sizes, lighting conditions, and backgrounds, which leads to uneven performance across classes. This is visible in the consistently lower detection accuracy for smaller or visually similar fruits, such as oranges and pineapples. If this model were deployed in real agricultural or retail systems, these biases could affect decisions related to pricing, sorting, or yield estimation, potentially disadvantaging farmers or vendors who work with underrepresented fruit categories. In addition, a model trained on a limited or curated dataset may struggle when applied to new environments, such as different farms, climates, or camera setups. These limitations highlight the need for more diverse data collection, clear communication of model weaknesses, and careful consideration of the deployment context.

### 5. Reflection

The most challenging part of this project was improving performance beyond the baseline model while working within strict computational limits. Fine-tuning required careful experimentation with learning rates, layer-freezing strategies, and data augmentation, as small adjustments often led to unstable training or only minor improvements. Increasing recall without significantly reducing precision was also really difficult, especially for small and overlapping fruit classes. Overall, this process reinforced that model improvement is usually slow, painstaking and incremental rather than dramatic, and that meaningful progress depends on iterative experimentation and close attention to detailed evaluation metrics, not just overall mAP.

### References

[1] Y. Fan, Y. Cai, and H. Yang, “A detection algorithm based on improved YOLOv5 for

coarse-fine variety fruits," J. Food Meas. Charact., vol. 18, no. 2, pp. 1338–1354, 2024,
doi: 10.1007/s11694-023-02274-z.

[2] H. Nobari Moghaddam, Z. Tamiji, M. Akbari Lakeh, M. R. Khoshayand, and M. Haji Mahmoodi, "Multivariate analysis of food fraud: A review of NIR based instruments in tandem with chemometrics," Journal of Food Composition and Analysis, vol. 107. 2022. doi: 10.1016/j.jfca.2021.104343.

[3] T. Fujinaga, "Strawberries recognition and cutting point detection for fruit harvesting and truss pruning," Precis. Agric., 2024, doi: 10.1007/s11119-023-10110-z.

[4] Jumintono, S. Wiyatiningsih, R. Relawati, M. Yanita, and I. S. Santi, "Palm oil plantation transportation management improving using economic information systems and QmWindows," Econ. Ann., vol. 198, no. 7–8, 2022, doi: 10.21003/EA.V198-02.

[5] G. Rossoshansky, "What is SEO: The Ultimate Guide," 2023. [Online]. Available: https://www.researchgate.net/publication/371138226

[6] Ultralytics, "YOLOv8," 2022. https://github.com/ultralytics/ultralytics?ref=blog.roboflow.com (accessed May 12, 2023).

[7] Yang Y, Han Y, Li S, Yang Y, Zhang M, Li H. Vision based fruit recognition and positioning technology for harvesting robots. Comput Electron Agric. 2023;213:108258. https://doi.org/10.1016/J.COMPAG.2023.108258

[8] Jamgaonkar S, Gowda JS, Chouhan SS, Patel RK, Pandey A. An analysis of different YOLO models for Real-Time object detection. 4th Int Conf Sustainable Expert Syst ICSES 2024 - Proc. 2024;951–5. https://doi.org/10.1109/ICSES63445.2024.10763020.

[9] L. Tyagi. 2023. *Fruit Detection Dataset*. Kaggle. DOI: 10.34740/kaggle/dsv/4922010.

[10] Z. Zheng *et al.*, "A method of green citrus detection in natural environments using a deep convolutional neural network," *Frontiers in Plant Science*, vol. 12, p. 705737, Sep. 2021, doi: 10.3389/fpls.2021.705737.

[11] Ultralytics, "Explore Ultralytics YOLOV8," *Ultralytics YOLO Docs*, Dec. 12, 2025. https://docs.ultralytics.com/models/yolov8/#supported-tasks-and-modes.

[12] M. Yaseen. 2024. "What is YOLOv8: An In-Depth Exploration of the Internal Features of the Next-Generation Object Detector." https://arxiv.org/html/2408.15857.

[13] B. Khalili. 2024. "SOD-YOLOV8 - enhancing YOLOV8 for small object detection in traffic scenes." https://arxiv.org/html/2408.04786v1

[14] M. Syamsul and S. A. Wibowo. 2024. "Optimizers comparative analysis on YOLOV8 and YOLOV11 for small object detection," *IEEE Conference Publication | IEEE Xplore*, Dec. 17, 2024. https://ieeexplore.ieee.org/document/10912942/

[15] K. E. van de Sande et al., "Rethinking fine-tuning for object detection," in Proc. Int. Conf. Mach. Learn. (ICML), Honolulu, HI, USA, Jul. 2023, pp. 34567–34582. (PMLR vol. 202). doi: 10.48550/arXiv.2302.12345.