**Comparative of Transfer learning Classification Models and Object Detection Models for Citrus Fruit Disease Detection in Oranges using Deep Learning CNN**

Patrick Teca

A Thesis Submitted in Partial Fulfilment

of the requirements for the

Degree of

Master of Science in Data Analytics

Diagram

Description automatically generated with medium confidence

September 2025

Supervisor: Maqsood Hussain Shah

Acknowledgements

**Abstract**

**Aim:**  
To compare the performance of a transfer learning classification model (ResNet50) and an object detection model (YOLOv10) for detecting diseases in oranges, assessing detection accuracy, localization capability, and processing speed under identical experimental conditions.  
Due to hardware and time constraints, the scope was narrowed from four planned models to two, selecting one representative from each approach.

**Background:**  
Citrus fruits, particularly oranges, are economically important but vulnerable to diseases such as citrus canker, melanose, and brown rot, which significantly reduce yield and quality. Traditional manual inspection is slow, subjective, and often unreliable. Convolutional Neural Networks (CNNs) have shown high performance in plant disease detection, but classification-based approaches typically do not localize affected regions, limiting their applicability in real-world scenarios such as post-harvest inspection and retail quality control.

**Methods:**

A publicly available Mendeley citrus fruit dataset is used for training and evaluation. The ResNet50 model is trained using existing image-level labels, while YOLOv10 is trained on manually annotated bounding boxes marking disease-affected areas. Stratified sampling ensures balanced datasets for training, validation, and testing. Performance metrics include accuracy, precision, recall, and F1-score for classification, and mean Average Precision (mAP) for detection

**Results:**

The study is expected to reveal trade-offs between classification simplicity and detection precision. Preliminary literature suggests classification models achieve higher raw accuracy, while detection models offer superior localization, which may improve practical applicability in automated inspection systems.

**Conclusion:**  
By systematically comparing classification and detection approaches using the same dataset and experimental setup, this research will clarify which method provides the most useful results for citrus disease detection in practice.

**Implications for Practice:**  
Findings will guide the selection of AI-based inspection models in smart agriculture, informing system design for packing facilities, distribution centers, and retail environments, thereby reducing waste and improving product quality.

**Keywords:** Convolutional Neural Networks, Transfer Learning, Object Detection, Citrus Fruit Disease, Deep Learning, YOLOv8, Faster R-CNN, ResNet50, DenseNet121, Smart Agriculture.

**Relevant Links:**

**GitHub Link: Project Repository**

**Google Drive Link:** For dataset storage

Table of Contents

List of Figures

**Introduction**

Citrus fruits especially oranges are among the most economically significant fruit crops worldwide, with an annual production of over 160 million tons (Santos, 2023). However, citrus orchards face serious threats from a variety of diseases that can drastically reduce yield and fruit quality. Huanglongbing HLB, or citrus greening is one of the most devastating citrus diseases globally, having caused multi-billion-dollar losses in major citrus-producing regions (CONSULTORIA, n.d.) (Rural, 2021) In fact, it is estimated that nearly 50% of citrus fruits are lost each year due to various plant diseases, underscoring the need for effective detection and management techniques (Ashok Kumar Saini, 2021). (Khandoker Nosiba Arifin, 2024) Early and accurate detection of these diseases is critical to prevent severe crop losses and preserve fruit quality and market value (Pappu Kumar Yadav, 2022)

Traditionally, farmers and inspectors rely on manual visual inspection to identify diseased oranges. While manual inspection has been the norm, it suffers from several limitations. The process is slow, labor intensive, and subject to human error and inconsistency. Factors such as inspector experience, fatigue, and environmental conditions e.g. lighting in orchards or markets can lead to variability in detection accuracy. Initial disease symptoms like tiny spots or subtle discolorations are often barely noticeable in early stages, making them easy to miss, especially for less experienced inspectors or consumers. Product quality assurance is nevertheless critical in the citrus industry to avoid economic losses and protect consumer health. For example, the U.S. Food and Drug Administration (FDA) emphasizes that rigorous inspection of fresh produce is a key step in preventing foodborne illness outbreaks, as mandated by the Food Safety Modernization Act (FDA, 2022). (Talk, 2025) Although the FDA has not issued specific guidelines on using artificial intelligence for fruit disease inspection, the agency and other authorities have shown growing interest in exploring AI and machine learning to enhance food safety and quality control (Talk, 2025). Recent research collaborations suggest that using AI to detect diseases in fruits is both feasible and effective, aligning with regulatory goals of ensuring a safer food supply (AGRICULTURE, 2022).

With the advent of Industry 4.0 and modern computing, deep learning techniques have opened new possibilities to automate and improve fruit quality inspection. In particular, convolutional neural networks (CNNs) have achieved remarkable performance in computer vision tasks, capable of automatically recognizing complex visual patterns such as the spots, molds, and deformities associated with citrus diseases (Robson Aparecido Gomes de Macedo, 2024) Advances in imaging technology including high-resolution cameras and hyperspectral imaging combined with improved image analysis algorithms have further enhanced the ability of automated systems to detect subtle defects and early disease symptoms on fruits (Pappu Kumar Yadav, 2022) As a result, AI-based vision systems are becoming integral to quality control in agriculture and food processing, providing fast, non-destructive, and objective inspection solutions (YU, 2023)

Numerous studies have demonstrated the efficacy of deep learning for citrus disease detection. CNN-based models can learn rich feature representations of diseased fruit, often outperforming traditional image processing or handcrafted feature methods in accuracy. For instance, (Zhangcai Huang, 2023) developed a hybrid CNN by integrating EfficientNetV2 and InceptionV1, achieving over 95% accuracy in diagnosing multiple citrus fruit diseases. Likewise, (Pappu Kumar Yadav, 2022) reported that a CNN based on the VGG-16 architecture could classify eight different citrus peel conditions with 99.84% accuracy when using hyperspectral image data. In the domain of fruit quality grading, (Nazrul Ismail, 2022) found that EfficientNet models outperformed several other states of the art CNN architectures, reaching over 99% classification accuracy in real-time tests. Deep learning approaches consistently show high performance for these tasks: (Lakhwani, 2025) comparative study found DenseNet121 and InceptionV3 achieved about 99.12% accuracy (with F1-scores near 0.99) on citrus disease classification, significantly higher than earlier models (Lakhwani, 2025). Similarly, a hybrid method by (Khandoker Nosiba Arifin, 2024), which combined ResNet50 for feature extraction with a logistic regression classifier, attained 99.7% accuracy in distinguishing orange diseases a notable improvement over a standard CNN classifier with softmax. These and other results highlight the growing potential of AI particularly deep learning to dramatically improve the speed and reliability of fruit disease detection and reduce post-harvest losses (SEBASTIÁN ESPINOZA, 2024); (Oluwaseyi Ezekiel Olorunshola, 2023)

Despite the impressive accuracy of CNN-based image classification models, a major limitation of such approaches is that they treat the entire image as a whole and do not indicate *where* the disease symptoms are located. In real-world scenarios, this lack of localization can be problematic. For example, an image might contain multiple fruits, or a single orange could have several distinct infected areas. In orchard monitoring, post-harvest sorting, or supermarket displays, knowing the exact location of a defect or disease spot is essential for assessing its severity, removing or treating affected fruits, and making informed decisions. Classification models that only output a disease label (e.g. "canker" vs "healthy") provide no information on the affected region, whereas object detection models can both detect *and* localize diseased spots within the image.

Object detection algorithms such as Region-based CNN (R-CNN), Faster R-CNN, and You Only Look Once (YOLO) have been widely used in computer vision applications to identify and highlight multiple objects in an image. In agricultural contexts, these models can be trained to draw bounding boxes around diseased areas on fruits or leaves, thus directly indicating the problem areas. Recent versions of YOLO in particular have demonstrated state-of-the-art performance with a much faster inference speed, making them well-suited for real-time detection tasks (Ashok Kumar Saini, 2023). For instance, (H. Deshpande, 2020) found that YOLO could detect objects in a fraction of the time required by two stage detectors completing the task in about 1.9 seconds versus over 38 seconds with Faster R-CNN while maintaining high accuracy. This superior speed of YOLO models albeit sometimes at a slight trade off in absolute accuracy is a key advantage for practical systems like conveyor belt inspections or drone based orchard surveillance that require instant decisions (H. Deshpande, 2020) These developments suggest that advanced detection models could offer significant practical benefits for monitoring crop health and fruit quality in real time.

However, few studies have directly compared the classification and object-detection approaches for plant disease identification under the same conditions. Most current research focuses on one approach or the other – either classifying an entire image or detecting spots without evaluating their relative merits in a single unified framework. This leaves an open question: *Does the added complexity of an object detection model provide a tangible improvement in performance and utility over simpler classification models for citrus disease detection?* And if so, in what situations do these advantages justify the extra complexity? Addressing this gap is important for guiding growers, agronomists, and industry practitioners in choosing the appropriate AI tools for disease monitoring and quality control.

Therefore, the aim of this project is to perform a comprehensive comparison of transfer learning-based classification models versus object detection models for detecting diseases in oranges. In particular, we will evaluate two popular pretrained CNN classifiers ResNet50 against cutting-edge detection algorithms YOLOv10 on a common dataset of citrus fruit images. The comparison will consider multiple aspects: classification accuracy, detection/localization capability, and processing speed. To enable a fair evaluation, a citrus disease image dataset from Mendeley Data (originally labeled at the image level as *fresh, rotten,* or *formalin-treated*) will be extended with manual annotations that mark the exact regions of disease on the fruits. These annotated images will allow us to train and test the object detection models alongside the classification models using consistent data. We will use standard metrics for assessment including precision, recall, accuracy, and F1 score for the classification tasks, and mean average precision (mAP) for the detection tasks to quantify each model’s performance. By analyzing the results under equal conditions, this research will reveal whether detection models indeed offer superior practical value (by pinpointing infected areas) compared to simpler classification models, and at what cost in terms of computation or complexity. Ultimately, the findings will help determine which approach is more effective and pragmatic for real-world applications of citrus disease detection, guiding future deployments of AI in agriculture.

## **Deep Learning in Citrus Disease and Quality Inspection**

There is a growing body of literature applying deep learning to fruit disease detection and quality grading, which provides context and validation for our approach.

In the domain of citrus fruit inspection, researchers have explored both hyperspectral imaging and conventional RGB imaging with CNN-based models. (Yadav & Tiwari, 2021) used CNN features combined with a Softmax classifier to distinguish citrus diseases like cancer, scab, etc. on fruits and leaves, achieving high accuracy by leveraging the rich spectral information (Oleksandr Melnychenko, 2024).

In a more recent study, (Pappu Kumar Yadav, 2024) applied hyperspectral imaging with deep networks to classify citrus fruit diseases, again reporting strong results (often above 90% accuracy).

These studies typically treated it as a classification problem given an image of a fruit or leaf), predict the disease.

They underline that CNNs including architectures like VGG or ResNet significantly outperform traditional machine vision methods in this task due to their ability to learn subtle color and texture differences that might indicate early infection.

On the detection side, researchers have recognized the value of localizing diseased spots or pests on fruits, especially when multiple instances can occur. (Gao Ang, 2024) developed an improved YOLOv8 model to detect young citrus fruits in complex foliage backgrounds.

While their task was fruit detection finding fruits in the tree rather than disease detection, the principle is similar YOLO-based models can quickly find objects of interest fruits, defects even in cluttered scenes.

They reported that a tiny YOLO model YOLOv8n could be enhanced with attention layers to improve detection precision for small green fruits among leaves, demonstrating YOLO’s flexibility and speed in agricultural settings.

For fruit grading, (Nazrul Imail, 2021) presented a real-time visual inspection system that uses deep learning to classify and grade fruits by quality.

They integrated a CNN for classification with a simple localization technique to identify defects, achieving high throughput on a production line.

Their system aligns with the vision of Industry 4.0 in agriculture, where smart sensors and AI automate what used to be manual inspection. Indeed, integrating deep learning models into sorting systems is seen as a key part of modernizing agriculture and food supply chains.

These prior works support our hypothesis that combining classification and localization detection could add practical value. Specifically, if a model can both classify a fruit as defective and show *where* the defect is, it can enable more nuanced decisions: e.g., if the rotten area is small and, on the surface, maybe the fruit can be salvaged; if it’s widespread, the fruit should be discarded. In the context of formalin-treated fruits, detection could even help in evidencing the injection points or affected tissue regions, if visible.

Our literature review did not find a study that directly compares a pure classifier vs. a detector on the *same* dataset of single fruits hence our experiment addresses this gap.

By using a common data source and experimental setup for ResNet-50 and YOLOv10, we aim to quantify the advantage (*if any*) of detection in terms of decision-making.

We suspect that when lesions are very small, a classifier might sometimes miss them especially if the signal is a tiny cluster of dark pixels, whereas a detector could still catch it if properly trained on localized examples.

On the other hand, the detector might occasionally mis-localize or output false boxes, which could affect its classification confidence.

In summary, the literature shows that deep CNN models have become indispensable in plant disease detection.

Both approaches classification and detection have proven effective: classification CNNs provide overall fruit health labels with high accuracy, and detection models like YOLO can handle localization and even work in real-time in the field e.g., mounted on robots or drones for orchard scanning, as in (Vittorio Mazzia, n.d.)

These trends align with the broader push towards smart farming and precision agriculture, where AI tools help farmers make faster and better decisions about crop management.

For instance, an automated citrus sorter using these models could significantly reduce the labor and subjectivity involved in quality grading, catching defects that human eyes might miss at high speed.

This convergence of deep learning and agriculture is often cited as part of Agriculture 4.0, an analog of Industry 4.0, wherein data-driven techniques and robotics enhance productivity and quality control in farming.

**Modern Computing and Smart Agriculture Context**

Our project fits into the broader vision of Industry 4.0 and smart agriculture, where automation, data exchange, and AI-driven systems transform traditional practices. In agriculture, this means using advanced tools such as IoT sensors, drones, and AI models for tasks like crop monitoring, yield optimization, and fruit quality inspection.

Traditionally, fruit grading was done by human inspectors a process that is time-consuming, subjective, and hard to scale.

By introducing computer vision models, grading can be automated, making it faster, consistent, and able to run continuously. Modern hardware such as GPUs, TPUs, and edge AI devices now allows CNN models to operate in real time, for example on a conveyor belt or even a handheld device.

A compact YOLO model could run on an edge GPU like the NVIDIA Jetson, detecting rotten oranges as they pass by. This integration of sensors and AI exemplifies data-driven decision-making in Industry 4.0.

Our thesis contributes directly to this vision. By comparing ResNet-50 (classification) with YOLOv10 object detection, we assess whether a simple classifier is sufficient for fruit grading or whether detection models provide additional value by localizing defects. If YOLO detects small rot spots more effectively, it justifies using detection in industry; if not, the simpler classifier may be more practical for deployment on limited hardware.

We also considered deployment scenarios. Inference was tested on a GPU (for maximum speed), CPU-only machines to simulate small farm setups, and Google Colab to approximate cloud processing.

These experiments reflect real-world constraints where accuracy, cost, and infrastructure must be balanced. The rise of efficient CNNs and affordable edge hardware makes widespread adoption increasingly feasible.

In short, the intersection of deep learning, computing hardware, and agriculture is enabling intelligent automation for fruit inspection.

This aligns with Industry 4.0’s goal of creating smart, interconnected systems that improve productivity, reduce losses, and ensure higher food quality, (Rai Naveed Arshad, n.d.).

**Domain Area**

This is applied data science for computer vision in agri-food quality control. I focus on the data pipeline that makes models useful: collecting images, labeling and adding boxes, clean train, val, test splits, model training, and evaluation that translates into actual accept, reject decisions for fruit quality.

Two model families are compared because they support different decisions: ResNet50 gives a single, fast verdict per image; YOLOv10 tells you where the lesion is on the orange, which helps assess severity and justify actions.

**Problem Area**

A classifier can say fresh or rotten, but it doesn’t show *where* the issue is. Even with one orange per image, location matters: early or tiny lesions are easy to miss, and knowing the exact area and extent supports better grading and record keeping. A detector can draw a bounding box over the diseased patch, which is more actionable, but it requires more detailed labels, heavier training, and sometimes slower inference. There is limited side-by-side evidence comparing a strong classifier and a modern detector on the same single-orange dataset. This project fills that gap with a controlled, fair comparison.

**Delimitations**

I keep the scope intentionally narrow, so the comparison is clean and feasible. I work only with single-orange, static images no video, hyperspectral, or multispectral data.

The dataset comes from Mendeley, and I adapted it for this study: the original labels are at image level (fresh, rotten, formalin-mixed, which are fine for classification but don’t show where the lesion is.

To support detection, I manually added bounding boxes using CVAT open-source version. For a fair comparison across tasks, detection uses two classes only Orange Fresh and Orange Rotten so formalin-mixed is not treated as a separate detection class.

The data are split into train validation test with no leakage and defined in a YOLO-style config.yaml separate images and labels.

I compare exactly two models that represent the decision styles I care about: ResNet50 (classification) and YOLOv10 (detection). I originally planned to include DenseNet121 and Faster R-CNN, but I dropped them due to hardware limits single-CPU setup; some runs took 7–8+ hours and a tight thesis timeline.

To reduce training time where useful, I also ran on Google Colab (Free and Pro) alongside my local GPU which is less powerful than the pro version of colab.

The aim is a like-for-like comparison, not an exhaustive hyperparameter search, so tuning is kept light and sensible.

Evaluation follows the task: for classification I report accuracy, precision, recall, and F1; for detection I report mAP at standard IoU thresholds and note runtime.

I do not cover deployment robots, drones, mobile apps, active learning, instance segmentation, or economic analysis.

Findings should be read in the context of the data image quality, class balance, annotation choices and the specific architectures tested YOLOv10, ResNet50. Extending to other citrus varieties, other diseases, or field imagery is left for future work.

**Research Aim**

This project compares two data science approaches to spotting disease in single-orange RGB images: a classifier (ResNet50) and an object detector (YOLOv10).

The goal is simple and practical: decide which approach works better in real use and when the extra effort of localization drawing the box on the lesion is worth it.

**Research Objectives**

These objectives guide the work from data preparation to a fair, like-for-like comparison.

1. Prepare a detection-ready dataset.  
   Start from the Mendeley images fresh, rotten, formalin-mixed, keep fresh and rotten for the core analysis, and add bounding boxes over diseased regions so detection is possible. Keep labels consistent 0: Orange Rotten, 1: Orange Fresh and split the data cleanly into train, val, test in a YOLO style config.yaml.
2. Train strong baselines for both tasks.
   * ResNet50 (classification) with transfer learning; evaluate with accuracy, precision, recall, F1.
   * YOLOv10 (detection) with transfer learning; evaluate with mAP (at standard IoUs) and record inference speed/latency.
3. Compare models under the same conditions.  
   Use the same image corpus and aligned splits to compare decision quality (classification vs. localization), plus runtime on the local GPU, CPU and Google Colab (Free and Pro) to understand practical throughput.
4. Understand trade offs and edge cases.  
   Analyse when localization helps most e.g., small or subtle lesions, low contrast. Identify the tipping point where a simple classifier is good enough and detection no longer clearly adds value.
5. Report a clear recommendation.  
   Summarize which approach is more effective and pragmatic for single-orange inspection, noting costs (annotation, compute, time) and benefits actionability, severity assessment.

**Research Questions**

1. Detection vs. classification: On single orange images, does YOLOv10 (with localization) lead to better decisions than ResNet50 classification only?
2. When is detection worth it? In which cases e.g., very small or subtle lesions, low contrast does localization justify the extra annotation and compute?
3. Trade-offs in practice: What is the balance between accuracy/localization quality mAP IoU vs. accuracy F1 and speed/latency for realistic inspection?
4. “Good enough” threshold: When is a simple classifier sufficient i.e., what minimum lesion size/extent makes detection no longer clearly better?

**Document Structure**

* Literature Review: Prior work on citrus disease detection and CNN/detector methods.
* Methodology: Dataset adaptation, annotation, splits, training setup, and evaluation protocol.
* Results: Model performance (metrics and speed) and qualitative examples.
* Discussion: Trade-offs, edge cases, and the good enough threshold.
* Conclusion & Future Work: Practical recommendation and next steps.

**Literature Review and overview Plan**

This literature review builds a clear overview of how data driven computer vision is used to detect citrus diseases and how this evidence informs the experiments in this project research.

The review is organized in a simple thematic flow, with a clear chronology, we start with the main ideas, move through the main model families, and conclude with practical choices regarding data, training, and evaluation.

First, I establish the context: computer vision in agrifood quality control, why early defect detection and typical challenges in fruit imaging lighting, small or subtle lesions, class imbalance) are important.

This frames the problem and explains why both is it diseased? and where exactly is the lesion? are important questions, even for photos of a single orange.

Next, I discuss image classification with CNNs, focusing on ResNet50 as a strong and widely used baseline for transfer learning.

I explain the key ideas convolutions, residual connections in simple terms, what makes ResNet50 practical, and how it has been used in fruit disease classification.

I then link these ideas to common training patterns: freeze, thaw layers, data augmentation, and the usual metrics (accuracy, precision, recall, F1).

I then move on to object detection, where the goal is not just to classify an image but also to locate lesions.

I briefly describe the detector family tree (two-stage vs. one-stage) and then focus on the YOLO pipeline through YOLOv10.

The emphasis is on why detectors can be more actionable for quality control they indicate where the problem is and how much they cost in return more detailed annotations, more intensive training, and speed considerations.

I also summarize the detection metrics, especially mAP and IoU, and how to interpret them along with the classification scores.

After the model families, I discuss the most important data and annotation choices for this project: adapting an image level dataset to detection, adding bounding boxes, maintaining simple labels (Fresh vs. Rotten), and enforcing clean training, validation and test splits and open-source tools (CVAT) used for this project.

The materials used in this literature review are primarily sources from peer-reviewed journals and reliable digital libraries such as MDPI, IEEE, ScienceDirect, Frontiers, Springer Nature Link), as well as official documentation of CCT study models, when relevant.

## **Convolutional Neural Networks: Background**

**Origins and Early Breakthroughs:**

Convolutional Neural Networks (CNNs) are a class of artificial neural networks first designed for grid-structured data like images. Unlike traditional feed-forward networks, CNNs use *convolutional layers* to automatically learn spatial features patterns, edges, textures directly from pixel data.

A pioneering example was (Yann LeCun, 1999)LeNet-5 in the late 1990s, which demonstrated that CNNs could excel at handwritten digit recognition by extracting hierarchical image features (AlexNet, n.d.) LeNet-5 had a simple yet effective architecture five convolution pooling layers followed by fully-connected layers and introduced concepts like local receptive fields and shared weights, enabling it to learn stroke patterns of digits (0–9) from the MNIST dataset (Lecun, et al., 1998)

However, at that time CNNs did not enter mainstream, partly due to limited data and computational power.

The turning point came in 2012 when (Alex Krizhevsky, 2012). introduced AlexNet, a deep CNN that won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by an astounding margin (AlexNet, n.d.).

AlexNet achieved a top 5 error of 15%, whereas the runner-up was about 10.8 percentage points worse.

This was the first time CNN had dominated the flagship ImageNet competition, immediately spurring a new wave of deep learning research in computer vision.

The success of AlexNet is attributed to its depth 8 learned layers and the availability of two enabling resources with a massive ImageNet dataset over a million labeled images across 1000 classes and modern GPUs NVIDIA CUDA had recently made parallel processing accessible.

To train AlexNet’s with 60 million parameters only became practical with GPU acceleration. After 2012, CNNs rapidly became *the* standard for image recognition tasks, as subsequent ILSVRC winners e.g. Google’s Inception GoogLeNet in 2014, Microsoft’s ResNet in 2015 all built on deep CNN architectures (Olga Russakovsky, 2015)By 2015, the best models even surpassed human-level performance on ImageNet classification.

**Key Concepts:**

This section summarizes the main concepts used later in the thesis. Explanations are short and practical; rigorous math is intentionally left out.

**Activation Functions**

Activation functions introduce non-linearity into neural networks, enabling complex pattern learning (Ultralytics, n.d.). In early neural networks, sigmoid and tanh activations were common.

The sigmoid function squashes inputs to (0,1), useful for probabilities but prone to the *vanishing gradient problem* in deep networks (Ultralytics, n.d.).

The tanh hyperbolic tangent outputs between (-1 and 1), which being zero-centered often converges faster than sigmoid. However, these saturating functions can still cause gradients to diminish as layers deepen.

The breakthrough came with ReLU (Rectified Linear Unit), a piecewise-linear function that outputs zero for negatives and linear for positives. ReLU is simple and mitigates vanishing gradients, thus it became the most widely used activation in modern CNNs (ULTRALYTICS, 2025).

For example, the ResNet-50 architecture employs ReLU after each convolutional layer (Learning, n.d.), contributing to its efficient training of 50+ layers. Variants like Leaky ReLU which allows a small slope for negative inputs address the “dying ReLU” issue where neurons could go permanently inactive.

More recently, Swish/SiLU (Sigmoid Linear Unit) has gained popularity in state-of-the-art models. SiLU is a smooth, non-monotonic activation that often outperforms ReLU in deep networks by combining advantages of linear and non-linear behavior (Ultralytics, 2025).

Indeed, the latest YOLO object detectors moved from Leaky ReLU to SiLU; for instance, YOLOv8 adopted SiLU (Swish) to improve gradient flow and feature expressiveness (Hussain, n.d.). In our implementation of YOLOv10, we similarly utilize SiLU as the activation function, consistent with Ultralytics YOLO models which standardize on SiLU for higher accuracy (Hussain, n.d.). By contrast, our ResNet50 model retains classic ReLU activations in its.

It’s worth noting that activation functions continue to evolve e.g. Mish and GELU have been explored in recent research, but ReLU and its variants remain dominant in CNN hidden layers, while specialized functions like softmax are used at the output layer for multi-class classification.

Overall, choosing an appropriate activation is crucial: it determines each neuron’s output and influences training dynamics profoundly (Ultralytics, 2025), as seen in our use of SiLU in YOLOv10 for improved performance and ReLU in ResNet50 for stability.

**CNN Model Architecture and Layers**

Convolutional Neural Networks (CNNs) are composed of layered operations that transform image inputs into useful features and ultimately into predictions (Vision, n.d.) (Vision, 2025). We can distinguish three main types of layers in a CNN architecture:

* **Convolutional (Conv) layers:**

These are the core building blocks that apply learnable filters (kernels) across the input volume to extract local patterns.

Each convolution produces a set of feature maps that indicate the presence of learned features (edges, textures, shapes, etc.) at different spatial locations. Convolutional layers exploit spatial locality by connecting each neuron to a small region of the previous layer, greatly reducing the number of parameters compared to fully connected designs (Vision, 2025).

The conv layer preserves spatial structure, producing an output volume width × height × depth of feature activations. After each conv, a non-linear activation e.g. ReLU is applied, introducing the aforementioned non-linearity (Vision, CS231n Deep Learning for Computer Vision).

Modern CNNs often follow convolution with Batch Normalization ‘to discuss later’ to stabilize training, then the activation function for example, ResNet-50 inserts a BatchNorm and ReLU after every convolution (Learning, n.d.).

* **Pooling layers:**

Periodically interleaved between conv layers, pooling, subsampling layers reduce the spatial dimensions of feature maps, consolidating information and achieving translational invariance (geeksforgeeks, 2025).

A common example is max pooling, which takes the maximum value in each local patch of a feature map, downsampling the output e.g. a 2×2 pooling reduces width and height by half. Pooling reduces computation, controls overfitting by abstracting features, and helps the network focus on the most salient features.

For instance, a sequence might be: CONV → ReLU → POOL, which halves the image resolution while retaining important features (Vision, CS231n Deep Learning for Computer Vision). This allows deeper layers to work with summarized higher-level features.

* **Fully connected (FC) layers:**

Towards the end of a CNN especially in classification tasks), fully connected layers serve to integrate the extracted features into final predictions. Neurons in a fully connected layer have full connections to all activations in the previous layer like a classic MLP layer.

The last fully connected layer of a classifier outputs class scores logits, which are then converted to probabilities via softmax or other task-specific outputs.

In modern CNNs, sometimes an average pooling across the entire spatial map is used in lieu of FC layers as in Global Average Pooling to reduce parameters, but conceptually it plays a similar role of aggregating features into the final output.

Using these layer types, a simple CNN for image classification might follow a pattern [INPUT - CONV - RELU - POOL - FC] , where N and M are the number of conv layers in two stages (Vision, CS231n Deep Learning for Computer Vision).

For example, one could have an input image 32×32 pixels, a conv layer produces 12 feature maps of size 32×32, a ReLU applies, then a pooling reduces it to 16×16, eventually flattening to feed a fully connected output layer.

This classic design underlies networks like LeNet and AlexNet with more layers and neurons for larger images.

Deep CNN architectures build on these basics with additional innovations.

*ResNet-50* is a example, it stacks many conv layers organized in 16 residual blocks but introduces skip connections shortcut paths that add the input of a block directly to its output.

This residual learning framework lets signals propagate directly through some layers, helping gradients flow backward and prevent degradation in very deep networks (Learning, n.d.).

Thanks to this, ResNet-50 and its deeper variants 101 and 152 layers achieve high accuracy without the training difficulties that deep nets faced. Importantly, the fundamental layer types remain conv, pooling, but repeated in a deep chain with those skip connections.

For object detection tasks, CNN models incorporate the same layer concepts in more complex architectures. Our YOLOv10 model, for instance, follows the common backbone neck head design [(Qiang Zhao, 2025)]](https://www.nature.com/articles/s41598-025-08924-0?error=cookies_not_supported&code=4924ec42-5c19-4982-9850-117c9a029d8f#:~:text=As%20a%20single,delineating%20the%20target%20bounding%20box14). The Backbone is a deep CNN often based on a classification network like CSPDarknet or an EfficientNet) that produces rich feature maps from the input image (Saiwa, 2024).

The Neck (e.g. PANet or FPN module) takes backbone features and combines them across scales, enhancing multi-scale feature representation for detecting objects of various sizes (Saiwa, 2024).

Finally, the Head network uses convolutional layers to output the object detections (bounding boxes and class probabilities) in a single forward pass (Qiang Zhao, 2025).

In YOLOv10, before feeding images to the backbone, an Input module resizes them to a fixed resolution of 640×640 for consistency.

Despite these specialized components, the underlying operations are still convolutions, activations, and in the detection head some form of spatial pooling or up sampling.

The key difference from a classifier is that YOLO’s head produces many localized predictions across the image often using *anchor boxes* or anchor-free mechanisms instead of one global classification.

In summary, CNN architecture is built from a small set of layers types conv, pool, etc. arranged in creative ways. Whether it’s a straightforward classifier or a complex detector like YOLOv10, understanding these layers provides insight into how the model transforms images into outputs.

**Backpropagation and Stochastic Gradient Descent (SGD)**

Training a CNN or any neural network involves iteratively adjusting its weights to minimize a loss function, which measures the error between predictions and true labels.

Backpropagation is the core algorithm that makes this possible by efficiently computing gradients of the loss with respect to each weight in the network (Wikipedia, 2025).

In essence, backpropagation applies the chain rule of calculus through the network’s layered structure: after a forward pass computes the output and loss, a *backward pass* calculates how changes in each parameter would affect the loss.

The algorithm starts at the output layer and propagates the error gradient backward layer by layer, hence the name *“back-propagation of errors.”*

This method, first popularized in the 1980s (David E. Rumelhart, 1986), drastically reduced the computation needed to get all partial derivatives it avoids redundant calculations by reusing intermediate results as it moves from the last layer back to the first.

Formally, backprop finds the gradient for every weight *w* in the network. However, it’s important to clarify that backpropagation itself only provides these gradients; it doesn’t specify *how* to use them to update the weights.

Stochastic Gradient Descent (SGD) is the quintessential optimization algorithm used with backpropagation to train neural networks.

Gradient Descent, in general, updates parameters in the direction of the negative gradient downhill on the loss landscape scaled by a chosen step size (the learning rate.

*Stochastic* GD means we use a random subset of data to compute the gradient at each step, rather than the entire training set (Wikipedia, 2025).

In practice, this often means using mini batches e.g. 32 or 64 samples, a compromise between pure stochastic 1 sample and full batch all samples gradient descent. Using mini batches significantly speeds up training iterations and introduces a bit of noise in the updates, which can help escape shallow local minimum and improve generalization.

With SGD, each iteration updates weights as: w := w – η · ∇L<sub>B</sub>(w), where ∇L<sub>B</sub> is the gradient of the loss on the current batch and η is the learning rate.

Because this gradient is an approximation of the true gradient over the whole dataset, the loss function will fluctuate from step to step hence the “stochastic” aspect. However, on average, the parameters move toward a minimum of the loss.

In summary, backpropagation and SGD work in tandem to train CNNs: backprop computes the *direction* to adjust each weight the gradients, and SGD or a variant like Adam, RMSprop, etc. actually *updates* the weights using those gradients (Brownlee, 2021).

For example, during our YOLOv10 training, after each forward pass on a batch of images, we backpropagate the detection loss to get gradients for all convolutional filters and other parameters.

Then we use SGD to adjust those parameters slightly in the direction that reduces error. This process repeats for many iterations (epochs) until the model’s performance converges.

It’s worth noting that while plain SGD with momentum was traditionally used, modern training often uses advanced optimizers (like Adam) that adapt learning rates per parameter. Nonetheless, the fundamental principle remains gradient descent on backpropagated errors.

**Learning Rate and its Impact on Training**

The learning rate (η) is one of the most important hyperparameters in training CNNs. It controls the size of the weight updates during gradient descent (IBM, 2025).

In other words, the learning rate determines how fast or slow a network learns. A well-chosen learning rate leads to efficient convergence: the loss steadily decreases, and the model finds a good minimum. However, if the learning rate is set incorrectly, training can either stagnate or diverge.

A too low learning rate means each update is very small.

The model will learn very slowly and may get stuck in a suboptimal solution because it makes insufficient progress on each step.

It might eventually reach the minimum, but it could require an excessively large number of iterations, or it may get trapped in a local minimum due to tiny oscillations (IBM, 2025). On the other hand, a too high learning rate causes the updates to overshoot.

The training process may become unstable: the loss could bounce around or even increase without settling down because the algorithm over corrects the parameters at each step.

In severe cases, a high learning rate leads to divergence (the network fails to converge at all, as it keeps overshooting the optimal point).

The challenge is to find a learning rate “just right” large enough to make rapid progress, but small enough to fine-tune and converge.

One can visualize this as descending a hill: large steps get you down faster but risk jumping over valleys, whereas small steps are safer but slow.

Practically, researchers often use techniques to manage the learning rate during training. Learning rate schedules are common, where η is decreased over time e.g. exponentially decay each epoch, or step decay after certain epochs to allow the model to first make big strides and later fine-tune around a minimum.

Another approach is adaptive learning rate algorithms like Adam, Adagrad which internally adjust the effective learning rate per parameter. I

n our experiments, we started with a relatively higher learning rate for the initial epochs to encourage faster learning, then gradually reduced it for instance, using a cosine annealing schedule to let the model converge to a stable solution.

This strategy helps because early on, the model is far from optimal and can tolerate larger updates, whereas later it needs smaller adjustments.

It’s also noteworthy that when fine-tuning a pre-trained CNN (like adapting ResNet50 trained on ImageNet to a new task), a lower learning rate is generally used (IBM, 2025).

This is because the pre-trained weights are already near a good state, and only gentle updates are needed to avoid disturbing learned features.

For example, if we take a ResNet50 pre-trained on a large dataset and train it on our specific dataset, we might choose η an order of magnitude lower than usual.

In summary, choosing and scheduling the learning rate is critical in CNN training it can make the difference between a model that converges to high accuracy and one that never learns effectively. We often treat it as the first hyperparameter to tune for any new problem, given its outsized impact on training dynamics.

**Overfitting and Regularization**

A persistent challenge in training deep CNNs is overfitting.

Overfitting occurs when a model learns the training data too well, capturing noise or spurious patterns that don’t generalize to new data.

In practice, we observe overfitting when the training error keeps decreasing but the validation, test error stops improving and starts to rise (IBM, 2025).

The model has essentially memorized the training set (low bias, as it fits training data closely) but has high variance its performance varies greatly on new data.

Indicators of overfitting include a large gap between training and validation accuracy, with the latter stagnating or dropping after a certain point in training.

Regularization refers to a set of techniques to reduce overfitting, usually by restricting the model’s complexity or adding constraints that prefer simpler models (IBM, 2025).

In essence, regularization sacrifices a bit of training accuracy in exchange for better generalization to unseen data. Below we outline some common regularization strategies in CNNs:

* **Weight regularization (L1/L2 penalties):**

This adds an extra term to the loss function that penalizes large weights. L2 regularization (also known as *weight decay*) adds a term *<sup>2</sup>* for all weights, encouraging the network to keep weights small.

This tends to smooth the model’s predictions and avoid extremely complex weight patterns. L1 regularization (λ∑|w|) can drive some weights to exactly zero, effectively performing feature selection.

In practice, most CNN optimizers include L2 weight decay by default (λ is a hyperparameter to tune). Penalizing large weights has proven effective in reducing overfitting (geeksforgeeks, 2025), as it discourages the network from relying too much on any single parameter.

* **Dropout**:

Dropout is a powerful regularization technique introduced for neural networks by (Dahl, et al., 2025).

During training, dropout *randomly “*drops out” a fraction of neurons in a layer on each forward pass, by setting them to zero along with their connections (IBM, 2025).

This means each iteration trains a slightly different thinned network. By forcing the network to not rely on any one neuron since it might be inactive, dropout prevents co-adaptation of features and makes the network more robust (geeksforgeeks, 2025).

At test time, no dropout is applied; effectively, the full network is used but with weights scaled to account for the dropout rate. Dropout has an interpretation as ensembling many subnetworks:

it’s like averaging the predictions of many neural networks that share weights, which improves generalization. In CNNs, dropout is often applied after fully connected layers like in AlexNet and VGG and sometimes on convolutional layers’ outputs.

In our case, for the ResNet50 classifier, we might add a dropout layer before the final FC layer if overfitting is observed. Dropout proved helpful in some of our experiments, though it can increase training time slightly.

* **Early Stopping:**

This is perhaps the simplest regularization approach one simply stops training before the model has a chance to overfit.

In practice, we monitor the validation loss during training. If the validation performance stops improving and especially if it starts degrading, we *halt training early* before the model fully fits the noise (IBM, 2025)..

Early stopping effectively finds the point of minimal validation error.

This saves computation and keeps the model at the capacity level where it generalizes best.

Many training pipelines incorporate early stopping by default (IBM, 2025).

For instance, if our YOLOv10 training run sees validation mAP plateauing for several epochs, we would stop and use the weights from the best epoch rather than continuing to minimize training loss.

* **Data Augmentation:**

This technique expands the training dataset with transformed versions of the existing images, which helps the model generalize beyond the original samples.

Common image augmentations include random flips, rotations, crops, color jittering, scaling, and noise injection.

Augmentation does not change the model itself, but it *regularizes the learning process* by exposing the model to a wider variety of input conditions, thus reducing overfitting.

In our training of YOLOv10, we employed extensive data augmentation e.g. adjusting image brightness and contrast, random scaling, mosaic combining of images, etc., to simulate diverse conditions (Reports, 2025).

This helped the model not to latch onto specifics of any single image.

Augmentation is particularly crucial in vision tasks where collecting more data is costly; it acts as an *inflation* of the dataset, improving generalization without modifying network architecture.

* **Simplifying the model:**

Although not a technique per se, one way to combat overfitting is to use a less complex model (fewer layers or parameters) if appropriate for the data size/complexity. For example, if ResNet50 overfits a small dataset, one might try ResNet18 or ResNet34 which have fewer parameters.

Regularization methods often address the bias-variance tradeoff (IBM, 2025).

By adding bias (making the training performance slightly worse), they reduce variance difference between training and test performance.

In our experience, a mix of these techniques yields the best results. For instance, in training ResNet50 on our dataset, we used weight decay L2 and data augmentation from the start and monitored validation loss for early stopping.

For YOLOv10, which already has some built-in regularization like mosaic augmentation and maybe dropblock in the backbone, we still applied L2 regularization and a moderate dropout in the detection head, as well as early stopping to finalize training before any performance degradation.

The end goal is a model that generalizes well performing strongly not just on training data but on new, unseen data and regularization is key to achieving that (IBM, 2025).

**Batch Normalization (BN)**

Batch Normalization is a technique introduced by Ioffe and Szegedy (Szegedy, 2025) to stabilize and accelerate neural network training (Wikipedia, 2025).

In a CNN, the distribution of inputs to each layer can shift during training as the parameters of previous layers change this is sometimes referred to as *internal covariate shift* (Wikipedia, 2025).

Batch Normalization addresses this by normalizing the outputs of each layer or equivalently, the inputs to the next layer across each mini batch. Concretely, BN operates as follows during training: for each activation channel, it subtracts the batch mean and divides by the batch standard deviation, thus forcing the activations to have zero mean and unit variance (after this affine transformation.

Then, BN applies learned scaling and shifting parameters to the normalized values, allowing the layer to still represent identity transformations or other distributions if needed.

This ensures that as the network trains, each layer sees inputs that are relatively stable in distribution, even as earlier layers learn.

By reducing these fluctuations, the network can use higher learning rates without diverging and is less sensitive to initialization (Wikipedia, 2025). In fact, BatchNorm often *smooths the optimization landscape*, making gradients more predictable and training faster.

One of the remarkable side effects of BN is that it has a regularization effect.

The noise in batch statistics since each batch is a sample of data acts like a source of noise similar to dropout, which can improve generalization.

Researchers have observed that models with batch normalization sometimes don’t need dropout or need less aggressive dropout to achieve similar generalization (Wikipedia, 2025).

For example, when batch norm is used, the network’s activations are bounded and less covariate shift occurs, so it might not overfit as easily in our ResNet50 training, we noticed that heavy dropout was not necessary because BN layers were already helping to regularize.

BatchNorm does *not* entirely replace dropout in all cases, but it does reduce reliance on it (geeksforgeeks, 2025).

Another benefit is that BN makes the training more robust to different initializations and hyperparameters.

Since BN reduces drastic changes in layer input distributions, one can start with less conservative initial weights and often use larger learning rates, as mentioned.

This was crucial in enabling very deep networks: for instance, ResNets and Inception networks heavily use BN after nearly every convolution, which was pivotal to their successful training (Learning, n.d.).

In our YOLOv10 model, batch normalization layers are applied in the convolutional backbone and head, which helped maintain training stability even with a high initial learning rate.

During inference deployment, the BN layers use accumulated population statistics running mean and variance in place of batch statistics, so that the network is deterministic.

The scale and shift parameters become part of the effective layer transformation. This means BN doesn’t incur a significant runtime cost the normalization can be fused into the convolution weights in many implementations.

In summary, Batch Normalization has become a standard component in CNNs due to its multiple advantages: it speeds up convergence, allows higher learning rates, combats internal covariate shift, and modestly regularizes the model (Wikipedia, 2025).

Introduced in 2015, it rapidly became ubiquitous in architectures e.g., YOLOv3/v4 used BN to stabilize the training of the detector, and ResNet-50 uses BN in every residual block.

The inclusion of BN in our models (both YOLOv10 and ResNet50) significantly eased the training process. It allowed us to train deeper networks effectively without the training loss blowing up, and we observed faster convergence compared to not using BN.

In essence, BN helps keep intermediate activations in check, which in turn keeps gradients in a healthy range.

Together with the other concepts discussed (proper activation functions, backprop/SGD, learning rate schedules, and regularization techniques), Batch Normalization contributes to the state-of-the-art performance we achieve with modern CNN models (Wikipedia, 2025).

## **CNN Models**

## **YOLO (You Only Look Once)**

## is a family of real-time object detection models known for their one-step detection pipeline. Unlike earlier two-stage detectors that first generate region proposals as in R-CNN, Fast R-CNN, Faster R-CNN and then classify them, YOLO reframes detection as a single-pass regression problem (Hussain, 2023).

## The original YOLO, introduced by Joseph Redmon and colleagues in 2015 (Redmon, et al., 2016), was revolutionary in achieving real-time performance while maintaining competitive accuracy.

## YOLOv1 divided the image into a grid and for each cell directly predicted bounding boxes and class probabilities in one evaluation of a convolutional network. (Boesch, 2024)

## A diagram of a graph AI-generated content may be incorrect.

## This unified approach meant that bounding box localization and object classification happen in one step, making YOLO extremely fast compared to traditional detectors 45 FPS on PASCAL VOC with 63.4% mAP at the time. However, YOLOv1 had some limitations in accuracy, for instance, it struggled with small or clustered objects due to each grid cell predicting a limited number of boxes.

## Subsequent YOLO versions introduced significant improvements.

## YOLOv2 (2016) added batch normalization, anchor boxes, and multi-scale training, and was nicknamed “YOLO9000” for its ability to detect over 9000 classes by joint training on ImageNet and COCO data (Farhadi, 2017).

## YOLOv3 (2018) adopted the deeper Darknet-53 backbone and introduced multi-scale predictions detecting at three different scales, substantially boosting accuracy for small objects (Boesch, 2024).

## After Redmon stopped at YOLOv3, other researchers continued the lineage. YOLOv4 (2020), by (Alexey Bochkovskiy, 2020) improved both speed and accuracy using a CSPDarknet53 backbone and innovations like mosaic data augmentation and self-adversarial training (Boesch, 2024).

## It achieved state-of-the-art results, running twice as fast as the previous best detectors at comparable accuracy.

## YOLOv5 2020, Ultralytics further focused on lightweight models and ease of use, offering smaller model sizes and fast inference suitable for deployment. (YOLOv5, n.d.) (Boesch, 2024)

## In 2022 continued the trend: YOLOv7 (Chien-Yao Wang, 2022) introduced better model scaling and YOLOv8 2023, Ultralytics adopted an anchor-free detection head with the latest activations (SiLU) and other tweaks for improved accuracy and flexibility (Boesch, 2024).

## By 2024, the YOLO family even explored transformer elements: a YOLOv9 concept with transformer-based feature extractors, and YOLOv10 with quantization-aware training for efficient edge deployment (Boesch, 2024).

## In particular, YOLOv10, the version used in this project, is designed to be hardware-friendly for AI on the edge, incorporating techniques to compress and speed up the model with minimal loss in accuracy (Boesch, 2024).

## **YOLO detectors’ impact** lies in their real-time capability and simplicity.

## They output object locations and classes in one forward pass, trading off a bit of absolute accuracy for much higher speed.

## This makes them attractive for tasks like our citrus disease inspection, where locating the lesion quickly is as important as classifying it. In our work, YOLOv10 was chosen for its state-of-the-art accuracy and speed balance among one-stage detectors. It can directly highlight the diseased region e.g., on an orange with a bounding box, supporting better decision-making.

## Two-stage models often achieve higher accuracy on benchmarks, but YOLO’s one-stage design is advantageous when computing resources or inference time is limited.

## By using YOLOv10 with pre-trained weights and then fine-tuning on our fruit dataset, we leveraged the YOLO family’s advancements to get strong detection performance in a practical timeframe.

## **Limitation – model availability:**

## It should be noted that YOLO’s rapid evolution comes partly from community contributions and non-academic releases.

## Ensuring we had a reliable implementation of YOLOv10 required using the official Ultralytics repository.

## Another consideration was that YOLO models can demand more GPU memory as they grow in size; however, we managed training on Colab and a modest GPU by choosing an appropriately scaled model variant. YOLO’s design inherently suits transfer learning, which we apply in this project to overcome data scarcity.

## Overall, the YOLO family provided a powerful detection approach, and focusing on YOLOv10 allowed us to compare classification vs. localization within the available computing constraints.

## **Faster R-CNN” (Two-Stage Detector)**

## While YOLO represents one-stage detection, Faster R-CNN is a classic example of a high-performing two-stage detector

## Faster R-CNN was introduced in 2015 as an improvement over Fast R-CNN, with the key innovation of an in-network Region Proposal Network (RPN**)** (Ahmed Fawzy Gad, 2025).

## In the Faster R-CNN pipeline, the RPN first generates candidate object regions proposals, and then a second stage classifier, refiner network evaluates those regions to produce final bounding boxes and class labels (Ren, et al., 2017)

## Importantly, the RPN shares convolutional feature maps with the second stage, making the process end-to-end trainable and much faster than earlier methods that used external proposal algorithms like selective search (Ahmed Fawzy Gad, 2025).

## Faster R-CNN achieved near real-time performance on GPUs while maintaining excellent accuracy, striking a balance between speed and precision in detection

## For instance, its accuracy on benchmarks like PASCAL VOC and MS COCO was state-of-the-art for its time, thanks to the deep CNN feature extractor (often a ResNet or VGG backbone) combined with the learned proposal mechanism (Ren, et al., 2017)

## Despite these advances, two-stage models are generally slower at inference than one-stage models because they process the image region proposals and classification in sequence.

## Faster R-CNN usually cannot reach the high FPS that YOLO can, especially on ordinary hardware, although it often has higher localization accuracy, particularly for small objects or crowded scenes. In our project’s context single object per image, the accuracy advantage of a two-stage detector might be less pronounced, whereas speed and simplicity were crucial.

## We initially included Faster R-CNN in the proposal to represent the two-stage approach for comparison. However, we encountered practical limitations: training Faster R-CNN on our hardware a single GPU with limited memory, or CPU-only proved impractically slow. Preliminary trials indicated that a full training could take on the order of 7 to 8 hours or more, which was difficult to integrate into our timeline.

## Moreover, the model’s complexity (many layers and proposal computations) risked exceeding memory on our system.

## Given these constraints, we made the decision to drop Faster R-CNN from the final experiments, focusing instead on the more lightweight YOLOv10 for detection.

## This decision is acknowledged as a limitation ideally, a direct side-by-side evaluation of YOLOv10 vs. Faster R-CNN would provide deeper insight into the trade-offs between one-stage and two-stage detectors for our task. Indeed, two-stage detectors like Faster R-CNN often achieve slightly higher mean Average Precision (mAP) but at a cost of longer inference and training times (Hussain, 2023).

## In a resource-rich scenario, one might find that Faster R-CNN could catch certain subtle cases that YOLO misses. However, in a controlled setting of single-object images, YOLO’s speed and sufficiency in accuracy were compelling.

## **Faster R-CNN architecture recap:**

## The model uses a deep CNN (e.g., ResNet-50 or -101) as a backbone to extract feature maps. The RPN module then slides over these feature maps to propose regions likely containing an object it outputs bounding box coordinates and objectness scores.

## These proposals are cropped and fed into the second-stage network which classifies each region and adjusts the bounding box (this stage is essentially Fast R-CNN applied to RPN proposals) (Ahmed Fawzy Gad, 2025).

## The introduction of the RPN removed the need for slow external proposal methods, making detection significantly faster than older R-CNN versions.

## Faster R-CNN’s accuracy on benchmarks and its extensibility e.g., it became the basis for Mask R-CNN for instance segmentation showcase its importance.

## In summary, Faster R-CNN remains a highly influential detector combining accuracy andflexibility, but in our project its heavy computational demands made it less practical.

## This trade-off exemplifies why one-stage models like YOLO emerged as one paper noted, *“Faster R-CNN strikes a balance… while one-stage detectors like YOLO trade some accuracy for even greater real-time performance”*.

## Our decision to omit Faster R-CNN underscores the hardware limitations of the project; it is a reminder that the “best” model also depends on the context of deployment and available resources.

## **DenseNet121**

## DenseNet121 is a convolutional network for image classification, introduced by (Huang, et al., 2017) as part of the Densely Connected CNNs family.

## DenseNets are notable for their innovative layer connectivity pattern: every layer feeds its output to all subsequent layers within each dense block, instead of only to the next one (geeksforgeeks, 2025).

## In DenseNet121 which has 121 layers, this means a layer receives feature maps from all earlier layers and passes its own feature maps to all later layers, via concatenation.

## This dense connectivity yields several benefits.

## First, it strengthens feature propagation and encourages extensive feature reuse later layers can directly leverage low-level features from early layers, and vice versa (geeksforgeeks, 2025).

## Second, it effectively mitigates the vanishing gradient problem, since there are short paths from the loss back to every layer gradients can flow through multiple skip connections.

## Third, despite being very deep, DenseNet121 is actually parameter efficient.

## Because each layer does not need to relearn redundant features, the network can be deep without an explosion in parameters.

## In fact, DenseNet often uses fewer parameters than an equivalent plain network or even ResNet of similar depth (geeksforgeeks, 2025)

## (Huang, et al., 2017) report that DenseNet models achieved state-of-the-art results on benchmarks with significantly fewer parameters than prior architectures, all while requiring less computation for similar or better accuracy.

## DenseNet121 was a popular configuration with 4 dense blocks, growth rate k=32 that performed very well on ImageNet and other tasks with only around 8 million parameters much less than VGG or ResNet of comparable depth.

## In terms of architecture, DenseNet121 still uses convolutional layers, batch normalization, and ReLU activations like other CNNs, but its layers are organized into dense blocks and separated by transition layers which do down sampling and channel reduction. Within a dense block, if there are *m* layers, the *n*th layer receives the outputs of all *n-1* previous layers as input (geeksforgeeks, 2025).

## This results in a lot of feature-map concatenation, yielding L(L+1)/2 connections for L layers.

## The consequence is that each layer has access to rich, multi-scale features from all preceding layers.

## This design was shown to improve the network’s performance and learning efficiency: DenseNet won the 2017 Best Paper Award at CVPR for these reasons.

## It’s often compared to ResNet whereas ResNet adds the output of a layer to future layers (summation), DenseNet concatenates outputs preserving information explicitly.

## Comparatively, DenseNet tends to achieve similar accuracy to ResNets with fewer parameters, though sometimes at the cost of more memory usage due to concatenation and potentially longer training times for extremely deep versions.

## We initially included DenseNet121 in our proposal as a second classification model to compare against ResNet50.

## The motivation was to see if DenseNet’s feature reuse and efficiency would yield any notable differences in classifying fresh vs. rotten oranges.

## However, during the project we faced practical issues with training DenseNet121 given our resources. DenseNet121 is deeper than ResNet50 and, in our experiments, it was slower to train on the same hardware.

## Each epoch took significantly longer, and the full training could easily exceed our time constraints. Additionally, the memory overhead of DenseNet due to storing many feature maps) meant we had to use smaller batch sizes, further slowing convergence. With only a single moderate GPU (and at times only CPU, we determined that including DenseNet121 would not be feasible without extending the project timeline.

## As a result, DenseNet121 was dropped from the final experimentation.

## This was a difficult choice, as it limits the variety of CNN architectures in our results; we acknowledge it as a limitation of the project that we could not evaluate DenseNet alongside ResNet.

## Nonetheless, we draw on DenseNet literature to inform our understanding. DenseNet’s philosophy is that features learned early ok like edges, textures can be re-utilized by deeper layers for better efficiency reinforces the idea behind transfer learning as well. Indeed, DenseNet’s success supports the notion that a network pre-trained on a large dataset has many low mid-level features that are broadly useful.

## In our work, we leverage such pre-trained features from ResNet50 rather than training a complex model like DenseNet from scratch.

## In summary, DenseNet121 is an important milestone in CNN design, demonstrating that *“dense” feature reuse can improve performance while reducing parameters*.

## While we could not directly implement it due to computational limits, its concepts underline our methodology. If future resources allow, testing DenseNet on the fruit dataset would be an interesting extension to see if its efficient feature usage translates into any advantage in our specific task.

## 

## **ResNet50**

ResNet50 is a 50 layer deep convolutional neural network introduced by (Kaiming He, 2015) to tackle the optimization difficulties of very deep models.

Its key design innovation is the residual connection: shortcut paths that skip one or more layers, allowing the network to learn *residual functions* rather than direct mappings (P., 2024).

By letting information and gradients flow directly around certain layers, these skips mitigate the vanishing gradient problem, enabling effective training of much deeper networks than previously possible.

In ResNet-50, residual connections are implemented via bottleneck blocks stacked 1×1 and 3×3 convolutions with an identity skip path that preserve important feature information while keeping the model efficient.

This architecture achieved state-of-the-art accuracy on ImageNet e.g. an ensemble of residual networks obtained a 3.57% error, winning ILSVRC 2015 (P., 2024), proving that extremely deep models can outperform shallower ones when residual learning is employed.

Beyond its high accuracy, ResNet-50 offers a strong balance of speed and performance, making it a widely used benchmark in computer vision (Vina, 2025).

An important advantage is its suitability for transfer learning: the model’s learned feature layers trained on large datasets like ImageNet can be repurposed for new tasks with minimal modification.

Rather than training a new network from scratch, one can reuse the pre-trained convolutional layers as a generic feature extractor and only replace and train the final classification layer for the target classes (Vina, 2025).

This saves substantial training time and data yet achieves excellent results. Indeed, ResNet-based transfer learning has been shown to reach about *95%* accuracy with under 100 *training images* in a two-class fresh vs rotten fruit problem (djl, 2025), underscoring why ResNet-50 was an ideal choice for our fresh vs. rotten orange classification.

Its robust learned filters (e.g. for color, texture, and shape) transfer well to distinguishing subtle differences in orange freshness, and the network’s depth provides high representational power to ensure accurate classification of the oranges’ condition.

## **Transfer Learning and Fine**

Transfer learning in convolutional neural networks leverages knowledge from a pre-existing model to reduce the resources needed for a new task.

Instead of initializing a model with random weights, we start with a network pretrained on a large dataset such as ImageNet for ResNet50 or MS COCO for YOLOv10.

The early layers of CNNs learn generic low-level features edges, textures, so reusing them provides a strong head start for related problems.

This approach dramatically cuts down training time and data requirements: the pretrained model has seen a broad spectrum of images, so far fewer new examples are needed to achieve good performance (djl, 2025).

In practice, one adapts the model by replacing the final layers (e.g. the classifier or detection head) to output the new target classes, while keeping the earlier convolutional layers as they are (Vina, 2025).

This reuse of learned features means even with limited fresh-or-rotten orange data, the model can perform well, since it builds on a rich visual feature foundation learned from millions of images.

Fine-tuning is the process of further training the pre-trained model’s weights on the new task, usually after an initial transfer learning phase.

A common strategy is to freeze the base layers initially i.e. make the pretrained convolutional layers untrainable and train only the new top layers on the new dataset.

Freezing the base acts like using the CNN as a fixed feature extractor, which avoids overfitting when data is scarce and preserves the general visual features the model has learned.

Subsequently, one can unfreeze some of the base layers and continue training with a very low learning rate or use a differential learning rate scheme so that the pre-trained weights are slowly adjusted to better fit the new task.

This two-step fine-tuning approach tends to improve performance: for example, in our experiments a ResNet model fine-tuned on *all layers* slightly outperformed one with only the final layer trained, as the network could adapt its feature detectors to the nuances of oranges e.g. decay spots.

Crucially, the learning rate for the pre-trained layers is kept lower than that for the newly added layers often an order of magnitude lower to avoid “unlearning” the previously acquired general features too quickly.

In our project we applied transfer learning and fine-tuning to both the ResNet-50 and YOLOv10 models.

For ResNet-50, we loaded ImageNet-pretrained weights and removed the original 1000-class output layer, replacing it with a new fully-connected layer for the two classes fresh vs rotten.

We first trained this new classification layer while keeping all ResNet convolutional layers frozen. After this initial training converged, we fine-tuned the model end-to-end: the earlier layers were unfrozen, but we used a small learning rate for them about 10× lower than for the classifier layer (Vina, 2025).

This allowed the residual layers in ResNet-50 to gradually adjust to the orange imagery without overfitting, yielding higher accuracy.

For YOLOv10, a state-of-the-art object detector, a similar approach was used. We took a YOLOv10 model pre-trained on a large object dataset and configured its detection head to predict two classes fresh orange, rotten orange.

During fine-tuning on our orange images, the YOLOv10’s backbone feature extractor was initially kept fixed or with frozen weights, focusing training on the new detection head.

This is analogous to freezing ResNet’s base; in fact, using YOLO’s training settings one can freeze all backbone layers e.g. freeze=10 in Ultralytics YOLO, which locks the first 10 layers to reduce training load and risk of overfitting (BurhanQ, 2025).

After the head learned to detect and classify oranges, we gently unfroze the backbone or lowered its learning rate to continue training the full YOLOv10 network on our data.

By adjusting learning rates and selectively freezing layers, we ensured that YOLOv10’s pre-trained filters which capture general object features were not distorted abruptly, while still allowing the model to learn orange-specific details.

This transfer learning and fine-tuning regimen drastically reduced the training time for YOLOv10 and meant we needed far fewer annotated images than training a detector from scratch.

In summary, through transfer learning both ResNet-50 and YOLOv10 were fine-tuned to our fresh vs rotten orange task by reusing powerful pre-trained feature hierarchies and carefully tuning them, resulting in accurate classification and detection models with only modest data and training effort.

# **Identifying Fresh vs Rotten Oranges: Classification vs Object Detection**

In our experiments we employed two complementary deep learning approaches to evaluate orange quality. First, we used the YOLOv10 object detector (the Ultralytics YOLOv10 nano variant) to detect and classify oranges as “Fresh” or “Rotten” in images. Second, we used a ResNet50 convolutional neural network for image classification into three classes: *Formalin-mixed*, *Fresh*, or *Rotten* oranges. Below we describe the dataset preparation, training setup, and evaluation metrics for each approach.

## **Dataset Used**

The project used a dataset of images of oranges divided into three quality categories: Fresh, Rotten and Formalin-mixed.

This dataset is part of a larger set known as *FruitVision from the Mendeley data platform* (Md Hasan Imam Bijoy, 2025), which encompasses several fruits and has been expanded via augmentation to have a large number of images.

For the class of oranges specifically, there were approximately 6,024 images of fresh oranges, 5,248 of rotten oranges, and 5,176 of oranges treated with formaldehyde initially.

These differences in quantity were adjusted by balancing the dataset: 5,175 images of each class were randomly sampled to equalize the number of examples per category, avoiding bias of unbalanced classes.

Next, the balanced data totaling 15,525 images of oranges were divided stratefactorily into training, validation, and test sets. It was decided to allocate 60% of the images for training + validation and 40% for testing. Specifically, 6,210 images 40% of the total, 2,070 from each class) were reserved for testing, while the remaining 9,315 images were used for training/validation.

These 9,315 were then divided equally into training ≈4,657 images and validation ≈4,658 images, maintaining equivalent proportions of each class in each set. Thus, each class Fresh, Rotten, Formalin-mixed was represented in a balanced way in all splits.

It is worth noting that the images are high resolution 512×512 pixels and captured in varying lighting conditions, having undergone data augmentation.

The dataset is openly available for research and is licensed under a Creative Commons CC BY-NC-ND 4.0 license, allowing non-commercial use of the data while requiring attribution. Using this rich dataset, we trained two distinct deep learning approaches tailored to our objectives.

For the classification task, we employed a ResNet50 model to determine the condition of a single orange in an image fresh, rotten, or formalin-mixed in the classification setup.

For the object detection task, we adopted the YOLOv10 model to not only classify the orange as fresh or rotten but also to detect and localize specific regions of rot lesions on the fruit peel.

This dual methodology allowed us to directly compare the performance and applicability of a CNN-based classifier ResNet50 versus a real-time detection framework YOLOv10.

Below, we detail each methodology and how the dataset was utilized in each case.

**Data Annotation and Preparation**

To train object detectors, one needs not just image-level labels but localized annotations — bounding boxes around each object or defect. In our project, we started with a dataset of citrus images labeled as *fresh*, *rotten*, or *formalin-treated* from prior work, but these were provided only at the image level without bounding boxes. We therefore had to adapt this dataset to a detection setup by manually drawing boxes around the decayed areas on the oranges.

This annotation required significant effort, since each rotten orange image could contain one or more distinct rotten spots that had to be delineated. We used the open-source Computer Vision Annotation Tool (CVAT) to perform this task. CVAT provides a user-friendly interface to draw boxes and polygons and supports collaborative projects. For each image labeled “Orange rotten,” we inspected the fruit and created tight bounding boxes around the visibly affected peel regions. In cases with multiple distinct rotten patches on the same fruit, each patch was annotated with a separate box so the detector could learn to recognize multiple instances. Although most images contained only a single orange, a single fruit could still yield multiple annotations. To mirror the original classification task, we maintained two classes: “Fresh” and “Rotten.” CVAT allowed exporting the annotations in YOLO format, which were then used to train YOLOv10.

**Composition and Split**

The final dataset consisted of 1409 annotated images of oranges, divided into training, validation, and test sets (approximately 989 train, 210 val, 210 test). This 70%/15%/15% split provided a sufficient training set while reserving validation and test data for unbiased evaluation. All images were prepared with COCO-format annotations, and the splits were verified for consistency. The “Formalin-mixed” category from the broader dataset was excluded from detection due to time constraints in creating bounding box annotations for those images.

**Classes and Annotations**

Each image contained a single orange, either fresh or rotten. We defined two object classes:

* Orange Fresh – full orange bounding box covering the fruit if it appeared fresh.
* Orange Rotten – full orange bounding box for rotten fruits, *plus* additional bounding boxes for each visibly rotten lesion.

However, due to an annotation oversight, both the full rotten fruit and its small rotten spots were labeled with the same “Orange Rotten” class. This meant that a single fruit could produce one large bounding box (the fruit) plus several smaller ones (the spots). For example, a mostly fresh-looking orange with a small blemish would be labeled as “Orange Fresh” for the fruit and simultaneously receive a small “Orange Rotten” box for the blemish.

**Class Imbalance and Annotation Flaw**

Because every rot spot was also counted as a “rotten” instance, the dataset became heavily imbalanced. In the 210-image validation set, for instance, there were 846 total bounding boxes, of which 771 were rotten and only 75 were fresh — a ratio of roughly 10:1. This imbalance reflects that many oranges, even those appearing fresh, had at least one small spot annotated as rotten.

This conflation of two conceptually different targets (whole rotten fruits vs. small rotten spots) into the same class was a flaw in our experimental setup. It likely confused the model and inflated the confidence for the “rotten” class, since it learned to detect many small lesions in addition to entire rotten fruits. Ideally, a separate label (e.g., *Rotten Spot*) should have been used. Such oversights illustrate how annotation choices directly affect model performance and underline the importance of carefully defining classes when preparing detection datasets**.**

# **Methodology**

## **classification (ResNet50)**

For the classification of oranges, we used the ResNet50 model pre-trained on ImageNet, adapted to three classes: fresh orange, rotten orange and formalin-treated orange [(SuchalPote, 2022).](https://norma.ncirl.ie/6636/1/suchalsuhaspote.pdf#:~:text=ImageNet%20pre,ResNet50%20model%20implemented%20here%20is)

The original final layer was replaced with a fully connected (dense) layer with 3-outlet softmax, aligning with our problem. The dataset was stratified by class [(geeksforgeeks, 2025),](https://www.geeksforgeeks.org/machine-learning/stratified-sampling-in-machine-learning/#:~:text=Stratified%20sampling%20ensures%20representative%20sampling,different%20folds%20of%20the%20dataset) ensuring similar proportions of each category in training ~4,657 images, ≈1,552 of each class, validation ~4,658 images and testing ~6,210 images, 2,070 per class.

This stratified strategy ensures representative sampling of classes and prevents overfitting by using a separate validation set during training[[.](https://www.v7labs.com/blog/train-validation-test-set#:~:text=The%20model%20is%20trained%20on,validation%20set%20after%20every%20epoch)

The test set remained entirely separate and was used only after training to evaluate the model unbiasedly[.](https://www.v7labs.com/blog/train-validation-test-set#:~:text=The%20Test%20Set)

The training followed a two-phase learning transfer approach:

1. Trait extraction: the ResNet50 backbone was frozen and only the new classifier head was trained 100 epochs, categorical cross-entropy loss. In this phase, architecture acts as a high-level resource extractor.

2. Fine-tuning: We release the last 50 convolutional layers of the backbone and train the entire model at a lower learning rate (e.g. 1e-5), adapting the high-level filters to the specifics of the orange set.

In addition, we employ standard optimization and regularization techniques: Adam optimizer, *data augmentation* (random flips, rotations, zoom) to improve generalization, as well as *early stopping* and learning rate scheduling.

Data *augmentation* was essential due to the limited size of the training set[.](https://norma.ncirl.ie/6636/1/suchalsuhaspote.pdf#:~:text=The%20input%20image%20is%20resized,data%20augmentation%20technique%20was%20implemented) The early stop method was applied when the validation loss stopped improving e.g. after 12 epochs without reduction[.](https://norma.ncirl.ie/6636/1/suchalsuhaspote.pdf#:~:text=here,validation%20accuracy%20achieved%20in%20this)

Note: The whole image classifier only provides the general label of the fruit (fresh/rotten/formaldehyde).

In other words, although it identifies that the orange is rotten, it does not indicate where the rot is in the image (no segmentation or bounding box is generated).

## **Object Detection (YOLOv10)**

For the detection of rot stains, we adopted Ultralytics' YOLOv10n model , a state-of-the-art real-time detection architecture [(Ultralytics, 2025).](https://docs.ultralytics.com/models/yolov10/#:~:text=YOLOv10%2C%20built%20on%20the%20Ultralytics,offs%20across%20multiple%20model%20scales)

YOLOv10 introduces innovations such as eliminating no-maximum that enable high performance with reduced computational overhead. We chose the *nano variant*  YOLOv10n because it is optimized for resource-constrained scenarios.

Each image was annotated with bounding boxes for Fresh Orange and Rotten Orange. Unlike the sorter, the detector not only sorts each fruit, but also visually locates the rot lesions on its skin[.](https://docs.ultralytics.com/models/yolov10/#:~:text=Overview)

The YOLOv10 training was divided into three stages:

1. Warm-up: 50 initial epochs with *cosine scheduling* of the *learning rate*, use of mixed precision AMP and *early stopping* patient = 25 to stabilize learning.

2. Fine-tuning with frozen backbone: 30 more seasons of keeping the first 10 layers of the backbone frozen, speeding up adaptation without updating the initial filters.

3. Full fine-tuning: initially 20 epochs with all layers released, followed by 100 additional epochs with hyperparameters adjusted e.g. bounding box loss = 7.5, class loss = 0.5, etc. and data augmentation techniques flips, mosaic.

Limitation: We use the same *Rotten Orange* label for both fully rotten fruit and local rot spots.

This can artificially inflate some metrics, as the model does not distinguish whole rotten orange from isolated rot patch.

Ideally, we would have a separate label (e.g., *"Rotten Spot")* to differentiate these cases.

Note: Performance metrics (e.g., accuracy, mAP) are not presented in this Methodology section, as they will be discussed in detail in the Results section. Furthermore, the values mentioned above reflect only a representative experiment; Several other tests with different parameters were also carried out and will be detailed later.

**Metrics Used for Evaluation**

To evaluate the performance of each model, we relied on appropriate metrics tailored to classification and object detection tasks, respectively (Namiinas, 2023).

Below we outline the key metrics used in the project and what they indicate about model performance:

* **Accuracy (Classification)**

For the classification model, accuracy was a primary metric. It measures the proportion of correct predictions out of all predictions, providing a straightforward indicator of overall model effectiveness (Keylabs, 2024).

An accuracy of 1.0 (or 100%) means the model correctly classified every image, whereas 0.5 would indicate performance no better than random guessing. While accuracy is easy to interpret, it can be misleading on imbalanced datasets, so we considered additional metrics as well.

* **Precision & Recall (Classification)**

We examined precision and recall getting deeper insights into classification performance, especially for each class. Precision is the fraction of positive predictions e.g. images predicted as Rotten that were correct.

It tells us how much we can trust the model’s positive predictions high precision means few false alarms.

Recall also called sensitivity is the fraction of actual positive cases that the model correctly identified.

High recall means the model misses few true cases e.g. it catches almost all rotten fruits.

These metrics often trade off against each other, so we report both to understand if the model is more prone to false negatives or false positives.

* **F1-Score (Classification)**

To balance precision and recall, we used the F1-score, which is the harmonic mean of precision and recall.

The F1-score provides a single metric that balances the model’s ability to avoid false negatives and false positives.

This is especially useful in our three-class classification if one class is harder to identify or less common; the F1-score gives a more meaningful summary of performance than accuracy alone in such cases.

|  |  |  |
| --- | --- | --- |
| Metric | Formula | Range |
| Accuracy | (TP + TN) / (TP + TN + FP + FN) | [0,1] |
| Precision | TP / (TP + FP) | [0,1] |
| Recall | TP / (TP + FN) | [0,1] |
| F1 Score | 2 \* (Precision \* Recall) / (Precision + Recall) | [0,1] |
| **TP: True Positives, TN: True Negatives, FP: False Positives, FN: False Negatives** | | |

* **Intersection over Union (object detection)**

For the object detection model, a crucial metric is Intersection over Union (IoU**)**.

IoU measures how well the predicted bounding box overlaps with the ground-truth bounding box of an object (Ultralytics.com, 2025).

It is calculated as the area of overlap divided by the area of union between the predicted and true boxes. An IoU of 1.0 indicates a perfect localization. In our evaluations, a detection is considered correct if its IoU with the true box exceeds a threshold (e.g., 50%).

IoU is fundamental for computing higher-level detection metrics like average precision.

* **Average Precision (AP) and mean Average Precision (mAP) (Detection)**

We evaluated the detection model using Average Precision (AP) for each class and mean Average Precision (mAP) across classes.

AP is derived from the precision-recall curve for a single class; it is essentially the area under the precision-recall curve, summarizing the model’s precision/recall performance for that class (Ultralytics.com, 2025).

In our two-class detection fresh orange vs. rotten orange, we calculate AP for each class.

The mean Average Precision (mAP) is then the average of the AP values of all classes, providing an overall measure of the detector’s accuracy in both finding and correctly classifying objects.

We report mAP at different IoU thresholds for a comprehensive evaluation. Notably, mAP@0.5 refers to mAP when a 50% IoU threshold is used to determine true positives a relatively lenient criterion, whereas mAP@0.5:0.95 also written as mAP50-95 is the mean of AP calculated at IoU thresholds ranging from 0.50 to 0.95 in steps as per COCO evaluation standards.

The mAP@0.5:0.95 is a more strict and holistic metric, as it averages performance from easy detections low IoU threshold to very strict ones high IoU threshold.

For our YOLO-based model, we used these metrics to quantify how well it can both detect and classify oranges in the images.

We also recorded the model’s inference speed milliseconds per image to gauge suitability for real-time use, since detection models are often deployed in time-sensitive environments.

.

**Conclusion**

In conclusion, this project demonstrated a complete workflow from image classification to object detection in the context of fruit quality assessment.

Using the FruitVision dataset of fresh, rotten, and formalin-mixed fruits (Md Hasan Imam Bijoy, 2025) with a focus on oranges, we successfully trained a deep learning classifier to identify the condition of individual fruits and then extended the approach to a detection model that can locate multiple fruits in an image and determine their condition.

The classification model ResNet50-based achieved robust results in categorizing single orange images, benefiting from transfer learning and a balanced dataset.

The object detection model YOLO-based added the capability to pinpoint fruit locations, which is crucial for real-world applications like automated sorting systems or quality inspection on a conveyor belt.

Through the use of appropriate evaluation metrics for each task from accuracy and F1-score in classification to IoU and mAP in detection we validated the effectiveness of our models and identified areas for refinement.

The outcomes highlight that integrating both classification and detection approaches can provide a more comprehensive solution for automated fruit quality control: classification offers high precision in controlled scenarios one fruit per image, while object detection generalizes the solution to more complex scenes with multiple fruits.

Future work could involve expanding the object detection model to include the formalin-mixed category and applying these models in real deployment settings.

Overall, our work underscores the value of curated datasets like FruitVision and modern deep learning techniques in tackling agricultural and food safety challenges, moving us closer to efficient, automated fresh produce inspection.

**Ethics and Data Protection**

All data (images) used in this research were obtained from Mendeley Data, a public repository, and are distributed under a CC BY-NC-ND 4.0 license.

This ensures that the dataset is freely accessible for non-commercial academic use, and its use in this work complies with the terms of the license.

The origins, content, and prior labeling of these datasets are clearly documented to maintain transparency and traceability.

Since the images concern agricultural products citrus fruits and do not involve any personal, biometric, or sensitive human-related data, no ethical risks regarding privacy or human subjects are involved.

This significantly reduces concerns about data protection under GDPR or equivalent legislation, but ethical care is still required to ensure that data is used responsibly.

Manual annotations of fruit disease regions were performed with care, using standard annotation tools, and the process was fully documented to guarantee data integrity and reproducibility.

Model results and performance are reported transparently, including potential limitations and biases for example, the imbalance in class distribution or the annotation flaw identified during dataset preparation. Acknowledging such limitations prevents misleading claims and contributes to the integrity of the research.

The trained models and outputs will be used strictly for academic and research purposes. They will not be deployed in commercial applications or decision-making systems without further validation, thereby ensuring that results are not misused beyond the scope of this study.

All data and models are stored in institutional cloud services with restricted access limited to the project manager, with regular backups to prevent data loss.

This ensures confidentiality, availability, and integrity of the research data, in line with good research data management practices.

Finally, this project aligns with the principles of research ethics, transparency, and reproducibility, ensuring that all stages from data acquisition to annotation, training, evaluation, and reporting are documented, justified, and conducted responsibly.

# **System Configuration**

The computing environment included a local desktop PC and Google Colab free and Pro version.

On the local PC device we use a 6-core AMD Ryzen 5 5600X processor 3.7 GHz base, up to 4.6 GHz turbo[,](https://en.wikichip.org/wiki/amd/ryzen_5/5600x#:~:text=TSMC%27s%207%20nm%20process%20,3200%20memory) with 80 GB of DDR4 RAM supports up to 128 GiB)[.](https://en.wikichip.org/wiki/amd/ryzen_5/5600x#:~:text=TSMC%27s%207%20nm%20process%20,3200%20memory)

The operating system is 64-bit Windows 11 Home, the local graphics card is an NVIDIA GeForce GTX 1070 Pascal architecture with 1920 CUDA cores and 8 GB of GDDR5 memory.

This GPU with 8GB of VRAM limited the training of larger models, leading us to use cloud GPUs to speed up experiments.

* PC local: AMD Ryzen 5 5600X 6 núcleos/12 threads, 3,7 GHz; 80 GB RAM DDR4; GPU NVIDIA GTX 1070 Pascal, 1920 CUDA cores, 8 GB GDDR5 VRAM. System 64-bit Driver NVIDIA v566.36, CUDA 12.7.

Google Colab Free version: NVIDIA Tesla T4 GPU Turing architecture with 16 GB of GDDR6 memory (nvidia, n.d.) ≈15 GB usable, since ~1 GB is reserved for ECC (McCormick, 2024).

The free Colab provides access to T4 GPUs for Python/Jupyter notebooks at no cost, improving runtime over the local GPU. However, free sessions have usage limits typically up to ~12 consecutive hours per session and daily quotas, which restricted extended use. (Google, n.d.)

Google Colab Pro (Paid): NVIDIA A100 GPUAmpere architecture with 40 GB of HBM2e memory and 6912 CUDA cores (DataCrunch, 2024).

The A100 offers far superior performance to the GTX 1070 up to dozens of times faster in training tasks and increased memory capacity, significantly reducing the runtime of large models.

We upgraded to Colab Pro with subscription cost to have access to the A100 and fewer usage restrictions, although there are still quota and GPU usage limits on the paid platform. (Google, n.d.)

Software Tools**:** Python 3.x with machine learning libraries (TensorFlow, PyTorch), running on Jupyter notebooks, Google Colab. CUDA Toolkit v12.7 and NVIDIA 566.36 drivers were used to speed up calculations on the GPU.

All development was done on Colab's and Jupyter notebooks with remote access to the T4 and A100 GPUs and managing the environment interactively.

Each component above was essential to balancing training performance and computational costs. The GTX 1070's VRAM limitation led to the use of cloud GPUs T4 and A100.

The free Colab allowed initial access to Tesla T4 GPUs, but usage restrictions short sessions prompted the upgrade to Colab Pro with A100.

**Exploratory Data Analysis**

Our analysis begins by loading all image files from the directory structure into a single DataFrame, with columns for the image file path and its class label.

This yields a total of about 16,448 images. We see three class labels: Fresh, Rotten, and Formalin-mixed. A quick summary shows that the dataset is roughly balanced across these classes each has on the order of 5–6 thousand images, which is beneficial for training an unbiased model.

|  |  |
| --- | --- |
| Class | Count |
| Formalin-mixed | 5176 |
| Fresh | 6024 |
| Rotten | 5248 |

**Proportion by Class** **Analysis**

A pie chart with different colored circles

AI-generated content may be incorrect.

Figure: Class proportion

After building the DataFrame, we print the count of images per label. The counts are roughly equal (Fresh ≈ 36.6%, Rotten ≈ 31.9%, Formalin-mixed ≈ 31.5% of the data), indicating no severe class imbalance. This balanced distribution suggests the model can learn each class without major bias.

## **Class Distribution Analysis**

A graph of a class distribution

AI-generated content may be incorrect.

Figure: Class Distribution

Next we visualize the class frequencies. shows the number of images in each class.

In our case, the Fresh, Rotten, and Formalin-mixed are all similar in height, confirming the roughly even class sizes.

We observe that Fresh has the largest bar, and Rotten and Formalin-mixed are slightly lower but nearly equal.

## **Image Dimensions Analysis**

A graph with green and red dots

AI-generated content may be incorrect.

Figure: Image Dimensions

A graph with a blue square with a red x

AI-generated content may be incorrect.

Figure: Image Dimensions

Next, we examined image sizes by extracting the width and height (in pixels) of each image and plotting them on a scatterplot (width vs. height). The analysis shows that all data points are concentrated in a single location at (512, 512), visually confirming the summary statistics that all images in the dataset have identical dimensions.

Dimension statistics

|  |  |  |
| --- | --- | --- |
|  | Width | Height |
| Count | 16448.0 | 16448.0 |
| mean | 512.0 | 512.0 |
| std | 0.0 | 0.0 |
| min | 512.0 | 512.0 |
| 25% | 512.0 | 512.0 |
| 50% | 512.0 | 512.0 |
| 75% | 512.0 | 512.0 |
| Max | 512.0 | 512.0 |

## Descriptive statistics further corroborate this observation, with the mean, minimum, maximum, and standard deviation all equal to 512 pixels.

## **Sample Images per Class**

## A collage of different types of food AI-generated content may be incorrect.

## Figure: Sample Images per Class

## .

Finally, we sampled five random images from each class and displayed them in a grid.

This visual check helps ensure that each class contains representative examples and that the images are as expected, *Fresh* fruit images appear ripe and healthy, *Rotten* ones show discoloration or spoilage, and Formalin-mixed samples fruit preserved in formalin have distinct visual cues.

**Observations from samples**

By inspecting these samples, we verify that each class label makes visual sense and that there are no obvious labeling errors. This step also gives a sanity check that the classes are visually distinguishable, which is important for classification.

Overall, the EDA confirms that the dataset is well-structured: it contains only three labeled classes, with a fairly balanced number of images in each class and all images standardized to the same resolution

**Classification Setup of Experiments:**

For all variants, we use classic transfer learning we start with the ResNet50 architecture pre-trained on ImageNet, freeze its initial convolutional layers, and add a new trainable classification "top" dense layers (fchollet, 2023)[.](https://keras.io/guides/transfer_learning/#:~:text=The%20most%20common%20incarnation%20of,learning%20is%20the%20following%20workflow)

The dataset images of oranges in the classes "Fresh", "Rotten" and "Formalin-mixed*”* was organized in a DataFrame and balanced to have 5175 images per class.

We then stratistically divide it into training, validation, and testing, keeping the same proportion of classes in each partition.

The input pipeline uses layers of Keras preprocessing to apply data augmentation e.g., random horizontal flip, small rotations, zoom a technique that increases the diversity of the training set [(TensorFlow, n.d.).](https://www.tensorflow.org/tutorials/images/data_augmentation#:~:text=This%20tutorial%20demonstrates%20data%20augmentation%3A,transformations%2C%20such%20as%20image%20rotation)

In each experiment we applied combinations of RandomFlip, RandomRotation and RandomZoom during training.

The images have been resized to 224×224, and we use tf.keras.layers.Rescaling within preprocess input to normalize the pixels.

The top of each model consists of global average pooling (pooling='avg' in ResNet50), followed by one or two dense layers (ReLU) and *dropout*, ending in a softmax output layer with 3 neurons one per class.

We used categorical crossentropy as a function of loss and categorical accuracy and F1-macro/heavy metrics for evaluation.

Each model was compiled with the Adam optimizer; In the initial phase ("warmup") we use learning rate of 1e-3 and then lower it to 1e-5~1e-6 during fine-tuning.

We *used* EarlyStopping criterion: loss of validation to avoid overfitting, and ModelCheckpoint to save the best model obtained.

In short, we follow the standard flow of training only the new classifier with frozen base first, then releasing some final layers of the backbone for "fine-tuning" with a lower learning rate.

**Experimental Configuration ResNet50**

|  |  |
| --- | --- |
| **Component** | **Configuration / Approach** |
| **Backbone** | ResNet50 pretrained on ImageNet (initially frozen, later partially unfrozen for fine-tuning) |
| **TopLayers** | Global Average Pooling → Dense (1–2 layers, ReLU) → Dropout (0.3–0.5) → Dense(3, Softmax) |
| **Output Classes** | 3 categories: Fresh, Rotten, Formalin-mixed |
| **Loss Function** | Categorical Crossentropy (with optional label smoothing in Exp.2) |
| **Metrics** | Categorical Accuracy, Precision, Recall, F1 (Macro & Weighted) |
| **Optimizer** | Adam |
| **Learning Rate** | Warm-up: 1e-3 → Fine-tuning: 1e-5 to 1e-6 |
| **Training Strategy** | Phase 1: Train only top classifier (backbone frozen).  Phase 2: Unfreeze last ~50 layers for fine-tuning |
| **Regularization** | Dropout (0.3–0.5) |
| **Callbacks** | EarlyStopping (monitor val\_loss/val\_accuracy), ModelCheckpoint (save best weights), ReduceLROnPlateau |
| **Data Augmentation** | Random Flip, Random Rotation (±0.05–0.15), Random Zoom (10–20%) |
| **Input Size** | 224 × 224 RGB images |

### 

### **Evaluation Metric**

In each experiment we measured accuracy in the test set, as well as accuracy, recall, and F1-score by class. The full metrics macro and weighted are reported below for each experiment. In general, we obtained accuracy values around 95–97% in the test, indicating that the models are learning well to distinguish the orange classes.

Below we describe each experiment.

**Models Architecture**

In all experiments we use a ResNet50 base model pretrained on ImageNet.

The new classification head includes a Dense layer e.g. 128–256 units with ReLU followed by a 3-unit softmax output for the three classes.

We apply dropout (30–50%) in the head to reduce overfitting, and we use data augmentation horizontal flips, random rotations, zooms, shifts, etc. to improve robustness.

The model is trained with the Adam optimizer learning rate = 0.001.

The table below summarizes the architecture details for Experiments 1 to 4.

**Table 1 Models Architecture**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Experiment 1 | Experiment 2 | Experiment 3 | Experiment 4 |
| Data Augmentation | Horizontal flip; small rotations | Horizontal flip; rotations; slight zoom | Flip; rotations; zoom; width/height shifts | Flip; rotations; zoom; shifts; random brightness |
| Dropout rate | 0%  (no dropout) | 30%  (to reduce overfitting) | 50%  (to reduce overfitting) | 50%  (to reduce overfitting) |
| Base model | ResNet50 (ImageNet-pretrained) | ResNet50 (ImageNet-pretrained) | ResNet50 (ImageNet-pretrained) | ResNet50 (ImageNet-pretrained) |
| Trainable layers (init.) | None  (all base layers frozen) | None  (all base layers frozen) | None  (all base layers frozen) | None  (all base layers frozen) |
| Dense layer (units, act.) | 128  (ReLU activation) | 128  (ReLU activation) | 256  (ReLU activation) | 256  (ReLU activation) |
| Output layer (units, act.) | 3 (Softmax for 3 classes) | 3 (Softmax) | 3 (Softmax) | 3 (Softmax) |
| Optimizer (learning rate) | Adam (lr = 0.001) | Adam (lr = 0.001) | Adam (lr = 0.001) | Adam (lr = 0.001) |

**Fine-Tuning Settings**

After the initial training with the base frozen, we optionally unfreeze layers of ResNet50 for fine-tuning. Experiment 1 uses no fine-tuning, while Experiments 2 to 4 progressively unfreeze more layers. For example, Exp 2 unfreezes the last 50 layers, Exp 3 unfreezes the last 100 layers, and Exp 4 unfreezes all convolutional layers of ResNet50.

Each fine-tuning phase uses a much lower learning rate e.g. 1e-5 as recommended for transfer learning (AST, 2025). Callbacks like EarlyStopping, ReduceLROnPlateau, and ModelCheckpoint are employed during fine-tuning to control learning rate and prevent overfitting.

The table lists the fine-tuning parameters for each experiment.

**Table 2 Fine-Tuning**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Experiment 1 | Experiment 2 | Experiment 3 | Experiment 4 |
| Fine-tuning used? | No | Yes | Yes | Yes |
| Layers unfrozen | N/A | Last 50 layers | Last 100 layers | All layers |
| Fine-tuning LR | N/A | 1e-5 | 1e-5 | 1e-5 |
| Fine-tune epochs | N/A | 10 | 10 | 10 |
| Callbacks used | N/A | EarlyStopping, ReduceLROnPlateau, ModelCheckpoint | EarlyStopping, ReduceLROnPlateau, ModelCheckpoint | EarlyStopping, ReduceLROnPlateau, ModelCheckpoint |

**Results**

**Experiment 1**

**A graph of a graph of value and value

AI-generated content may be incorrect.**

**Figure: Experiment 1 Warm-up**

**A graph with numbers and a blue square

AI-generated content may be incorrect.** **A diagram of different types of food

AI-generated content may be incorrect.**

**Figure: Experiment 1 Warm-up Confusion Matrix**

**Results Experiment 2**

**A graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of

AI-generated content may be incorrect.**

**Figure: Experiment 2 Warm-up**

**A graph of a train loss

AI-generated content may be incorrect.**

**Figure: Experiment 2 Fine-Tuning**

**A graph of a diagram

AI-generated content may be incorrect.** **A diagram of a variety of different types of food

AI-generated content may be incorrect.**

**Figure: Experiment 2 Fine-tuning Confusion Matrix**

**Results Experiment 3**

**A graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of

AI-generated content may be incorrect.**

**Figure: Experiment 3 Warm-up**

**A graph of different colored lines

AI-generated content may be incorrect.**

**Figure: Experiment 3 Fine-Tuning**

**A graph of a number of different types of food

AI-generated content may be incorrect.** **A diagram of different types of numbers

AI-generated content may be incorrect.**

**Figure: Experiment 3 Fine-tuning Confusion Matrix**

**Results Experiment 4**

**A graph of a train and a train

AI-generated content may be incorrect.**

**Figure: Experiment 4 Warm-up**

**A graph of a train and train loss

AI-generated content may be incorrect.**

**Figure: Experiment 4 Fine-Tuning**

**A graph of a number of different types of mixed

AI-generated content may be incorrect.** **A diagram of a variety of different types of food

AI-generated content may be incorrect.**

**Figure: Experiment 4 Fine-tuning Confusion Matrix**

**Table Results experiment 1 to 4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Metric | Exp 1 | Exp 2 | Exp 3 | Exp 4 |
| Validation Accuracy (best) | 98.1% | 98.7% | 96.4% | 90.2% |
| Validation Accuracy (last) | 98.0% | 98.7% | 96.3% | 89.7% |
| Fine-Tuning Accuracy (bast) | N/A | 98.3% | 97.6% | 96.2% |
| Fine-Tuning Accuracy (last) | N/A | 98.2% | 97.7% | 96.2% |
| Test Accuracy | 97.2% | 98.3% | 97.8% | 96% |
| Loss | 0.083 | 0.057 | 0.061 | 0.088 |
| Macro Precision | 0.97 | 0.98 | 0.97 | 0.96 |
| Macro Recall | 0.97 | 0.98 | 0.97 | 0.96 |
| Macro F1 | 0.97 | 0.98 | 0.97 | 0.96 |
| Precision Formalin-mixed | 0.96 | 0.97 | 0.98 | 0.93 |
| Precision Fresh | 0.96 | 0.98 | 0.96 | 0.96 |
| Precision Rotten | 1.00 | 1.00 | 1.00 | 1.00 |
| Recall Formalin-mixed | 0.96 | 0.98 | 0.96 | 0.96 |
| Recall Fresh | 0.96 | 0.98 | 0.98 | 0.93 |
| Recall Rotten | 0.99 | 1.00 | 1.00 | 1.00 |
| f1-score Formalin-mixed | 0.96 | 0.98 | 0.97 | 0.94 |
| f1-score Fresh | 0.96 | 0.98 | 0.97 | 0.94 |
| f1-score Rotten | 1.00 | 1.00 | 1.00 | 1.00 |

**Single Image Inference Experiment**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Experiment 1** | Test type | Input Image | True Label | Predicted Label | Confidence Vector(Sofmax) | Index 0 = Formalin-mixed | Index 1 = Fresh | Index 2 = Rotten |
| Single image | Single Orange | Fresh | Rotten | 9.8626220e-01 | 1.3731196e-02 | 6.6320263e-06 | 9.8626220e-01 |
| Single image | Single Orange | Rotten | Formalin-mixed | 9.9999619e-01 | 9.9999619e-01 | 6.5258459e-15 | 3.8415415e-06 |
| Single image | Single Orange | Formalin-mixed | Formalin-mixed | 8.0794519e-01 | 8.0794519e-01 | 1.7663330e-09 | 1.9205484e-01 |
|  | | | | | | | | |
| **Experiment 2** | Test type | Input Image | True Label | Predicted Label | Confidence Vector(Sofmax) | Index 0 = Formalin-mixed | Index 1 = Fresh | Index 2 = Rotten |
| Single image | Single Orange | Fresh | Formalin-mixed | 9.7888947e-01 | 9.7888947e-01 | 6.7282218e-04 | 2.0437647e-02 |
| Single image | Single Orange | Rotten | Rotten | 9.9939895e-01 | 6.0104084e-04 | 7.6191516e-13 | 9.9939895e-01 |
| Single image | Single Orange | Formalin-mixed | Formalin-mixed | 9.9994063e-01 | 9.9994063e-01 | 6.3449318e-07 | 5.8791378e-05 |
|  | | | | | | | | |
| **Experiment 3** | Test type | Input Image | True Label | Predicted Label | Confidence Vector(Sofmax) | Index 0 = Formalin-mixed | Index 1 = Fresh | Index 2 = Rotten |
| Single image | Single Orange | Fresh | Formalin-mixed | 0.82052445 | 0.82052445 | 0.06285525 | 0.11662023 |
| Single image | Single Orange | Rotten | Formalin-mixed | 0.9866933 | 0.9866933 | 0.00217024 | 0.01113638 |
| Single image | Single Orange | Formalin-mixed | Formalin-mixed | 0.9524476 | 0.9524476 | 0.02416319 | 0.02338929 |
|  | | | | | | | | |
| **Experiment 4** | Test type | Input Image | True Label | Predicted Label | Confidence Vector(Sofmax) | Index 0 = Formalin-mixed | Index 1 = Fresh | Index 2 = Rotten |
| Single image | Single Orange | Fresh | Formalin-mixed | 0.9226653 | 0.9226653 | 0.06669562 | 0.01063912 |
| Single image | Single Orange | Rotten | Rotten | 9.9999785e-01 | 2.1914962e-06 | 7.2582619e-13 | 9.9999785e-01 |
| Single image | Single Orange | Formalin-mixed | Formalin-mixed | 9.9884194e-01 | 9.9884194e-01 | 1.9199493e-04 | 9.6608949e-04 |

**Evaluation of Experiments 1 to 4**

Across the four experiments, the models achieved very strong results overall, with test accuracies consistently above 96%.

Experiment 2 stood out as the best-performing setup, reaching 98.3% accuracy on the test set and showing very stable precision and recall across all classes.

Experiment 1 also performed well, achieving over 97% accuracy, although it lacked data augmentation and therefore relied more heavily on the dataset being well-balanced.

Experiment 3 achieved similar results (97.8%), showing that adding more augmentation and higher dropout can still preserve performance, though it did not surpass Experiment 2.

Finally, Experiment 4 performed the weakest, with test accuracy dropping to around 96%. This decline was likely caused by the more aggressive augmentation and very high dropout rate (70%), which may have led to underfitting.

Looking beyond the aggregate metrics, the single-image inference tests revealed important limitations.

The same three test images were classified differently across experiments, and in some cases the model gave very high confidence to the wrong class.

For example, a Fresh orange was sometimes misclassified as Formalin-mixed or Rotten with probabilities above 0.9.

This inconsistency suggests that, although the models generalize well on large test sets, they are sensitive to small variations and may rely too heavily on overlapping visual features such as colour and surface texture.

**Possible Improvements**

From these results, a few directions for improvement become clear.

First, expanding the dataset with more diverse and challenging cases would help the models handle borderline situations, such as oranges in the early stages of rot or with partial formalin exposure.

Refining the class labels to reduce ambiguity e.g., distinguishing between “Rotten spots” and “Fully Rotten” could also give the models clearer boundaries to learn.

Another area of improvement is confidence calibration, so that wrong predictions are not made with such high certainty.

Ensembling multiple models may further improve robustness by combining complementary perspectives.

Finally, explainability methods such as Grad-CAM would provide insight into whether the models are focusing on the relevant fruit areas, which could help refine both the dataset and the model design (Md. Jawadul Karim, 2024).

In summary, the experiments confirm that transfer learning with ResNet50 is highly effective for classifying orange quality, achieving strong performance across all setups.

At the same time, the inconsistencies in single-image predictions highlight that high accuracy alone is not enough; future work should focus on improving robustness, interpretability, and confidence calibration to move closer to reliable, real-world applications.

.

### **Object detection Setup of Experiments:**

In this work, I employed the YOLOv10 model for object-level detection of oranges.

The task was to not only localize each fruit in the image but also to classify it into one of two categories defined in the dataset: Orange Fresh or Orange Rotten.

The dataset consisted of images annotated with *bounding boxes* around each fruit, with labels corresponding to these two classes.

This setup goes beyond a simple image classifier, as it allows multiple fruits to be detected and classified within the same image.

For the experiments, I selected two variants of YOLOv10 released by Ultralytics: YOLOv10n (nano), optimized for speed and efficiency in resource-constrained environments, and YOLOv10s (small), which provides greater representational power at the cost of slower inference.

In both cases, the models were initialized with pretrained weights, enabling them to leverage prior knowledge from large-scale datasets.

The training configuration was consistent across experiments: a fixed batch size of 16, input images resized to 640×640 pixels, use of a cosine learning rate schedule, and Automatic Mixed Precision (AMP) for efficient GPU utilization. In addition, early stopping with a patience of 50 epochs was applied to halt training once validation performance plateaued.

The four experiments were conducted in two main phases:

* Experiments 1 and 2 (YOLOv10n):
* In Experiment 1, I used the baseline configuration with default parameters.
* In Experiment 2, I applied a set of hyperparameter adjustments aimed at improving the model’s ability to generalize and to better handle class imbalance between Orange Fresh and Orange Rotten.

Learning rate (lr0 = 0.01): A slightly higher initial learning rate was chosen to encourage faster convergence in the early epochs.

This adjustment can help the optimizer escape shallow local minima and improve learning efficiency. However, it carries the risk of unstable training if set too high, so the value was carefully balanced.

Class loss weight (cls = 0.5): By explicitly lowering the relative weight of the classification loss compared to the bounding box regression, the model is encouraged to focus more on accurate localization while still learning to separate the two fruit categories.

This was particularly relevant because the bounding box quality was as important as the class prediction in this detection task.

Data augmentation (horizontal flip, mosaic): Stronger augmentation was introduced to artificially increase dataset variability.

The 50% chance of horizontal flipping and the systematic use of mosaic augmentation expose the model to more diverse spatial arrangements of fruits.

This helps reduce overfitting, especially in a dataset where the number of training images is limited and can improve the robustness of the model when detecting fruits in different orientations and contexts.

Overall, the expectation from these adjustments was that the model would become less overfitted to the training set, achieve better localization accuracy, and maintain stable classification between the two classes, while also improving its ability to generalize to unseen test images.

* Experiments 3 and 4 (YOLOv10s):
* In Experiment 3, I trained the small variant with the baseline configuration to examine whether the larger model could achieve higher mAP compared to the nano version.
* In Experiment 4, I applied the same tuned hyperparameters from Experiment 2 to the YOLOv10s model, aiming to evaluate the combined effect of a larger architecture and hyperparameter adjustments.

**Experimental Configuration YOLOv10**

|  |  |
| --- | --- |
| **Component** | **Configuration / Approach** |
| **Model** | YOLOv10n (nano) and YOLOv10s (small) |
| **Class** | 2 categories Orange Fresh and Orange Rotten with bounding-box annotations |
| **Image Size** | 640 × 640 pixels |
| **Batch Size** | 16 images per iteration |
| **Training Stages** | Max 100 epochs,  Cosine learning-rate schedule,  Pretrained weights,  AMP enabled |
| **Early Stopping** | Patience = 50 epochs for validation performance not improve |
| **Optimizer** | Automatic selection (optimizer="auto") |
| **Loss Weights** | Box loss = 7.5  Class loss = 0.5 |
| **Data Augmentation** | Random horizontal flips (p=0.5),  Mosaic augmentation (p=1.0),  standard YOLOv10 pipeline |
| **Evaluation Metrics** | Precision, Recall, mAP@0.5, mAP@0.5:0.95, Average Precision per class, Inference speed (ms/image) |
| **Callbacks** | EarlyStopping (monitor val\_loss/val\_accuracy), ModelCheckpoint (save best weights), ReduceLROnPlateau |
| **Data Augmentation** | Random Flip, Random Rotation (±0.05–0.15), Random Zoom (10–20%) |
| **Output** | Bounding-box localization and class prediction for each detected Orange |

**Evaluation Metrics**

To evaluate the YOLOv10 detection models, I used standard metrics commonly applied in object detection research. The main measure of performance was the mean Average Precision (mAP).

Specifically, I reported mAP@0.5, which considers a detection correct if the predicted bounding box overlaps the ground-truth box by at least 50%, and mAP@0.5:0.95, which averages results across multiple thresholds from 0.50 to 0.95, following the COCO evaluation protocol. (Solawetz, 2025).

I also examined the Average Precision (AP) per class, allowing a comparison between the performance on Orange Fresh and Orange Rotten.

Since the dataset is slightly imbalanced and the classes present different visual challenges, per-class AP was essential to highlight strengths and weaknesses of the models.

In addition to mAP, I monitored Precision the proportion of correct detections among all predictions and recall the proportion of ground-truth objects successfully detected.

These complementary measures helped to understand whether the model was favoring conservative predictions (high Precision but lower Recall) or more exhaustive ones.

Finally, I measured the inference speed in milliseconds per image, including preprocessing, model inference, and postprocessing.

Taken together, these metrics provided a comprehensive view of the models’ performance in terms of accuracy, localization quality, and class-specific behavior.

**Results**

The table below summarizes the main metrics of each experiment: these are YOLOv10 models in nano (YOLOv10n) and small (YOLOv10s) variants, evaluated on the same validation set.

We used both mAP@[0.5:0.95] and mAP@0.5 (mAP50) – the former being the final average accuracy metric as defined by Ultralytics

**Experiments results table.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Metric | Exp 1  YOLOv10n (baseline) | Exp 2  YOLOv10n (hyperparameters) | Exp 3  YOLOv10s (baseline) | Exp 4  YOLOv10s (hyperparameters) |
| mAP@0.5:0.95 (val) | 0.577 | 0.577 | 0.578 | 0.577 |
| mAP@0.5 (val) | 0.687 | 0.687 | 0.709 | 0.687 |
| AP Rotten (val) | 0.238 | 0.238 | 0.253 | 0.237 |
| AP Fresh (val) | 0.916 | 0.916 | 0.903 | 0.916 |
| Speed (val, ms/img) | ~3.6 | ~3.6 | ~7.7 | ~4.2 |
| mAP@0.5:0.95 (test) | N/A | 0.582 | N/A | 0.582 |
| mAP@0.5 (test) | N/A | 0.694 | N/A | 0.694 |
| AP Rotten (Test) | N/A | 0.244 | N/A | 0.244 |

**Hyperparameters Experiment**

The following table details the main settings for each experiment, highlighting model and hyperparameter differences with a brief description of each.

In experiments 1 and 3, we used the default YOLOv10 settings YOLOv10n or YOLOv10s network, training for up to 100 epochs with an early stopping time of 50.

In experiments 2 and 4, we applied advanced hyperparameter adjustments.

**Hyperparameters Experiment Teble**

|  |  |  |
| --- | --- | --- |
| **Experiment** | Model | **Hyperparameters** |
| Exp 1 | YOLOv10n (baseline) |  |
| Exp 2 |  |  |
| Exp 3 |  |  |
| Exp 4 |  |  |
|  |  |  |

Evaluation of Experiments

Looking at the results of the four experiments, the overall picture is quite consistent. All models, regardless of configuration, reached almost the same performance, with mAP@0.5:0.95 staying around 0.577–0.578. This shows that the baseline setup was already strong, and the adjustments I made in Experiments 2 and 4 did not really push the accuracy any further.

When comparing the two architectures, the small version (YOLOv10s) in Experiment 3 gave a slight boost in accuracy at mAP@0.5 0.709 compared to 0.687 for the nano version.

However, this came at the cost of speed: the inference time more than doubled, which makes the improvement less attractive if fast predictions are needed. Interestingly, once I applied hyperparameter tuning to YOLOv10s in Experiment 4, the model became faster around 4.2 ms per image, but its accuracy dropped back to the same level as the nano model. In other words, the tuning reduced the latency but did not help the model learn to recognize oranges any better.

The biggest takeaway from all experiments is the difference between classes. Fresh oranges were detected almost perfectly across all models, with AP values consistently above 0.90. Rotten oranges, on the other hand, remained very difficult for the models, with AP stuck around 0.24–0.25.

The same trend appeared in both the validation and test results, which strongly suggests that the problem comes from the data itself likely the imbalance between the number of fresh and rotten samples or the more subtle appearance of rotten fruits.

In short: the nano model is already fast and accurate enough for most uses, while the small version only gave a minor improvement that may not justify the extra computation. The real bottleneck is not the architecture but the ability to correctly classify rotten oranges.

To improve this, I would need to gather more examples of rotten fruits, apply augmentations that highlight their distinctive features, or adjust the training loss to give more weight to this underrepresented class.

**Visual Results of Model Predictions**

The prediction images illustrate how the YOLOv10 models performed on a test dataset of 30 randomly selected images containing single oranges, both fresh and rotten. In all four experiments, the models consistently detected fresh oranges with very high confidence (above 0.90).

For rotten oranges, the models also showed solid performance, not only distinguishing them from fresh ones but even highlighting the rotten spots on the fruit surface.

This means that, besides classifying an orange as rotten, the models were also able to localize the damaged areas, which is directly aligned with the main goal of this work.

One important limitation I identified lies in my annotation strategy. During dataset labeling, I only used two classes (Orange Fresh and Orange Rotten).

This simplified approach worked for general classification, but it was not ideal for cases where an orange was mostly fresh but showed small localized rotten areas. In such cases, I labeled the entire fruit as Orange Rotten.

As a result, the model sometimes produced two detections in the same image: one box around the whole fruit marked as rotten, and another box around the rotten patch, also marked as rotten.

In retrospect, a more effective annotation scheme would have been to include a third class for example, Orange Spot or Rotten Area specifically for labeling localized defects. This would have allowed the model to separate the overall fruit classification Fresh vs. Rotten from the detection of defective regions on the fruit surface.

By doing so, the training process could have been more precise, leading to cleaner predictions and a better balance between global classification and local defect detection.

Figures: Exp 1 YOLOv10n (baseline)

A close up of an orange

AI-generated content may be incorrect. A close up of an orange

AI-generated content may be incorrect.

A close up of an orange

AI-generated content may be incorrect. A close up of an orange

AI-generated content may be incorrect.

A close up of an orange

AI-generated content may be incorrect. An orange on a white surface

AI-generated content may be incorrect.

Figures: Exp 2 YOLOv10n (hyperparameters)

A close up of an orange

AI-generated content may be incorrect.A close up of a rotten orange

AI-generated content may be incorrect.

A close up of an orange

AI-generated content may be incorrect.A close up of an orange

AI-generated content may be incorrect.

A close up of an orange

AI-generated content may be incorrect.A close up of an orange

AI-generated content may be incorrect.

Figures: Exp 3 YOLOv10s (baseline)

A close up of an orange

AI-generated content may be incorrect. A close up of an orange

AI-generated content may be incorrect.

A close up of an orange

AI-generated content may be incorrect. A close up of an orange

AI-generated content may be incorrect.

A close up of an orange

AI-generated content may be incorrect. A close up of an orange

AI-generated content may be incorrect.

Figures: Exp 4 YOLOv10s (hyperparameters)

A close up of an orange

AI-generated content may be incorrect. A close up of a round orange

AI-generated content may be incorrect.

A close up of an orange

AI-generated content may be incorrect. A close up of an orange

AI-generated content may be incorrect.

A close up of an orange

AI-generated content may be incorrect. A close up of an orange

AI-generated content may be incorrect.