**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.

**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 a 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.

## 4.2.6 “DenseNet-121”

## **DenseNet-121** 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[[19]](https://arxiv.org/abs/1608.06993#:~:text=layers%20have%20L%20connections%20,on%20most%20of%20them%2C%20whilst)[[20]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=Key%20Characteristics%20of%20DenseNet). In DenseNet-121 (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[[21]](https://arxiv.org/abs/1608.06993#:~:text=all%20preceding%20layers%20are%20used,four%20highly%20competitive%20object%20recognition)[[22]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=from%20the%20loss%20function%20and,parameters%20compared%20to%20traditional%20networks). 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)[[20]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=Key%20Characteristics%20of%20DenseNet). Third, despite being very deep, DenseNet-121 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[[21]](https://arxiv.org/abs/1608.06993#:~:text=all%20preceding%20layers%20are%20used,four%20highly%20competitive%20object%20recognition)[[23]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=3,parameters%20compared%20to%20traditional%20networks). Huang et al. 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[[24]](https://arxiv.org/abs/1608.06993#:~:text=all%20preceding%20layers%20are%20used,on%20most%20of%20them%2C%20whilst). DenseNet-121 in particular 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, DenseNet-121 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[[25]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=DenseNet%20introduces%20a%20paradigm%20shift,layers%2C%20significantly%20enhancing%20information%20flow)[[26]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=Within%20a%20dense%20block%2C%20each,channels%20to%20the%20dense%20block). This results in a lot of feature-map concatenation, yielding L(L+1)/2 connections for L layers[[19]](https://arxiv.org/abs/1608.06993#:~:text=layers%20have%20L%20connections%20,on%20most%20of%20them%2C%20whilst). 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[[27]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=Parameter%20Efficiency%20High%20Moderate%20Low,art%20Good%2C%20but%20outperformed).

## We initially included DenseNet-121 in our proposal as a second classification model to compare against ResNet-50. 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 DenseNet-121 given our resources. DenseNet-121 is deeper than ResNet-50 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 DenseNet-121 would not be feasible without extending the project timeline. As a result, **DenseNet-121 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 – that **features learned early on (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[[21]](https://arxiv.org/abs/1608.06993#:~:text=all%20preceding%20layers%20are%20used,four%20highly%20competitive%20object%20recognition). In our work, we leverage such pre-trained features (from ResNet-50) rather than training a complex model like DenseNet from scratch.

## In summary, **DenseNet-121** is an important milestone in CNN design, demonstrating that *“dense” feature reuse can improve performance while reducing parameters*[[21]](https://arxiv.org/abs/1608.06993#:~:text=all%20preceding%20layers%20are%20used,four%20highly%20competitive%20object%20recognition). 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.

## 4.2.7 “ResNet-50”

## **ResNet-50** is a deep convolutional neural network with 50 layers, belonging to the **Residual Networks (ResNet)** family pioneered by He et al. (2016). ResNets introduced the concept of **skip connections** (or residual connections), which largely solved the training difficulties of very deep networks[[28]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=GoogleNet%20introduced%20the%20cascading%20concept,later%20layers%20of%20the%20architecture)[[29]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=Application). A skip connection effectively allows the network to learn a residual function on top of the identity input , by adding the original input to the output of a few stacked layers. This simple idea enables **information and gradients to flow directly** from later layers back to earlier layers, mitigating the vanishing gradient problem that plagued ultra-deep nets[[28]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=GoogleNet%20introduced%20the%20cascading%20concept,later%20layers%20of%20the%20architecture)[[30]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=Skip%20Connections). In practice, ResNet-50 is organized into a series of convolutional blocks, each with internal shortcuts that add the block’s input to its output. This architecture allows training of networks far deeper than previously possible – the original ResNet paper famously trained a 152-layer network and an ensemble of ResNets which won the **ILSVRC 2015 ImageNet competition**, with a top-5 error of only ~3.6%, surpassing human-level performance on that task[[31]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=titled%20%E2%80%98Deep%20Residual%20Learning%20for,Image%20Recognition%E2%80%99). The success of ResNet-152 demonstrated that extremely deep models can generalize well when properly optimized; meanwhile, **ResNet-50** emerged as a popular configuration balancing depth and computational load. It has ~25.6 million parameters (much fewer than VGG-16’s 138M, for comparison) and can be run on common GPU hardware, making it a **practical choice for transfer learning** in many applications.

## ResNet-50’s architecture builds on earlier ideas (e.g., VGG’s 3×3 conv layers) but with the crucial addition of the identity shortcuts. Each residual block in ResNet-50 is actually a “bottleneck” block with three conv layers: a 1×1 reduce, a 3×3 conv, and a 1×1 expand. The input is added to the output of this three-layer stack (after matching dimensions, typically via the 1×1 conv)[[32]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=ResNet). This design was chosen to make the 50-layer model more efficient while still effectively 50 layers deep. The impact of residual learning was immediately clear – even the 34-layer plain network vs. 34-layer with residual connections showed a substantial improvement in accuracy and ease of training (He et al., 2016). In essence, ResNet-50 can **learn complex features** but also easily propagate gradients; if any layer were not needed, the network could simply learn an identity mapping for that layer (the skip connection makes that possible)[[30]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=Skip%20Connections). This contributed to the mantra “go deeper” in CNN design after 2015, as ResNets proved depth could be achieved without sacrificing trainability.

## In our project, **ResNet-50** is used as the backbone model for image classification (fresh vs. rotten orange). We chose ResNet-50 for several reasons: (1) Its strong track record – it’s a *“standard”* in transfer learning, often used as a baseline in many vision tasks due to its robust feature extraction capabilities[[33]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=ResNet%20stands%20for%20Residual%20Network,Residual%20Learning%20for%20Image%20Recognition%E2%80%99)[[34]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=Application). (2) The availability of **pre-trained weights** on ImageNet for ResNet-50 is excellent (via Keras or PyTorch libraries), which we leverage for transfer learning (see Section 4.3). (3) ResNet-50 is computationally lighter than very deep models like DenseNet-121 or ResNet-101/152, which was important given our hardware constraints. Training ResNet-50 on our dataset (with frozen layers then fine-tuning) was feasible within a few hours on a free Colab GPU, whereas deeper models might not converge in time. Moreover, ResNet’s design tends to generalize well – the earlier layers learn very general features (edges, colors, shapes), which are likely relevant to oranges, and later layers learn more specific concepts. By fine-tuning those later layers on our citrus dataset, we adapt the powerful pre-trained features to our task.

## **Performance-wise**, ResNet-50 in our experiments served as a solid baseline. We expected it to achieve high accuracy in classifying images as diseased or healthy, given similar use cases in plant disease detection literature have reported good results with ResNet variants (Mohanty et al., 2016; Zhang et al., 2019). Indeed, ResNet architectures have been used in agriculture for disease classification with success, often reaching >90% accuracy[[35]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=modules%2C%20whilst%20ResNet%20introduced%20the,later%20layers%20of%20the%20architecture)[[29]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=Application). In our results (discussed later), ResNet-50 delivered reliable accuracy, precision, and recall, validating our choice. One reason is that **ResNet-50’s residual features transfer well** – as one study noted, even features from intermediate ResNet layers are quite general and can be re-purposed to new tasks[[36]](https://proceedings.neurips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf#:~:text=target%20network%20to%20be%20trained,without%20overfitting%3B%20Recent%20studies%20have). Our use of ResNet-50 underscores this benefit of modern CNNs: one can take a model proven on a massive dataset and adapt it to a smaller, specific dataset with relative ease. This model, along with YOLOv10, forms one of the two pillars of our comparative study. The limitation, however, is that we could not compare ResNet-50 to an alternative like DenseNet-121 due to resource limits. Still, within the scope, ResNet-50 provided a compelling classifier that is likely near “optimal” for our needs – going for a more complex model might have yielded diminishing returns given the size of our dataset.

## In summary, **ResNet-50** brought the advantages of **deep residual learning** to our project: it is deep enough to capture intricate features of rot lesions, yet thanks to residual connections it was tractable to train and fine-tune. Its ImageNet-trained foundation gave us a head start through transfer learning. The model’s proven performance and manageable size made it an appropriate choice, and it stands as a key component in evaluating the central question of our project: *Is a purely classification approach (ResNet-50) “good enough” compared to a detection approach (YOLOv10) for identifying citrus disease in images?*

## 4.3 Transfer Learning and Fine-Tuning

## Building high-performance CNNs from scratch typically requires enormous labeled datasets and extensive training time. **Transfer learning** offers a practical shortcut by **reusing knowledge** from models pre-trained on large datasets (Pan and Yang, 2010). The core idea is that a CNN trained on a very large and diverse dataset (such as ImageNet, with 1,000 classes and millions of images) will have learned a variety of low-level and mid-level features that are broadly useful – edges, textures, shapes, color patterns, etc. These learned feature representations can be **transferred** to a new but related task, rather than starting the new model from random initialization. As Pan and Yang (2010) note, successful knowledge transfer can **improve learning performance and reduce the need for extensive new labeled data**, effectively leveraging prior learning to avoid costly annotation efforts[[37]](https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf#:~:text=knowledge%20transfer%2C%20if%20done%20successfully%2C,This%20survey%20focuses%20on). The underlying assumption is that many visual features – especially **low-level features** – are *generic* and **shared across different visual tasks**[[36]](https://proceedings.neurips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf#:~:text=target%20network%20to%20be%20trained,without%20overfitting%3B%20Recent%20studies%20have). For example, the detectors for edges, gradients, or basic textures that a network learns from ImageNet are likely also useful for distinguishing a rotten spot on an orange. Empirical studies support this: Yosinski et al. (2014) found that the first layers of a CNN learn very general features that transfer well to other datasets, whereas later layers become more task-specific. Thus, using a pre-trained CNN’s early layers as a fixed feature extractor for a new task is often extremely effective (Razavian et al., 2014). In our case, we use models pre-trained on ImageNet (ResNet-50 and YOLO’s backbone) and adapt them to the orange disease dataset, following standard transfer learning procedures.

## The **typical workflow** we followed for transfer learning is: (1) **Initialize** the model with pre-trained weights (e.g., ResNet-50 trained on ImageNet). (2) **Replace** the top-layer classifier with a new layer (or small network) appropriate for our task – for ResNet, we removed the 1000-class ImageNet fully-connected layer and added a new output layer for our two classes (Fresh vs. Rotten). (3) **Freeze** the pre-trained base layers initially, so that their weights do not change during the first round of training[[38]](https://keras.io/guides/transfer_learning/#:~:text=1,model%20on%20your%20new%20dataset)[[39]](https://keras.io/guides/transfer_learning/#:~:text=base_model). We only train the new top layers at first, which typically have random initial weights. Freezing is important because if we allowed the large base network to train on our small dataset immediately, the randomly initialized new layer could cause large gradient updates that *“would destroy the pre-trained features”*[[40]](https://keras.io/guides/transfer_learning/#:~:text=It%20is%20critical%20to%20only,trained%20features). In other words, the pre-trained convolutional filters contain valuable general features; we don’t want them to adjust too quickly or erratically due to the new random weights at the top. By training only the new classifier layer for some epochs, we let it learn to interpret the pre-existing features for the new classification task. During this phase, the base CNN is effectively an immovable feature extractor – and we also take care to keep special layers like **Batch Normalization** in inference mode to preserve their learned statistics[[41]](https://keras.io/guides/transfer_learning/#:~:text=Many%20image%20models%20contain%20,things%20to%20keep%20in%20mind)[[42]](https://keras.io/guides/transfer_learning/#:~:text=,what%20the%20model%20has%20learned). (Batch Norm layers, if left in training mode while frozen, would still update their running mean/var and could skew the model’s feature distribution, hence we pass training=False to the base model as Keras recommends[[43]](https://keras.io/guides/transfer_learning/#:~:text=inputs%20%3D%20keras,1%3A%5D%60%20to%20vectors)[[42]](https://keras.io/guides/transfer_learning/#:~:text=,what%20the%20model%20has%20learned).)

## After this initial training of the top layer (often called “feature extraction” stage), we move to **fine-tuning**. Fine-tuning means unfreezing some or all of the base model’s layers and continuing training, but typically with a **very low learning rate**[[44]](https://keras.io/guides/transfer_learning/#:~:text=pre)[[45]](https://keras.io/guides/transfer_learning/#:~:text=,trainable%20%3D%20True). The rationale is that we want to slightly adjust the pre-trained weights to better fit our specific dataset, without over-adjusting and forgetting the previously learned general features (a phenomenon known as “catastrophic forgetting”). In practice, we usually fine-tune the **later layers** of the network first – those are more specialized and thus more likely to benefit from adaptation. The earlier layers (which capture generic patterns like edges) usually remain frozen or use an even smaller learning rate, since they are already appropriate for most vision tasks (Ahmed et al., 2021). In our project, after training the new top layer on ResNet-50 until it was reasonably converged on the validation set, we unfroze the last couple of ResNet blocks and continued training with a learning rate about 10x lower than the initial one. This allowed the model to **“fine-tune”** high-level feature detectors (like perhaps those detecting decay textures or color changes on orange peel) to our data, while not distorting the fundamental edge and shape detectors in earlier layers. We monitored validation performance to avoid overfitting during fine-tuning – often just a few epochs of fine-tuning are sufficient, as was our case. Fine-tuning led to a modest but important boost in accuracy and F1-score, indicating that the model gained from that extra adaptivity.

## We applied a similar transfer learning approach to YOLOv10. The YOLO model’s backbone (CSP-Darknet) came pre-trained (likely on a mix of large datasets). We froze the backbone initially and trained the detection head on our labeled oranges (this is analogous to training only the new classifier in ResNet). Then we unfroze the backbone partially to fine-tune. In doing so, one must be careful with **learning rates** and **training schedules**. We used the Ultralytics YOLO training procedure which by default uses pre-trained weights and applies a one-cycle learning rate schedule, effectively fine-tuning the whole model gently on the new data.

## Overall, transfer learning dramatically **reduced training time and data requirements** for our project. Instead of training ResNet-50 from scratch (which would likely require tens of thousands of images to not overfit), we benefited from the rich feature representations that were already present[[46]](https://proceedings.neurips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf#:~:text=task,compute%20features%20that%20are%20fairly)[[47]](https://proceedings.neurips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf#:~:text=its%20first%20n%20layers%20to,training%20on%20the%20new%20task). Our results validate this strategy: the pre-trained ResNet-50 reached high accuracy on our task quickly, and YOLOv10 likewise learned to detect lesions with relatively few epochs, all thanks to standing on the shoulders of giants (the prior training on massive datasets). This technique is common in modern deep learning workflows because it **leverages the universality of learned representations** – as one publication succinctly put it, *“the low-level features extracted from data are similar and can be applied to other datasets”* (Ahmad et al., 2021). Transfer learning, combined with careful fine-tuning, allowed us to achieve our project goals within limited resources and is a key **enabler in applied computer vision** today[[37]](https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf#:~:text=knowledge%20transfer%2C%20if%20done%20successfully%2C,This%20survey%20focuses%20on). Without it, training our models to high performance would have been significantly more difficult, if not infeasible, given our dataset size and hardware.

## 

## [[1]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=shown%20in%20Figure%201%20,demanding%2C%20generating%20higher%20FPS%20and) [[2]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=YOLO%20was%20introduced%20to%20the,improving%20both%20speed%20and%20accuracy) [[3]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=proposed%20regions.%20Examples%20of%20two,stage%20detectors) [[5]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=Performance%20wise%2C%20the%20simple%20YOLO,at%2045%20FPS) [[28]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=GoogleNet%20introduced%20the%20cascading%20concept,later%20layers%20of%20the%20architecture) [[35]](https://www.mdpi.com/2075-1702/11/7/677#:~:text=modules%2C%20whilst%20ResNet%20introduced%20the,later%20layers%20of%20the%20architecture) YOLO-v1 to YOLO-v8, the Rise of YOLO and Its Complementary Nature toward Digital Manufacturing and Industrial Defect Detection

## <https://www.mdpi.com/2075-1702/11/7/677>

## [[4]](https://viso.ai/computer-vision/yolo-explained/#:~:text=The%20essence%20of%20YOLO%20models,and%20probabilities%20for%20each%20region) [[6]](https://viso.ai/computer-vision/yolo-explained/#:~:text=,much%20greater%20effect%20on%20IOU) [[7]](https://viso.ai/computer-vision/yolo-explained/#:~:text=A%20couple%20of%20years%20later%2C,performing%20model) [[8]](https://viso.ai/computer-vision/yolo-explained/#:~:text=YOLOv4%3A%20Improved%20accuracy%20and%20speed,with%20CSPDarknet53%20as%20its%20backbone) [[9]](https://viso.ai/computer-vision/yolo-explained/#:~:text=match%20at%20L475%20However%2C%20this,in%20speed%2C%20but%20also%20accuracy) [[10]](https://viso.ai/computer-vision/yolo-explained/#:~:text=However%2C%20this%20is%20not%20all%2C,in%20speed%2C%20but%20also%20accuracy) [[11]](https://viso.ai/computer-vision/yolo-explained/#:~:text=YOLOv4%3A%20Improved%20accuracy%20and%20speed,with%20CSPDarknet53%20as%20its%20backbone) [[12]](https://viso.ai/computer-vision/yolo-explained/#:~:text=YOLOv6%3A%20Faster%20and%20more%20efficient,time%20object%20detection) [[13]](https://viso.ai/computer-vision/yolo-explained/#:~:text=YOLOv9%3A%20Transformer,scale%20detection) YOLO Explained: From v1 to Present

## <https://viso.ai/computer-vision/yolo-explained/>

## [[14]](https://www.digitalocean.com/community/tutorials/faster-r-cnn-explained-object-detection#:~:text=%2A%20While%20Faster%20R,time%20performance) [[15]](https://www.digitalocean.com/community/tutorials/faster-r-cnn-explained-object-detection#:~:text=%2A%20Faster%20R,proposing%20regions%20and%20classifying%20them) [[16]](https://www.digitalocean.com/community/tutorials/faster-r-cnn-explained-object-detection#:~:text=proposal%20methods.%20,and%20precision%20in%20computer%20vision) [[17]](https://www.digitalocean.com/community/tutorials/faster-r-cnn-explained-object-detection#:~:text=as%20a%20balance%20of%20speed,time%20performance) [[18]](https://www.digitalocean.com/community/tutorials/faster-r-cnn-explained-object-detection#:~:text=datasets%2C%20including%20multi,time%20performance) Faster R-CNN Explained for Object Detection Tasks | DigitalOcean

## <https://www.digitalocean.com/community/tutorials/faster-r-cnn-explained-object-detection>

## [[19]](https://arxiv.org/abs/1608.06993#:~:text=layers%20have%20L%20connections%20,on%20most%20of%20them%2C%20whilst) [[21]](https://arxiv.org/abs/1608.06993#:~:text=all%20preceding%20layers%20are%20used,four%20highly%20competitive%20object%20recognition) [[24]](https://arxiv.org/abs/1608.06993#:~:text=all%20preceding%20layers%20are%20used,on%20most%20of%20them%2C%20whilst) [1608.06993] Densely Connected Convolutional Networks

## <https://arxiv.org/abs/1608.06993>

## [[20]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=Key%20Characteristics%20of%20DenseNet) [[22]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=from%20the%20loss%20function%20and,parameters%20compared%20to%20traditional%20networks) [[23]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=3,parameters%20compared%20to%20traditional%20networks) [[25]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=DenseNet%20introduces%20a%20paradigm%20shift,layers%2C%20significantly%20enhancing%20information%20flow) [[26]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=Within%20a%20dense%20block%2C%20each,channels%20to%20the%20dense%20block) [[27]](https://www.geeksforgeeks.org/computer-vision/densenet-explained/#:~:text=Parameter%20Efficiency%20High%20Moderate%20Low,art%20Good%2C%20but%20outperformed) DenseNet Explained - GeeksforGeeks

## <https://www.geeksforgeeks.org/computer-vision/densenet-explained/>

## [[29]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=Application) [[30]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=Skip%20Connections) [[31]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=titled%20%E2%80%98Deep%20Residual%20Learning%20for,Image%20Recognition%E2%80%99) [[32]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=ResNet) [[33]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=ResNet%20stands%20for%20Residual%20Network,Residual%20Learning%20for%20Image%20Recognition%E2%80%99) [[34]](https://viso.ai/deep-learning/resnet-residual-neural-network/#:~:text=Application) Deep Residual Networks (ResNet, ResNet50) A Complete Guide- viso.ai

## <https://viso.ai/deep-learning/resnet-residual-neural-network/>

## [[36]](https://proceedings.neurips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf#:~:text=target%20network%20to%20be%20trained,without%20overfitting%3B%20Recent%20studies%20have) [[46]](https://proceedings.neurips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf#:~:text=task,compute%20features%20that%20are%20fairly) [[47]](https://proceedings.neurips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf#:~:text=its%20first%20n%20layers%20to,training%20on%20the%20new%20task) How transferable are features in deep neural networks?

## <https://proceedings.neurips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf>

## [[37]](https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf#:~:text=knowledge%20transfer%2C%20if%20done%20successfully%2C,This%20survey%20focuses%20on) untitled

## <https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf>

## [[38]](https://keras.io/guides/transfer_learning/#:~:text=1,model%20on%20your%20new%20dataset) [[39]](https://keras.io/guides/transfer_learning/#:~:text=base_model) [[40]](https://keras.io/guides/transfer_learning/#:~:text=It%20is%20critical%20to%20only,trained%20features) [[41]](https://keras.io/guides/transfer_learning/#:~:text=Many%20image%20models%20contain%20,things%20to%20keep%20in%20mind) [[42]](https://keras.io/guides/transfer_learning/#:~:text=,what%20the%20model%20has%20learned) [[43]](https://keras.io/guides/transfer_learning/#:~:text=inputs%20%3D%20keras,1%3A%5D%60%20to%20vectors) [[44]](https://keras.io/guides/transfer_learning/#:~:text=pre) [[45]](https://keras.io/guides/transfer_learning/#:~:text=,trainable%20%3D%20True) Transfer learning & fine-tuning

## <https://keras.io/guides/transfer_learning/>

## 4.2 From Image Classification to Object Detection

While image classification (assigning a single label to an entire image) is useful, many real-world tasks require **object detection** – identifying *what* objects are present *and where* they are in the image. In our context of citrus inspection, a classifier like ResNet-50 might tell us an image likely contains a rotten orange, but it cannot pinpoint the diseased spot on the peel. Object detection models address this by outputting bounding boxes around objects of interest along with class labels. This section reviews the development of object detectors, leading into the YOLO family of models which we leverage (specifically YOLOv10).

**Two-Stage vs. One-Stage Detectors:** Early deep learning detectors (c. 2014–2015) were often *two-stage* systems. A representative example is **Faster R-CNN** (Ren et al., 2015)[[18]](https://viso.ai/deep-learning/faster-r-cnn-2/#:~:text=Faster%20R,world%20images). Faster R-CNN first uses a Region Proposal Network (RPN) to generate candidate regions in the image that might contain an object, and then a second stage classifies each region and refines the bounding box. In essence, the model looks *twice* at the image: once to propose object-like regions, and a second time to classify those regions precisely. This approach was a big improvement over earlier methods (it was *“more efficient and accurate”* than its predecessors R-CNN and Fast R-CNN[[19]](https://viso.ai/deep-learning/faster-r-cnn-2/#:~:text=locate%20objects%20in%20complex%20real,images)) and became a cornerstone of high-accuracy detection in the mid-2010s. However, two-stage detectors tend to be computationally heavy – the sequential process of proposal generation and per-proposal classification makes them slower, which is problematic for real-time applications.

In 2016, Redmon et al. introduced **YOLO (You Only Look Once)**, a fundamentally different approach that reframed detection as a single-stage regression problem (Redmon et al., 2016). YOLO’s philosophy is to do everything in one forward pass of a single network – *one look* to both identify and localize objects. Instead of generating explicit proposals, YOLO divides the image into an $S \times S$ grid and directly predicts bounding boxes and class probabilities for each cell[[20]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=What%20is%20YOLO%3F)[[21]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=While%20algorithms%20like%20Faster%20RCNN,a%20single%20fully%20connected%20layer). This end-to-end design trades a bit of accuracy for a significant gain in speed. For instance, the original YOLO could run in real-time (45+ frames per second) on standard hardware, an order of magnitude faster than Faster R-CNN, though with slightly lower mean Average Precision (mAP) (Redmon et al., 2016). Subsequent research showed the general trade-off: **single-stage detectors** (like YOLO and SSD) are faster and more suitable for real-time processing, whereas **two-stage detectors** (like Faster R-CNN) often achieve higher accuracy, especially for small objects, at the cost of speed[[22]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=the%20specific%20requirements%20and%20constraints,of%20the%20application)[[23]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=Generally%2C%20single,where%20accuracy%20is%20more%20important). The choice depends on the application’s requirements.

Over the years, YOLO has undergone many improvements (v2, v3, v4, etc.), steadily closing the accuracy gap while retaining speed. The YOLO v2 (“YOLO9000”) introduced batch normalization and higher resolution training; YOLO v3 added multi-scale predictions; YOLO v4 and v5 incorporated new backbone networks (e.g., CSPDarknet) and training tricks to further boost mAP. These refinements have made one-stage detectors extremely competitive with two-stage methods on benchmarks, to the point that recent versions of YOLO rival or even surpass Faster R-CNN in accuracy, while still being faster.

**YOLOv10: A Modern One-Stage Detector:** In our project, we consider **YOLOv10**, a state-of-the-art member of the YOLO family (Wang et al., 2024). YOLOv10 continues the one-stage, end-to-end philosophy but introduces several innovations to improve both speed and accuracy. One key innovation is an improved assignment strategy during training that eliminates the need for a separate Non-Maximum Suppression (NMS) step[[24]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=One%20key%20technique%20used%20in,each%20object%20in%20the%20image). Normally, detection models generate multiple overlapping boxes for the same object and then apply NMS to filter out all but the best box. YOLOv10’s designers proposed a *“consistent dual assignment”* scheme that trains the network to inherently produce sparse, non-redundant outputs, making NMS optional (and thus simplifying the pipeline) (Wang et al., 2024). Another aspect of YOLOv10 is an overall focus on efficiency: every component from the backbone and neck (feature fusion layers) to the detection head was optimized for the speed–accuracy trade-off. The result is a family of model variants (from small to large) that deliver strong accuracy at low inference latency, suitable for real-time inspection tasks. According to the authors, YOLOv10 models achieve comparable mAP to other cutting-edge detectors while being faster in throughput, due to this holistic design (Wang et al., 2024). The official implementation is publicly available[[25]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=Several%20new%20versions%20of%20the,YOLO%27s%20development%20in%20recent%20years), and we leverage those pretrained weights and configuration in our experiments. In practice, using YOLOv10 means we can feed an image of an orange to the network and get back, in one step, a prediction like “Rotten: 95% confidence” along with a bounding box showing where the rot is – all in a fraction of a second. This capability is crucial for a future automated grading system on a production line, where decisions must be made on the fly.

It’s worth noting that YOLO’s progress is part of a broader trend in detection: *one-stage detectors have greatly matured*, thanks to techniques like better **backbones** (e.g. CSPDarknet, adapted from ResNet/Darknet), **feature pyramids** (FPN, PANet) to handle multiple object scales, and improved **loss functions** for localization. Meanwhile, two-stage detectors also incorporated improvements (Faster R-CNN itself was extended to Mask R-CNN for instance segmentation, and transformers have been used in DETR), but for our single-object-per-image scenario, the simplicity and speed of YOLO-type models are very attractive. In summary, object detection models have evolved from complex, multi-step pipelines to lean, end-to-end networks, mirroring the general deep learning trend of unified models. Our work specifically uses YOLOv10 as a representative advanced detector to compare against a classifier (ResNet-50) on the task of citrus disease identification.

*Figure 4.2.* *One-Stage vs. Two-Stage Object Detection.* *Modern detection architectures consist of a* *backbone* *CNN (often pretrained, e.g. ResNet) that extracts feature maps, a* *neck* *(such as a Feature Pyramid Network) that merges multi-scale features, and a detection* *head* *that outputs bounding boxes and class labels*[*[26]*](https://viso.ai/deep-learning/yolov8-guide/#:~:text=The%20diagram%20below%20illustrates%20the,of%20an%20object%20detection%20model)[*[27]*](https://viso.ai/deep-learning/yolov8-guide/#:~:text=The%20architecture%20consists%20of%20a,objects%20and%20predicting%20bounding%20boxes)*. One-stage detectors like YOLO and SSD perform dense predictions in a single pass over the image (green path), directly predicting classes and locations for all cells in a grid. Two-stage detectors like Faster R-CNN have an initial region proposal step and then classify each proposed region (purple path). One-stage methods are faster since they avoid per-candidate processing, while two-stage methods historically achieved higher accuracy on challenging benchmarks. Advances in network design have narrowed this gap significantly in recent years.*

**Detection Outputs and Metrics:** In object detection, predictions are evaluated by how well the bounding boxes overlap with ground truth and whether the correct classes are assigned. Two common metrics are **Intersection over Union (IoU)** and **Mean Average Precision (mAP)**. IoU measures localization accuracy: it is the area of overlap between the predicted box and the true box divided by the area of their union[[28]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=). A prediction is usually considered a “True Positive” if IoU > 0.5 (or another threshold) with the correct object; otherwise it counts as a localization error[[29]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=In%20object%20detection%2C%20%C2%A0precision%20and,5%20is%20a%20negative%20prediction)[[30]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=differs%20from%20the%20approach%20taken,repurposed%20classifiers%20to%20perform%20detection). Average Precision (AP), on the other hand, combines precision and recall across different confidence thresholds into one number (the area under the precision–recall curve)[[31]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=)[[32]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=Recall%20and%20precision%20offer%20a,mAP). The **mean Average Precision (mAP)** is the mean of AP over all object classes (and sometimes over multiple IoU thresholds, as in COCO’s mAP@[.5:.95]). We will report mAP for our detection results to quantify how well the model finds rotten spots. For classification-only models, standard metrics like accuracy, precision, recall, and F1-score are used. It’s important to interpret these appropriately: a classifier might have high accuracy in telling fresh vs. rotten oranges, but a detector with slightly lower classification accuracy could still be more useful if it pinpoints defects. Hence we consider both classification metrics and detection metrics in our evaluation (explained further in Section 4.3).

## 4.3 Transfer Learning and Model Fine-Tuning

Training deep networks from scratch typically requires enormous labeled datasets and computational resources. In practice, **transfer learning** is a powerful shortcut: a model trained on a large generic dataset is adapted to a smaller, domain-specific dataset. Both major models in our project (ResNet-50 and YOLOv10) benefit from transfer learning.

For image classification, it is now common to start with a CNN pretrained on ImageNet (which has broad visual knowledge) and **fine-tune** it on the target task (Pan and Yang, 2010). For example, we take a ResNet-50 already trained on ImageNet, which presumably has learned to detect edges, textures, and object parts, and then retrain some of its layers on our citrus dataset. Early layers can even be frozen (kept at their pre-learned weights) since they capture very general features (like Gabor filters or color blobs) that are useful for almost any vision task. Later layers, which represent more dataset-specific combinations of features, are usually adapted to the new task by continuing training on the new data with a smaller learning rate. This approach drastically reduces training time and improves performance, especially when the new dataset is limited. Krizhevsky et al. (2012) noted this benefit early on, and since then **fine-tuning** has been a go-to technique in the deep learning toolkit. He et al. (2016) similarly demonstrated that fine-tuning deep ResNets yields excellent results across tasks beyond ImageNet classification[[33]](https://www.ultralytics.com/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision#:~:text=Transfer%20learning%20with%20ResNet).

In our classification experiment, we use transfer learning by taking a pretrained ResNet-50 and replacing its final output layer (originally a 1000-class softmax for ImageNet) with a new layer for our two classes (“fresh” vs “rotten/formalin”). We then train the network on our images of citrus, starting with weights that already encode generic vision features. This should converge faster and better than training from scratch on a small citrus dataset. For object detection, transfer learning is also prevalent. The YOLOv10 model we use comes with pretrained weights (in fact, likely pretrained on MS COCO or similar large detection datasets). We will fine-tune YOLOv10 on our specific detection task (finding defects on oranges). Typically, YOLO models have a backbone network (often a variant of a classification CNN) that is pretrained, and a detection head that might be trained from scratch or also partially pretrained. By training the model on our dataset, it will adjust from general object categories to our specific domain of citrus defects. The expectation is that models initialized with pretrained weights will require fewer epochs and will reach higher accuracy given the same data, compared to randomly initialized models[[33]](https://www.ultralytics.com/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision#:~:text=Transfer%20learning%20with%20ResNet).

One caveat during fine-tuning is handling layers like batch normalization: these layers learn dataset-specific normalization parameters. Best practice is often to freeze batch norm layers or use a very low learning rate for them, to avoid destroying the valuable statistics they learned from the large dataset (Chollet, 2018). In our experiments, we follow standard protocols (initial training of a new classification head, then gradual unfreezing of deeper layers for fine-tuning) to leverage transfer learning effectively.

**Evaluation Metrics for Classification vs. Detection:** Since we are comparing a classification model and a detection model on the same overall task (citrus disease identification), we need to use appropriate metrics for each and interpret them carefully. For the binary classification (fresh vs rotten) task, **accuracy** is a straightforward metric: the fraction of oranges correctly classified. We also consider **precision**, **recall**, and **F1-score** because our data may be imbalanced (e.g., fewer rotten samples than fresh). Precision tells us, out of all oranges the model labelled “rotten,” how many were truly rotten (it penalizes false alarms). Recall tells us, out of all truly rotten oranges, how many the model caught (it penalizes misses). The F1-score is the harmonic mean of precision and recall, summarizing overall classification effectiveness. In a context like food safety, high recall might be especially important – we don’t want to miss a rotten fruit – but we also want decent precision to avoid too many false positives. We will report all these metrics for ResNet-50.

For the detection model (YOLOv10), as mentioned, **mAP** at IoU 0.5 (and possibly across multiple IoUs) is the main metric. mAP integrates both the localization and classification performance of the detector into one number. We interpret mAP alongside the classifier’s metrics. For instance, if ResNet-50 achieves 95% classification accuracy but YOLOv10 gets, say, 90% mAP, we will analyze whether YOLO might be slightly less accurate at classification or if it missed some small lesions – but YOLO’s output is richer (it provides location). Another metric specific to detection is the average **IoU of true positives**, which indicates how precise the bounding boxes are on average. Additionally, detection outputs allow us to measure the **processing speed** (in frames per second or milliseconds per image) which is crucial for a potential real-time system. We plan to record inference time for both models on the same hardware to compare the practicality of each approach. Overall, using the appropriate metrics for each approach ensures a fair comparison and highlights the trade-offs (classification-only vs. detection) in terms of what they deliver to an end-user of a citrus inspection system.

## 4.4 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) without bounding boxes. We had to **adapt an image-level dataset to a detection dataset** by drawing boxes around the decayed areas on the oranges. This required a significant annotation effort, as each rotten orange image might have one or more rotten spots that needed to be delineated.

We used an open-source tool, the **Computer Vision Annotation Tool (CVAT)**, to perform this bounding-box annotation. CVAT provides a user-friendly interface to draw boxes and polygons, supporting collaborative annotation projects. For each image that was labeled “rotten” or “formalin-mixed” (indicating some defect present), we inspected the image and created a tight bounding box around the visibly affected region of the fruit peel. In cases where an image had multiple distinct rotten patches, we annotated each with a separate box (so that detectors can learn to handle multiple instances, although our images typically contain one orange each, there could be multiple spots on one fruit). We maintained the labels simply as “rotten” (for any kind of defect) vs. “fresh,” to mirror the classification task. CVAT allowed exporting the annotations in YOLO format (text files with each box’s coordinates and class), which we then used to train YOLOv10.

One challenge was ensuring **annotation consistency** and quality. We defined clear guidelines: include the entire diseased area in the box with a little padding, but avoid including too much healthy area; if unsure about a slight discoloration, err on the side of including it if it might indicate early rot. We also took care to use CVAT’s interpolation feature for any images where drawing a precise box was tricky due to blur or lighting. Given that our project is focusing on single-object images (one fruit per image), the annotation task was manageable – unlike typical detection datasets that have many objects per image. Nonetheless, careful annotation is crucial because the detector will learn from our labels; any systematically bad labels (e.g., consistently too tight or too loose boxes) could bias the model.

After annotation, we split the dataset into training, validation, and test sets, making sure that the split is stratified and *consistent across the classification and detection tasks*. That is, if an image is used to test ResNet-50’s classification, the same image (with its box) is used to test YOLOv10’s detection, so our comparison is fair. We also augment the training data with flips, rotations, and lighting changes (using the same augmentation pipeline for both models as appropriate) to increase robustness. The end result is two parallel sets of data: one of labeled images for classification (fresh/rotten labels), and one of annotated images for detection (fresh/rotten labels with coordinates). This allows us to train and evaluate both models under as similar conditions as possible.

## 4.5 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 et al. (2022) used CNN features combined with a Softmax classifier to distinguish citrus diseases (like canker, scab, etc.) on fruits and leaves, achieving high accuracy by leveraging the rich spectral information[[34]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=In%20the%20context%20of%20Industry,DL%29%2C%20and%20unmanned%20aerial)[[35]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=management%20and%20harvest%20preparation,data%20structure%20and%2C%20ultimately%2C%20a). In a more recent study, Yadav et al. (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. Ang et al. (2024) developed an improved YOLOv8 model to detect young citrus fruits in complex foliage backgrounds[[20]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=What%20is%20YOLO%3F). 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**, Ismail et al. (2022) 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[[34]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=In%20the%20context%20of%20Industry,DL%29%2C%20and%20unmanned%20aerial)[[36]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=sector%20increasingly%20relying%20on%20technological,data%20structure%20and%2C%20ultimately%2C%20a).

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[[35]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=management%20and%20harvest%20preparation,data%20structure%20and%2C%20ultimately%2C%20a), 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 Melnychenko et al., 2024). 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[[34]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=In%20the%20context%20of%20Industry,DL%29%2C%20and%20unmanned%20aerial).

## 4.6 Modern Computing and Smart Agriculture Context

It is important to view our project in the context of **Industry 4.0** and modern smart agriculture. Industry 4.0 refers to the fourth industrial revolution characterized by automation, data exchange, and AI-driven systems in manufacturing and related sectors. In agriculture, this translates to leveraging advanced technologies (IoT sensors, drones, AI models) for tasks like crop monitoring, yield optimization, and quality assurance. Our focus – automated fruit quality inspection – is a prime example. Traditionally, fruit grading was done by human inspectors who would visually check each orange for blemishes or signs of disease. This process is labor-intensive, subjective, and not scalable. By introducing computer vision systems that can assess fruit in real-time, we move towards an **AI-augmented inspection** process that is faster, more consistent, and can operate 24/7 if needed.

Modern computing capabilities (GPUs, TPUs, edge AI devices) have reached a point where running a CNN model in real-time on a conveyor belt or even on a handheld device is feasible. For instance, a small YOLO model can run on an NVIDIA Jetson (an edge GPU platform) attached to a fruit-sorting machine, instantly flagging rotten oranges as they pass by. This kind of deployment exemplifies Industry 4.0 because it integrates **intelligent sensors** (cameras + AI) into the production line, enabling *data-driven decisions* (e.g., divert this orange to waste) on the fly[[34]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=In%20the%20context%20of%20Industry,DL%29%2C%20and%20unmanned%20aerial)[[36]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=sector%20increasingly%20relying%20on%20technological,data%20structure%20and%2C%20ultimately%2C%20a). The data collected can further be analyzed to find patterns – maybe certain orchards or batches have higher defect rates, informing upstream decisions.

In research, there have been attempts to fully integrate such systems. For example, Pearson et al. (2022) discuss using robotics and AI (including deep learning vision) to reduce food loss and waste, which is directly related to automatically removing spoiled produce from supply chains (a component of food industry 4.0). In another study, Moghimi et al. (2020) deployed a CNN-based apple sorter and highlighted that *“the integration of deep learning in quality control is a promising pathway toward Food Industry 4.0”*. All these efforts point to a future where **deep learning techniques are standard tools in agriculture** – from detecting diseases on leaves using drones to sorting fruits by internal quality using X-ray imagery and CNNs (as explored by Perez et al., 2021). The citrus industry specifically can benefit given the scale of production and the economic impact of diseases like citrus canker, mold, etc. Early detection and removal of infected fruits can prevent the spread and maintain overall quality of batches, thus these AI systems contribute to both quality and safety.

Our thesis, though focused on a specific technical comparison, ultimately contributes to this vision. By determining whether a classification model suffices or a detection model adds value, we provide guidance on which type of AI system is more appropriate for citrus packhouses or processing facilities. If YOLOv10 (detection) proves to be significantly better at catching small defects than ResNet-50 (classification), it makes a case for adopting detection-based inspection in industry. If the difference is minor, one might opt for a simpler classifier, which could be easier to deploy on limited hardware. Either outcome is useful for designing the next generation of inspection systems.

In terms of **computing**, our experiments also consider the deployment aspect. We test inference on a local GPU (for maximum speed) but also on a CPU-only environment to simulate what happens if a small farm co-op wants to run this on a normal PC without a GPU. We even test on Google Colab (which simulates a cloud environment) to see if cloud processing is viable within necessary time frames. These considerations echo the practical constraints of applying AI in agriculture – one must balance accuracy with cost and infrastructure. Fortunately, as mentioned, the steady improvement of both algorithms (efficient CNNs) and hardware (cheap GPUs, even AI accelerators on mobile devices) is tilting the balance in favor of widespread adoption. The concept of **edge AI** is particularly relevant: models like MobileNet and later YOLO variants are optimized to run on edge devices, meaning the grading can happen right on the farm or packing site, avoiding latency of internet connectivity.

To sum up, the intersection of deep learning, modern computing hardware, and agriculture is enabling an “intelligent automation” of tasks like citrus disease detection. This is perfectly aligned with the goals of Industry 4.0 – creating smart, interconnected systems that improve efficiency and reduce reliance on manual labor. Our work is a small but integral part of this movement, as we explore the trade-offs between different AI approaches to find what works best for citrus quality control in a real-world setting.

## 4.7 Conclusion

Through this literature review, we traced the evolution of CNNs and their applications from academic curiosities to indispensable tools in industry. We saw how **CNN architectures** improved over time – deeper networks with innovations like residual connections and dense connectivity led to remarkable gains in image recognition performance. We reviewed how these networks, when pretrained on large datasets, can be repurposed via **transfer learning** to solve specific problems like ours with relatively little data. We also examined the parallel evolution in **object detection**, culminating in real-time one-stage detectors like YOLO that can localize defects as well as classify them.

From the perspective of citrus disease inspection, prior studies have mostly focused on classification accuracy (e.g., distinguishing diseased vs. healthy fruit) or on detection in more complex scenes (fruits on trees). Few have directly asked the question we pose: *what additional value does localization provide for single-fruit inspection?* This review established that while a strong classifier can achieve high accuracy, a detector could provide more actionable information. The expected benefits include pinpointing small rotten spots and potentially providing size/severity information of the defect, which a classifier cannot do. On the other hand, detection models are more complex and might require more annotation effort and slightly more computation.

By grounding our approach in the literature, we have solidified the rationale for our experiments. The review highlighted a gap – the need for a **direct comparison** of classification vs. detection under controlled conditions – which our work aims to fill. Moreover, the broader context of Industry 4.0 and smart farming emphasizes why this is important: as automated inspection systems become the norm, we must determine the most efficient and effective AI method to deploy. If our experiments show that YOLOv10 can catch defects that ResNet-50 misses (or vice versa), it will guide the design of future citrus grading systems. Ultimately, the literature confirms that integrating deep learning into agriculture is not only feasible but highly beneficial[[34]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=In%20the%20context%20of%20Industry,DL%29%2C%20and%20unmanned%20aerial). Our thesis builds on these insights, using state-of-the-art CNN techniques to enhance citrus fruit quality control and adding new knowledge about the interplay of classification and detection in this specific application.

**References:**

*(Selected key references are listed in Harvard style; full bibliography is provided at the end of the thesis.)*

* **LeCun et al. (1998)** – *Gradient-based learning applied to document recognition.* Introduced LeNet-5 CNN, early successful application of CNNs to handwritten digit recognition[[1]](https://www.pinecone.io/learn/series/image-search/imagenet/#:~:text=AlexNet%20was%20the%20first%20model,dismissed%20by%20most%20as%20impractical).
* **Krizhevsky et al. (2012)** – *ImageNet classification with deep CNNs (AlexNet).* Demonstrated a CNN winning ILSVRC 2012 by a large margin, rekindling interest in deep learning[[3]](https://www.pinecone.io/learn/series/image-search/imagenet/#:~:text=Results%20of%20ILSVRC%202012%20).
* **Simonyan & Zisserman (2015)** – *Very Deep CNNs for Large-Scale Image Recognition (VGG).* Explored pushing depth to ~19 layers with smaller filters, achieving top ILSVRC results.
* **Szegedy et al. (2015 & 2016)** – *Inception (GoogLeNet) series.* Introduced network-in-network modules and factorized convolutions for efficient depth; won ILSVRC 2014.
* **He et al. (2016)** – *Deep Residual Learning (ResNet).* Enabled ultra-deep networks (50-152 layers) with skip connections, setting new state-of-the-art[[14]](https://www.ultralytics.com/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision#:~:text=This%20design%20helps%20solve%20a,for%20the%20model%20to%20learn).
* **Huang et al. (2017)** – *Densely Connected CNNs (DenseNet).* Proposed dense connectivity to maximize feature reuse and efficiency[[17]](https://arxiv.org/abs/1608.06993#:~:text=in%20a%20feed,DenseNets%20obtain).
* **Redmon et al. (2016)** – *You Only Look Once (YOLO).* Proposed single-stage object detector running in real-time, with unified bounding box and class prediction[[20]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=What%20is%20YOLO%3F).
* **Ren et al. (2015)** – *Faster R-CNN.* Two-stage detector with Region Proposal Network, significantly improved detection accuracy at moderate speed.
* **Wang et al. (2024)** – *YOLOv10: Real-Time End-to-End Detection.* Latest YOLO variant focusing on NMS-free design and efficiency; reported on arXiv and implemented by Ultralytics.
* **Yadav et al. (2022, 2024)** – Applied CNNs (including ResNet variants) on hyperspectral data for citrus disease classification, achieving high accuracy in distinguishing healthy vs. various diseases.
* **Ang et al. (2024)** – Used YOLOv8 to detect citrus fruits in orchards; demonstrated improved small object detection in dense foliage.
* **Ismail et al. (2022)** – Developed a real-time fruit grading system using deep learning; highlighted the potential of AI in automated quality control.
* **Melnychenko et al. (2024)** – Integrated YOLO-based detection with UAV imaging for fruit counting; contextualized in Industry 4.0, emphasizing efficiency gains in agriculture[[34]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=In%20the%20context%20of%20Industry,DL%29%2C%20and%20unmanned%20aerial)[[36]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=sector%20increasingly%20relying%20on%20technological,data%20structure%20and%2C%20ultimately%2C%20a).
* **Pearson et al. (2022)** – Discussed reducing food waste with AI and robotics, relevant to automated removal of defective produce.
* **Pan & Yang (2010)** – Provided an overview of transfer learning methods, foundational to the fine-tuning approach we use.
* **Ioffe & Szegedy (2015)** – Introduced Batch Normalization, a technique we implicitly benefit from in modern CNN architectures to stabilize training.
* **Srivastava et al. (2014)** – Introduced Dropout regularization, which we use during training to improve generalization.

[[1]](https://www.pinecone.io/learn/series/image-search/imagenet/" \l ":~:text=AlexNet%20was%20the%20first%20model,dismissed%20by%20most%20as%20impractical) [[2]](https://www.pinecone.io/learn/series/image-search/imagenet/#:~:text=were%20stacked%20against%20these%20networks) [[3]](https://www.pinecone.io/learn/series/image-search/imagenet/#:~:text=Results%20of%20ILSVRC%202012%20) [[4]](https://www.pinecone.io/learn/series/image-search/imagenet/#:~:text=Several%20pieces%20came%20together%20for,the%20use%20of%20multiple%20GPUs) [[12]](https://www.pinecone.io/learn/series/image-search/imagenet/#:~:text=Image%3A%20Typical%20architecture%20of%20a,CNN) [[13]](https://www.pinecone.io/learn/series/image-search/imagenet/#:~:text=Convolutional%20Neural%20Networks%20will%20be,of%C2%A0Embedding%20Methods%20for%20Image%20Search) AlexNet and ImageNet: The Birth of Deep Learning | Pinecone

<https://www.pinecone.io/learn/series/image-search/imagenet/>

[[5]](https://en.wikipedia.org/wiki/Convolutional_neural_network#:~:text=The%20ImageNet%20Large%20Scale%20Visual,The%20best%20algorithms%20still) Convolutional neural network - Wikipedia

<https://en.wikipedia.org/wiki/Convolutional_neural_network>

[[6]](https://www.upgrad.com/blog/basic-cnn-architecture/#:~:text=Convolutional%20Layer) [[7]](https://www.upgrad.com/blog/basic-cnn-architecture/#:~:text=Example%3A%20Let%E2%80%99s%20consider%20an%20example,or%20ears%20in%20the%20image) [[8]](https://www.upgrad.com/blog/basic-cnn-architecture/#:~:text=Pooling%20Layer) [[9]](https://www.upgrad.com/blog/basic-cnn-architecture/#:~:text=Layer) [[10]](https://www.upgrad.com/blog/basic-cnn-architecture/#:~:text=Fully%20Connected%20Layer) [[11]](https://www.upgrad.com/blog/basic-cnn-architecture/#:~:text=connected%20layer) CNN Architecture: 5 Layers Explained Simply

<https://www.upgrad.com/blog/basic-cnn-architecture/>

[[14]](https://www.ultralytics.com/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision#:~:text=This%20design%20helps%20solve%20a,for%20the%20model%20to%20learn) [[15]](https://www.ultralytics.com/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision#:~:text=,car) [[16]](https://www.ultralytics.com/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision#:~:text=ResNet,image%20and%20where%20they%20are) [[33]](https://www.ultralytics.com/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision#:~:text=Transfer%20learning%20with%20ResNet) ResNet-50 and Its Role In Computer Vision | Ultralytics

<https://www.ultralytics.com/blog/what-is-resnet-50-and-what-is-its-relevance-in-computer-vision>

[[17]](https://arxiv.org/abs/1608.06993#:~:text=in%20a%20feed,DenseNets%20obtain) [1608.06993] Densely Connected Convolutional Networks

<https://arxiv.org/abs/1608.06993>

[[18]](https://viso.ai/deep-learning/faster-r-cnn-2/#:~:text=Faster%20R,world%20images) [[19]](https://viso.ai/deep-learning/faster-r-cnn-2/#:~:text=locate%20objects%20in%20complex%20real,images) The Fundamental Guide to Faster R-CNN

<https://viso.ai/deep-learning/faster-r-cnn-2/>

[[20]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=What%20is%20YOLO%3F) [[21]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=While%20algorithms%20like%20Faster%20RCNN,a%20single%20fully%20connected%20layer) [[22]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=the%20specific%20requirements%20and%20constraints,of%20the%20application) [[23]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=Generally%2C%20single,where%20accuracy%20is%20more%20important) [[24]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=One%20key%20technique%20used%20in,each%20object%20in%20the%20image) [[25]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=Several%20new%20versions%20of%20the,YOLO%27s%20development%20in%20recent%20years) [[28]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=) [[29]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=In%20object%20detection%2C%20%C2%A0precision%20and,5%20is%20a%20negative%20prediction) [[30]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=differs%20from%20the%20approach%20taken,repurposed%20classifiers%20to%20perform%20detection) [[31]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=) [[32]](https://www.v7labs.com/blog/yolo-object-detection#:~:text=Recall%20and%20precision%20offer%20a,mAP) YOLO Algorithm for Object Detection Explained [+Examples]

<https://www.v7labs.com/blog/yolo-object-detection>

[[26]](https://viso.ai/deep-learning/yolov8-guide/#:~:text=The%20diagram%20below%20illustrates%20the,of%20an%20object%20detection%20model) [[27]](https://viso.ai/deep-learning/yolov8-guide/#:~:text=The%20architecture%20consists%20of%20a,objects%20and%20predicting%20bounding%20boxes) YOLOv8: A Complete Guide

<https://viso.ai/deep-learning/yolov8-guide/>

[[34]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=In%20the%20context%20of%20Industry,DL%29%2C%20and%20unmanned%20aerial) [[35]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=management%20and%20harvest%20preparation,data%20structure%20and%2C%20ultimately%2C%20a) [[36]](https://www.mdpi.com/1424-8220/24/6/1913#:~:text=sector%20increasingly%20relying%20on%20technological,data%20structure%20and%2C%20ultimately%2C%20a) Intelligent Integrated System for Fruit Detection Using Multi-UAV Imaging and Deep Learning

<https://www.mdpi.com/1424-8220/24/6/1913>