# **Comparison Report**

## ***ResNet Model:***

First, I imported the necessary libraries for the project, including keras, NumPy, and pandas. I then defined the paths for the training and testing data directories.

To load the images, I used the ***tf.keras.preprocessing.image\_dataset\_from\_directory*** function. This utility reads images from the specified directories and generates a ***tf.data.Dataset*** object. It automatically infers the class labels from the subdirectory names within the main data directories. Using this function, I created separate dataset objects for training, validation, and testing.

To display a sample of the images with their corresponding labels, I used the ***os.listdir*** function to retrieve the class names from the subdirectory names. I then created a custom ***show\_images*** function to display a batch of 25 images. Additionally, I configured the dataset to use ***tf.data.AUTOTUNE*** to optimize performance by allowing TensorFlow to dynamically determine the optimal number of workers for data loading.

Next, I implemented data augmentation techniques to make the model more robust. By manipulating the images within the dataset, the model can learn to recognize features more effectively across a wider variety of examples.

For the ***ResNet*** model itself, I started with a pre-trained base model from ***Keras***. I set the ***include\_top*** parameter to **False** to remove the original top classification layer and leverage transfer learning for my specific dataset. I also set the base model's ***trainable*** attribute to **False** to freeze its weights during the initial training phase. Before building my final model, I defined a ***preprocess\_input*** function to ensure the images were formatted correctly for the ***ResNet*** model. I also created GlobalAveragePooling2D and Dense layers, which would serve as the new classification head.

I then built a new functional model by stacking the layers in the following order:

* Data augmentation layer
* Preprocessing layer (preprocess\_input)
* The pre-trained, non-trainable ResNet base model
* A GlobalAveragePooling2D layer
* A new Dense prediction layer

Finally, I compiled the model by defining the optimizer, loss function, and metrics. I also incorporated an EarlyStopping callback to prevent overfitting. After training the model for 10 epochs, the accuracy exceeded 95%. Due to the large number of classes, I was unable to display the full confusion matrix. Instead, I generated a confusion matrix for a single batch of test data to visualize the model's performance on a smaller, representative sample.

**ResNet Applications (Image Classification)**

ResNet's ability to classify images makes it valuable for applications where the primary goal is to identify what an image contains.

* **E-commerce:** ResNet can be used for product classification, automatically categorizing items in a store's catalog (e.g., shirts, shoes, electronics) based on their images. This improves search functionality and inventory management.
* **Medical Imaging**: In healthcare, ResNet can assist in the detection of diseases by classifying medical scans (like X-rays or MRIs). For example, it can be trained to identify whether a scan shows signs of a tumor or a specific condition.
* **Agriculture:** Farmers can use ResNet for crop and pest identification from drone or satellite imagery. This helps in monitoring crop health and identifying areas that need intervention.
* **Social Media:** Social media platforms use image classification to tag content, filter inappropriate images, or organize user-uploaded photos by category (e.g., "Food," "Travel," "Pets").

## ***YOLO Model:***

Training the YOLO model was a demanding process due to the manual image annotation required. I used Roboflow to create bounding boxes, which was time-consuming as their guidelines recommend annotating at least 70 images.

After completing the manual annotation, I used the code provided by ***Roboflow*** to integrate the dataset into my notebook. I then imported the necessary libraries and executed the YOLO command-line interface to perform the object detection task.

The model was set to train mode for 10 epochs. Despite the short training time, the results were surprisingly good. When switched to detect mode, the model accurately identified fruits with precise bounding boxes, validating the effort put into the annotation phase.

**YOLO Applications (Object Detection)**

YOLO excels at identifying multiple objects within a single image and locating them with bounding boxes. This makes it ideal for tasks that require not only identification but also the precise location of objects.

* **Inventory Systems:** In warehouses and retail, YOLO can be used for automated inventory management. Cameras can scan shelves to count products and identify out-of-stock items, significantly reducing the need for manual checks.
* **Self-Driving Cars:** YOLO is a key component in autonomous vehicles for real-time object detection . It helps cars identify and locate other vehicles, pedestrians, traffic signs, and obstacles on the road, which is critical for navigation and safety.
* **Security and Surveillance**: In a surveillance context, YOLO can automatically detect and track specific objects or people, alerting security personnel to unusual activity or unauthorized items.
* **Robotics**: Robots in manufacturing or logistics can use YOLO to locate and pick up specific objects from a conveyor belt or a cluttered environment, enabling automation of complex tasks.