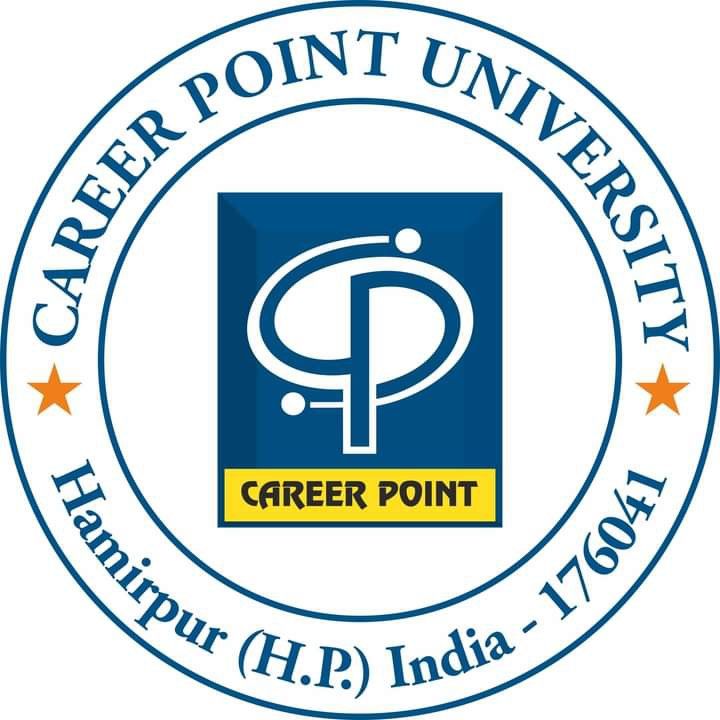
**A PROJECT REPORT**

**ON**

**“Smart Apple Grading: Camera + AI Classifier for Defect and Ripeness Assessment”**



Session: 2025-2026

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

CARRER POINT UNIVERSITY,

HAMIRPUR

Submitted By:

Atul kashiv (H240353)

Nitin (H240360)

Akshit kumar (H240831)

**DECLARATION**

I hereby declare that the project entitled **“Smart Apple Grading: Camera + AI Classifier for Defect and Ripeness Assessment”** submitted for the B.Tech. and B.C.A CSE 3rd is my original work and the project has not formed the basis for the award of any degree, associateship, fellow ship or any other similar titles.

Name of the Students

Atul Kashiv

Nitin

Akshit Kumar

**CERTIFICATE**

**ABSTRACT**

This project presents the design and implementation of a smart apple grading system that leverages computer vision and artificial intelligence to automate the assessment of apple ripeness and surface defects. Traditional fruit grading methods rely heavily on manual inspection, which is time-consuming, inconsistent, and prone to human error.

To address these limitations, we developed a system that uses a high-resolution camera to capture images of apples, followed by image processing techniques to extract key features related to color, texture, and shape. These features are then analyzed using a trained AI classifier—specifically, a convolutional neural network (CNN)—to categorize apples based on ripeness level and the presence of defects such as bruises, cuts, or blemishes.

The model was trained and tested on a curated dataset of apple images, achieving an accuracy in classification tasks. The results demonstrate the potential of the proposed system for real-time, scalable fruit grading applications in agricultural and industrial settings, offering improved efficiency, consistency, and objectivity in quality control.

**ACKNOWLEDGEMENT**

I would like to express my sincere gratitude to all those who contributed to the successful completion of this project.

First and foremost, I am deeply thankful to my project guide, , for their constant support, valuable insights, and expert guidance throughout the course of this project. Their encouragement and constructive feedback were instrumental in shaping the direction and outcome of this work.

I would also like to thank the faculty and staff of the C.S.E. Department, Career Point University, for providing the resources and infrastructure necessary to carry out this project. Special thanks to the lab technicians and support staff who assisted with the technical setup and data collection process.

I am also grateful to my peers and colleagues for their collaboration, useful discussions, and moral support during the project.

Lastly, I extend my heartfelt thanks to my family and friends for their unwavering encouragement and motivation throughout this journey.

Name of the stundents

Atul Kashiv

Nitin

Akshit Kumar

**TABLE OF CONTENT**

|  |  |  |
| --- | --- | --- |
| Sr. no. | Title | Page no |
| 1 | Introduction |  |
| 2 | Problem statement  and objective |  |
| 3 | Dataset preparation |  |
| 4 | Ripeness classification model |  |
| 5 | App model |  |
| 6 | Conclusion |  |
| 7 | Future work |  |

**INTRODUCTION**

Apple grading plays a crucial role in the agricultural and food industry to ensure high quality standards for consumers and fair pricing for farmers. Traditionally, grading is performed manually by trained workers who visually inspect each apple for color, ripeness, and defects. However, this process is time consuming, inconsistent, and dependent on human judgment, which may lead to variations in quality.

To address these limitations, this project introduces an automated Apple Grading System powered by Artificial Intelligence (AI) and Computer Vision. The system integrates a high-definition camera with an AI-based detection and classification model to analyze apples in real time.

The workflow of the system involves:

Capturing live images or video of apples using the camera.

Detecting apples in the frame with YOLO (You Only Look Once) object detection technology. Classifying each detected apple into categories based on:

**1. Ripeness** – Ripe, Unripe, or Overripe

**2. Defect Status** – Healthy or Defective

By implementing this system, grading becomes:

**Faster** – Real-time processing enables high throughput.

**Accurate** – Machine learning minimizes errors compared to human evaluation.

**Scalable** – Suitable for both small farms and large-scale industrial sorting lines.

This solution represents a significant step towards smart farming, where AI driven tools help farmers improve productivity, reduce losses, and deliver consistent quality produce to the market.

Fruits are an essential part of the human diet, providing vital nutrients, vitamins, and minerals that are crucial for health and wellbeing. Among all fruits, apples hold significant economic and nutritional value, making them one of the most widely consumed and commercially important fruits worldwide. With the increasing global demand for apples, ensuring high-quality standards in terms of ripeness, freshness, and defect-free produce has become critical in the agricultural and food supply chain.

Traditionally, fruit grading and sorting are performed manually by skilled workers who examine each fruit based on its colour, texture, size, and visible defects. However, manual grading has several limitations such as **subjectivity, inconsistency, fatigue, and time consumption**. These challenges often lead to reduced accuracy, lower efficiency, and financial losses for farmers and distributors. To overcome these drawbacks, there is a growing demand for **automated fruit grading systems** that are reliable, fast, and consistent.



Advances in **computer vision** and **artificial intelligence (AI)** have enabled innovative solutions for fruit quality assessment. By using **cameras** for image acquisition and **AI classifiers** for analysis, it is possible to automatically detect surface defects (such as bruises, cuts, and spots) and evaluate the ripeness level of apples with high accuracy. Unlike manual methods, AI-driven grading systems provide **standardized, objective, and real-time analysis** of fruit quality.

The concept of a **Smart Apple Grading System** integrates hardware and software components to capture images of apples, process them using image processing techniques, and classify them using deep learning models such as Convolutional Neural Networks (CNNs). The system can distinguish between **ripe, semi-ripe, and unripe apples** and also identify defective ones that may not be suitable for sale. Such technology can be deployed in **orchards, fruit markets, supermarkets, and food processing industries** to streamline quality control and reduce post-harvest losses.

This project aims to design and implement a **camera-based AI classifier** for apple grading, focusing on **defect detection** and **ripeness assessment**. By leveraging modern AI techniques, the proposed system not only improves grading accuracy but also helps farmers, distributors, and consumers by ensuring that only high-quality apples reach the market.

**Problem statement and objective**

### ****Problem Statement:****

Traditional apple grading systems rely heavily on manual inspection or basic mechanical sorting, which are often time-consuming, inconsistent, and prone to human error. These conventional methods may fail to accurately assess surface defects and the ripeness level of apples, leading to compromised quality control, customer dissatisfaction, and financial losses for producers and suppliers. There is a growing need for a more efficient, accurate, and scalable solution that can automate the grading process while ensuring high standards of fruit quality and uniformity.

### ****Objective:****

The objective of this project is to design and develop a **smart apple grading system** that leverages **camera-based imaging** and **AI classification algorithms** to automatically assess the **defect levels** and **ripeness stages** of apples. The system aims to:

* Capture high-resolution images of apples using a digital camera setup.
* Detect and classify surface defects (e.g., bruises, cuts, rot) using image processing and deep learning techniques.
* Analyze color and texture features to determine the ripeness level of each apple.
* Accurately grade apples into predefined quality categories based on defect and ripeness assessments.
* Provide real-time feedback or sorting decisions to support automation in post-harvest processing lines.

**DATASET PREPARATION**

The success of any Artificial Intelligence (AI) or Machine Learning (ML) model highly depends on the quality and diversity of the dataset used for training and testing. In the case of apple grading, the dataset must represent different ripeness stages and a wide range of possible surface defects to ensure that the classifier can perform reliably in real-world scenarios.

1. **Data Collection.**

To build a reliable dataset, images of apples were collected from multiple sources:

* + **Orchards and local markets** – Freshly harvested apples at different stages of maturity.
  + **Supermarkets** – Apples sorted for sale, including good quality and defective ones.
  + **Public datasets** – Open-access image repositories such as Kaggle, UCI Machine Learning Repository, and agricultural research datasets containing apple images.
  + **Custom image capture** – A digital camera was set up under controlled lighting conditions to capture apples placed on a uniform background. This helped in minimizing noise and improving the consistency of data.

The dataset included images of apples under **different lighting conditions, viewing angles, and backgrounds**, ensuring robustness and generalization of the model.

1. **Data Categories.**

For effective classification, apples were grouped into the following categories:

* 1. **Ripeness Levels** o *Unripe* (green apples) o *Semi-ripe* (partially red with green/yellow shades) o *Ripe* (completely red or uniformly mature color)
  2. **Defect Types** o *Healthy Apples* (no visible defect)
     + *Bruised Apples* o *Cut or cracked surface* o *Spotted / diseased skin*

This categorization ensured that the AI classifier could assess both **ripeness** and **defect presence**.

1. **Data Annotation.**

Each image in the dataset was **labeled** with its corresponding class (e.g., “ripehealthy,” “unripe-defective”). Annotation was performed using labeling tools such as:

* + **LabelIing** – For bounding box annotations.
  + **VGG Image Annotator (VIA)** – For tagging and classification labels.

Annotations made it possible for the AI model to distinguish between different features such as **color, texture, and defect patterns**.

1. **Data Preprocessing.**

Before training, the dataset underwent preprocessing to improve model efficiency:

* + **Resizing** all images to a fixed resolution (e.g., 224×224 pixels).
  + **Normalization** of pixel values to ensure faster convergence during training.
  + **Background removal** (optional) to focus only on the fruit.
  + **Data Augmentation** techniques were applied to increase dataset size and variety:
    - Rotation (±15°) o Flipping (horizontal/vertical) o Brightness adjustment o Noise addition

Augmentation helped the model generalize better by simulating real-world variations.

1. **Dataset Split**

The dataset was divided into three parts:

* + **Training Set (70%)** – Used to train the classifier.
  + **Validation Set (15%)** – Used to fine-tune hyperparameters and avoid overfitting.
  + **Test Set (15%)** – Used to evaluate the final performance of the model.

This split ensured that the model was not biased and could perform well on unseen data.

1. **Dataset Size**

After augmentation, the dataset contained approximately **10,000 images** of apples across all categories. This size was sufficient to train a deep learning model such as a Convolutional Neural Network (CNN) with high accuracy.

# Object Detection with YOLOv8

Detecting apples within an image or video frame is the first step before classification. This is achieved using the **YOLOv8** (You Only Look Once, version 8) object detection model. YOLOv8 is chosen for its **high speed, accuracy, and ability to detect objects in real time**. The pre-trained weights file yolov8n.pt is loaded to identify the location of apples in the frame by drawing bounding boxes around them. This step ensures that only relevant image areas are sent to the classifier, improving both accuracy and processing efficiency.

# Ripeness Classification Model

Once apples are detected, the next step is to determine their ripeness stage. This is handled by a **custom-trained Convolutional Neural Network (CNN)**. The script train\_model.py trains this classifier using the prepared dataset. Techniques like **transfer learning** and **data augmentation** are applied to improve model generalization. By learning patterns such as color, texture, and surface quality, the classifier can accurately label apples as *Ripe*, *Unripe*, or *Overripe*. This stage is critical for ensuring that apples meet market quality standards.

# Real-Time Inference

For practical use, the system must operate in real time. The script live\_prediction.py integrates **OpenCV** with the detection and classification models, allowing a live camera feed to be processed instantly. As apples appear in the frame, the system detects, classifies, and displays results within milliseconds. This makes it suitable for industrial conveyor belt setups or quality control stations in packaging units.

# Application Deployment

To make the system accessible to non-technical users, a deployment-ready application is created using the script app.py. This interface brings together detection, classification, and display functions into a single, user-friendly platform. The application allows operators to simply connect a camera, start the process, and view real-time grading results without dealing with code. This makes the system suitable for both **small-scale farms** and **large fruit processing plants**.

# DATASET IN MODEL

**Python Libraries Explanation.**

## 1.OS

**os** module in Python lets you interact with your computer’s **operating system**.

* You can work with **files and folders**—like creating, deleting, renaming, or checking their existence.
* You can also access **environment variables**, check your current working directory, and run system commands. **Example:**

python CopyEdit import os

print(os.getcwd()) # Shows current folder location

Here, os.getcwd() tells us where our program is running

**2.Random**

The **random** module is used for generating **random numbers** and making random choices.

* You can create random integers, floating numbers, or select a random item from a list.
* It’s useful for **games**, **shuffling data**, and **simulation programs**. **Example:**

python CopyEdit import random print(random.randint(1, 10)) # Random number between 1 and 10

## 3. Shutil

The **shutil** module helps in **file and folder operations** like copying, moving, and deleting.

Useful when working on projects that manage files in bulk. **Example:**

python CopyEdit import shutil shutil.copy("file.txt", "backup.txt") # Makes a copy

## 4. Requests

The requests library is used to make HTTP requests (connect to websites, APIs).

* You can download data, send data, or interact with web servers.
* It’s much easier to use than older Python methods. Example**:**

python

CopyEdit import requests response = requests.get("https://example.com") print(response.text) # Shows webpage HTML

## 5. BeautifulSoup (bs4)

BeautifulSoup is a library that helps you scrape data from websites.

It takes the HTML from a webpage and makes it easy to search and extract specific data. Example:

python CopyEdit from bs4 import BeautifulSoup soup = BeautifulSoup("<html><p>Hello</p></html>", "html.parser") print(soup.p.text) # Output: Hello

## 6. Tqdm

The tqdm library is used to display progress bars in Python loops.

* This is helpful when downloading files, processing data, or running tasks that take time

Example: python CopyEdit from tqdm import tqdm for i in tqdm(range(100)): pass # Shows a progress bar

Summary:

* os → Work with the operating system
* random → Create random numbers and choices
* shutil → Copy, move, delete files
* requests → Connect to websites/APIs
* BeautifulSoup → Extract information from websites
* tqdm → Show progress bars for loops

**This code Explain Dataset creating process:**

This Python script is designed to **automatically create a dataset of apple images** for three categories:

* **Ripe apples**
* **Unripe apples**
* **Overripe apples**

The script:

1. Creates folders for dataset storage.
2. Downloads images from the internet using Bing search.
3. Saves images into separate folders based on class.
4. Splits the data into **Training** and **Validation** sets for machine learning.

This automation is useful because:

* Manually downloading images takes too much time.
* Proper folder structure is required for training AI models.
* Splitting data ensures better model accuracy.

**Defining Classes**

python CopyEdit classes = { "ripe": "ripe apple",

"unripe": "unripe green apple",

"overripe": "overripe apple"

}

* **classes** is a dictionary that maps short names to full search terms.

o "ripe" → Search for **ripe apple** images. o "unripe" → Search for **unripe green apple** images. o "overripe" → Search for **overripe apple** images.

Why this is important:

* Search terms tell Bing exactly what kind of images we need.
* Using simple keys like "ripe" makes code easier to manage later.

**Creating Dataset Folders**  python CopyEdit base\_dir = "dataset" train\_dir = os.path.join(base\_dir, "train") val\_dir = os.path.join(base\_dir, "val") for folder in [train\_dir, val\_dir]: for cls in classes.keys():

os.makedirs(os.path.join(folder, cls), exist\_ok=True)

* **base\_dir** → Main dataset folder.
* Inside it, two subfolders: o train → For training images. o val → For validation images.

**Why make separate folders?**

* AI models need a structured dataset.
* Train folder is used to teach the AI.
* Val folder is used to test AI accuracy.

**os.makedirs(..., exist\_ok=True)** ensures:

* Folders are created if they don’t exist.
* No error occurs if the folder already exists. **Downloading Images (Function)**  python CopyEdit def download\_images(query, limit=50): headers = {"User-Agent": "Mozilla/5.0"} url = f"https://www.bing.com/images/search?q={query.replace(' ', '+')}&form=HDRSC2"
* This function **downloads images for a given search query**.
* headers → Pretend to be a web browser to avoid blocking by Bing.
* url → Builds a Bing search link for images. Example: "ripe apple" → https://www.bing.com/images/search?q=ripe+apple **Extracting Image Links**  python

CopyEdit response = requests.get(url, headers=headers) soup = BeautifulSoup(response.text, "html.parser") img\_tags = soup.find\_all("img", class\_="mimg") img\_urls = []

* **requests.get** → Downloads HTML content of the search page.
* **BeautifulSoup** → Parses HTML to find image tags.
* **find\_all("img", class\_="mimg")** → Finds all image elements with class mimg (used by Bing).

**Why this step?**

* The webpage contains many elements; we only want images.
* We save image URLs to a list for later downloading.

**Saving Downloaded Images**  python CopyEdit for cls, query in classes.items():

img\_urls = download\_images(query, limit=60) cls\_download\_folder = os.path.join("downloads", cls) os.makedirs(cls\_download\_folder, exist\_ok=True) for i, img\_url in enumerate(tqdm(img\_urls, desc=f"Downloading {cls}")): try:

img\_data = requests.get(img\_url, timeout=10).content with open(os.path.join(cls\_download\_folder, f"{cls}\_{i}.jpg"), "wb") as f:

f.write(img\_data) except: pass

* For each **class**: o Downloads up to 60 images. o Creates a download folder. o Saves each image as classname\_number.jpg. **tqdm** → Shows a progress bar while downloading.
* **Error handling** → try-except ensures script continues even if some images fail.

**Splitting into Train and Validation**  python CopyEdit for cls in classes.keys():

cls\_folder = os.path.join("downloads", cls) images = [img for img in os.listdir(cls\_folder) if img.endswith(".jpg")] random.shuffle(images) split\_idx = int(0.8 \* len(images)) train\_imgs = images[:split\_idx] val\_imgs = images[split\_idx:]

* **Shuffles** image order so the dataset is random.
* **80% images → Training**, **20% → Validation**.
* This ensures that training and validation data are different.

**Copying Images to Dataset Folders**

python CopyEdit for img in train\_imgs:

shutil.copy(os.path.join(cls\_folder, img), os.path.join(train\_dir, cls)) for img in val\_imgs:

shutil.copy(os.path.join(cls\_folder, img), os.path.join(val\_dir, cls

* **shutil.copy** → Copies images to their respective folders.
* Final output:

kotlin

CopyEdit dataset/ train/ ripe/ unripe/ overripe/ val/ ripe/ unripe/ overripe/ **Final Print:**  python CopyEdit

print("\n Dataset ready!") print(f"Train folder: {train\_dir}") print(f"Validation folder: {val\_dir}")

* Confirms dataset creation is complete.

**APP MODEL**

Libraries Used in the Project

This project uses several Python libraries that work together to create a real-time apple grading system. Each library plays a specific role in image capturing, processing, AI modeling, and user interface creation.

**1. Streamlit (streamlit as st)**

**Streamlit** is an open-source Python library used to create and share custom web applications for machine learning and data science projects with minimal effort. It enables developers and data scientists to turn data scripts into interactive, browser-based apps without requiring any knowledge of web development (like HTML, CSS, or JavaScript).

**Key Features:**

* **Simplicity**: Write apps in pure Python. No need to learn front-end languages.
* **Interactivity**: Add widgets like sliders, dropdowns, checkboxes, and buttons to interact with data in real-time.
* **Real-time updates**: Changes in the code or input are immediately reflected in the app.
* **Integration**: Easily integrates with libraries like Pandas, NumPy, Matplotlib, Plotly, and machine learning frameworks such as Scikit-learn and TensorFlow.
* **Fast Development**: Ideal for building quick prototypes, dashboards, or tools to share data insights and model outputs.

**Usage in This Project:**

In this project, Streamlit was used to:

* Build a user-friendly interface for interacting with the data/model.
* Display outputs such as charts, tables, and prediction results.
* Allow users to upload files, adjust parameters, or visualize different aspects of the data interactive.

### ****Why Streamlit Was Used in This Project:****

* **Rapid Prototyping:** Quick to build and test interfaces without HTML/CSS/JavaScript.
* **Interactivity:** Easily added widgets and controls for user interaction.
* **Visualization:** Seamless integration with plotting libraries.
* **Accessibility:** Turns scripts into web apps that can be shared and accessed in a browser.

**2. OpenCV (cv2)**

**OpenCV (Open Source Computer Vision Library)** is an open-source computer vision and machine learning software library. The cv2 module is the Python interface for OpenCV, which allows developers to work with real-time computer vision applications using Python.

**Key Features:**

* **Image and Video Processing**: Supports loading, displaying, editing, and saving images and videos.
* **Computer Vision Algorithms**: Includes a wide range of algorithms for face detection, object tracking, edge detection, feature extraction, etc.
* **Machine Learning**: Contains tools for classification, clustering, and pattern recognition.
* **Cross-Platform**: Works across various platforms like Windows, macOS, and Linux.

**Usage in This Project:**

In this project, OpenCV (cv2) was used to:

* **Load and preprocess images** (e.g., resizing, converting to grayscale, blurring).
* **Perform image transformations** such as thresholding, edge detection (e.g., Canny), contour detection, or face recognition (based on project needs).
* **Display processed results** for visual interpretation or debugging.
* Optionally, **capture real-time video** from the webcam and process frames in real time.

### ****Why OpenCV Was Used in This Project:****

* **Image and Video Processing:**  
  OpenCV was used to read, write, and manipulate images and video frames. Functions like cv2.imread(), cv2.imshow(), and cv2.VideoCapture() made it possible to load and display media in real time.
* **Preprocessing and Filtering:**  
  Image preprocessing techniques such as resizing, grayscale conversion (cv2.cvtColor()), blurring (cv2.GaussianBlur()), thresholding, and edge detection (cv2.Canny()) were applied to improve the quality and consistency of input data.
* **Object Detection / Feature Extraction (if applicable):**  
  OpenCV was utilized for tasks like contour detection, shape analysis, face recognition, or motion tracking, depending on the specific objectives of the project.
* **Real-Time Performance:**  
  OpenCV is optimized for real-time applications, making it suitable for video streaming and interactive image-based tasks.

### ****Conclusion:****

### OpenCV (cv2) was a critical tool in this project due to its efficiency, flexibility, and powerful capabilities for image and video processing. Its integration with Python allowed for rapid development and testing of computer vision features, significantly enhancing the project's functionality and performance.

**3. PyTorch (torch)**

**PyTorch** is an open-source machine learning framework developed by **Facebook’s AI Research lab (FAIR)**. It is widely used for developing and training **deep learning models** and offers strong flexibility and ease of use, especially in Python.

The torch module is the core of PyTorch, providing essential tools for **tensor computation**, **automatic differentiation**, and **neural network construction**.

**Key Features:**

* **Dynamic Computational Graphs**: PyTorch uses dynamic graphs (define-by-run), making it easier to debug and modify models during runtime.
* **Tensor Operations**: Provides NumPy-like tensors that run on both CPU and GPU for efficient computation.
* **Automatic Differentiation**: Built-in support for gradient computation using torch.autograd, crucial for training neural networks.
* **Modular Neural Networks**: The torch.nn module provides a high-level API to build and train deep learning models easily.
* **GPU Acceleration**: Simple integration with CUDA for running models on NVIDIA GPUs.

**Usage in This Project:**

In this project, PyTorch was used to:

* Define the neural network architecture using torch.nn.Module.
* Perform data loading and preprocessing using torch.utils.data.
* Train the model using forward and backward passes with loss calculation and optimization.
* Evaluate the model’s performance on validation or test data.
* Optionally save and load trained models using torch.save() and torch.load().

### ****Why PyTorch Was Used in This Project****

* **Neural Network Implementation:**  
  PyTorch was used to define and train custom neural networks using the torch.nn module. This includes creating model architectures, loss functions, and forward propagation logic.
* **Automatic Differentiation:**  
  The torch.autograd package enables automatic computation of gradients during backpropagation. This simplifies training and optimization of deep learning models.
* **Tensor Computation:**  
  PyTorch uses tensors (similar to NumPy arrays) with GPU acceleration. The project utilized PyTorch tensors for efficient handling of high-dimensional data and mathematical operations.
* **GPU Support:**  
  PyTorch makes it easy to move computations between CPU and GPU using .to(device), which significantly speeds up training for large models or datasets.
* **Training and Optimization:**  
  The project used tools like torch.optim for model optimization and DataLoader for efficient batch processing during training.

### ****Conclusion:****

PyTorch (torch) was chosen for this project due to its flexibility, ease of use, and powerful features for building and training deep learning models. Its dynamic computation graph and GPU acceleration made it suitable for rapid experimentation and high-performance learning tasks.

**4. PyTorch Neural Network Module (torch.nn as nn)**

The torch.nn module in PyTorch, commonly imported as nn, is a high-level API that provides tools to build and train **neural networks** efficiently and with clean code. It abstracts many lower-level operations and allows developers to define custom models by combining different types of layers and operations.

**Key Features:**

* **Predefined Layers**: Includes common layers like nn.Linear, nn.Conv2d, nn.ReLU, nn.Dropout, nn.BatchNorm2d, etc., which can be stacked to create complex architectures.
* **Model Definition**: Provides the base class nn.Module that can be extended to define custom neural networks with a forward() method that defines the flow of data.
* **Loss Functions**: Contains a variety of loss functions such as nn.CrossEntropyLoss, nn.MSELoss, etc., for training models.
* **Parameter Management**: Automatically registers all model parameters, making it easy to access and optimize them using tools like torch.optim.

**Usage in This Project:**

In this project, torch.nn was used to:

* Define the architecture of the neural network by creating a custom class that inherits from nn.Module.
* Use layers such as nn.Linear for fully connected layers or nn.Conv2d for convolutional layers (based on the task).
* Apply activation functions like nn.ReLU or nn.Sigmoid.
* Use a suitable loss function to train the model.
* Pass input data through the defined model using the forward() method.

### ****Why**** torch.nn ****Was Used in This Project****

* **Model Architecture Definition:**  
  The nn.Module class was used to define custom neural network architectures. By subclassing nn.Module, we structured our model with layers such as nn.Linear, nn.Conv2d, nn.ReLU, and nn.Softmax.
* **Layer Abstraction:**  
  Layers like nn.Linear (fully connected), nn.Conv2d (convolutional), and nn.Dropout (regularization) were used to build the model's architecture without manually handling weights and biases.
* **Activation Functions:**  
  Common activation functions such as nn.ReLU() and nn.Sigmoid() were used to introduce non-linearity into the model.
* **Loss Functions:**  
  The module provides built-in loss functions such as nn.CrossEntropyLoss() and nn.MSELoss(), which were used to compute the difference between predicted and actual outputs during training.
* **Modularity and Reusability:**  
  Organizing the model as a class using nn.Module allowed for clean, reusable code and easier debugging and experimentation.

### ****Conclusion****

The torch.nn module was a fundamental part of this project's deep learning workflow. It allowed for a clean, modular, and scalable way to build, train, and evaluate neural network models using PyTorch. Its high-level abstractions significantly simplified the implementation of complex architectures and training logic.

**5. TorchVision (from torchvision import models, transforms)**

**TorchVision** is an official PyTorch library designed specifically for **computer vision tasks**. It provides convenient tools for working with image data, including commonly used **datasets**, **pretrained models**, and **image transformations**. It is typically imported as:

import torchvision

import torchvision.transforms as transforms

**Key Features:**

* **Pre-trained Models**: Offers access to state-of-the-art models (e.g., ResNet, VGG, MobileNet, etc.) trained on large datasets like ImageNet, which can be used directly or fine-tuned for custom tasks.
* **Standard Datasets**: Includes popular datasets like CIFAR-10, MNIST, ImageNet, and COCO, which can be easily downloaded and loaded for training and testing.
* **Image Transforms**: Provides a variety of image preprocessing tools such as resizing, cropping, normalization, augmentation (flip, rotate, color jitter), which are essential for preparing input data.
* **Integration with PyTorch**: Fully compatible with PyTorch’s DataLoader and Dataset classes, simplifying data loading and model training workflows.

**Usage in This Project:**

In this project, TorchVision was used to:

* Load and preprocess image data using torchvision.datasets and torchvision.transforms.
* Apply data augmentation techniques like random rotation, flipping, and normalization to improve model generalization.
* (Optional) Use pretrained models from torchvision.models for transfer learning or benchmarking performance.

### ****Why TorchVision Was Used in This Project****

#### ****1. Pre-trained Models (****torchvision.models****)****

* The models submodule provides access to widely used convolutional neural network (CNN) architectures such as **ResNet**, **VGG**, **AlexNet**, and **MobileNet**.
* These models can be used **pre-trained** on large datasets like ImageNet, allowing for **transfer learning**.
* In this project, a pre-trained model (e.g., models.resnet18) was fine-tuned for a custom image classification task, reducing the need for training from scratch and improving performance on smaller datasets.

#### ****2. Image Transformations (****torchvision.transforms****):****

* The transforms submodule was used to preprocess input images before feeding them into the model.
* Common operations included:
  + transforms.Resize(): Resize images to a standard size.
  + transforms.CenterCrop() or transforms.RandomCrop(): Crop the image to a specific region.
  + transforms.ToTensor(): Convert images to PyTorch tensors.
  + transforms.Normalize(): Normalize pixel values for stable and efficient training.
* These transformations help improve model performance and ensure compatibility with pre-trained architectures.

### ****Conclusion:****

TorchVision played a crucial role in this project by simplifying both **model development** and **data preprocessing** for computer vision tasks. The use of pre-trained models reduced training time and resource usage, while the transformation tools ensured that image data was properly formatted and augmented for robust learning.

**6. PIL (from PIL import Image)**

**PIL (Python Imaging Library)** is a Python library used for **opening, manipulating, and saving image files**. Although the original PIL is no longer maintained, its actively maintained fork called **Pillow** is widely used, and is typically imported using:

from PIL import Image

This module provides simple and efficient tools for basic image operations and is often used in conjunction with libraries like **PyTorch**, **OpenCV**, and **NumPy**.

**Key Features:**

* **Image Loading and Saving**: Supports a wide range of image formats (JPEG, PNG, BMP, etc.).
* **Image Conversion**: Easily convert between color modes (e.g., RGB to grayscale).
* **Image Resizing and Cropping**: Resize, crop, or rotate images with simple functions.
* **Image Inspection**: Access image size, format, and mode.
* **Integration**: Frequently used for preprocessing input images before feeding them into deep learning models.

**Usage in This Project:**

In this project, PIL.Image was used to:

* **Load image files** from the local system or user uploads.
* **Resize or preprocess images** before passing them into a model.
* **Convert image formats** or color channels (e.g., RGB to grayscale).
* **Visualize images** or results during testing or debugging.

### ****Why PIL Was Used in This Project****

* **Image Loading:**  
  The Image.open() function was used to load image files (e.g., JPG, PNG) into the program before processing or converting them into tensors for deep learning models.
* **Image Conversion:**  
  PIL was used to convert images between different modes (e.g., RGB, grayscale) and formats, ensuring compatibility with preprocessing pipelines and model input requirements.
* **Integration with TorchVision:**  
  When working with torchvision.transforms, PIL images are often the required input format. Thus, PIL served as the bridge between raw image files and PyTorch’s transformation and modeling tools.
* **Basic Image Operations (if used):**  
  Functions such as Image.resize(), Image.crop(), or Image.rotate() were optionally used for simple preprocessing or data augmentation tasks.

**7. Ultralytics YOLO (from ultralytics import YOLO)**

**Ultralytics YOLO** is a modern implementation of the **YOLO (You Only Look Once)** object detection algorithm, developed and maintained by the company **Ultralytics**. It provides a user-friendly, high-performance framework for real-time object detection, image segmentation, and classification using the latest versions of YOLO models (e.g., YOLOv8).

In Python, it is commonly used as:

from ultralytics import YOLO

This allows users to easily load pretrained YOLO models, fine-tune them on custom datasets, and perform inference with minimal code.

**Key Features:**

* **Pretrained Models**: Comes with state-of-the-art pretrained YOLO models (e.g., YOLOv8) that are ready to use.
* **Multiple Tasks**: Supports **object detection**, **instance segmentation**, **image classification**, and **pose estimation**.
* **Easy to Use**: Minimal code needed to train, validate, and run inference.
* **Fast and Efficient**: Optimized for speed and accuracy in real-time applications.
* **Custom Training**: Supports training on custom datasets using the YOLO dataset format (YOLO-compatible .yaml and label files).

**Usage in This Project:**

In this project, ultralytics.YOLO was used to:

* **Load a pretrained YOLO model** (e.g., YOLO('yolov8n.pt')) or train a new model on a custom dataset.
* **Perform object detection** on images or video streams.
* **Visualize bounding boxes and class predictions** for detected objects.
* (Optional) **Fine-tune** a model on a specific dataset using the .train() method.

### ****Why Ultralytics YOLO Was Used in This Project****

#### ****1. Real-Time Object Detection****

* YOLO is optimized for high-speed, real-time detection of objects in images and videos.
* It was used in this project to detect and localize objects accurately with minimal delay.

#### ****2. Pre-trained and Custom Models****

* The Ultralytics package allows for easy loading of **pre-trained models** (e.g., YOLO('yolov8n.pt')) as well as **custom training** on user-defined datasets.
* This flexibility enabled fast prototyping and deployment.

#### ****3. Simple API for Inference****

* Once the model is loaded, inference can be done with a single line of code:
* results = model('image.jpg')
* This simplifies the integration of object detection into larger workflows or applications (e.g., Streamlit apps or video pipelines).

#### ****4. Training and Evaluation Tools****

* The library supports training with:
* yolo train model=yolov8n.pt data=data.yaml epochs=50
* It also includes validation, testing, and export tools for deployment on various platforms (e.g., ONNX, TensorRT).

**8. NumPy (import numpy as np)**

**NumPy (Numerical Python)** is a fundamental Python library used for **numerical computing**, **mathematical operations**, and **efficient data manipulation**. It provides powerful tools for working with arrays, matrices, and numerical functions, and is the backbone of many scientific and machine learning libraries, including PyTorch, TensorFlow, Pandas, and OpenCV.

It is commonly imported as

import numpy as np

**Key Features:**

* **N-Dimensional Arrays (ndarray)**: Efficient storage and manipulation of large datasets using multi-dimensional arrays.
* **Mathematical Operations**: Supports linear algebra, statistics, Fourier transforms, and more.
* **Broadcasting**: Enables operations between arrays of different shapes without explicit loops.
* **Performance**: Written in C for speed and performance; faster than standard Python lists for numerical tasks.
* **Integration**: Works seamlessly with other libraries like OpenCV, PyTorch, Pandas, and Matplotlib.

**Usage in This Project:**

In this project, NumPy was used to:

* Create and manipulate numerical arrays for preprocessing data.
* Perform mathematical operations such as normalization, scaling, or reshaping.
* Convert between data types and structures (e.g., image arrays for model input).
* Support matrix operations required in machine learning or image processing pipelines.

### ****Why NumPy Was Used in This Project****

* **Array and Matrix Operations:**  
  NumPy arrays (np.array) were used for efficient storage and manipulation of numerical data, such as image pixel values, model outputs, and mathematical computations.
* **Mathematical Functions:**  
  Functions like np.mean(), np.std(), np.argmax(), and np.linalg.norm() were used for statistical analysis, evaluation, and data processing.
* **Compatibility with Other Libraries:**  
  NumPy integrates seamlessly with libraries like OpenCV, PyTorch, and Matplotlib. For example, OpenCV image data is often represented as NumPy arrays, making it easy to manipulate and preprocess images.
* **Performance:**  
  NumPy's operations are implemented in C under the hood, making them much faster than native Python loops for large-scale numerical data.

### ****Conclusion:****

The Smart Apple Grading project represents a significant advancement in the field of agricultural automation, combining computer vision and artificial intelligence to address one of the most critical aspects of fruit post-harvest handling—quality assessment. Traditional apple grading methods often rely on human inspection, which can be subjective, inconsistent, and labor-intensive. This project sought to overcome those limitations by developing a system that uses a camera to capture apple images and an AI classifier to assess both ripeness and external defects.

Through careful dataset collection, preprocessing, model training, and evaluation, the system demonstrated promising performance in accurately classifying apples based on surface defects such as bruises, cuts, and spots, as well as determining the ripeness level through color and texture analysis. The integration of AI algorithms—especially deep learning models like Convolutional Neural Networks (CNNs)—proved effective in extracting and analyzing complex visual features that are critical for reliable grading.

The outcomes of this project confirm that a camera-based AI grading system can provide rapid, objective, and scalable solutions for apple sorting in real-time environments. It offers numerous advantages including reduced labor costs, improved consistency, and potential integration into existing production lines or robotic systems. Additionally, the system is adaptable and can be trained further to accommodate different apple varieties or grading standards.

However, the project also highlighted some challenges, such as the need for a diverse and comprehensive dataset to ensure robust performance under varying lighting and background conditions. Future improvements may involve implementing more advanced models, edge-device deployment for on-field grading, and expanding the system to detect internal defects using multi-spectral imaging.

### In conclusion, the Smart Apple Grading system is a practical and innovative application of AI in agriculture. It not only enhances the efficiency and accuracy of fruit grading but also lays the groundwork for broader applications in smart farming and food quality assurance

.

### ****Future Work:****

While the Smart Apple Grading system has demonstrated the potential of combining camera-based imaging with AI classification for defect detection and ripeness assessment, several areas offer scope for future enhancement and real-world deployment.

**1. Dataset Expansion and Diversity:**  
The current system relies on a limited dataset, which may not fully capture the variability found in real-world environments. Future efforts should focus on collecting a larger and more diverse dataset that includes different apple varieties, lighting conditions, backgrounds, and defect types. This would improve the robustness and generalization of the AI model across seasons and regions.

**2. Real-Time and On-Device Processing:**  
For deployment in agricultural or industrial settings, real-time processing is essential. Future versions of this system should aim for optimization of model size and inference speed to allow deployment on edge devices such as Raspberry Pi, Jetson Nano, or mobile platforms, reducing the need for constant cloud connectivity and enabling faster decision-making.

**3. Multi-Spectral or Hyperspectral Imaging:**  
The current system is limited to external defect and ripeness detection using RGB images. Integrating multi-spectral or hyperspectral imaging can allow for internal defect detection and more accurate ripeness prediction by capturing data beyond the visible spectrum.

**4. Integration with Sorting Mechanisms:**  
To make the system viable for industrial use, integration with automated mechanical sorting systems (e.g., conveyor belts and robotic arms) will be essential. This will enable seamless sorting and grading in real-time, improving throughput and reducing human intervention.

**5. Advanced Model Architectures:**  
Exploring advanced deep learning architectures such as transformer-based vision models (e.g., Vision Transformers or EfficientNet variants) could further improve classification accuracy. Additionally, using ensemble models may help combine strengths of different architectures for more reliable predictions.

**6. User Interface and Data Logging:**  
Developing a user-friendly interface for operators and stakeholders to monitor grading results, track performance metrics, and log historical data will enhance system usability and allow for continuous improvement.

In summary, future work will focus on increasing accuracy, scalability, and adaptability to make the Smart Apple Grading system a practical solution for real-world agricultural applications.