AI Smart Camera System for Real-Time Object Detection Using YOLOv8

Monica Bhandari

Table of Contents

1. Introduction

In today's world, cameras are everywhere, on our phones, in our laptops, on streets and buildings. But most of these cameras simply record what they see. They do not understand what is happening in the scene. With the growth of artificial intelligence, especially in computer vision, cameras are becoming much smarter. They can now recognize objects, track activities, and support real-time decision-making in ways that were not possible before.

This project focuses on building an AI Smart Camera System that can detect and recognize multiple everyday objects in real time using YOLOv8, one of the most advanced object detection models available today. The goal was to create a system that does more than just display a video feed, it interprets the scene and identifies what is present, similar to how a human would notice and label things around them.

To train the system, I used the COCO dataset, which contains 80 different object categories such as people, cars, laptops, bottles, animals, traffic signs, and many more. Because the dataset is so diverse, the model learns to recognize objects in many different environments, lighting conditions, and angles. This makes the system flexible and useful for a wide range of real-world applications, from safety and monitoring to automation and smart environments.

The process of building this project involved several key steps: understanding the dataset, configuring the YOLOv8 model, training it on thousands of images, evaluating the results using metrics like mAP and precision–recall, and finally testing the model on live camera input. Each step helped shape the system into something that can reliably detect objects in real time and respond quickly, which is essential for any intelligent camera application.

2. Data Collection

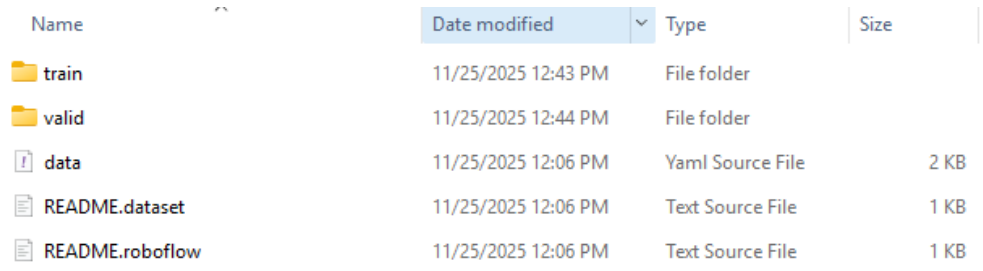
For this project, I used the COCO dataset, originally collected and published by Microsoft. It is one of the widely adopted datasets for training deep learning models in object detection.

The COCO dataset is publicly available and I used a YOLO-compatible version provided through Ultralytics/YOLO, which includes:

- Pre-split train and validation sets.
- YOLO-format label files for all images.
- Dataset metadata stored in data.yaml.
- Images organized into structured directories for direct use in training.

The dataset can be accessed at:
https://universe.roboflow.com/microsoft/coco

Below is the folder structure used during training:



3. Dataset Description

The version used in this project is the 2017 COCO Object Detection dataset, which contains:

- 121,408 images
- 883,331 object annotations
- 80 object classes
- 91 “stuff” categories
- A median image dimension of 640 × 480

Object Categories

The dataset includes 80 labeled object categories commonly found in everyday life. These categories span a wide range of domains, including:

- People: person.
- Vehicles: bicycle, car, motorcycle, bus, train, truck.
- Electronics: cell phone, laptop, mouse, keyboard, TV monitor.
- Animals: dog, cat, horse, cow, elephant, bear, zebra.
- Household items: chair, sofa, bed, dining table, toilet.
- Sports: ball, snowboard, skis, surfboard, tennis racket.
- Kitchen and food items: bottle, cup, bowl, fork, knife, spoon, pizza, orange, broccoli.

- Outdoor objects: stop sign, fire hydrant, traffic light.
- Miscellaneous everyday items: backpack, handbag, umbrella, suitcase.

This wide coverage helps the model understand scenes with multiple interacting objects, just like a real-world smart camera.

“Stuff” Categories

In addition to object categories, COCO contains 91 “stuff” classes, such as:

- sky
- grass
- road
- wall
- sand
- water

Stuff classes are not individual objects but are important for understanding the overall scene. Although YOLOv8 training for this project focuses on object categories, the richness of the dataset contributes to the model’s generalization.

Class Distribution Analysis

To analyze dataset imbalance, I generated a class frequency table covering all 80 object categories in the COCO dataset. The results reveal a highly skewed distribution, where certain classes appear far more frequently than others.

The most common class is “person”, with over 263,743 annotated instances, making it by far the dominant category in COCO. Other high-frequency classes include car, chair, book, bottle, and cup.

In contrast, several categories occur very rarely, such as hair drier, toaster, parking meter, etc. These classes represent a very small portion of the dataset.

Impact on Model Performance

Class imbalance directly affects YOLOv8’s learning:

- Frequent classes have more training examples, allowing the model to learn stronger and more robust representations.
- As a result, the model performs significantly better on classes like person, car, and dog.
- Rare classes provide limited training data, which often leads to weaker learning and reduced generalization.
- The model tends to perform poorly on low-frequency classes.

Understanding this distribution is important because it helps explain why some classes achieve high detection accuracy while others struggle.

The complete class frequency table is provided in the accompanying CSV file for reference.

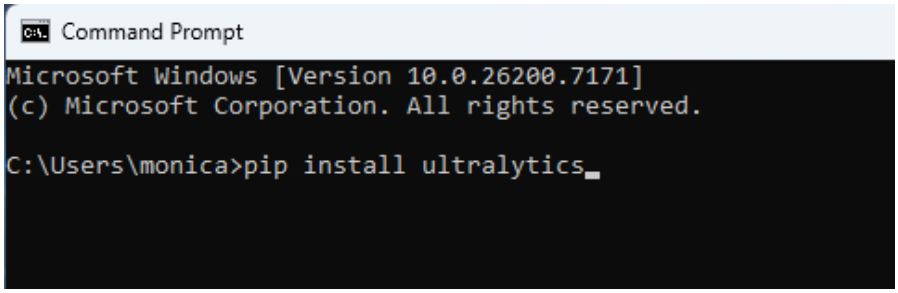
class_distribution_table

4. Environment Setup and Terminal Workflow

Before training the YOLOv8 model, I set up the project environment and installed all necessary libraries and dependencies. This step ensured that the training pipeline, dataset loading, and real-time testing could run smoothly without errors. The entire workflow was carried out using the terminal.

4.1 Installing Required Libraries

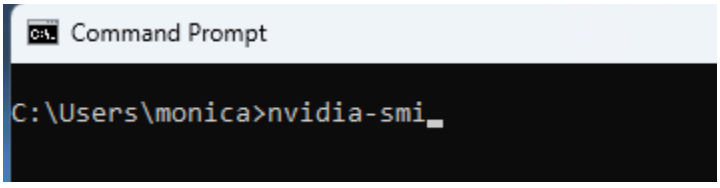
I began by installing the essential Python packages needed for YOLOv8 and object detection.



This installs YOLOv8 along with all necessary dependencies such as PyTorch, OpenCV, and other utilities required for training and evaluation.

4.2 Checking GPU Availability

Since training deep learning models performs better on GPUs, I verified that my system recognized the GPU. To check the GPU status, I used the following command in the terminal:



As shown below, the system successfully detected the NVIDIA GeForce RTX 3060 with 12 GB VRAM, running Driver Version 581.42 and CUDA Version 13.0. The GPU utilization was 0% at the time of checking, indicating that it was fully available for training.

```
C:\Users\monica>nvidia-smi
Tue Dec  9 16:23:19 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 581.42                 Driver Version: 581.42         CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3060      WDDM  |   00000000:01:00.0  On |                  N/A |
|  0%   31C    P8             14W /  170W |     774MiB /  12288MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A            364    C+G   ...t\Edge\Application\msedge.exe      N/A      |
|    0   N/A  N/A           1656    C+G   ...2txyewy\CrossDeviceResume.exe      N/A      |
|    0   N/A  N/A           5640    C+G   ...ntrolPanel\SystemSettings.exe      N/A      |
|    0   N/A  N/A           6792    C+G   C:\Windows\explorer.exe               N/A      |
|    0   N/A  N/A           9804    C+G   ...y\StartMenuExperienceHost.exe      N/A      |
|    0   N/A  N/A           9808    C+G   ..._cw5n1h2txyewy\SearchHost.exe      N/A      |
|    0   N/A  N/A          11656    C+G   ....0.3595.94\msedgewebview2.exe      N/A      |
|    0   N/A  N/A          12756    C+G   ...8bbwe\PhoneExperienceHost.exe      N/A      |
|    0   N/A  N/A          13204    C+G   ....0.3595.94\msedgewebview2.exe      N/A      |
|    0   N/A  N/A          13316    C+G   ...8wekyb3d8bbwe\M365Copilot.exe      N/A      |
|    0   N/A  N/A          15532    C+G   ...xyewy\ShellExperienceHost.exe      N/A      |
|    0   N/A  N/A          15660    C+G   ...(x86)\Notepad++\notepad++.exe      N/A      |
|    0   N/A  N/A          16104    C+G   ...Chrome\Application\chrome.exe      N/A      |
|    0   N/A  N/A          16160    C+G   ...Chrome\Application\chrome.exe      N/A      |
|    0   N/A  N/A          17376    C+G   ...crosoft OneDrive\OneDrive.exe      N/A      |
|    0   N/A  N/A          17784    C+G   ...em32\ApplicationFrameHost.exe      N/A      |
|    0   N/A  N/A          19260    C+G   ...indows\System32\ShellHost.exe      N/A      |
+-----------------------------------------------------------------------------------------+
```

5. Project directory structure

To set up the project environment, I created a folder on my desktop named Capstone, which served as the main project directory. Inside it, I created another folder called dataset, where the COCO dataset files (train, valid, and data.yaml) were stored.

```
C:\Users\monica>cd C:\Users\monica\Desktop\Capstone\datasets

C:\Users\monica\Desktop\Capstone\datasets>dir
 Volume in drive C has no label.
 Volume Serial Number is 126A-7C18

 Directory of C:\Users\monica\Desktop\Capstone\datasets

11/25/2025  12:06 PM    <DIR>          .
11/25/2025  12:39 PM    <DIR>          ..
11/25/2025  12:06 PM             1,121 data.yaml
11/25/2025  12:06 PM               673 README.dataset.txt
11/25/2025  12:06 PM               889 README.roboflow.txt
11/25/2025  12:43 PM    <DIR>          train
11/25/2025  12:44 PM    <DIR>          valid
               3 File(s)          2,683 bytes
               4 Dir(s)  15,918,940,160 bytes free

C:\Users\monica\Desktop\Capstone\datasets>
```

Navigating to the Correct Directory

Before starting training, I needed to make sure the YOLO command was run from the main project folder (Capstone), not from inside the datasets folder. I used the following command:

```
C:\Users\monica\Desktop\Capstone\datasets>cd..

C:\Users\monica\Desktop\Capstone>
```

The cd.. command moves one level up in the directory structure. After executing it, the terminal path was correctly updated. Running the training from the correct directory ensures that YOLOv8 can properly access the dataset path and save the model outputs in the right location.

6. YOLOv8 Training Setup

6.1 YOLOv8 Overview

YOLOv8 (You Only Look Once, version 8) is one of the latest and most efficient object detection architectures developed by Ultralytics. It is designed for real-time applications and supports both training from scratch and fine-tuning on custom datasets.

For this project, I used YOLOv8n (nano), the smallest and fastest variant of YOLOv8. It is lightweight, trains quickly, and is suitable for systems with limited GPU memory while still providing strong accuracy for general-purpose object detection tasks.

How YOLOv8 Learns?

For every batch of images, YOLOv8 performs:

1. Forward propagation
    - The input image is passed through the YOLO network.
    - The model predicts object classes and bounding boxes.
2. Loss calculation
   YOLOv8 uses three primary loss components:
    - Box loss – how accurate the bounding box locations are.
    - Classification loss – how accurately objects are classified.
    - DFL loss (Distribution Focal Loss) – improves bounding box precision.

3. Back propagation

    - The model adjusts its weights based on errors.

    - Loss values decrease as training progresses.

What is YOLOv8 originally trained on? Is COCO part of that pretraining dataset?

Yes, YOLOv8's object detection models are pretrained primarily on the COCO dataset. COCO is the main dataset used by Ultralytics to train the full YOLOv8 detection model.

YOLOv8's backbone may be initialized with ImageNet-pretrained weights, but the end-to-end detector is trained on COCO, not ImageNet.

In addition, Ultralytics sometimes incorporates extra COCO-style data and Roboflow Universe datasets during training to improve robustness, but COCO remains the core pretraining dataset.

Since YOLOv8 is pretrained on COCO:

1. Training becomes much faster, the model starts from a strong baseline instead of random weights.

2. Accuracy improves significantly, the model has already learned high-quality feature representations.

3. Transfer learning works correctly, the pretrained model features align perfectly with the dataset I used.

4. The model does not need huge epochs, because much of the learning already happened during pretraining.

6.2 Training Command

Once the environment was set up and the dataset was properly organized, I initiated the training process using YOLOv8. The training command was executed from the main project directory:

```
yolo detect train ^
 data="C:\Users\monica\Desktop\Capstone\datasets\data.yaml" ^
 model=yolov8n.pt ^
 epochs=50 ^
 imgsz=640 ^
 project="C:\Users\monica\Desktop\Capstone" ^
 name="my_yolo_model"
```

Explanation of Parameters

- data = "C:\Users\monica\Desktop\Capstone\datasets\data.yaml":

  Specifies the path to the data.yaml file, which defines class names and dataset locations.

- model=yolov8n.pt:

  Uses the YOLOv8 nano model.

- epochs=50:

  Trained the model for 50 cycles.

- imgsz=640:

  Resizes all images to 640×640 pixels, the recommended size for YOLOv8.

- project = "C:\Users\monica\Desktop\Capstone":

  Defines the folder where YOLO will store results (training logs, model weights, graphs).

- name="my_yolo_model":

  Creates a subfolder for this specific run, making it easy to track experiments.

6.3 Understanding Epochs

In machine learning, an epoch refers to one complete pass through the entire training dataset. During each epoch:

- The model sees every training image once.
- It learns patterns, adjusts weights, and reduces loss.
- Performance is evaluated on the validation set to track progress.

Training for multiple epochs helps the model gradually improve accuracy and generalization.

Why I Chose 50 Epoch

Choosing the number of epochs is important because:

- Too few epochs → the model underfits (does not learn enough).
- Too many epochs → the model overfits (memorizes the training data instead of generalizing).

For this project, 50 epochs was chosen for the following reasons:

- Balanced Training Time

  Training on the RTX 3060 GPU takes a noticeable amount of time.
  50 epochs provide enough learning without making training excessively long.

- Good Accuracy Without Overtraining

YOLOv8 typically reaches strong performance between 30–80 epochs for medium-sized datasets. 50 epochs is a widely recommended default for general COCO-based training.

- Validation Curves Usually Stabilize Around 40–50 Epochs

  Most YOLO models show decreasing loss and increasing mAP until around 40–50 epochs, after which improvements slow down.

In-depth understanding of the Loss Functions:

Box Loss: Measures how accurately the model predicts the position and size of objects in an image.

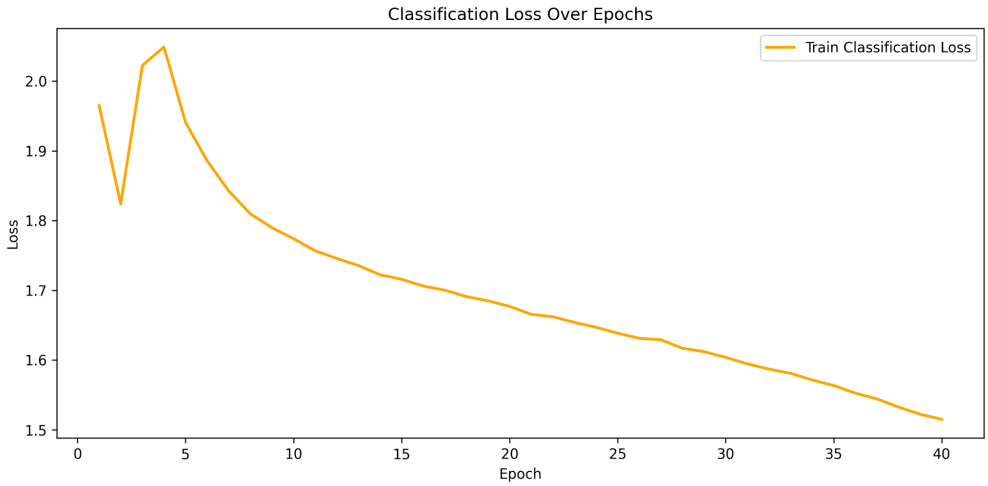
If the predicted box does not overlap well with the actual object, this loss increases.

- High box loss: boxes are inaccurate or poorly placed
- Low box loss: boxes match the objects closely.

Classification Loss: measures how accurately the model predicts what the object is (e.g., person, dog, car).

If the model predicts the wrong class or has low confidence in the correct class, this loss increases.

- High classification loss: many incorrect labels.
- Lower classification loss: better understanding of object categories.

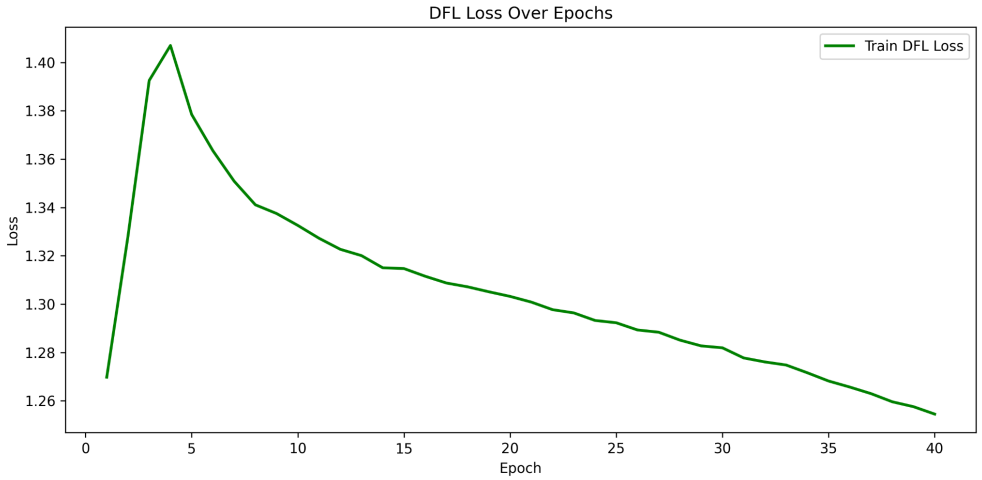


As we can see, the classification loss decreased throughout training, demonstrating that the model improved its ability to correctly identify object categories. The gradual slope indicates steady and reliable learning.

DFL Loss (Distribution Focal Loss): used to improve the precision of bounding box boundaries.
It helps the model refine the edges of boxes so they fit objects more tightly.

We can think of DFL as the component that makes the bounding boxes:

- sharper
- more accurate
- better aligned with object edges



A gradual decrease in DFL loss indicates that the model is improving in predicting fine-grained details of object locations.

To generate the graphs above, the following code was used:

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("results.csv")

# Boss Loss
plt.figure(figsize=(10,5))
plt.plot(df["epoch"], df["train/box_loss"], label="Train Box Loss", linewidth=2)
plt.plot(df["epoch"], df["val/box_loss"], label="Validation Box Loss", linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Box Loss Over Epochs")
plt.legend()
plt.tight_layout()
plt.savefig("box_loss_curve.png", dpi=300, bbox_inches="tight")
plt.show()

# Classification Loss
plt.figure(figsize=(10,5))
plt.plot(df["epoch"], df["train/cls_loss"], label="Train Classification Loss", color="orange", linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Classification Loss Over Epochs")
plt.legend()
plt.tight_layout()
plt.savefig("classification_loss_curve.png", dpi=300, bbox_inches="tight")
plt.show()

#DFL Loss
plt.figure(figsize=(10,5))
plt.plot(df["epoch"], df["train/dfl_loss"], label="Train DFL Loss", color="green", linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("DFL Loss Over Epochs")
plt.legend()
plt.tight_layout()
plt.savefig("dfl_loss_curve.png", dpi=300, bbox_inches="tight")
plt.show()
```

7. Model Training Process

When the training command was executed, YOLOv8 automatically began processing the dataset and optimizing the neural network.

7.1 Training and Validation Split

YOLOv8 reads the paths from data.yaml and loads:

- Training images from the train/ folder.
- Validation images from the valid/ folder.

During each epoch:

- The model learns from the training set.
- The model is evaluated on the validation set.
- Performance metrics are updated in real time.

7.2 Metrics Tracked During Training

YOLOv8 automatically logs several key performance metrics:

- **mAP50:** Mean Average Precision at IoU 0.50.
- **mAP50–95:** Average precision across multiple IoU thresholds.
- **Precision:** Measures how many detected objects were correct.
- **Recall:** Measures how many real objects were detected.
- **Training loss:** Should decrease with each epoch.
- **Validation loss:** Helps detect overfitting.

These metrics help evaluate how well the model is learning.

7.3 System-Generated Files

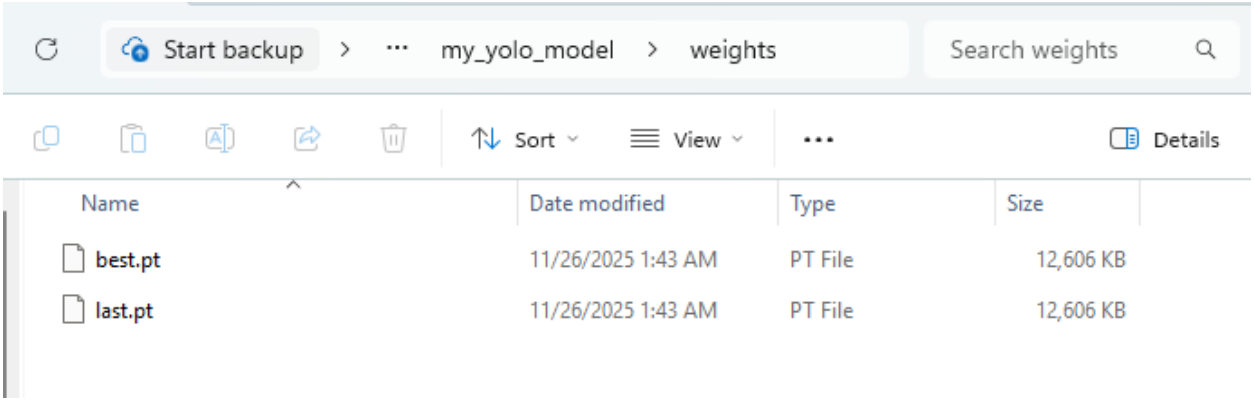
Saving Model Weights

YOLOv8 saves two important checkpoint files:

- best.pt
  The model with the highest validation mAP during training.

- last.pt
  The final model after the last epoch.

These files are used later for evaluation and real-time testing.



Output Files Generated

After training, YOLOv8 automatically generates:

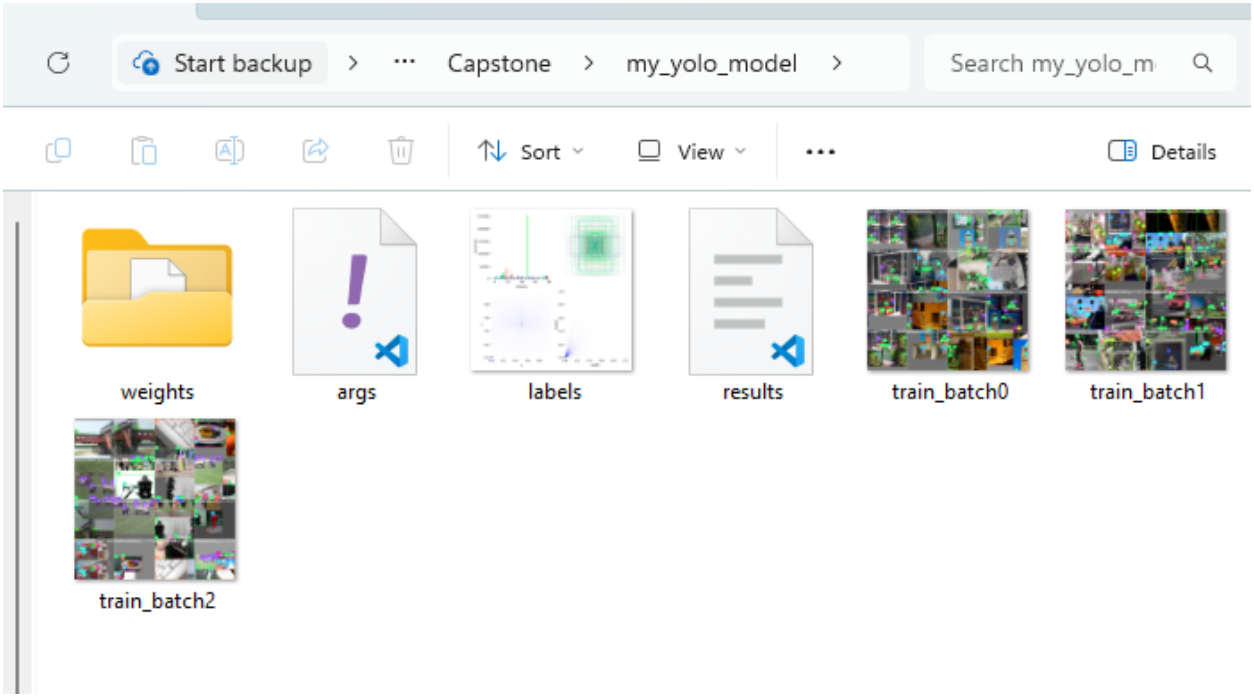
- **weights/** – contains best.pt and last.pt, the saved model weights.

- **results.png** – a summary graph showing training loss, validation loss, mAP50, and mAP50-95 over all epochs.

- **confusion_matrix.png** – shows how accurately each class was detected and how often classes were confused.

- **precision-recall curve** – evaluates detection quality at different confidence thresholds.
- **labels.png / labels.jpg** – visualization of dataset label distribution.
- **train_batch0.jpg, train_batch1.jpg, train_batch2.jpg** – sample training batches showing images with bounding boxes during training.
- **args.yaml** – contains the exact training parameters used, ensuring the experiment is reproducible.

These outputs make it easier to interpret the training process, understand model performance, and validate that the dataset and annotations were correctly processed.



8. Results and Performance Analysis

After completing training, YOLOv8 generated a results.csv file containing detailed metrics for each epoch, including training loss, validation loss, precision, recall, and mean Average Precision (mAP). To better visualize and interpret these results, I used Python (Pandas and Matplotlib) to generate performance graphs from this CSV file.

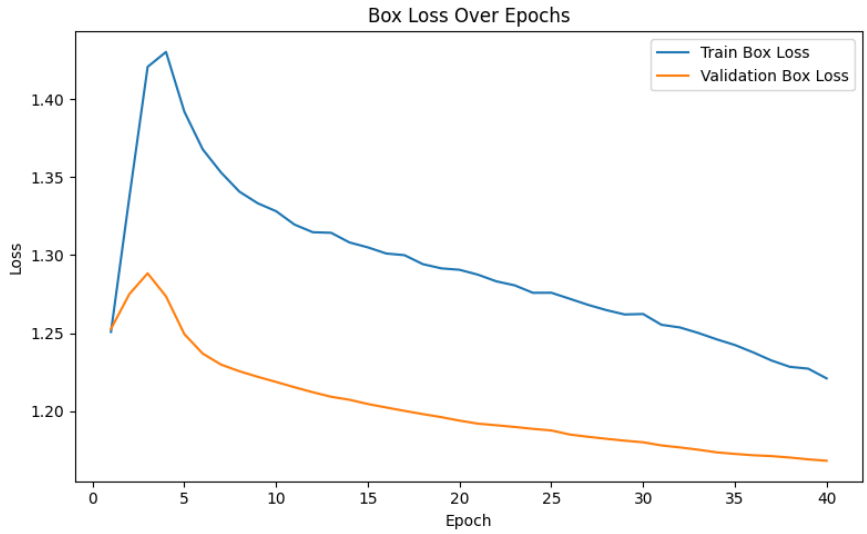
These visualizations help clearly show how the model improved during training and whether it converged properly.

The script used to generate these graph is as follows:

e.log | sample.py | training | plots.py

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("results.csv")

# Box Loss Curve
plt.figure(figsize=(8,5))
plt.plot(df['epoch'], df['train/box_loss'], label='Train Box Loss')
plt.plot(df['epoch'], df['val/box_loss'], label='Validation Box Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Box Loss Over Epochs')
plt.legend()
plt.tight_layout()
plt.savefig("box_loss_curve.png")
plt.close()

# mAP Curve
plt.figure(figsize=(8,5))
plt.plot(df['epoch'], df['metrics/mAP50(B)'], label='mAP50')
plt.plot(df['epoch'], df['metrics/mAP50-95(B)'], label='mAP50-95')
plt.xlabel('Epoch')
plt.ylabel('mAP')
plt.title('mAP Over Epochs')
plt.legend()
plt.tight_layout()
plt.savefig("map_curve.png")
plt.close()

# Precision-Recall Curve
plt.figure(figsize=(8,5))
plt.plot(df['epoch'], df['metrics/precision(B)'], label='Precision')
plt.plot(df['epoch'], df['metrics/recall(B)'], label='Recall')
plt.xlabel('Epoch')
plt.ylabel('Score')
plt.title('Precision and Recall Over Epochs')
plt.legend()
plt.tight_layout()
plt.savefig("precision_recall_curve.png")
plt.close()

print("Graphs generated successfully!")
```

8.1 Box Loss Over Epochs



The box loss measures how accurately the model predicts bounding box locations.

- Training box loss decreased consistently across epochs.
- Validation box loss followed a similar downward trend without major spikes.

This pattern indicates that the model was learning effectively and did not suffer from overfitting.

8.2 mAP (Mean Average Precision) Over Epochs

mAP is the most important metric in object detection. It measures how well the model detects and classifies objects.

It combines both:

- Localization accuracy: how well the bounding box fits the object.
- Classification accuracy: whether the predicted label is correct.

mAP50: is the mean average precision at IoU = 0.50.

- IoU (Intersection over Union) measures how much our predicted bounding box overlaps with the ground-truth box.
- If IoU ≥ 0.50, the prediction is counted as a correct detection.

A ground truth box is the manually annotated bounding box that represents the correct location and class of an object, used as the reference for training and evaluating object detection models.

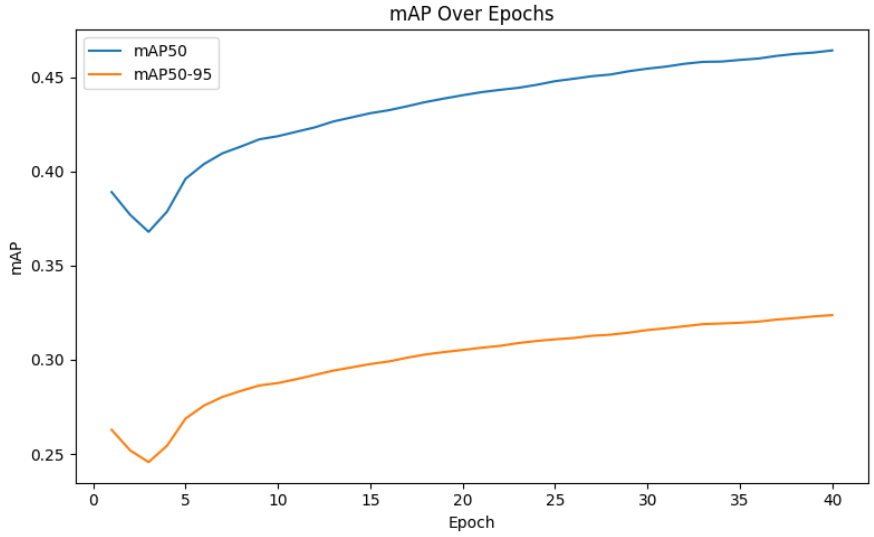
mAP50–95: averages the mAP values across 10 different IoU thresholds:

0.50, 0.55, 0.60, 0.65, … , 0.95.

This means the model is tested at:

- loose overlap (IoU 0.50)
- moderate overlap (IoU 0.70)
- very strict overlap (IoU 0.95)

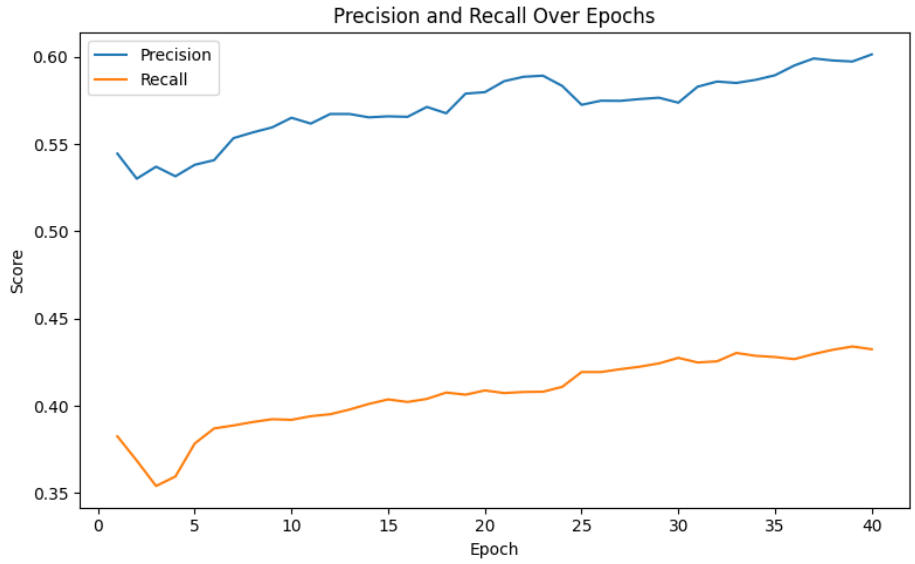
It tell us, “How well the model performs overall across easy, medium, and very strict localization requirements.”



mAP50 (IoU = 0.50) improved steadily from ~0.38 to ~0.46.
mAP50–95 increased from ~0.25 to ~0.32.

This shows that the model continued learning and achieved its highest performance at Epoch 40, where YOLO saved the best.pt file.

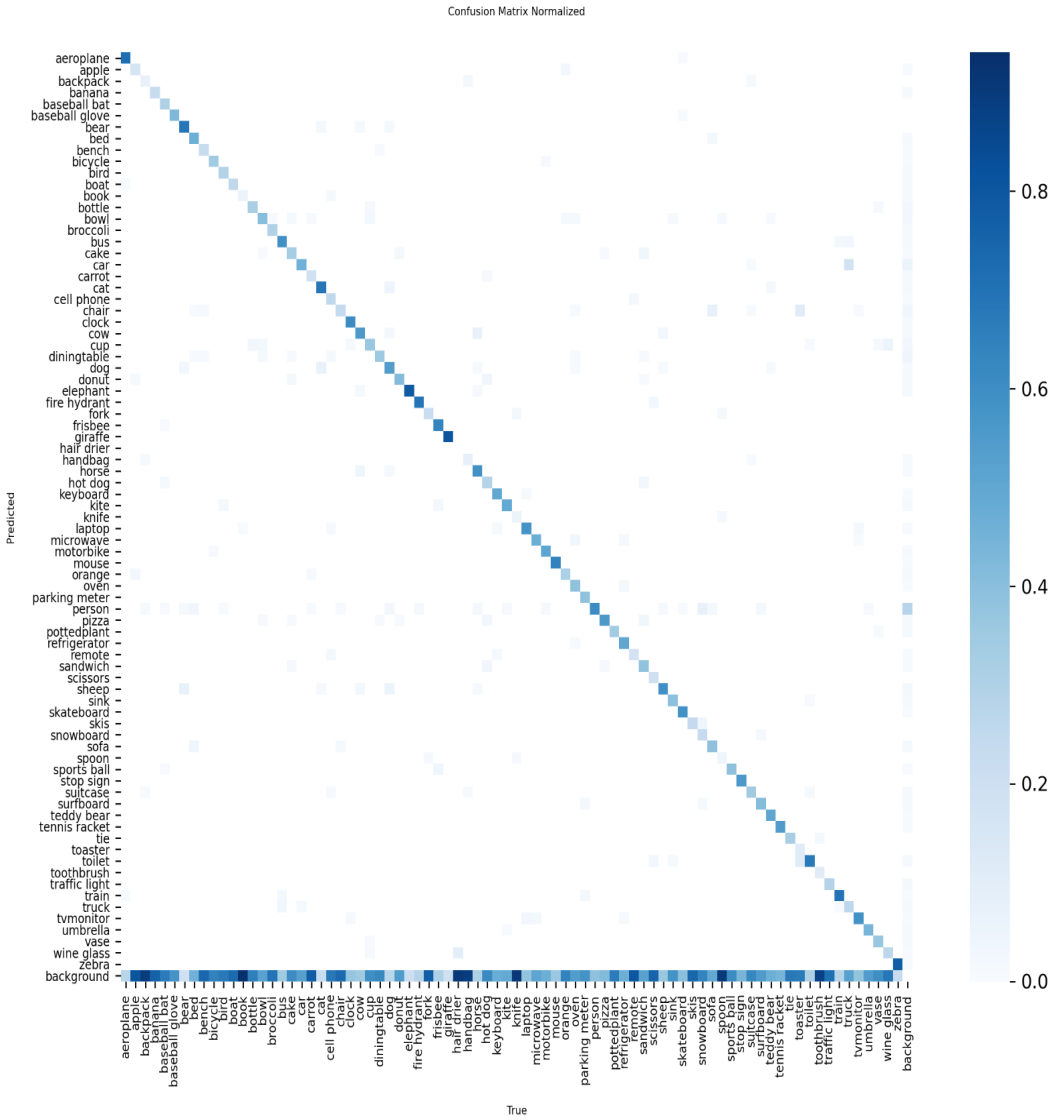
8.3 Precision and Recall Over Epochs



Precision increased throughout training, meaning the model produced fewer false positives.

Recall also improved, indicating it detected more of the actual objects in the images.

What Classes did the Model perform best/worst on?



To evaluate class-wise performance of my YOLOv8 model, I used the normalized confusion matrix generated during validation. In a normalized matrix, each row represents a ground-truth class and each column represents the predicted class, with values scaled between 0 and 1. The diagonal values indicate correct predictions, while off-diagonal values show misclassifications.

The confusion matrix for my model shows a strong and clear diagonal, meaning the model correctly identifies the majority of objects it detects. The strongest diagonal intensities occur for highly frequent COCO classes such as person, car, dog, cat, bottle, and cup. These classes achieve better accuracy because they appear more often in the dataset and have well-defined visual features that YOLOv8 learns easily.

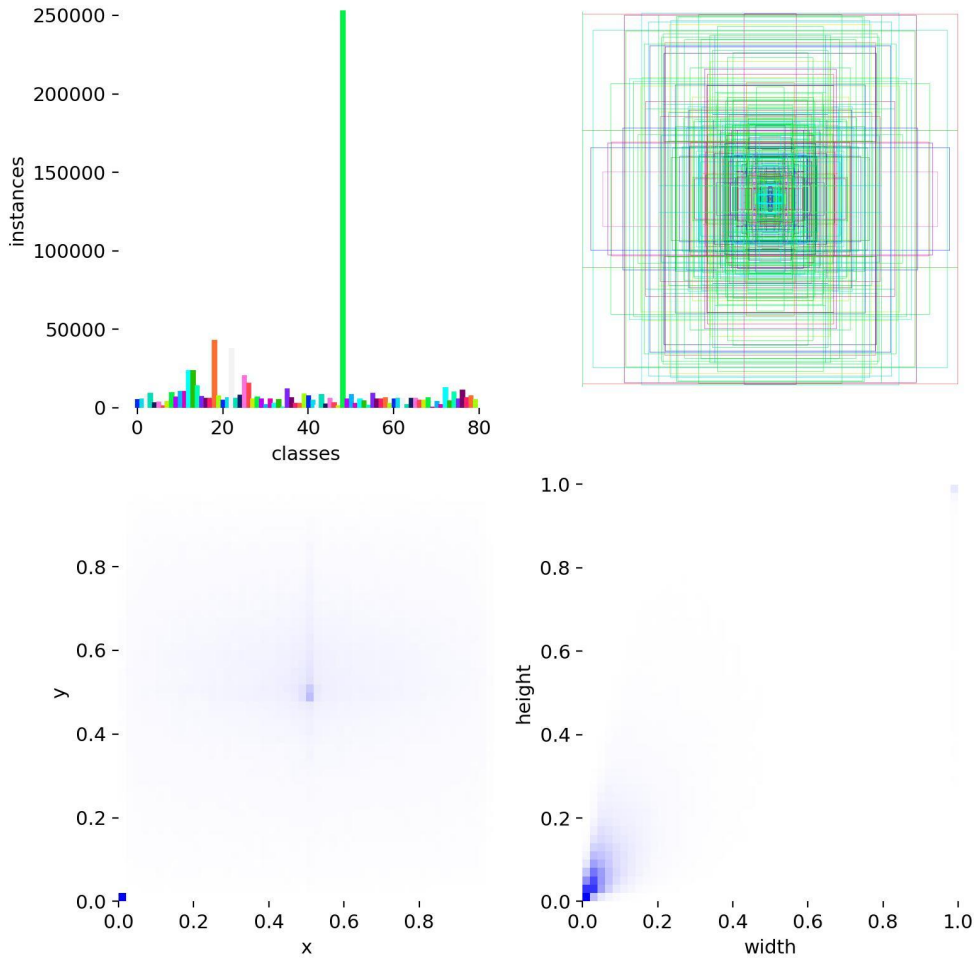
In contrast, classes such as hair drier, toaster, parking meter, snowboards, skis, and baseball gloves exhibit faint diagonal values, indicating weaker performance. These classes are rare in COCO and often visually ambiguous or very small within images, which makes learning more difficult for the model.

Overall, the model performs best on common classes and struggles on rarely occurring classes.
Due to its size, the confusion matrix is hard to view in the report; therefore, a link to the PNG version is provided as a reference:

confusion_matrix_normalized.png

9. Dataset Label Visualization

YOLOv8 automatically generates a labels.jpg file that provides a visual summary of the dataset's annotation patterns. This visualization is divided into four sections, each representing an important aspect of the dataset used for training.



1. Class Distribution (Top-Left Plot)

This bar chart shows the number of annotated instances for each of the 80 COCO object classes.

- Some classes (e.g., *person*) appear very frequently, which is expected because COCO contains many human-centered images.

- Other classes have fewer instances.

This imbalance is normal for COCO and helps explain why certain classes train faster or achieve higher accuracy.

2. Bounding Box Shape Distribution (Top-Right Plot)

This plot overlays bounding boxes from the entire dataset.

- The overlapping multi-colored rectangles show the variation in object sizes and positions.

- Large objects (e.g., cars, people) and small objects (e.g., phones, utensils) are represented.

This helps visualize the diversity of object sizes that the model must learn to detect.

3. Bounding Box Center Heatmap (Bottom-Left Plot)

This heatmap shows where bounding box centers most commonly appear in images.

- Brighter areas indicate more frequent bounding box centers.

- COCO tends to have many objects near the center of the image, so the center region is denser.

This helps illustrate the spatial distribution of objects in the dataset.

4. Bounding Box Size Distribution (Bottom-Right Plot)

This heatmap shows the distribution of bounding box widths and heights.

- Smaller objects are more common (bottom-left cluster).
- Larger bounding boxes also appear but less frequently.

This distribution explains why the model must learn to detect both tiny objects (like utensils) and very large ones (like buses).

10. Image Testing

To evaluate the performance of the trained YOLOv8 model outside of the training environment, I performed inference on a set of sample images. These images were placed in a folder named test on my desktop, containing various real-world images downloaded from the web.

Using the best.pt model generated, I ran YOLOv8 inference on all images in the test folder with the following script:

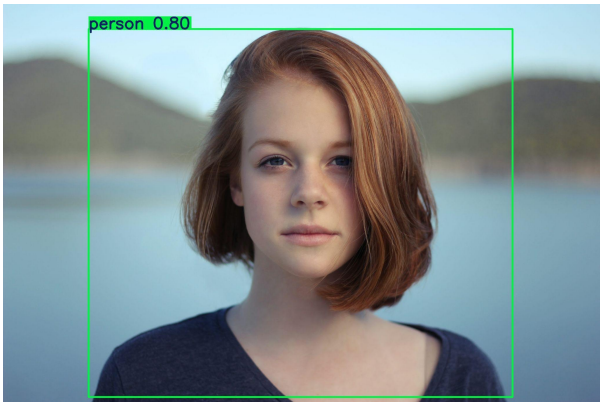
e.log | sample.py | training | plots.py | test.py

```python
from ultralytics import YOLO

# Loading the trained model
model = YOLO("my_yolo_model/weights/best.pt")

#detecting on the test folder
model.predict(source="C:/Users/monica/Desktop/test", save=True)

print("Done")
```

This command processes each image inside the test directory and automatically saves the prediction outputs, complete with bounding boxes, class labels, and confidence scores into the YOLO-generated folder. Sample prediction images are included below:


suitcase 0.50


umbrella 0.50


person 0.80


fire hydrant 0.94


traffic light 0.73
bus 0.84
SCHOOL BUS
STOP
135
CHAMPAIGN COMMUNITY UNIT SCHOOL DISTRICT NO. 4


pottedplant 0.38
pottedplant 0.33
chair 0.75
vase 0.61
0.53
chair 0.95
diningtable 0.69
chair 0.91

11. Real-Time Webcam Testing

To evaluate the model's performance in a live environment, I tested the trained YOLOv8 model using the system's webcam. This step is important because the goal of the project is to build an AI Smart Camera System capable of detecting objects continuously in real time. I used the following code to test:

```
Last login: Thu Dec  4 12:05:35 on console
(base) monicabhandari@Mac ~ % yolo predict model="/Users/monicabhandari/Desktop/best.pt" source=0 show=True
```

Here:

- model specifies the path to the trained weights (best.pt),
- source=0, opens the default webcam,
- show=True, displays the detection window in real time.

Once executed, YOLOv8 processed each video frame and displayed bounding boxes, object labels, and confidence scores directly on the live feed. The results generated were as follows:

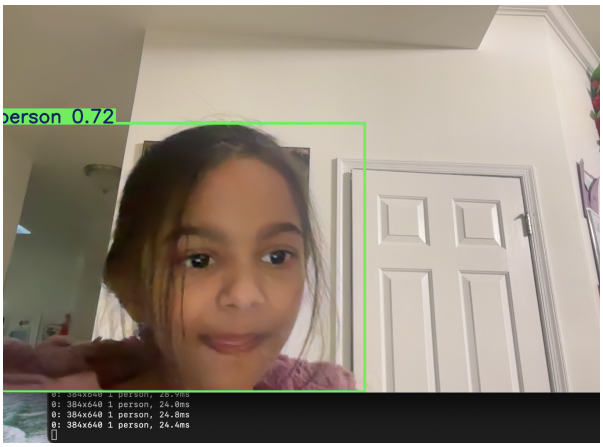


Fig: YOLOv8 detecting multiple objects in a living room.

12. Conclusion

This project successfully trained and evaluated a YOLOv8 object detection model using the COCO dataset. The model showed steady improvement during training, and the best checkpoint was saved at Epoch 40. Testing on new images and real-time webcam input confirmed that the system can accurately detect multiple object classes. Overall, the project demonstrates a working AI Smart Camera System capable of performing real-time object detection.