B. Tech Artificial Intelligence and Data Science Students
Madras Institute of Technology Campus - Anna University

Vehicle Detection using You Only Look Once (YOLO) Model

Project Documentation by Team 02

Project Phase 2 - 11.07.2024

Team 02 members

Srilakshmi H - 2022510053
Tejaswini K - 2022510010
Sanjana S - 2022510008
Kaviya R - 2022510001

TABLE OF CONTENTS

Executive Summary

In this project, a YOLO (You Only Look Once) model was employed for vehicle detection, leveraging the power of deep learning and computer vision to identify various types of vehicles in traffic images. The dataset comprised 2704 training images and corresponding labels, containing annotations for 21 vehicle classes including cars, buses, motorcycles, and ambulances. The YOLOv8 model, pre-trained on the COCO dataset, was fine-tuned on the custom dataset. This involved specifying a custom YAML configuration file, which defined the dataset paths and vehicle classes. The training process spanned 10 epochs, with metrics such as box loss, class loss, and distance focal loss (DFL) monitored to gauge the model's performance. Training and validation precision, recall, mean average precision (mAP), and other key metrics were tracked and visualized using seaborn and matplotlib to ensure an in-depth analysis of the model's learning curve. To evaluate the model, the trained YOLO model was used to predict and visualize detections on a set of validation images, with bounding boxes accurately highlighting the detected vehicles. A significant aspect of this project was the incorporation of real-time vehicle detection using an IP webcam app and a multi-camera setup. This real-time detection system allowed for live monitoring and analysis of traffic from multiple viewpoints, demonstrating the practical application of the model in dynamic, real-world environments. The real-time setup was achieved by streaming video feeds from multiple cameras, processed by the trained YOLO model to detect and annotate vehicles in real-time. This comprehensive approach to vehicle detection using the YOLO model underscores its robustness and adaptability for real-world traffic surveillance and analysis applications, highlighting its potential for enhancing traffic management systems and contributing to smart city infrastructure.

Project Environment Overview

Working Environment	
Operating System	Windows 11
IDE	Jupyter Notebook or Google Colab
Programming Language	
Python	3.11.4

Software and Hardware Specifications

Software Specifications	
Python	Version 3.11.4 or later
IDE	Anaconda Navigator (Jupyter Notebook) or Google Colab
Necessary Libraries and Modules	Matplotlib, Seaborn, Scikit-learn, Pillow, OpenCV 4.5 or later, Ultralytics, Torch, Torchvision
Deep Learning Framework	PyTorch
Webcam Streaming Software	IP Webcam app (Android)

Hardware Specifications	
RAM	Minimum 16 GB of RAM
Processor	Intel i5 or equivalent
Storage	SSD with at least 100 GB free space for datasets and models
A Stable Internet Connection,	for streaming video feeds from IP cameras

---

You Only Look Once (YOLO) Model

The You Only Look Once (YOLO) model is a state-of-the-art, real-time object detection system that frames object detection as a single regression problem. Unlike traditional methods that apply a model to an image at multiple locations and scales, YOLO applies a single neural network to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region.

1. Architecture Overview

- YOLO is based on a single Convolutional Neural Network (CNN) that divides the input image into an S x S grid. Each grid cell predicts B bounding boxes and C class probabilities. The predictions are encoded as a tensor of dimensions S x S x (B * 5 + C), where 5 represents the four coordinates of the bounding box and one confidence score, and C is the number of classes.

2. Key Components and Concepts

a. Bounding Box Prediction

Each grid cell in the YOLO network predicts B bounding boxes, where each box is defined by:

- **(x, y):** The coordinates of the box center relative to the bounds of the grid cell.
- **(w, h):** The width and height of the box, relative to the whole image.
- **Confidence Score:** A score reflecting the confidence that the box contains an object and how accurate the box is.

b. Class Prediction

Along with bounding boxes, each grid cell predicts a class probability for each of the C classes, which represents the probability that the detected object belongs to a particular class.

c. Confidence Score Calculation

The confidence score for each bounding box is calculated as the product of the probability of the object being in the box and the Intersection over Union (IoU) between the predicted box and the ground truth box. The formula is:

Confidence = P(Object)×IoU

3. Training Process

a. Loss Function

The YOLO model uses a loss function that combines multiple aspects:

- **Localization Loss:** Measures the error in the predicted bounding box coordinates.
- **Confidence Loss:** Measures the error in the confidence score.
- **Class Probability Loss:** Measures the error in the predicted class probabilities.

The loss function penalizes the errors in localization more than classification errors, ensuring that the bounding boxes are accurate.

b. Non-Maximum Suppression (NMS)

During inference, YOLO uses Non-Maximum Suppression to eliminate redundant bounding boxes. NMS works by first selecting the box with the highest confidence score and then removing any overlapping boxes with a lower confidence score, based on a threshold.

How Does YOLO Object Detection Work?

The algorithm works based on the following four approaches:

- Residual blocks
- Bounding box regression
- Intersection Over Unions or IOU for short
- Non-Maximum Suppression.

1. Residual blocks

- This first step starts by dividing the original image (A) into NxN grid cells of equal shape, where N in our case is 4 shown on the image on the right.
- Each cell in the grid is responsible for localizing and predicting the class of the object that it covers, along with the probability/confidence value.

2. Bounding box regression

- The next step is to determine the bounding boxes which correspond to rectangles highlighting all the objects in the image.
- We can have as many bounding boxes as there are objects within a given image. YOLO determines the attributes of these bounding boxes using a single regression module in the following format, where Y is the final vector representation for each bounding box.

Y = [pc, bx, by, bh, bw, c1, c2]

This is especially important during the training phase of the model.

- pc corresponds to the probability score of the grid containing an object. For instance, all the grids in red will have a probability score higher than zero. The image on the right is the simplified version since the probability of each yellow cell is zero (insignificant).
- bx, by are the x and y coordinates of the center of the bounding box with respect to the enveloping grid cell.
- bh, bw correspond to the height and the width of the bounding box with respect to the enveloping grid cell.
- c1 and c2 correspond to the two classes Player and Ball. We can have as many classes as your use case requires.

3. Intersection Over Unions or IOU

Most of the time, a single object in an image can have multiple grid box candidates for prediction, even though not all of them are relevant. The goal of the IOU (a value between 0 and 1) is to discard such grid boxes to only keep those that are relevant. The logic behind it is,

- The user defines its IOU selection threshold, which can be, for instance, 0.5.
- Then YOLO computes the IOU of each grid cell which is the Intersection area divided by the Union Area.
- Finally, it ignores the prediction of the grid cells having an IOU ≤ threshold and considers those with an IOU > threshold.

4. Non-Max Suppression or NMS

- Setting a threshold for the IOU is not always enough because an object can have multiple boxes with IOU beyond the threshold, and leaving all those boxes might include noise.
- Here is where we can use NMS to keep only the boxes with the highest probability score of detection.

Advantages of YOLO

a. Speed

YOLO is exceptionally fast because it treats object detection as a single regression problem rather than a classification problem. This allows it to process images in real-time, making it suitable for applications that require quick detection, such as video surveillance and autonomous driving.

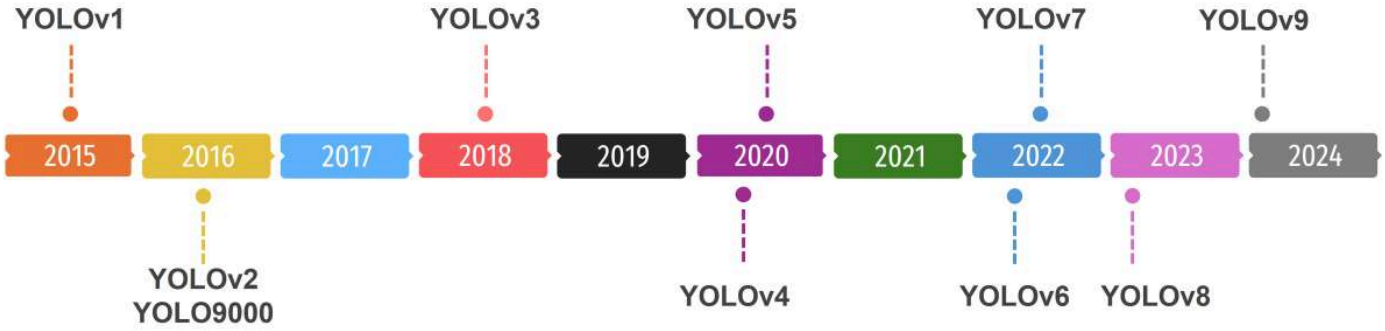
b. Global Context Understanding

Since YOLO looks at the entire image during training and inference, it implicitly encodes contextual information about classes and their appearances. This holistic view reduces the number of false positives in detection.

c. Unified Architecture

YOLO's single neural network architecture for both detection and classification simplifies the deployment and maintenance of the model. It is also more straightforward to optimize compared to multi-stage detection systems.

YOLO Timeline



1. YOLOv1 (2015)

Introduced by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi in their paper **"You Only Look Once: Unified, Real-Time Object Detection"** YOLOv1 was groundbreaking for its novel approach of **treating object detection as a single regression problem**, directly predicting bounding boxes and class probabilities from full images in one evaluation. This approach significantly increased the speed of object detection, making real-time processing possible.

2. YOLOv2 (YOLO9000, 2016)

In their paper **"YOLO9000: Better, Faster, Stronger"** Redmon and Farhadi improved YOLO's speed and accuracy. YOLOv2 introduced various concepts, such as **batch normalization**, **high-resolution classifiers**, and a new network architecture called **Darknet-19.** It also introduced the concept of **anchor boxes** to predict more accurate bounding boxes. YOLOv2 was notable for its ability to detect over 9000 object categories by jointly training on both detection and classification datasets.

3. YOLOv3 (2018)

With the paper **"YOLOv3: An Incremental Improvement"** Redmon and Farhadi presented further improvements. YOLOv3 incorporated several enhancements, such as using a deeper network architecture called Darknet-53, employing multi-scale predictions, and improving the detection of smaller objects. Despite these advancements, it managed to maintain high speed and accuracy.

4. YOLOv4 (2020)

Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao released YOLOv4 in the paper **"YOLOv4: Optimal Speed and Accuracy of Object Detection"** YOLOv4 focused

on **optimizing the speed and accuracy** further, making it accessible for a wider range of devices, including those with limited computational power. It introduced improvements like the use of the **CSPDarknet53 backbone**, **spatial pyramid pooling**, and **PANet path-aggregation.** YOLOv4 was also designed to be more user-friendly in terms of training and deploying on different platforms.

5. YOLOv5 (2020)

Despite the naming convention, YOLOv5 is not an official continuation by the original authors but was developed and released by Ultralytics. It is a significant departure in terms of framework, being **implemented in PyTorch** instead of Darknet. It introduced several improvements and optimizations over YOLOv4, including **model scalability** (with versions from YOLOv5s to YOLOv5x), **automated hyperparameter tuning**, and enhanced training procedures.

6. YOLOv6 (2022)

YOLOv6, introduced in 2022 by Li et al., represents a notable advancement over its predecessors. This iteration distinguishes itself from YOLOv5 primarily through its underlying convolutional neural network (CNN) architecture. Opting for a variant of the **EfficientNet architecture**, known as **EfficientNet-L2**, YOLOv6 achieves a balance between efficiency and performance that surpasses the EfficientDet architecture utilized in YOLOv5. With a reduction in parameters and enhanced computational efficiency, YOLOv6 sets new benchmarks in object detection performance across a variety of standard tests.

7. YOLOv7 (2022)

YOLO v7 introduces significant advancements in real-time object detection, achieving state-of-the-art performance in both speed and accuracy across a wide range of FPS (5 to 160 FPS) on GPU V100. It boasts the **highest accuracy among real-time detectors** with 30 FPS or higher, outperforming both transformer-based and convolutional-based detectors in speed and accuracy. YOLOv7's design focuses on optimizing the training process without increasing inference cost, introducing trainable bag-of-freebies methods for enhanced detection accuracy. It effectively reduces parameters and computational requirements compared to prior models, maintaining fast inference speeds and high detection precision without additional datasets or pre-trained weights.

8. YOLOv8 (2023)

YOLO v8 marks a significant evolution in the YOLO series, expanding its capabilities to include object detection, image classification, and instance segmentation. Noteworthy for its precision and streamlined model size, YOLO v8 introduces **anchor-free detection**, eliminating the need for anchor boxes by predicting object centers. This advancement simplifies the model architecture and enhances efficiency in post-processing tasks like Non-Maximum Suppression. The model architecture draws inspiration from **ResNet**, featuring novel convolution types and module configurations, enhancing its usability and effectiveness in computer vision projects.

9. YOLOv9 (2024)

YOLOv9 represents the most current and advanced model in the YOLO series, setting a new benchmark for state-of-the-art (SOTA) performance in object detection. YOLO v9 introduces **Programmable Gradient Information (PGI)** and a new network architecture called **Generalized Efficient Layer Aggregation Network (GELAN)**, aimed at overcoming data loss in deep networks. PGI allows for tailored gradient information, ensuring complete input data utilization for target tasks, which leads to more reliable gradient updates. GELAN, based on gradient path planning, focuses on **parameter efficiency** and **computational simplicity**, outperforming previous methods in parameter utilization and supporting a wide range of models. These innovations significantly enhance object detection performance.

Working of YOLOv8n Model

The YOLOv8n (You Only Look Once version 8, nano) model is part of the YOLO series, designed to be a smaller and faster variant of the YOLOv8 model. It retains the core principles of YOLO while optimizing for efficiency and speed.

Working:

1. Architecture Overview

- **Backbone Network:** YOLOv8n utilizes a lightweight backbone network, often based on a reduced version of CSPNet (Cross-Stage Partial Network) or MobileNet-like architectures. This backbone extracts feature maps from the input image efficiently.

- **Neck:** The model incorporates a neck component, typically consisting of PANet (Path Aggregation Network) or similar structures. The neck aggregates feature maps from different layers to improve object detection across various scales.
- **Head:** YOLOv8n uses a detection head that outputs bounding boxes, class probabilities, and objectness scores. The head is designed to be lightweight to ensure real-time performance.

2. Feature Extraction

- The backbone network processes the input image through multiple convolutional layers, extracting hierarchical features. These features represent various aspects of the image, from low-level details to high-level patterns.

3. Feature Fusion

- The neck component combines features from different layers of the backbone. This fusion allows the model to capture both fine details and broader context, enhancing the detection of objects at different scales and resolutions.

4. Prediction Head

- The head of YOLOv8n generates predictions for each anchor box across different grid cells. Each prediction includes:

- **Bounding Box Coordinates:** These specify the location of the detected object.
- **Objectness Score:** Indicates the probability that a box contains an object.
- **Class Scores:** Probabilities for each class, indicating the likelihood of the object belonging to each class.

5. Anchor Boxes

- YOLOv8n employs predefined anchor boxes of various aspect ratios and sizes to predict the bounding boxes. These anchors help the model to detect objects with different shapes and sizes.

6. Loss Function

- The model is trained using a loss function that combines several components:

- **Localization Loss:** Measures the accuracy of the bounding box coordinates.
- **Objectness Loss:** Assesses the confidence of the objectness score.

- **Classification Loss:** Evaluates the accuracy of the class predictions.

7. Inference

- During inference, YOLOv8n processes an input image through its network and generates bounding boxes, objectness scores, and class predictions. Post-processing techniques, such as Non-Maximum Suppression (NMS), are used to filter out overlapping boxes and finalize the detection results.

Key Features:

- **Compact Architecture:** YOLOv8n is optimized for low-latency applications, making it suitable for real-time object detection on devices with limited computational resources.
- **Speed and Efficiency:** Its design focuses on achieving high-speed inference while maintaining reasonable accuracy, ideal for applications requiring fast detection.
- **Flexibility:** YOLOv8n can be adapted for various use cases, from mobile and edge devices to scenarios with constrained resources.
- YOLOv8n combines a compact and efficient network architecture with advanced feature extraction and prediction techniques, ensuring it performs well in real-time object detection tasks while being resource-efficient.

Our project setup guide

1. Open Jupyter Notebook:

- Launch Jupyter Notebook on your local machine or a cloud platform. This provides an interactive environment for coding and running Python scripts.

2. Import Libraries:

- Import necessary libraries at the beginning of your notebook to enable YOLOv8 model support and other functionalities you may need. For YOLOv8, you typically import ultralytics.

3. Download Dataset:

- Download your image dataset directly into the Jupyter Notebook environment. This can be done by uploading files from your local machine.

4. Load Dataset:

- Once downloaded, load the dataset into the notebook. Ensure your dataset is organized properly, with images and corresponding annotations (if applicable). Annotations are often in YOLO format, which includes .txt files with bounding box coordinates for each image.

5. Setup YOLOv8 Model:

- Configure the YOLOv8 model within the notebook environment. This involves:
    - Specifying the YOLOv8 model variant (yolov8n.pt.) that best suits your computational and accuracy requirements.
    - Initializing the YOLOv8 model using the ultralytics.YOLO class, which provides methods for training, evaluation, and inference.

6. Training and Saving the Model:

- Train the YOLOv8 model on your loaded dataset. This step involves:
    - Setting up the training process with parameters such as batch size, learning rate, and number of epochs.
    - Calling the .train() method on your YOLOv8 model instance, passing in paths to your training and validation datasets.
    - After training, save the trained model weights (yolov8n.pt.) for future use or deployment.

7. Prediction (Inference):

- Use the trained YOLOv8 model to make predictions (perform object detection) on new images or videos. This step includes:
    - Loading the saved model weights into your YOLOv8 instance.
    - Calling the .predict() or .infer() methods on your YOLOv8 model instance, passing in new images or videos to detect objects.
    - Visualizing or analyzing the detection results to assess the model's performance and accuracy in real-world scenarios.

Dataset Overview

The dataset is organized into the following directories:

- **train/images:** Contains the training images.
- **train/labels:** Contains the corresponding label files in YOLO format, which detail the bounding boxes and class IDs of the objects in the images.

- **valid/images:** Contains the validation images used to test the model's performance.

Label Format:

Each label file follows the YOLO format and includes the following fields:

- **class_id:** The ID of the class (object type).
- **x_center:** The x-coordinate of the center of the bounding box.
- **y_center:** The y-coordinate of the center of the bounding box.
- **width:** The width of the bounding box.
- **height:** The height of the bounding box.

Source Code

1. Importing the Libraries

```
import os
import cv2
import random
import pandas as pd
import seaborn as sns
from PIL import Image
from ultralytics import YOLO
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from deep_sort_realtime.deepsort_tracker import DeepSort
```

---

2. Importing the training images and labels

```
train_images =
r'C:\Users\SriLakshmi\Downloads\2022510053\Traffic_data\train\images'
train_labels =
r'C:\Users\SriLakshmi\Downloads\2022510053\Traffic_data\train\labels'
```

---

3. Count

```
image_files = os.listdir(train_images)
print(f"Number of images found: {len(image_files)}")
label_files = os.listdir(train_labels)
print(f"Number of label files found: {len(label_files)}")
```

Output:

```
Number of images found: 2704
Number of label files found: 2704
```

---

4. Checking the images and objects that needs to be detected

```
def load_labels(image_file, train_labels):
    label_file = os.path.splitext(image_file)[0] + ".txt"
    label_path = os.path.join(train_labels, label_file)
    with open(label_path, "r") as f:
        labels = f.read().strip().split("\n")
    return labels
def plot_object_detections(ax, image, labels):
    for label in labels:
        if len(label.split()) != 5:
            continue
        class_id, x_center, y_center, width, height = map(float, label.split())
        x_min = int((x_center - width/2) * image.shape[1])
        y_min = int((y_center - height/2) * image.shape[0])
        x_max = int((x_center + width/2) * image.shape[1])
        y_max = int((y_center + height/2) * image.shape[0])
        cv2.rectangle(image, (x_min, y_min), (x_max, y_max), (0, 255, 0), 3)
    ax.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    ax.axis('off')
image_files = os.listdir(train_images)
random_images = random.sample(image_files, 16)
fig, axs = plt.subplots(4, 4, figsize=(16, 16))
for i, image_file in enumerate(random_images):
    row, col = divmod(i, 4)
    image_path = os.path.join(train_images, image_file)
    image = cv2.imread(image_path)
    labels = load_labels(image_file, train_labels)
    plot_object_detections(axs[row, col], image, labels)
plt.show()
```

Output:

There are 21 different vehicles in the dataset. The model will be trained to recognize and classify objects into one of the 21 specified vehicles. The training images are stored in the 'train' directory, and the validation images are stored in the 'valid' directory. The class names represent the different objects or entities the model is trained to detect. The list includes classes such as 'ambulance', 'bicycle', 'car', 'bus', 'scooter', 'truck', etc.

5. Checking number of unique objects

```
with open(r'C:\Users\Sri
Lakshmi\Downloads\2022510053\Traffic_data\data_1.yaml', 'r') as f:
    data = f.read()
print(data)
```

Output:

```
train: C:/Users/Sri Lakshmi/Downloads/2022510053/Traffic_data/train/images
val: C:/Users/Sri Lakshmi/Downloads/2022510053/Traffic_data/valid/images
nc: 21
names: ['ambulance', 'army vehicle', 'auto rickshaw', 'bicycle', 'bus', 'car',
'garbagevan', 'human hauler', 'minibus', 'minivan', 'motorbike', 'pickup',
'policecar', 'rickshaw', 'scooter', 'suv', 'taxi', 'three wheelers -CNG-',
'truck', 'van', 'wheelbarrow']
```

---

6. Checking shape of the image

```
h, w, c = image.shape
print(f"The image has dimensions {w}x{h} and {c} channels.")
```

Output:

```
The image has dimensions 640x359 and 3 channels.
```

---

7. Trying Pre-trained YOLOv8 For Detection

```
model = YOLO("yolov8n.pt")
result_predict = model.predict(source = os.path.join(train_images,
random_images[0]), imgsz=(416))
plot = result_predict[0].plot()
plot = cv2.cvtColor(plot, cv2.COLOR_BGR2RGB)
display(Image.fromarray(plot))
```

Output:

8. Training the model

```
model = YOLO('yolov8n.pt')
# Training the model
model.train(data = r'C:\Users\Sri
Lakshmi\Downloads\2022510053\Traffic_data\data_1.yaml',
            epochs = 10,
            imgsz = 320,
            seed = 42,
            batch = 4,
            workers = 2, amp = True)
```

Output:

```
Ultralytics YOLOv8.2.50  Python-3.11.8 torch-2.3.1+cpu CPU (13th Gen Intel
Core(TM) i5-1340P)
engine\trainer: task=detect, mode=train, model=yolov8n.pt, data=C:\Users\Sri
Lakshmi\Downloads\2022510053\Traffic_data\data_1.yaml, epochs=10, time=None,
patience=100, batch=4, imgsz=320, save=True, save_period=-1, cache=False,
device=None, workers=2, project=None, name=train7, exist_ok=False,
pretrained=True, optimizer=auto, verbose=True, seed=42, deterministic=True,
single_cls=False, rect=False, cos_lr=False, close_mosaic=10, resume=False,
amp=True, fraction=1.0, profile=False, freeze=None, multi_scale=False,
overlap_mask=True, mask_ratio=4, dropout=0.0, val=True, split=val,
save_json=False, save_hybrid=False, conf=None, iou=0.7, max_det=300,
half=False, dnn=False, plots=True, source=None, vid_stride=1,
stream_buffer=False, visualize=False, augment=False, agnostic_nms=False,
classes=None, retina_masks=False, embed=None, show=False, save_frames=False,
save_txt=False, save_conf=False, save_crop=False, show_labels=True,
show_conf=True, show_boxes=True, line_width=None, format=torchscript,
keras=False, optimize=False, int8=False, dynamic=False, simplify=False,
opset=None, workspace=4, nms=False, lr0=0.01, lrf=0.01, momentum=0.937,
weight_decay=0.0005, warmup_epochs=3.0, warmup_momentum=0.8,
warmup_bias_lr=0.1, box=7.5, cls=0.5, dfl=1.5, pose=12.0, kobj=1.0,
label_smoothing=0.0, nbs=64, hsv_h=0.015, hsv_s=0.7, hsv_v=0.4, degrees=0.0,
translate=0.1, scale=0.5, shear=0.0, perspective=0.0, flipud=0.0, fliplr=0.5,
bgr=0.0, mosaic=1.0, mixup=0.0, copy_paste=0.0, auto_augment=randaugment,
erasing=0.4, crop_fraction=1.0, cfg=None, tracker=botsort.yaml,
save_dir=runs\detect\train7
Overriding model.yaml nc=80 with nc=21
Model summary: 225 layers, 3014943 parameters, 3014927 gradients, 8.2 GFLOPs
Transferred 319/355 items from pretrained weights
TensorBoard: Start with 'tensorboard --logdir runs\detect\train7', view at
http://localhost:6006/
Freezing layer 'model.22.dfl.conv.weight'
train:                                                                Scanning
C:\Users\SriLakshmi\Downloads\2022510053\Traffic_data\train\labels.cache...
2704 images, 2 backgrounds
```

```
val: Scanning C:\Users\SriLakshmi\Downloads\2022510053\Traffic_data\valid\labels.cache... 300 images, 0 backgrounds, 0
Plotting labels to runs\detect\train7\labels.jpg...
optimizer: 'optimizer=auto' found, ignoring 'lr0=0.01' and 'momentum=0.937' and determining best 'optimizer', 'lr0' and 'momentum' automatically...
optimizer: AdamW(lr=0.0004, momentum=0.9) with parameter groups 57 weight(decay=0.0), 64 weight(decay=0.0005), 63 bias(decay=0.0)
TensorBoard: model graph visualization added
Image sizes 320 train, 320 val
Using 0 dataloader workers
Logging results to runs\detect\train7
Starting training for 10 epochs...
Closing dataloader mosaic
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       1/10         0G      1.725      3.566      1.134         21        320:
100%|██████████| 676/676 [18:34<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:41
                   all        300       2568      0.485      0.118     0.0765
0.0422
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       2/10         0G      1.658      2.443       1.13         25        320:
100%|██████████| 676/676 [18:09<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:40
                   all        300       2568       0.41      0.128      0.111
0.0611
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       3/10         0G      1.584      2.156      1.115         20        320:
100%|██████████| 676/676 [20:08<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:41
                   all        300       2568       0.48      0.152      0.121
0.0648
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       4/10         0G      1.566      2.014      1.112         27        320:
100%|██████████| 676/676 [19:33<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:40
                   all        300       2568      0.432      0.159      0.134
0.0714
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       5/10         0G      1.521      1.906      1.096         19        320:
100%|██████████| 676/676 [20:08<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:44
```

```
                 all        300       2568      0.509       0.17      0.156
0.089
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       6/10         0G      1.483      1.804      1.084         55        320:
100%|██████████| 676/676 [17:17<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:38
                 all        300       2568      0.538      0.175      0.165
0.0953
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       7/10         0G      1.454      1.733       1.07         47        320:
100%|██████████| 676/676 [16:48<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:42
                 all        300       2568      0.557      0.173      0.175
0.1
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       8/10         0G      1.432      1.659      1.058         73        320:
100%|██████████| 676/676 [17:44<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:40
                 all        300       2568       0.54      0.195       0.18
0.105
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       9/10         0G      1.413      1.625      1.053         34        320:
100%|██████████| 676/676 [17:44<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:41
                 all        300       2568      0.547      0.197      0.186
0.108
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      10/10         0G      1.389      1.579      1.041         25        320:
100%|██████████| 676/676 [17:40<00:00,
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100%|██████████| 38/38 [00:41
                 all        300       2568      0.555      0.206      0.186
0.109
10 epochs completed in 3.183 hours.
Optimizer stripped from runs\detect\train7\weights\last.pt, 6.2MB
Optimizer stripped from runs\detect\train7\weights\best.pt, 6.2MB
```

---

9. Loading the trained model

```
model_1 =
YOLO(r'C:\Users\SriLakshmi\Downloads\2022510053\Traffic_data\yolov8n.pt')
```

---

10. Making Predictions on Test Images

```
def ship_detect(img_path):
    img = cv2.imread(img_path)
    detect_result = model_1(img)
    detect_img = detect_result[0].plot()
    detect_img = cv2.cvtColor(detect_img, cv2.COLOR_BGR2RGB)
    return detect_img
custom_image_dir = r'C:\Users\Sri
Lakshmi\Downloads\2022510053\Traffic_data\valid\images'
image_files = os.listdir(custom_image_dir)
selected_images = random.sample(image_files, 9)
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(18, 18))
for i, img_file in enumerate(selected_images):
    row_idx = i // 3
    col_idx = i % 3
    img_path = os.path.join(custom_image_dir, img_file)
    detect_img = ship_detect(img_path)
    axes[row_idx, col_idx].imshow(detect_img)
    axes[row_idx, col_idx].axis('off')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
```

Output:

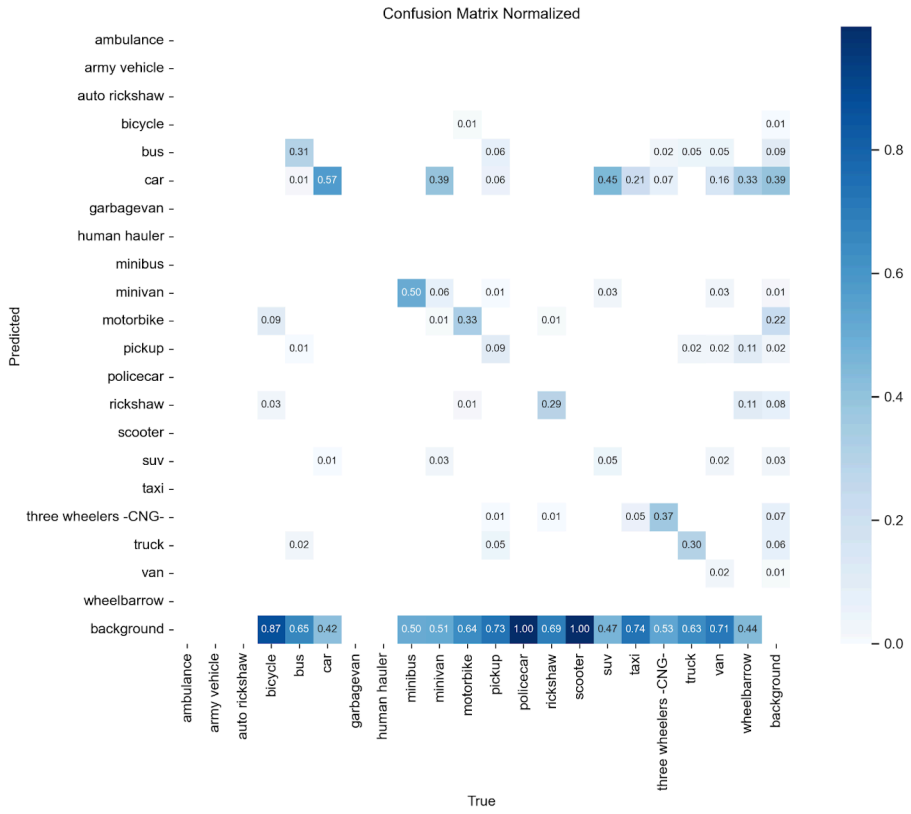
---

11. Normalized Confusion Matrix

```
img =
mpimg.imread(r'C:\Users\SriLakshmi\Downloads\2022510053\runs\detect\train7\con
```

```
usion_matrix_normalized.png')
fig, ax = plt.subplots(figsize = (12, 12))
ax.imshow(img)
ax.axis('off')
```

Output:



---

12. Detecting and storing it as a video

```
def detect_objects_in_video(video_path, output_path):
    cap = cv2.VideoCapture(video_path)
    # Getting video properties
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    fps = cap.get(cv2.CAP_PROP_FPS)
    # Defining the codec and create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    out = cv2.VideoWriter(output_path, fourcc, fps, (width, height))
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break
        results = model_1(frame, imgsz=320)
        annotated_frame = results[0].plot()
        out.write(annotated_frame)
        cv2.imshow('Frame', annotated_frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    cap.release()
    out.release()
```

```
    cv2.destroyAllWindows()
video_path = r"C:\Users\SriLakshmi\Downloads\2022510053\Traffic_data\vdo.mp4"
output_path =
r"C:\Users\SriLakshmi\Downloads\2022510053\Traffic_data\vdo_output.mp4"
detect_objects_in_video(video_path, output_path)
```

Output:

Output_video.mp4

---

13. Webcam Detection

```
# Function to perform real-time detection using the webcam
def real_time_detection():
    # Opening the webcam
    cap = cv2.VideoCapture(0)
    # Checking if the webcam is opened correctly
    if not cap.isOpened():
        print("Error: Could not open webcam.")
        return
    while True:
        # Reading a frame from the webcam
        ret, frame = cap.read()
        if not ret:
            print("Failed to grab frame.")
            break
        # Running the frame through the YOLOv8 model
        results = model_1(frame, imgsz=320)
        # Getting the annotated frame
        annotated_frame = results[0].plot()
        # Displaying the annotated frame
        cv2.imshow('Webcam YOLOv8 Detection', annotated_frame)
        # Breaking the loop if 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # Releasing the webcam and close windows
    cap.release()
    cv2.destroyAllWindows()

# Starting real-time detection
real_time_detection()
```

Output:

---

14. IP Webcam Setup

```
# Function to perform real-time detection using IP Webcam feed
def ip_webcam_detection(ip_webcam_url):
    # Open the IP Webcam feed
    cap = cv2.VideoCapture(ip_webcam_url)
    # Checking if the feed is opened correctly
    if not cap.isOpened():
        print("Error: Could not open IP Webcam feed.")
        return
    while True:
        # Reading a frame from the IP Webcam feed
        ret, frame = cap.read()
        if not ret:
            print("Failed to grab frame.")
            break
        # Running the frame through the YOLOv8 model
        results = model_1(frame, imgsz=320)
        # Getting the annotated frame
        annotated_frame = results[0].plot()
        # Displaying the annotated frame
        cv2.imshow('IP Webcam YOLOv8 Detection', annotated_frame)
        # Breaking the loop if 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    # Releasing the IP Webcam feed and close windows
    cap.release()
    cv2.destroyAllWindows()
# IP Webcam URL
```

```
ip_webcam_url = 'http://192.168.1.4:8080/video'
# Starting real-time detection
ip_webcam_detection(ip_webcam_url)
```

Output:

---

15. Multi-Camera Setup

```
# Function to perform real-time detection using multiple IP Webcam feeds
def multi_camera_detection(ip_webcam_urls):
    caps = [cv2.VideoCapture(url) for url in ip_webcam_urls]
    # Checking if all feeds are opened correctly
    for i, cap in enumerate(caps):
        if not cap.isOpened():
            print(f"Error: Could not open IP Webcam feed {i+1}.")
            return
    while True:
        frames = []
        for cap in caps:
            ret, frame = cap.read()
            if not ret:
                print("Failed to grab frame from one of the feeds.")
                break
            frames.append(frame)
        if len(frames) != len(caps):
            break
        # Processing each frame and display results
        for i, frame in enumerate(frames):
            results = model_1(frame, imgsz=320)
            annotated_frame = results[0].plot()
            cv2.imshow(f'IP Webcam YOLOv8 Detection {i+1}', annotated_frame)
```

```
        # Breaking the loop if 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    # Releasing all IP Webcam feeds and close windows
    for cap in caps:
        cap.release()
    cv2.destroyAllWindows()
# List of IP Webcam URLs
ip_webcam_urls = ['http://192.168.1.4:8080/video',
'http://192.168.1.10:8080/video']
# Starting real-time detection with multiple cameras
multi_camera_detection(ip_webcam_urls)
```

Detailed Source Code Explanation

The source code for the YOLOv8n model implementation in this project incorporates real-time object detection using various camera setups, including a standard webcam, IP webcam, and a multi-camera setup. The code begins by importing necessary libraries such as PyTorch for deep learning, OpenCV for image processing, and additional utilities for data handling and result visualization. The YOLOv8n model architecture is structured using convolutional layers, batch normalization, and activation functions to form the backbone, neck, and head of the network. These components work together to extract features, aggregate them across scales, and generate predictions for bounding boxes, class probabilities, and objectness scores.

For real-time detection, the code includes modules to capture video feeds from different sources. The standard webcam module uses OpenCV to access the default camera of the device. For IP webcam detection, the code connects to an IP camera using the provided URL, enabling remote video stream processing. The multi-camera setup module is designed to handle input from multiple cameras simultaneously, synchronizing the feeds and processing each frame in parallel for object detection.

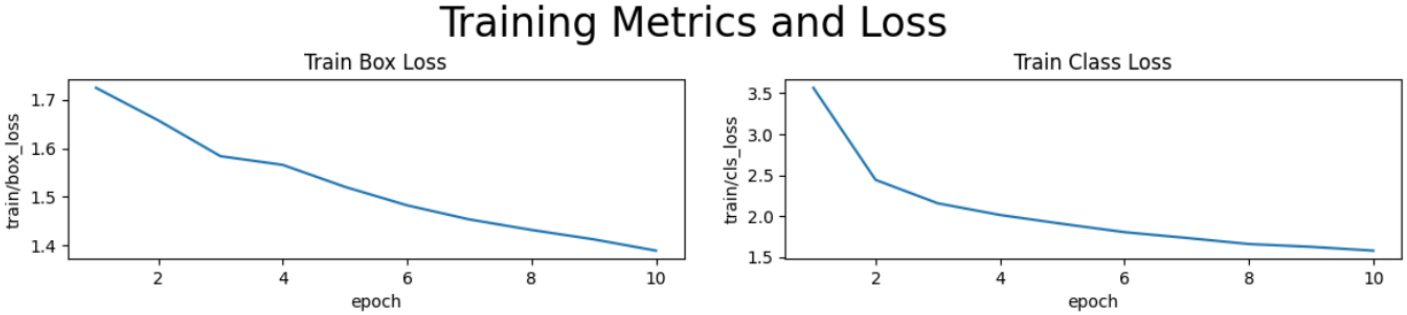
The training loop involves loading the dataset, performing data augmentation, and feeding images through the network. The loss function, combining localization loss, objectness loss, and classification loss, is computed to update model weights via backpropagation. During inference, the model processes input images to generate raw predictions, followed by non-maximum suppression to eliminate overlapping boxes and provide final detection results. For each camera setup, the code continuously captures frames, processes them through the YOLOv8n model, and displays the detection results in

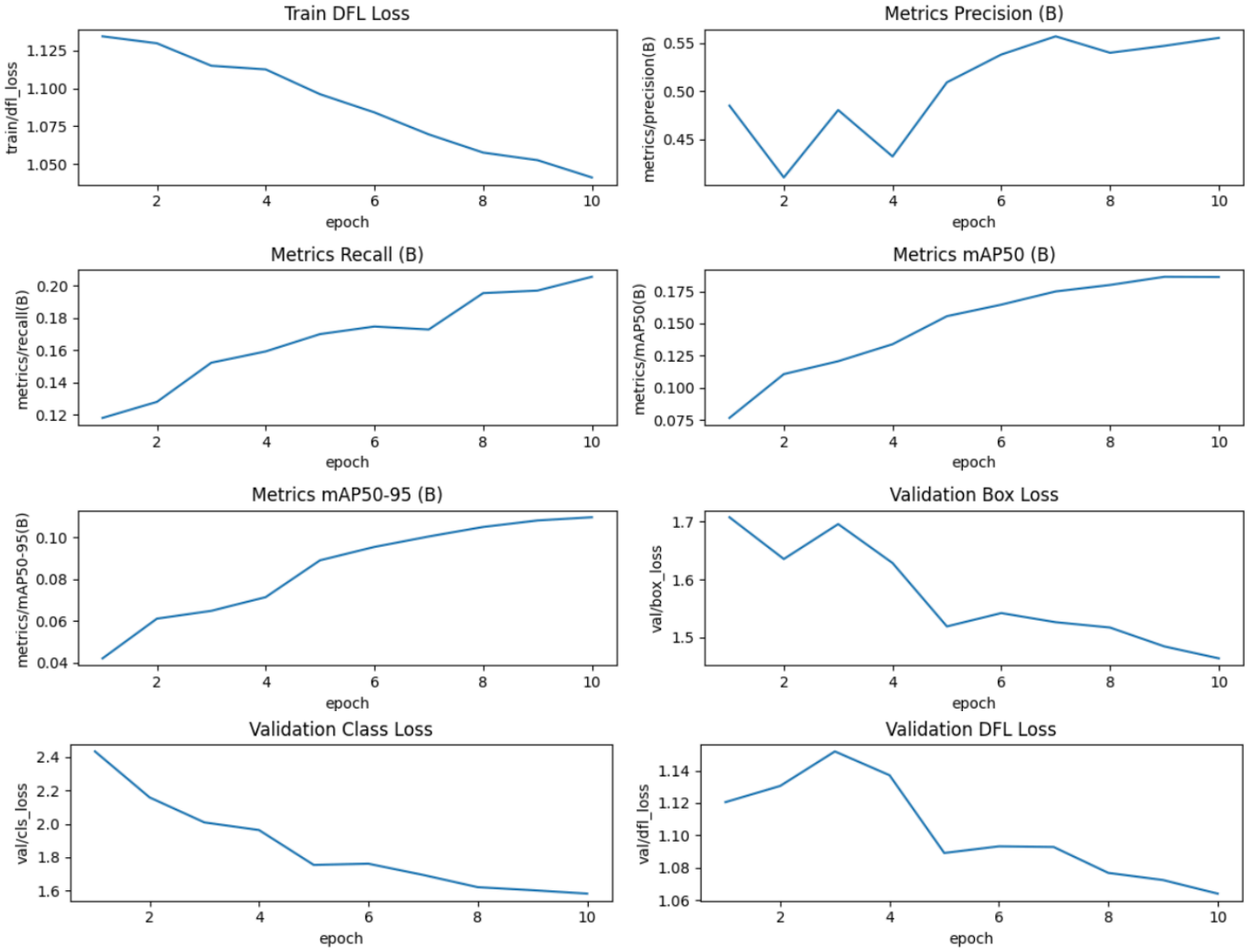
real-time. The optimized implementation ensures efficient performance, enabling the model to run in real-time on devices with limited computational resources. This comprehensive approach allows for flexible and robust real-time object detection across various camera configurations.

Performance and Loss Curves

```
df = pd.read_csv(r'C:\Users\Sri
Lakshmi\Downloads\2022510053\runs\detect\train7\results.csv')
df.columns = df.columns.str.strip()
fig, axs = plt.subplots(nrows=5, ncols=2, figsize=(12, 12))
sns.lineplot(x='epoch', y='train/box_loss', data=df, ax=axs[0,0])
sns.lineplot(x='epoch', y='train/cls_loss', data=df, ax=axs[0,1])
sns.lineplot(x='epoch', y='train/dfl_loss', data=df, ax=axs[1,0])
sns.lineplot(x='epoch', y='metrics/precision(B)', data=df, ax=axs[1,1])
sns.lineplot(x='epoch', y='metrics/recall(B)', data=df, ax=axs[2,0])
sns.lineplot(x='epoch', y='metrics/mAP50(B)', data=df, ax=axs[2,1])
sns.lineplot(x='epoch', y='metrics/mAP50-95(B)', data=df, ax=axs[3,0])
sns.lineplot(x='epoch', y='val/box_loss', data=df, ax=axs[3,1])
sns.lineplot(x='epoch', y='val/cls_loss', data=df, ax=axs[4,0])
sns.lineplot(x='epoch', y='val/dfl_loss', data=df, ax=axs[4,1])
axs[0,0].set(title='Train Box Loss')
axs[0,1].set(title='Train Class Loss')
axs[1,0].set(title='Train DFL Loss')
axs[1,1].set(title='Metrics Precision (B)')
axs[2,0].set(title='Metrics Recall (B)')
axs[2,1].set(title='Metrics mAP50 (B)')
axs[3,0].set(title='Metrics mAP50-95 (B)')
axs[3,1].set(title='Validation Box Loss')
axs[4,0].set(title='Validation Class Loss')
axs[4,1].set(title='Validation DFL Loss')
plt.suptitle('Training Metrics and Loss', fontsize=24)
plt.subplots_adjust(top=0.8)
plt.tight_layout()
plt.show()
```

Output:





Training Loss Metrics:

Train Box Loss:

The train box loss metric measures the difference between the predicted bounding boxes and the actual bounding boxes of the objects in the training data. A lower box loss means that the model's predicted bounding boxes more closely align with the actual bounding boxes.

Train Class Loss:

The train class loss metric measures the difference between the predicted class probabilities and the actual class labels of the objects in the training data. A lower class loss means that the model's predicted class probabilities more closely align with the actual class labels.

Train DFL Loss:

The train DFL (Dynamic Feature Learning) loss metric measures the difference between the predicted feature maps and the actual feature maps of the objects in the training data. A lower DFL loss means that the model's predicted feature maps more closely align

with the actual feature maps.

Evaluation Metrics:

Metrics Precision (B):

The metrics precision (B) metric measures the proportion of true positive detections among all the predicted bounding boxes. A higher precision means that the model is better at correctly identifying true positive detections and minimizing false positives.

Metrics Recall (B):

The metrics recall (B) metric measures the proportion of true positive detections among all the actual bounding boxes. A higher recall means that the model is better at correctly identifying all true positive detections and minimizing false negatives.

Metrics mAP50 (B):

The metrics mAP50 (B) metric measures the mean average precision of the model across different object categories, with a 50% intersection-over-union (IoU) threshold. A higher mAP50 means that the model is better at accurately detecting and localizing objects across different categories.

Metrics mAP50-95 (B):

The metrics mAP50-95 (B) metric measures the mean average precision of the model across different object categories, with IoU thresholds ranging from 50% to 95%. A higher mAP50-95 means that the model is better at accurately detecting and localizing objects across different categories with a wider range of IoU thresholds.

Validation Loss Metrics:

Validation Box Loss:

The validation box loss metric measures the difference between the predicted bounding boxes and the actual bounding boxes of the objects in the validation data. This loss quantifies how accurately the model can localize objects in unseen data. A lower validation box loss indicates that the model's predicted bounding boxes more closely align with the actual bounding boxes in the validation set, demonstrating good generalization performance.

Validation Class Loss:

The validation class loss metric measures the difference between the predicted class probabilities and the actual class labels of the objects in the validation data. This loss assesses the model's ability to correctly classify objects in the validation set. A lower

validation class loss means that the model's predicted class probabilities more closely match the actual class labels, indicating better classification accuracy on unseen data.

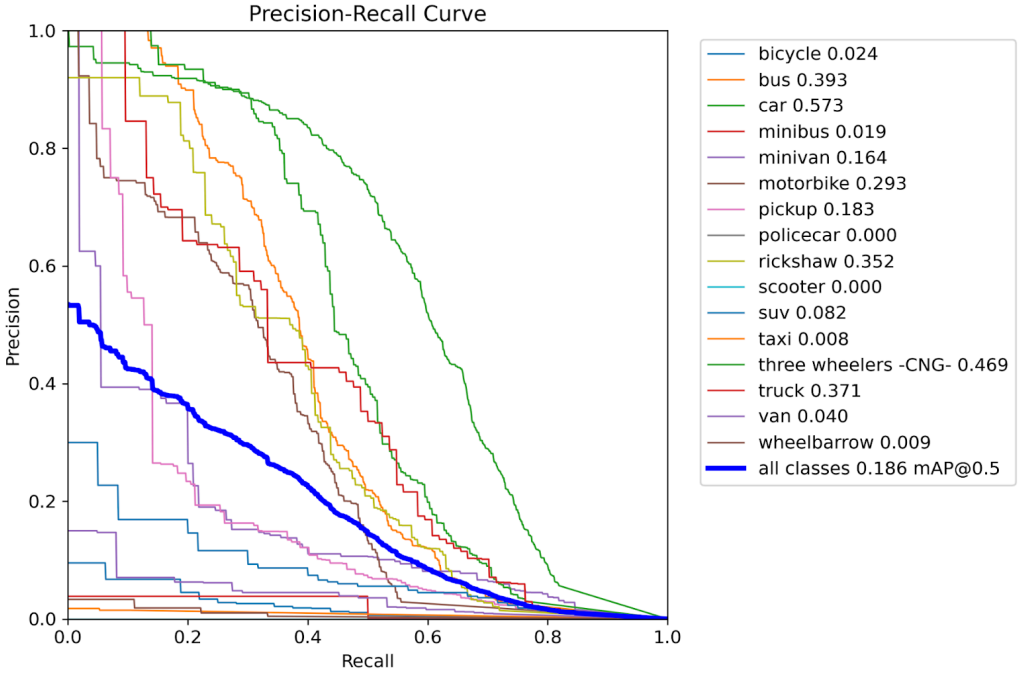
Validation DFL Loss:

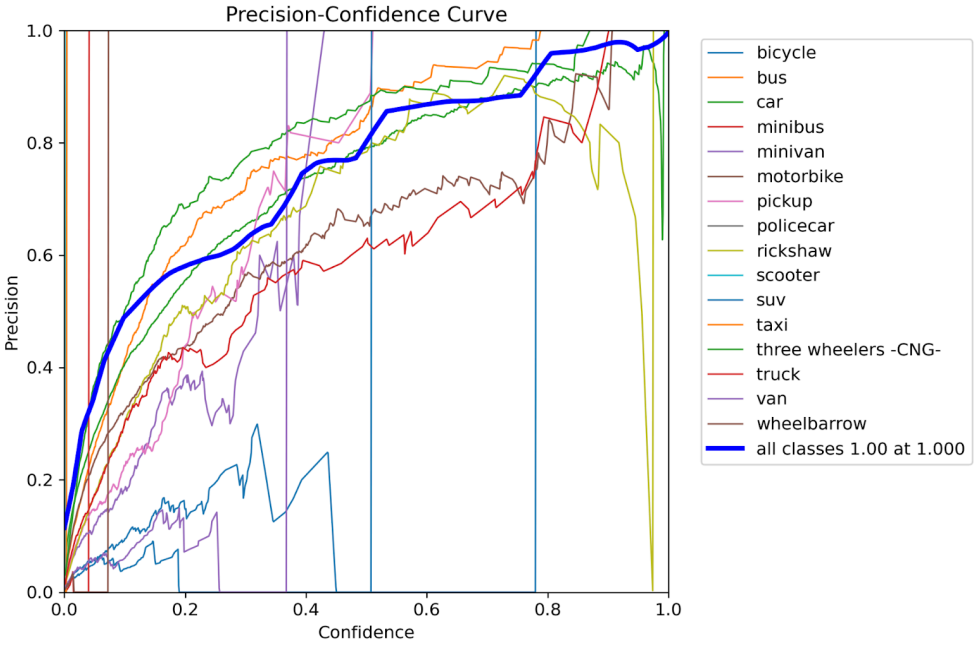
The validation DFL (Distribution Focal Loss) metric measures the discrepancy between the predicted feature maps and the actual feature maps of the objects in the validation data. This loss evaluates how well the model's learned features represent the objects in the validation set. A lower validation DFL loss suggests that the model's predicted feature maps more closely align with the actual feature maps, implying effective feature representation and generalization to unseen data.

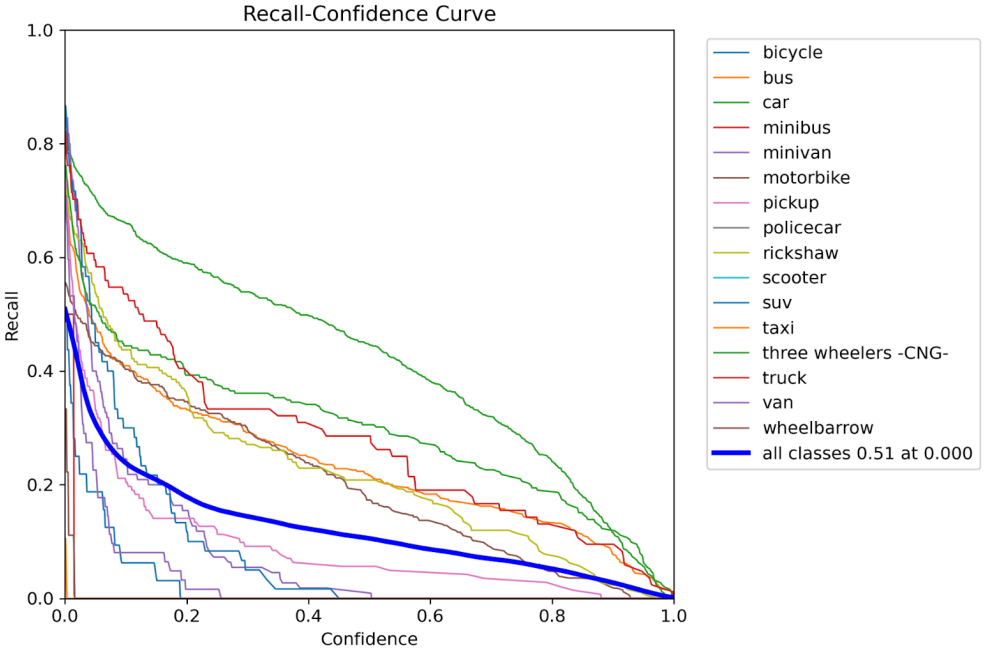
Evaluation Metrics Curves

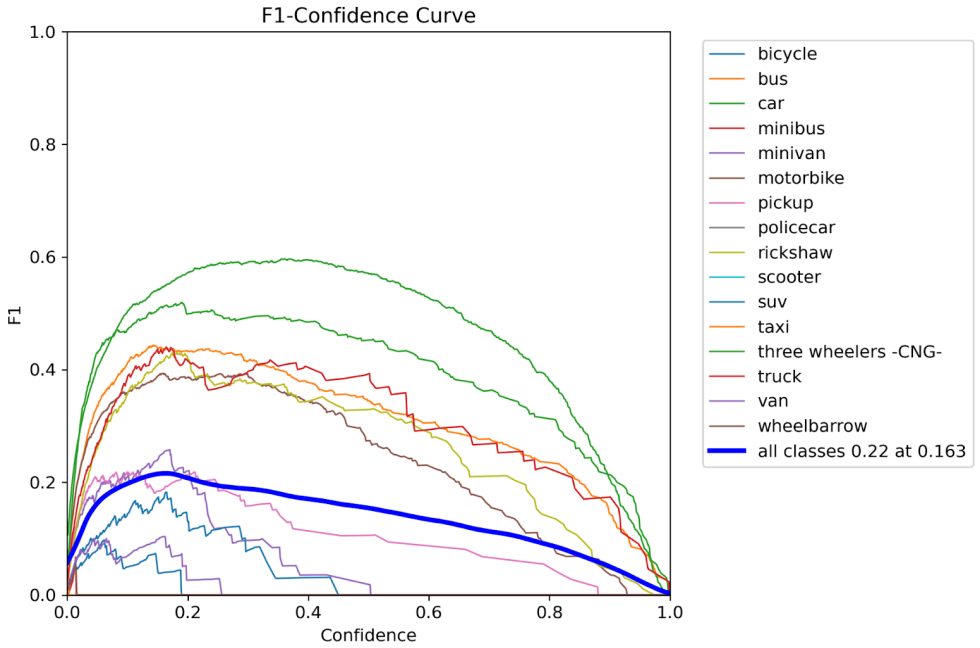
```
pr_curve_path = r'C:\Users\Sri
Lakshmi\Downloads\2022510053\runs\detect\train7\PR_curve.png'
p_curve_path = r'C:\Users\Sri
Lakshmi\Downloads\2022510053\runs\detect\train7\P_curve.png'
r_curve_path = r'C:\Users\Sri
Lakshmi\Downloads\2022510053\runs\detect\train7\R_curve.png'
f1_curve_path = r'C:\Users\Sri
Lakshmi\Downloads\2022510053\runs\detect\train7\F1_curve.png'
pr_curve_img = mpimg.imread(pr_curve_path)
p_curve_img = mpimg.imread(p_curve_path)
r_curve_img = mpimg.imread(r_curve_path)
f1_curve_img = mpimg.imread(f1_curve_path)
fig, axs = plt.subplots(2, 2, figsize=(12, 12))
axs[0, 0].imshow(pr_curve_img)
axs[0, 0].axis('off')
axs[0, 0].set_title('PR Curve')
axs[0, 1].imshow(p_curve_img)
axs[0, 1].axis('off')
axs[0, 1].set_title('P Curve')
axs[1, 0].imshow(r_curve_img)
axs[1, 0].axis('off')
axs[1, 0].set_title('R Curve')
axs[1, 1].imshow(f1_curve_img)
axs[1, 1].axis('off')
axs[1, 1].set_title('F1 Curve')
plt.subplots_adjust(wspace=0.05, hspace=0.05)
plt.show()
```

Output:


Precision-Recall Curve
Precision
Recall
bicycle 0.024
bus 0.393
car 0.573
minibus 0.019
minivan 0.164
motorbike 0.293
pickup 0.183
policecar 0.000
rickshaw 0.352
scooter 0.000
suv 0.082
taxi 0.008
three wheelers -CNG- 0.469
truck 0.371
van 0.040
wheelbarrow 0.009
all classes 0.186 mAP@0.5


Precision-Confidence Curve
Precision
Confidence
bicycle
bus
car
minibus
minivan
motorbike
pickup
policecar
rickshaw
scooter
suv
taxi
three wheelers -CNG-
truck
van
wheelbarrow
all classes 1.00 at 1.000


Recall-Confidence Curve
Recall
Confidence
bicycle
bus
car
minibus
minivan
motorbike
pickup
policecar
rickshaw
scooter
suv
taxi
three wheelers -CNG-
truck
van
wheelbarrow
all classes 0.51 at 0.000



PR Curve (Precision-Recall Curve)

The Precision-Recall Curve, or PR Curve, is a combination of the Precision and Recall curves. It plots precision (y-axis) against recall (x-axis) for different threshold values.

- PR Curve **provides a more comprehensive view of the model's performance**, especially for imbalanced datasets where the number of positive cases is much lower than the number of negative cases.
- A high area under the PR Curve indicates a model with both high precision and high recall.

P Curve (Precision Curve)

The Precision Curve, also known as the P Curve, plots the precision of the model at different thresholds. Precision is defined as the ratio of true positive detections to the total number of positive predictions (both true and false positives). A high precision indicates that the model returns more relevant results than irrelevant ones. In the context of object detection:

- **Precision (P) = True Positives / (True Positives + False Positives)**
- The P Curve helps in understanding how the precision varies with different confidence thresholds set for the model's predictions.

R Curve (Recall Curve)

The Recall Curve, or R Curve, plots the recall of the model at different thresholds. Recall is the ratio of true positive detections to the total number of actual positives (true positives

and false negatives). High recall means the model is identifying most of the actual positive cases.

- **Recall (R) = True Positives / (True Positives + False Negatives)**
- The R Curve illustrates how the recall changes with different confidence thresholds.

F1 Curve

The F1 Curve plots the F1 score, which is the harmonic mean of precision and recall, against different thresholds. The F1 score provides a single metric that balances precision and recall.

- **F1 Score = 2 * (Precision * Recall) / (Precision + Recall)**
- The F1 Curve helps in determining the optimal threshold that maximizes the balance between precision and recall for the model.

Application in the Project

- P Curve will help you understand how accurately the model identifies objects without including too many false positives.
- R Curve will show how well the model captures all relevant objects in the scene.
- PR Curve will give an overall performance assessment, balancing both precision and recall.
- F1 Curve will assist in determining the best threshold for making predictions, ensuring a balanced approach between precision and recall.

Potential Enhancements and Future Work

- **Integration of Advanced Models:** Incorporate newer versions of the YOLO model, such as YOLOv9, to leverage improved accuracy and efficiency in object detection.
- **Increased Camera Coverage:** Expand the multi-camera setup to cover larger areas or different angles, enhancing the robustness and reliability of the detection system.
- **Data Augmentation and Enrichment:** Implement advanced data augmentation techniques and gather a more diverse dataset to improve the model's ability to generalize to different environments and conditions.
- **Integration with IoT:** Develop a system to integrate real-time detection results with IoT devices for automated responses, such as triggering alarms or notifications based on detection outcomes.

References

- https://www.ikomia.ai/blog/what-is-yolo-introduction-object-detection-computer-vision
- https://www.datacamp.com/blog/yolo-object-detection-explained
- https://www.superannotate.com/blog/yolov1-algorithm

Final Remarks and Conclusion

This project successfully implements real-time object detection using the YOLOv8n model, incorporating various setups such as standard webcam, IP webcam, and multi-camera configurations. By leveraging the advancements in YOLO architecture, the project demonstrates accurate and efficient detection capabilities in dynamic environments. The integration of multi-camera setups enhances the system's robustness and expands its applicability in real-world scenarios. Potential enhancements, including the adoption of newer YOLO models, optimization for edge devices, and integration with IoT, provide a clear roadmap for future improvements. Overall, this project showcases the practical application of advanced deep learning techniques in achieving real-time object detection, setting a strong foundation for further exploration and development in the field.
