Author : Kena Hemnani
Date : 8th July 2024

Object detection

Over the last ten years, deep learning has garnered significant attention and emerged as a crucial technology in the field of artificial intelligence. One of the prominent areas within deep learning and computer vision is object detection. Object detection has found numerous applications in computer vision, including object tracking, image retrieval, video surveillance, image captioning, image segmentation, medical imaging, and many more.

Traffic Detection and Monitoring :

Computer vision has numerous significant applications in traffic surveillance and management, enhancing safety, efficiency, and overall traffic flow. Some of the key applications include:

1. **Traffic Flow Monitoring**: Analyzing real-time traffic conditions to monitor vehicle flow, identify congestion, and detect incidents such as accidents or breakdowns.
2. **Automated Number Plate Recognition (ANPR)**: Identifying and recording vehicle license plates for purposes such as toll collection, law enforcement, and vehicle tracking.
3. **Traffic rules Violation Detection**: Detecting and recording traffic violations such as running red lights, speeding, illegal parking, and lane violations.
4. **Pedestrian and Cyclist Detection**: Enhancing safety by detecting pedestrians and cyclists, particularly at crosswalks and intersections, to prevent accidents.
5. **Vehicle Classification**: Classifying vehicles into categories (e.g., cars, trucks, motorcycles) to better understand traffic composition and manage lane usage.
6. **Smart Traffic Signals**: Using real-time traffic data to optimize traffic signal timings, reducing congestion and improving traffic flow.
7. **Accident Detection and Analysis**: Automatically detecting accidents and providing detailed analysis to improve emergency response times and understand causes for future prevention.
8. **Parking Management**: Monitoring parking spaces to provide real-time information on availability, manage illegal parking, and improve the efficiency of parking operations.
9. **Road Condition Monitoring**: Detecting road conditions such as potholes, debris, or obstacles to maintain road quality and ensure driver safety.

These applications leverage computer vision technologies to create smarter, safer, and more efficient traffic management systems.

In this report we will implement a Real time traffic Detection System which includes detecting of Vehicles like bicycle, bus, car and motorbike as well as pedestrians which can be used for several downstream tasks such as:

1. Counting the number of Vehicles and Pedestrians and hence determining the flow of traffic in different areas of the City at different times of a day.
2. Determining the flow of traffic from different directions at cross-roads thereby optimizing the traffic signal to reduce congestion and improve traffic flow.
3. Determining heavy vehicles, ambulances, school buses etc. for statistical analysis of traffic.

Dataset:

We will use the Open Source available dataset in Kaggle i.e., Traffic Detection Dataset for implementing our Traffic Detection System.

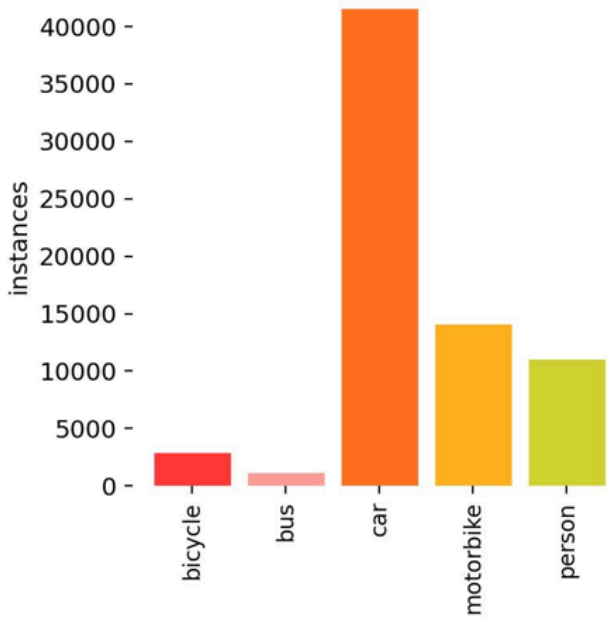
This dataset encompasses traffic camera images from different countries, providing a global perspective on traffic monitoring and management. The dataset includes images captured under different weather conditions, lighting, and traffic scenarios, making it suitable for real-world applications.

It consist of following number of Images for Training, Validation and Testing

Number of Training Samples : 5805
Number of Validation Samples : 549
Number of TestingSamples : 249

The bounding box annotations are given for the following 5 classes:
Bicycle, Bus, Car, Motorbike, Person

Following graph represents the distribution of instances of the above class in the Traffic Detection Dataset in the training set:



Detection Models:

In this report we will Benchmark, Compare and Analyze some of the most recent and State of the Art Object Detection models to be able to choose what suits best for our Traffic Detection Dataset.

Following are the Object Detection models we will explore:

1. YOLOv8
2. YOLOv9
3. YOLOv10

Before going through the differences between each of above recent YOLO architectures, let us go through the basics of 1st version of YOLO i.e., YOLOv1

Yolov1 theory

The first YOLO architecture was released with the paper You Only Look Once in 2016. It is the First Single Stage Object Detector which was faster as compared to the the Two Stage Object Detectors researchers explored till then i.e., RCNN, Fast-RCNN, Faster-RCNN and so on.

YOLO's simple and faster approach to process the image in one stage to output all the Object bounding box predictions facilitated the deployment of Object Detection models in real time.

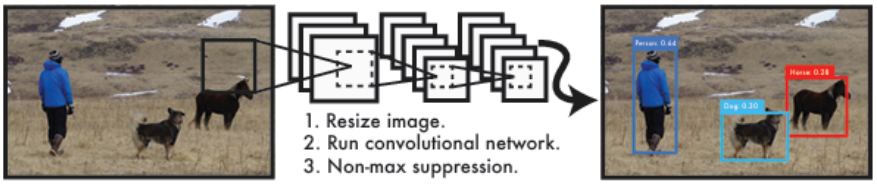
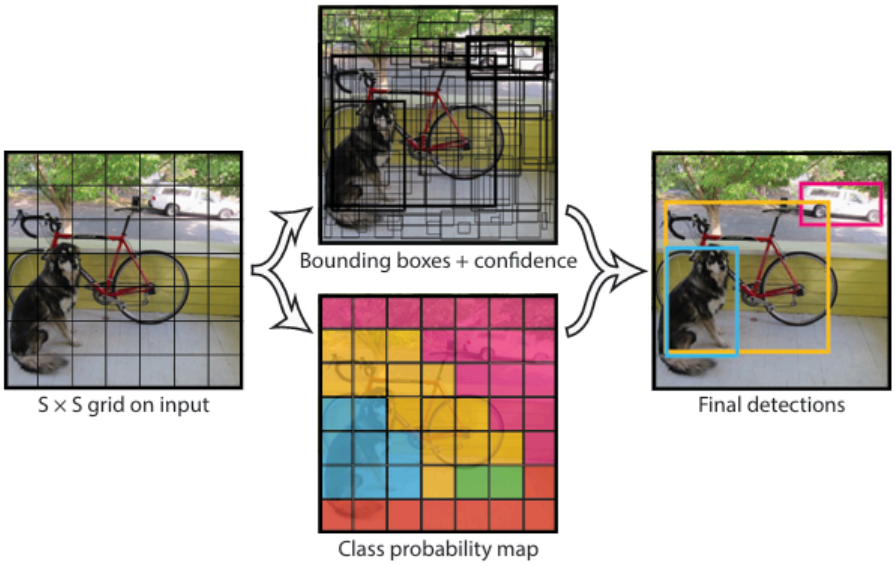


Figure 1: The YOLO Detection System. Processing images with YOLO is simple and straightforward. Our system (1) resizes the input image to 448 × 448, (2) runs a single convolutional network on the image, and (3) thresholds the resulting detections by the model's confidence.

YOLO models detection as a regression problem. It divides the image into an S * S grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an S * S * (B * 5 + C)tensor.



After this several YOLO model got released in last decade i.e., YOLOv2 which used anchors to increase detection accuracy and several other improvements, YOLOv3 focused on increasing speed along with accuracy and used Feature Pyramid Network to accurately detect objects of different size and aspect ratios, and so on ..

The most recent YOLO architectures are YOLOv8, YOLOv9 and YOLOv10

YOLOv8 : YOLOv8 features a more efficient and powerful backbone architecture, improving the model's ability to extract features from input images. Improvements in the detection head allow YOLOv8 to produce more accurate bounding boxes and class predictions, enhancing overall detection performance. It also employs Anchor-free detections. The benefit of anchor-free detection lies in its flexibility and efficiency, as it eliminates the need for manually specifying anchor boxes. This process can be challenging and may result in suboptimal outcomes in earlier YOLO models like v1 and v2.

YOLOv9 : YOLOv9 introduces groundbreaking techniques like Programmable Gradient Information (PGI) and the Generalized Efficient Layer Aggregation Network (GELAN). These innovations tackle information loss challenges in deep neural networks, ensuring high efficiency, accuracy, and adaptability. PGI preserves crucial data across network layers, while GELAN optimizes parameter utilization and computational efficiency.

YOLOv10 : YOLOv10 introduces NMS-free training to reduce latency and an efficiency-accuracy driven design strategy. It also introduces consistent dual assignments for NMS-free training, achieving competitive performance alongside low inference latency. Additionally, they introduced a comprehensive efficiency-accuracy driven model design strategy, optimizing various YOLO components for both efficiency and accuracy. This approach minimizes computational overhead while boosting performance.

Experiments:

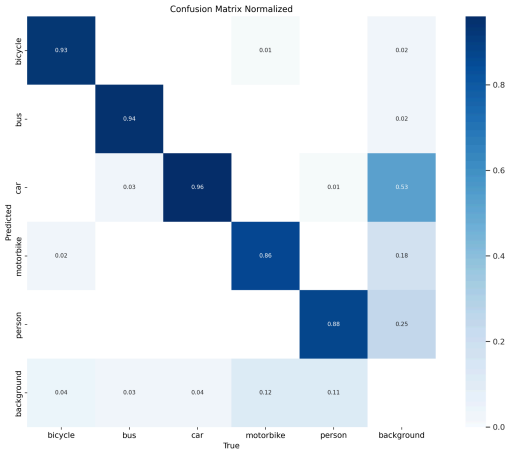
Following models were trained on the given dataset for 60 epochs and batch size of 32.

1. YOLOv8-n
2. YOLOv8-s
3. YOLOv9-c
4. Gelan-c
5. YOLOv10-n
6. YOLOv10-s

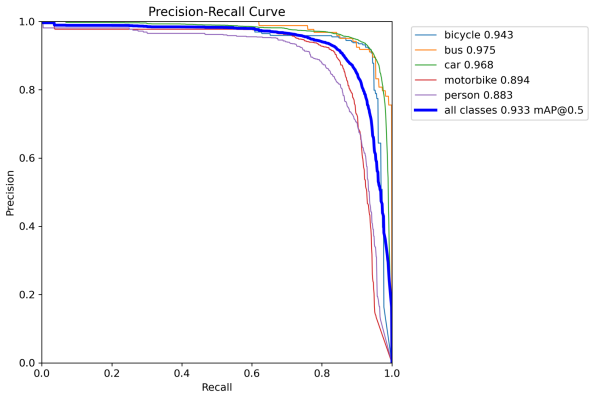
Following are the Performance metrics on Validation data for each of the above models after trained on Traffic Detection Dataset:

1.) Yolov8-n:

Model	**Speed on GPU [FPS]**	**Speed on CPU [FPS]**	**mAP@0.5**	**mAP@0.5: 0.95**	**# of parameters (M)**	**GFLOPs**
YOLOv8-n	89.0	6.3	0.933	0.725	3.01	8.1



Confusion Matrix

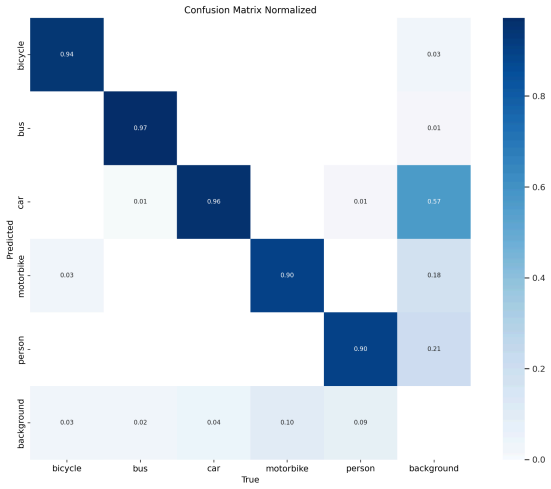


PR-Curve

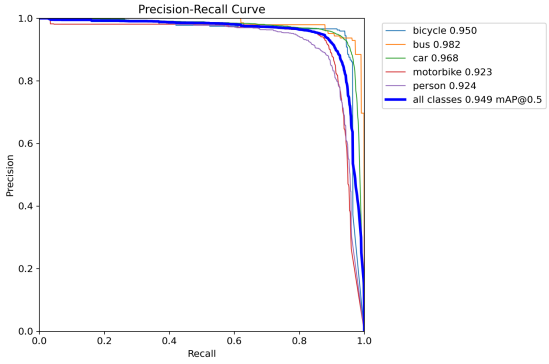
```
Validating runs/detect/yolov8n_custom4/weights/best.pt...
Ultralytics YOLOv8.2.48 🚀 Python-3.10.12 torch-2.3.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 3006623 parameters, 0 gradients, 8.1 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 10
                   all        549       6270      0.904      0.892      0.933      0.725
               bicycle        189        250      0.918      0.936      0.943      0.772
                   bus         81        108       0.91      0.944      0.975      0.881
                   car        520       3842      0.917      0.941      0.968      0.785
            motorbike        331       1238      0.919      0.821      0.894      0.584
                person        196        832      0.857       0.82      0.883      0.601
Speed: 0.3ms preprocess, 2.4ms inference, 0.0ms loss, 8.0ms postprocess per image
```

2.) yolov8-s:

Model	**Speed on GPU [FPS]**	**Speed on CPU [FPS]**	**mAP@0.5**	**mAP@0.5: 0.95**	**# of parameters (M)**	**GFLOPs**
YOLOv8-s	82.64	2.6	0.949	0.78	11.12	28.4



Confusion Matrix

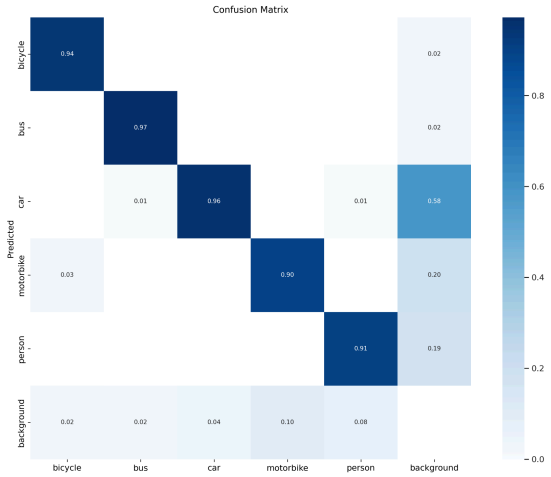


PR-Curve

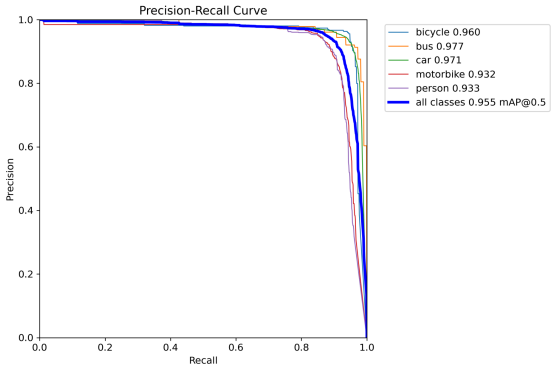
```
Validating runs/detect/yolov8s_custom/weights/best.pt...
Ultralytics YOLOv8.2.49 🚀 Python-3.10.12 torch-2.3.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 11127519 parameters, 0 gradients, 28.4 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 1
                   all        549       6270      0.929      0.915      0.949       0.78
               bicycle        189        250      0.941      0.944       0.95      0.816
                   bus         81        108      0.925      0.972      0.982      0.906
                   car        520       3842      0.932       0.94      0.968      0.817
             motorbike        331       1238      0.947      0.859      0.923      0.642
                person        196        832      0.902      0.861      0.924      0.722
Speed: 0.6ms preprocess, 5.3ms inference, 0.0ms loss, 7.6ms postprocess per image
Results saved to runs/detect/yolov8s_custom
```

3) yolov9-c :

Model	**Speed on GPU [FPS]**	**Speed on CPU [FPS]**	**mAP@0.5**	**mAP@0.5: 0.95**	**# of parameters (M)**	**GFLOPs**
YOLOv9-c	16.77	0.323	0.955	0.792	50.96	237.7



Confusion Matrix

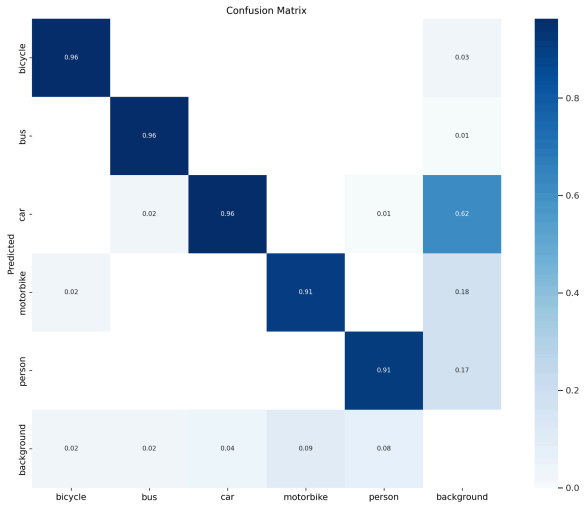


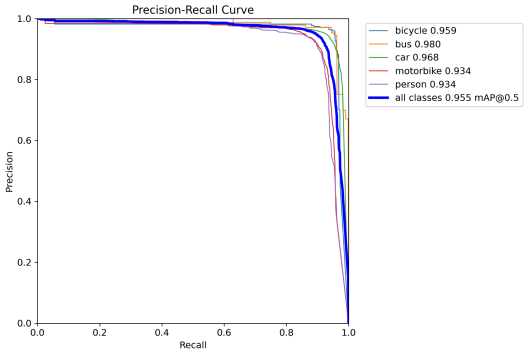
PR-Curve

```
Validating runs/train/exp/weights/best.pt...
Fusing layers...
yolov9-c summary: 724 layers, 50967870 parameters, 0 gradients, 237.7 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 35/35 [00:31<00:00,  1.12it/s]
                   all        549       6270      0.926      0.928      0.955      0.792
               bicycle        549        250      0.956      0.946       0.96      0.832
                   bus        549        108       0.92      0.963      0.977      0.917
                   car        549       3842      0.927      0.951      0.971       0.82
             motorbike        549       1238      0.922      0.883      0.932      0.662
                person        549        832      0.904      0.895      0.933      0.728
Results saved to runs/train/exp
```

4) gelan-c:

Model	**Speed on GPU [FPS]**	**Speed on CPU [FPS]**	**mAP@0.5**	**mAP@0.5: 0.95**	**# of parameters (M)**	**GFLOPs**
Gelan-c	32.89	0.378	0.955	0.793	25.41	102.5

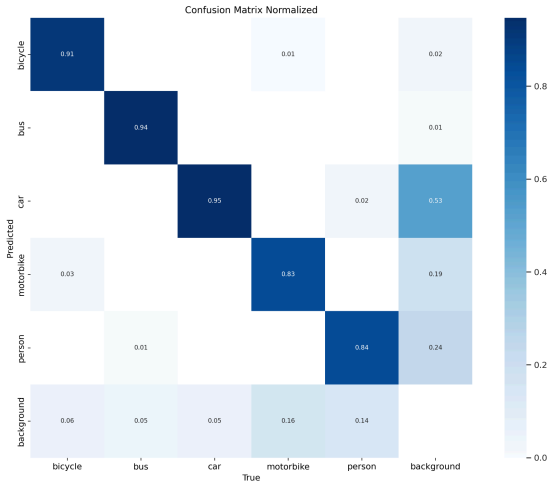


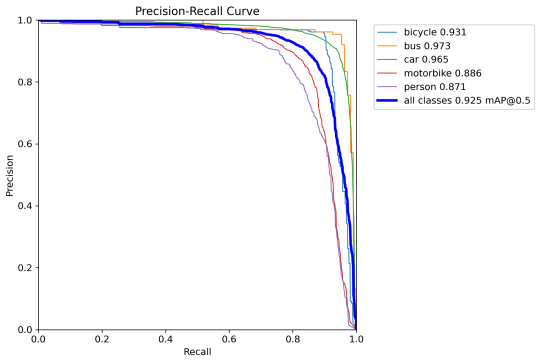



```
Validating runs/train/exp/weights/best.pt...
Fusing layers...
gelan-c summary: 467 layers, 25414815 parameters, 0 gradients, 102.5 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100% 18/18 [00:25<00:00,  1.40s/it]
                   all        549       6270      0.944       0.92      0.955      0.793
               bicycle        549        250      0.957      0.956      0.959      0.832
                   bus        549        108      0.953      0.946       0.98      0.905
                   car        549       3842      0.935      0.941      0.968      0.823
             motorbike        549       1238      0.947      0.863      0.934      0.665
                person        549        832      0.927      0.893      0.934      0.741
Results saved to runs/train/exp
```

5) yolov10-n:

Model	**Speed on GPU [FPS]**	**Speed on CPU [FPS]**	**mAP@0.5**	**mAP@0.5: 0.95**	**# of parameters (M)**	**GFLOPs**
YOLOv10-n	67.56	76.0	0.925	0.714	2.69	8.2

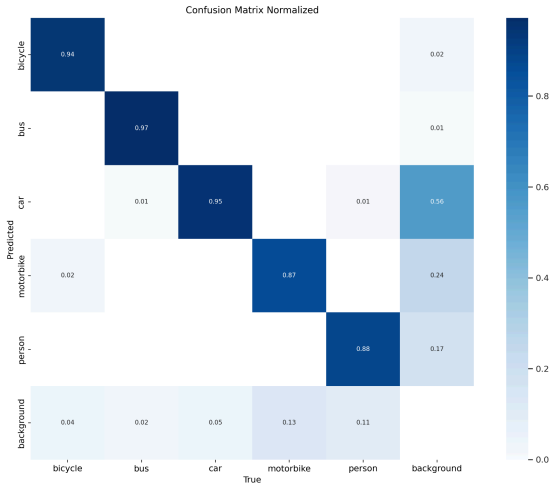


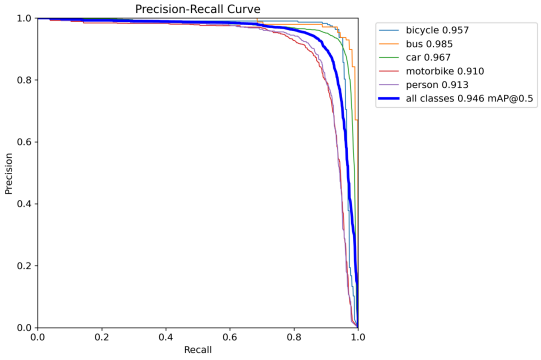



```
Validating runs/detect/train4/weights/best.pt...
Ultralytics YOLOv8.1.34 🚀 Python-3.10.12 torch-2.3.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
YOLOv10n summary (fused): 285 layers, 2696366 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 9/9 [00:19<00:00,  2.14s/it]
                   all        549       6270      0.905      0.874      0.925      0.714
               bicycle        549        250       0.92      0.904      0.931      0.758
                   bus        549        108      0.953      0.948      0.973       0.87
                   car        549       3842      0.914      0.926      0.965       0.77
            motorbike        549       1238        0.9      0.791      0.886      0.574
                person        549        832      0.837      0.799      0.871      0.596
Speed: 0.4ms preprocess, 5.4ms inference, 0.0ms loss, 0.1ms postprocess per image
```

6) yolov10-s:

Model	**Speed on GPU [FPS]**	**Speed on CPU [FPS]**	**mAP@0.5**	**mAP@0.5: 0.95**	**# of parameters (M)**	**GFLOPs**
YOLOv10-s	67.56	73.52	0.946	0.765	8.039	24.5






```
Validating runs/detect/train/weights/best.pt...
Ultralytics YOLOv8.1.34 🚀 Python-3.10.12 torch-2.3.0+cu121 CUDA:0 (Tesla T4, 15102MiB)
YOLOv10s summary (fused): 293 layers, 8038830 parameters, 0 gradients, 24.5 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 9/9 [00:17<00:00,  1.93s/it]
                   all        549       6270      0.924      0.907      0.946      0.765
               bicycle        549        250      0.963      0.933      0.957      0.811
                   bus        549        108      0.919      0.972      0.985      0.897
                   car        549       3842       0.93      0.939      0.967      0.796
             motorbike        549       1238      0.904      0.837       0.91      0.624
                person        549        832      0.906      0.853      0.913      0.696
Speed: 0.2ms preprocess, 11.4ms inference, 0.0ms loss, 0.8ms postprocess per image
Results saved to runs/detect/train
```

Insights:

1) Following is the Table which shows different performance metrics other than mAP@0.5 i.e., mAP@0.5:0.95 , Performance in FPS in GPU as well as CPU,
   Number of model parameters and GFLOPs

Model	Speed on GPU [FPS]	Speed on CPU [FPS]	mAP@0.5	mAP@0.5: 0.95	# of parameters (M)	GFLOPs
YOLOv8-n	89.0	6.3	0.933	0.725	3.01	8.1
YOLOv8-s	82.64	2.6	0.949	0.78	11.12	28.4
YOLOv9-c	16.77	0.323	0.955	0.792	50.96	237.7
Gelan-c	32.89	0.378	0.955	0.793	25.41	102.5
YOLOv10-n	67.56	76.0	0.925	0.714	2.69	8.2
YOLOv10-s	67.56	73.52	0.946	0.765	8.039	24.5

Following graph shows the mAP v/s FPS(GPU) of all models:

Following graph shows the mAP v/s FPS(CPU) of all models:

From the above data we can see that:

1.) yolov9-c and gelan-c have the highest mAP but model FPS is less in GPU as well as CPU.

2.) yolov8-s and yolov10-s are comparable in terms of mAP but FPS is more for yolov8-s on GPU and less in CPU whereas yolov10-s has high FPS in both GPU as well as CPU.

3.) yolov8-n and yolov10-n are comparable in terms of mAP but FPS is more for yolov8-s on GPU and less in CPU whereas yolov10-s has high FPS in both GPU as well as CPU.

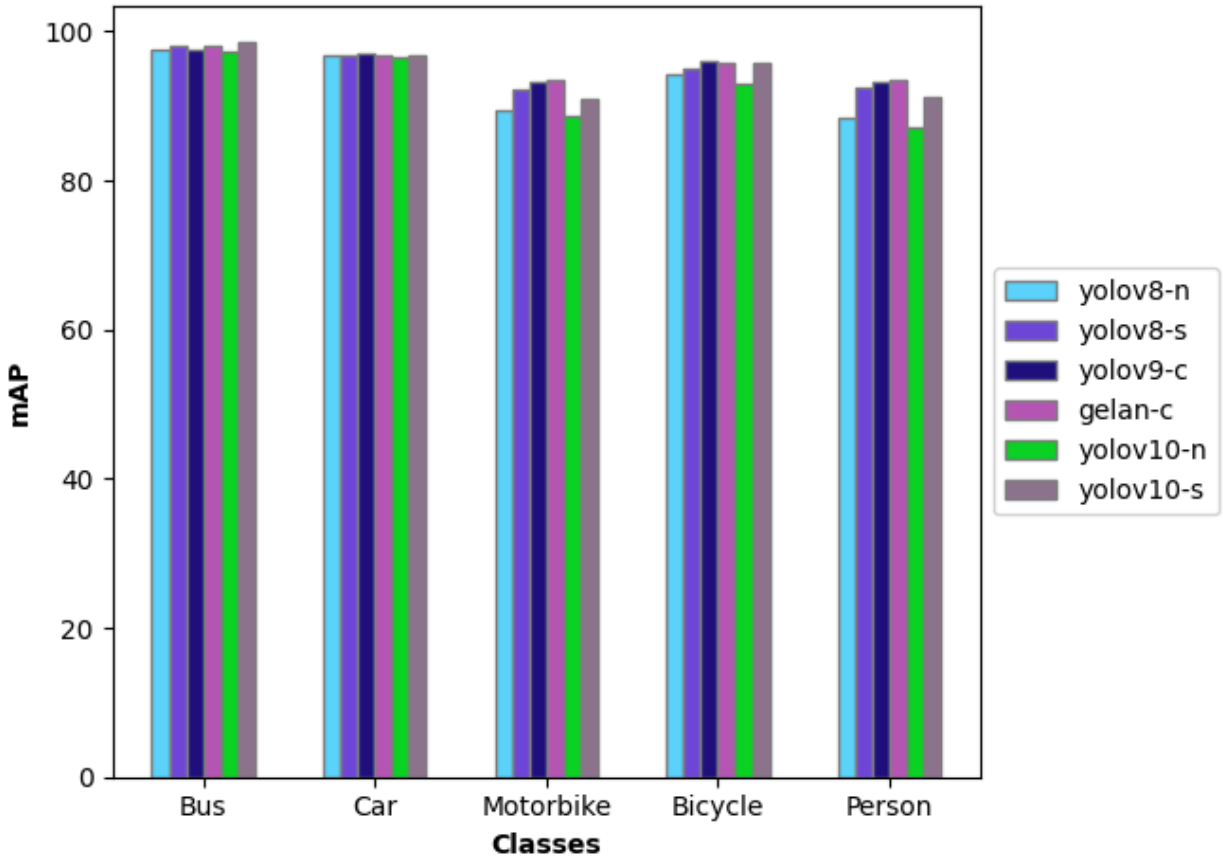
4.) yolov10-s and yolov10-n have less inference time on CPU than on GPU. It looks strange but it is possible for a machine learning model to be faster on a CPU than on a GPU in certain situations. Here are some reasons why this might happen:

1. **Model Size**: Small models, especially those with a small number of parameters, might not benefit much from the parallel processing capabilities of a GPU. The overhead of transferring data to and from the GPU might outweigh the computational benefits.
2. **Model Type**: Some models or algorithms that are not well-suited to parallelization might run faster on a CPU.

   Since, yolov10 is NMS-free as well as its architectural design consist of pointwise and depthwise convolutions which might be more efficient in terms of speed in CPU. Also one of the probable reasons could be the overhead because data transfer between GPU to CPU is drastically reduced in yolov10 when it runs on CPU because of the compact model architecture.

2) mAP across all classes:

Following graph shows mAP of all models across all classes:



From above graph we can say that:

1. yolov9-c and gelan-c are also performing well while detecting smaller objects like person and motorbike.
2. yolov8-n and yolov10-n are not doing good in detecting smaller objects.
3. yolov8-s and yolov10-s are giving comparable performance in detecting small objects as well.

Conclusion:

From the above comparison on several metrics, yolov10-s will be a good choice to get good performance in terms of mAP for all kind of objects i.e., small as well as large. Also it has very high FPS in both GPU and CPU.

For deployment on edge devices:

Deploying models on edge devices often requires reducing model size due to limited computational resources and storage. Here are some techniques and algorithms to achieve this:

1. Model Architecture Optimization

Designing or choosing more efficient model architectures that are inherently smaller and faster

2. Quantization

Quantization reduces the precision of the model's weights and activations, typically from 32-bit floating point to 8-bit integers. This can significantly reduce the model size and improve inference speed without a large loss in accuracy.

- Post-training Quantization: Applies quantization after the model is trained.
- Quantization-aware Training: Incorporates quantization in the training process to reduce accuracy loss.

3. Pruning

Pruning removes weights or entire neurons that contribute little to the model's predictions. This can reduce the model size and improve speed.

- Weight Pruning: Removes individual weights.
- Neuron Pruning: Removes entire neurons or filters in convolutional layers.

4. Knowledge Distillation

In knowledge distillation, a smaller model (student) is trained to replicate the behavior of a larger model (teacher). The student model learns from the teacher's outputs, enabling it to achieve comparable performance with fewer parameters.

In practice, a combination of these techniques can be used to achieve the best results in terms of model size reduction, computational efficiency, and maintaining accuracy.

We can use several tools and libraries such as Pytorch, Tensorflow, ONNX etc. to achieve the model compression.

Possible Approaches for Anomaly detection:

There could be several types of anomalies in traffic surveillance and monitoring. I will write the approach considering the type of anomaly.

1) To detect the vehicles that crosses the speed limits of the roads:

   Each road and highways have their defines speed threshold i.e., it could be min and max allowed speed or only only max speed. To detect this we can detect and track objects using algorithms such as YOLO for Detection of Vehicles and DeepSort for tracking of Vehicles and estimate the speed of the Vehicles. If the speed of the Vehicle is not within the defined speed limits we can say that those Vehicles have broken the traffic rules.

2) To detect heavy vehicles to determine their lane of travel and speed:

   We can detect and track heavy Vehicles and also estimate their speed same as above method. In addition to that we can also detect lanes on the road hence, we can map detected heavy vehicles in which lane it is moving. If it is traveling in the restricted lane for heavy vehicles it can be considered as anomaly.

3) Video Anomaly Detection:

   We can detect the anomalies in videos such as Vehicle-Vehicle collision, Vehicle-Pedestrian Collision, Collision with the Obstacle on roadways, etc.

   In the bonus question-2 I have benchmarked the Algorithm i.e., MOVAD on DoTA Dataset . Go through it to know more.

Possible Approaches if we have to deal with dust or camera is tempered :

1) Image Quality Assessment :

   We can access the quality of an image based on metrics such as Blur, Luminosity, Contrast, IQA which can indicate that the quality of the camera is bad for some reason But we do not know the cause.

2) Blur and dust detection:

   We can segment the region in the image which is dusty or blurr or totally occluded and categorize the cause of it i.e., dust/blur/tempered.

By these methods we can identify whether the camera is tempered or the images are dusty.

After we access the quality of the image we can take necessary steps like not considering the detections for downstream tasks corresponding to the blurred/dusty/tempered region in an

image. Also we can use Advance Image Restoration techniques based on GANs i.e., Generative Adversarial Network.

References:

YOLOv10 vs YOLOv9 — Review of Implementation | by Hamdi Boukamcha | May, 2024 | Medium
Mastering YOLOv10: A Complete Guide with Hands-On Projects | by ProspexAI | May, 2024 | Medium
Object Detection from a Traffic Video (kaggle.com)
2402.13616 (arxiv.org)
YOLOv9 - Ultralytics YOLO Docs
From YOLO to YOLOv8: Tracing the Evolution of Object Detection Algorithms | by Abirami Vina | Nerd For Tech | Medium
YOLOv8 for Object Detection Explained [Practical Example] | by Encord | Encord | Medium
1506.02640 (arxiv.org)
Introducing Ultralytics YOLOv8 by Ultralytics Team
Brief summary of YOLOv8 model structure · Issue #189 · ultralytics/ultralytics · GitHub
2405.14458 (arxiv.org)