YOLOv10: Revolutionizing Real-Time Object Detection

Sunidhi Ashtekar

9 min read · May 28, 2024

Listen Share More

Image from Ultralytics Assets

In the fast-evolving world of computer vision, the YOLO (You Only Look Once) series has consistently set benchmarks for real-time object detection. The latest iteration, YOLOv10, promises to push these boundaries even further. In this blog post, we'll explore the architecture of YOLOv10, highlight its key features, discuss how it outperforms its predecessors and train on custom dataset.

Introduction

YOLOv10 builds upon the strengths of its predecessors, addressing the limitations in post-processing and model architecture. The primary goal is to enhance both performance and efficiency, making YOLOv10 a state-of-the-art model for real-time end-to-end object detection.

YOLOv10 offers a range of model scales, ensuring that users can choose the best model according to their specific application requirements. This flexibility allows for better adaptation in various real-world scenarios, from lightweight models for mobile applications to larger models for high-performance tasks. The YOLOv10 model is available in six sizes:

- Nano (n): 2.3 million parameters
- Small (s): 7.2 million parameters
- Medium (m): 15.4 million parameters
- Big (b): 19.1 million parameters
- Large (l): 24.4 million parameters
- Extra Large (x): 29.5 million parameters

Architecture Overview

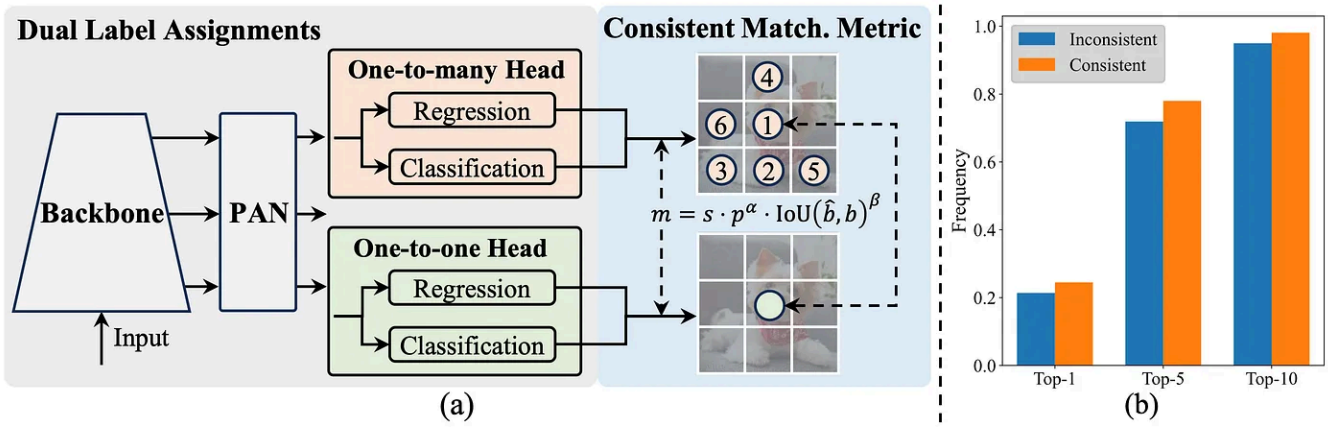


Figure 2: (a) Consistent dual assignments for NMS-free training. (b) Frequency of one-to-one assignments in Top-1/5/10 of one-to-many results for YOLOv8-S which employs α_{o2m}=0.5 and β_{o2m}=6 by default [20]. For consistency, α_{o2o}=0.5; β_{o2o}=6. For inconsistency, α_{o2o}=0.5; β_{o2o} =2.

YOLOv10 Model Diagram from official YOLOv10 repository.

Backbone:

- Responsible for feature extraction, the backbone in YOLOv10 uses an enhanced version of CSPNet (Cross Stage Partial Network). This improvement facilitates better gradient flow and reduces computational redundancy, making the model more efficient.

Neck:

- The neck component aggregates features from different scales and passes them to the head. It includes PAN (Path Aggregation Network) layers, which are designed for effective multiscale feature fusion, ensuring that features from various levels are combined effectively for better object detection.

Consistent Dual Assignments for NMS-free Training:

- Traditional YOLO models rely on non-maximum suppression (NMS) for post-processing, which can increase inference latency. YOLOv10 adopts a consistent dual assignment strategy, eliminating the need for NMS during inference.
- The model employs dual label assignments during training: one-to-many for rich supervision and one-to-one for efficient inference. This approach harmonizes the supervision, leading to better performance without additional inference overhead.

Holistic Efficiency Driven Design

The holistic efficiency driven design includes the modifications mentioned below.

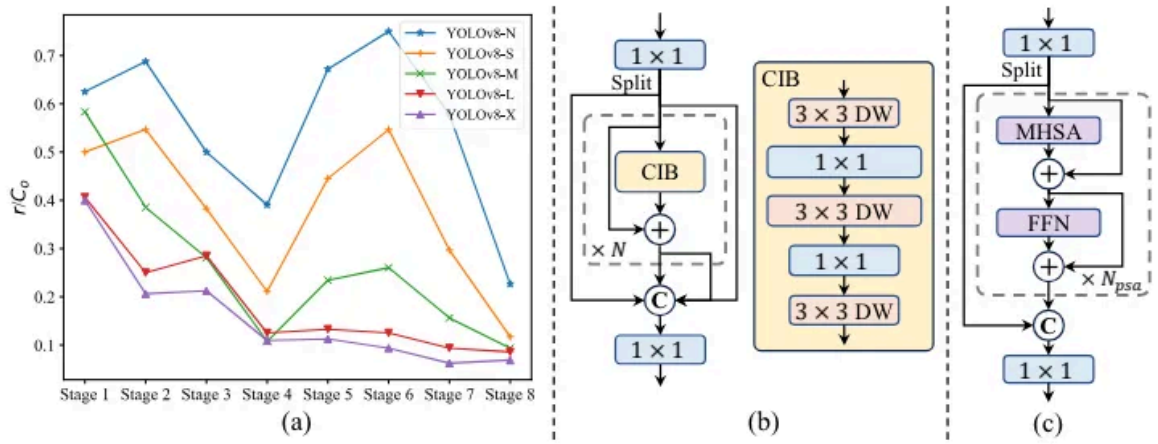


Figure 3: (a) The intrinsic ranks across stages and models in YOLOv8. The stage in the backbone and neck is numbered in the order of model forward process. The numerical rank r is normalized to r/C_o for y-axis and its threshold is set to $\lambda_{max}/2$, by default, where C_o denotes the number of output channels and λ_{max} is the largest singular value. It can be observed that deep stages and large models exhibit lower intrinsic rank values. (b) The compact inverted block (CIB). (c) The partial self-attention module (PSA).

YOLOv10 Holistic Efficiency-Accuracy Driven Model Design from official YOLOv10 paper.

NMS-Free Training:

- Traditional object detection models often use non-maximum suppression (NMS) to remove duplicate bounding boxes. However, this can be computationally expensive, particularly with a large number of detected boxes. YOLOv10 addresses this by training the model to naturally avoid generating multiple bounding boxes for the same object.
- This is achieved through consistent dual assignments, ensuring each object is assigned a single unique bounding box during both training and inference.
- By eliminating the NMS post-processing step, YOLOv10 significantly reduces inference latency, computational cost and inference time. Also, improves accuracy by ensuring each object is represented by a single, high-quality bounding box.

Lightweight Classification Head:

- The classification head is the final layer in a neural network that assigns labels to detected objects. YOLOv10 introduces lightweight classification heads

designed to be efficient without compromising accuracy. This is achieved by reducing the number of parameters and operations in the classification head.

- It reduces computational overhead by simplifying the classification head while maintaining performance.

Spatial-Channel Decoupled Downsampling:

- Downsampling reduces the spatial dimensions of feature maps while increasing channel dimensions. The standard approach using 3x3 convolutions with a stride of two is computationally expensive.
- YOLOv10 improves this by separating spatial and channel operations. Pointwise convolutions adjust the channel dimension without changing the spatial dimension, and depthwise convolutions reduce the spatial dimension while maintaining the adjusted channel dimension.
- Decoupling spatial reduction and channel transformation, reduces computational cost and maximizes information retention.

Rank-Guided Block Design:

- Previous YOLO models used the same basic building blocks across all stages, leading to inefficiencies. YOLOv10 introduces a rank-guided block design that identifies stages with higher redundancy which reduces complexity and replaces them with a more compact inverted block (CIB), which effectively removes unnecessary information and enhances efficiency.
- It uses intrinsic rank analysis to optimize the complexity of different stages, improving efficiency without sacrificing accuracy.

Holistic Accuracy Driven Design

To boost YOLOv10's performance with minimal computational cost, two key features are introduced: large-kernel convolution and partial self-attention (PSA).

Large-Kernel Convolutions:

Large-kernel convolutions expand the receptive field, enhancing the model's capability, especially for larger objects. However, using them universally can dilute small object details and increase latency. YOLOv10 addresses this by:

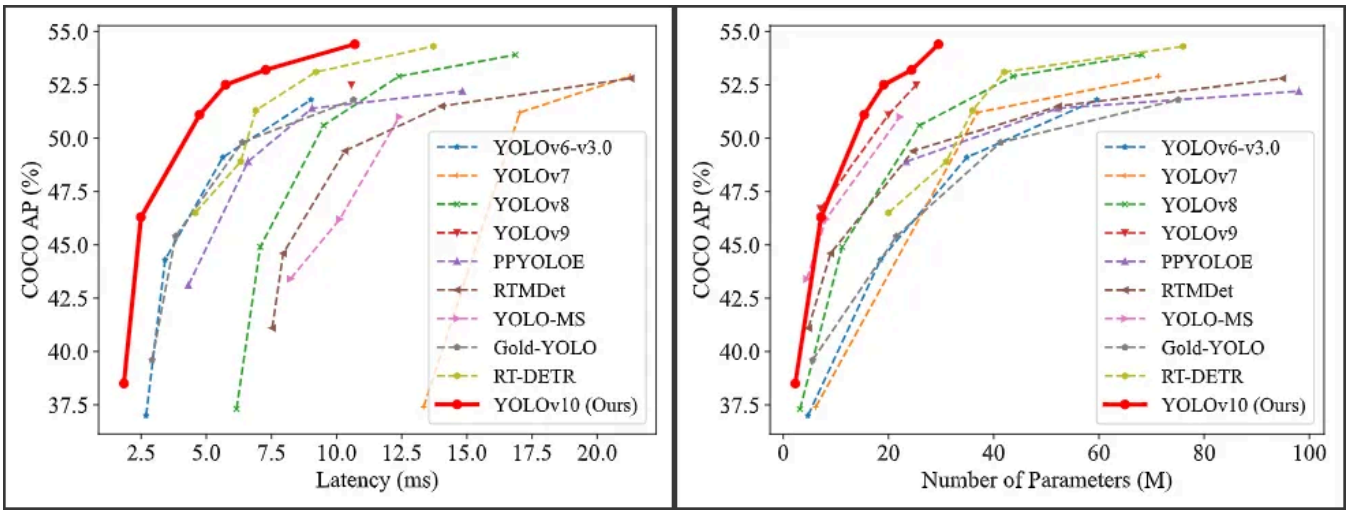
- Selective Application: Using 7×7 depthwise convolutions only in deep stages.
- Structural Reparameterization: Adding an extra 3×3 depthwise convolution branch during training to aid optimization, merged without inference overhead.
- Model Scale Adaptation: Employing large kernels primarily in smaller model scales to avoid diminishing returns as model size increases.

Partial Self-Attention (PSA)

Self-attention excels at global feature modeling but is computationally intensive. YOLOv10 optimizes this with PSA:

- Feature Partitioning: Dividing channels after a 1×1 convolution.
- Selective Processing: Applying self-attention to only one part of the split features.
- Efficient Fusion: Recombining features with another 1×1 convolution.
- Dimension Reduction: Reducing query and key dimensions to half the size of the value in MHSA.
- Optimized Placement: Using PSA only after Stage 4, minimizing computational overhead.

Key Highlights and Improvements



YOLOv10 Performance from the official YOLOv10 repository.

Enhanced Efficiency:

- YOLOv10's architectural improvements lead to substantial reductions in parameters, FLOPs (Floating Point Operations), and latency across various model scales (N/S/M/B/L/X). For instance, YOLOv10-S is 1.8 times faster than RT-DETR-R18 with a similar average precision (AP), and YOLOv10-B achieves 46% less latency and 25% fewer parameters compared to YOLOv9-C.

Superior Accuracy:

- The model achieves state-of-the-art performance across different benchmarks, with significant improvements in AP compared to previous YOLO versions. YOLOv10-L outperforms YOLOv8-L by 0.3 AP with 1.8 times fewer parameters, demonstrating efficient parameter utilization.

Performance:

- Speed: YOLOv10 processes images at remarkable speeds, achieving 2.0ms for preprocessing, 13.4ms for inference, and 1.3ms for postprocessing per image at a resolution of (1, 3, 384, 640). This swift processing time makes YOLOv10 highly suitable for applications requiring rapid and accurate object detection, all while maintaining high accuracy.
- The smallest size processes each image in just 1 millisecond (1000fps), making it the ideal choice for real-time video processing on edge devices.
- High speed on CPU!

Getting Started

To train YOLOv10 on custom dataset these are the steps to follow,

- Environment Setup
- Download Pre-trained Model Weights for YOLOv10
- Install Supervision and Ultralytics packages
- Download datasets from Roboflow and train the model
- Visualization and Analysis
- Get Detection Results

Environment Setup

A Google account is required to access Google Colab, a free cloud service providing necessary computational resources for deep learning tasks, including access to GPUs and TPUs up to 16 GB.

GPU Status Check

Firstly we ensure the availability and readiness of the GPU for processing and running the YOLOv10 model.

```
!nvidia-smi
```

```
!nvidia-smi
```

```
Sat May 25 20:34:08 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA A100-SXM4-40GB          Off | 00000000:00:04.0 Off |                    0 |
| N/A   41C    P0              47W / 400W |      2MiB / 40960MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Mounting Google Drive

Next, we need to navigate to folder where the dataset is stored if you have downloaded the dataset or else we can directly use Roboflow to load the dataset.

```
from google.colab import drive
drive.mount('/content/drive')
```

Display Current Directory

Store current working directory's path in the `HOME` variable for reference.

```
import os
HOME = os.getcwd()
print(HOME)
```

Install YOLOv10 and Supervision Dependencies

To set up the environment for running the YOLOv10 model, we need to install the `supervision` library and clone the YOLOv10 repository. These installations are crucial for ensuring that all necessary tools and libraries are available for the model to function correctly.

```
!pip install -q supervision
!pip install -q git+https://github.com/THU-MIG/yolov10.git
```

Download YOLOv10 Pre-trained Weights

This step involves creating a directory for storing the model weights and downloading the pre-trained weights for various YOLOv10 model sizes. These weights are essential for initializing the model and can be used for both inference and fine-tuning.

```
!mkdir -p {HOME}/weights
!wget -P {HOME}/weights -q https://github.com/jameslahm/yolov10/releases/downlc
!wget -P {HOME}/weights -q https://github.com/jameslahm/yolov10/releases/downlc
!wget -P {HOME}/weights -q https://github.com/jameslahm/yolov10/releases/downlc
!wget -P {HOME}/weights -q https://github.com/jameslahm/yolov10/releases/downlc
!wget -P {HOME}/weights -q https://github.com/jameslahm/yolov10/releases/downlc
!wget -P {HOME}/weights -q https://github.com/jameslahm/yolov10/releases/downlc
!ls -lh {HOME}/weights
```

Load YOLOv10 Model with Pre-trained Weights

In this step, we load the YOLOv10 model using the pre-trained weights downloaded earlier. This allows us to leverage the pre-trained model for inference or further training. The `ultralytics` library is used to facilitate this process.

```
from ultralytics import YOLOv10

model = YOLOv10(f'{HOME}/weights/yolov10n.pt')
```

Perform Inference and Annotate Image with YOLOv10

This cell demonstrates how to perform inference using the YOLOv10 model on an input image. The results are then processed to extract detections, which are used to annotate the image with bounding boxes and labels. The `supervision` library helps in handling the annotations and displaying the results.

```
import cv2
import supervision as sv
from ultralytics import YOLOv10

model = YOLOv10(f'{HOME}/weights/yolov10n.pt')
image = cv2.imread(f'{HOME}/data/dog.jpeg')
results = model(image)[0]
detections = sv.Detections.from_ultralytics(results)

bounding_box_annotator = sv.BoundingBoxAnnotator()
label_annotator = sv.LabelAnnotator()

annotated_image = bounding_box_annotator.annotate(
    scene=image, detections=detections)
annotated_image = label_annotator.annotate(
    scene=annotated_image, detections=detections)

sv.plot_image(annotated_image)
```

Download and Prepare Dataset with Roboflow

This cell sets up the environment for dataset preparation. It creates a directory for datasets, installs the Roboflow library, and uses it to download a specific dataset for YOLOv10. The dataset will be used for training and evaluation purposes.

The project utilizes the RF100 Vehicles Road-Traffic-4 dataset from Roboflow.

```
!mkdir {HOME}/datasets
%cd {HOME}/datasets

!pip install -q roboflow

from google.colab import userdata
from roboflow import Roboflow


rf = Roboflow(api_key="YOUR API KEY")
project = rf.workspace("roboflow-100").project("road-traffic")
version = project.version(4)
dataset = version.download("yolov8")
```

Train YOLOv10 Model on Custom Dataset

This cell initiates the training of the YOLOv10 model using the custom dataset downloaded from Roboflow. The model is trained for a specified number of epochs with a given batch size.

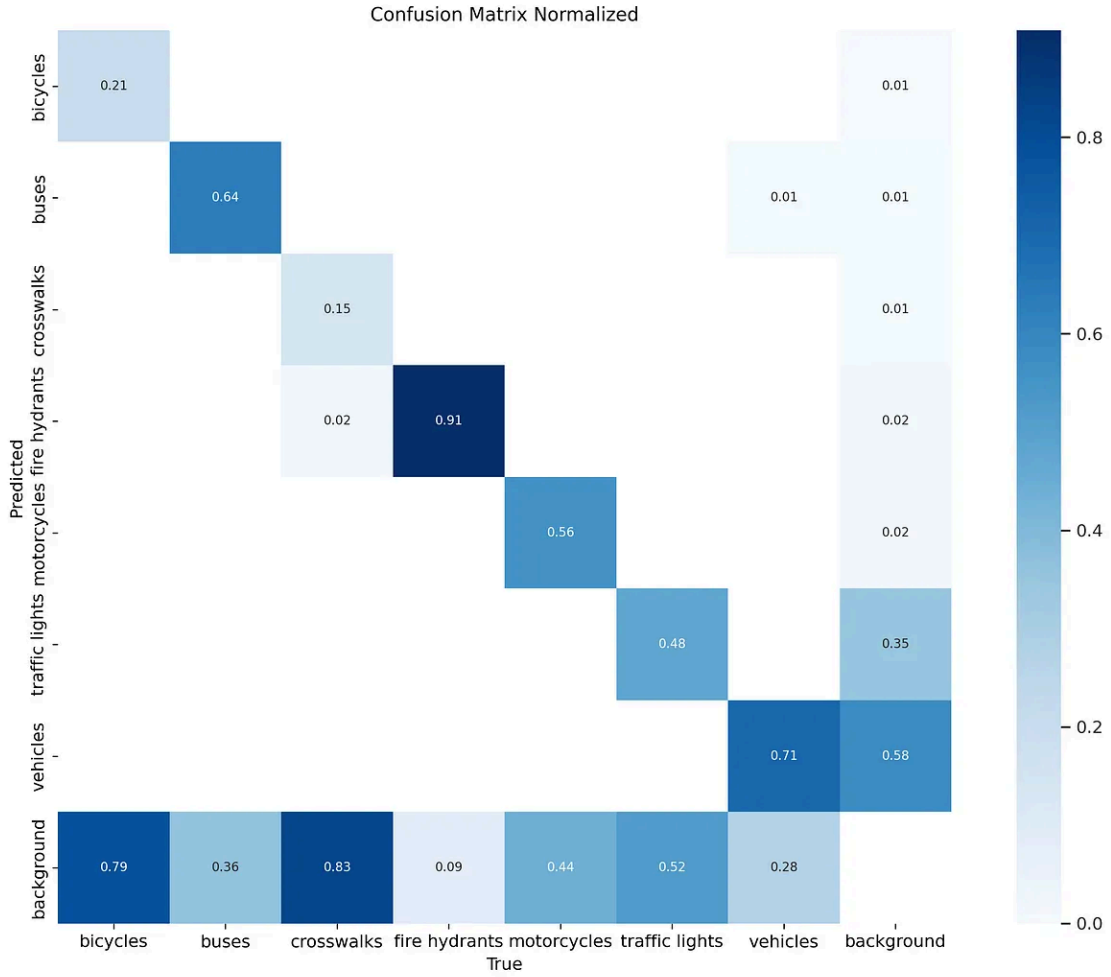
```
%cd {HOME}

!yolo task=detect mode=train epochs=25 batch=32 plots=True \
model={HOME}/weights/yolov10n.pt \
data = /content/datasets/road-traffic-4/data.yaml
```

Display Confusion Matrix of YOLOv10 Training

After training the YOLOv10 model, it's important to evaluate its performance. This cell displays the confusion matrix generated during training. The confusion matrix helps in visualizing the performance of the model by showing the accuracy of its predictions.

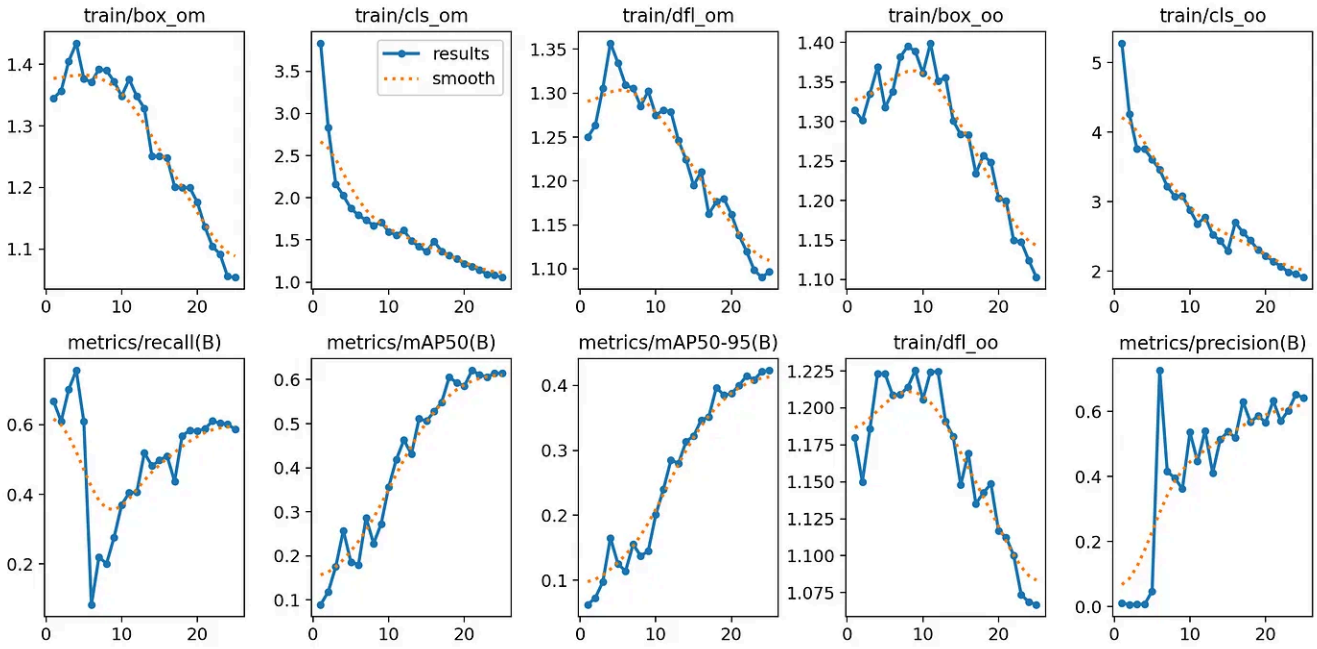
```
%cd {HOME}
Image(filename=f'{HOME}/runs/detect/train/confusion_matrix_normalized.png', wic
```



Display YOLOv10 Training Results

This cell displays the training results image generated by the YOLOv10 training process. This image provides a summary of the model's performance metrics and training progress, helping to visualize the effectiveness of the training.

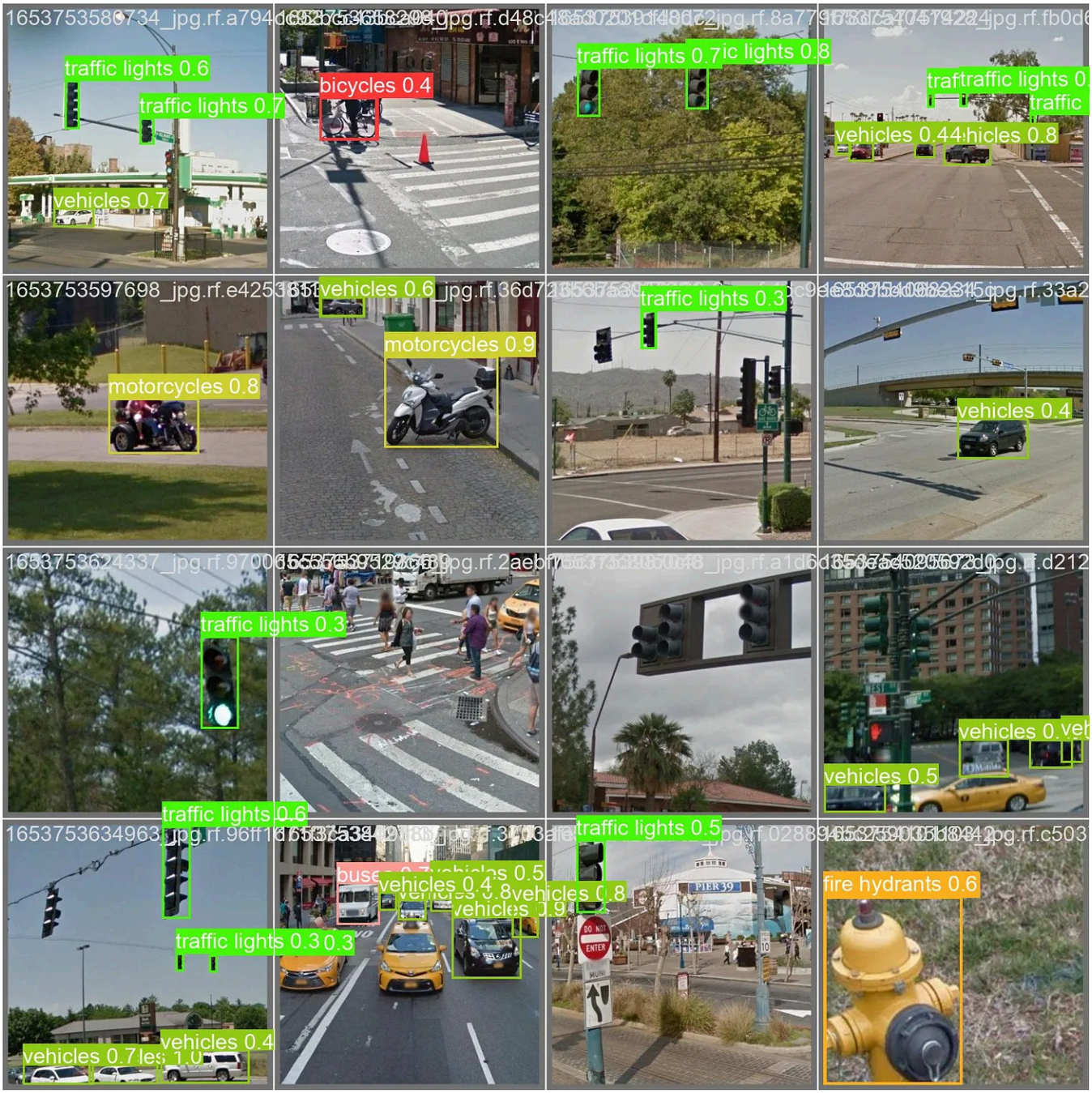
```
%cd {HOME}
Image(filename=f'{HOME}/runs/detect/train10/results.png', width=600)
```



Display Validation Predictions from YOLOv10 Training

This cell displays an image from the validation set, this helps in visualizing how well the YOLOv10 model is performing on the validation data.

```
%cd {HOME}
Image(filename=f'{HOME}/runs/detect/train10/val_batch0_pred.jpg', width=600)
```



Conclusion

I hope this guide has provided a comprehensive overview of the YOLOv10 architecture, its innovative features, and the process of training it on a custom dataset. By leveraging the road traffic dataset from Roboflow, we demonstrated how to utilize YOLOv10 for advanced object detection tasks. This article aimed to highlight the practical applications and potential of YOLOv10 in real-world scenarios.

If you have any questions, recommendations, or critiques, please don't hesitate to reach out on LinkedIn. I'm open to discussions and eager to hear your feedback or

assist with any challenges you might encounter.

References

To further explore the concepts, I recommend visiting the following resources:

- YOLOv10 Official Repository
- YOLOv10 Official Paper
- Ultralytics YOLO Documentation: Detailed documentation and guides on using the YOLOv10 model provided by Ultralytics. Ultralytics YOLO Documentation
- Roboflow Blog: A great resource for the latest updates and tutorials on using Roboflow for computer vision projects. Check out their blog for insightful articles.

This article illustrates the potential of YOLOv10's cutting-edge architecture and innovations. To know more about implementation details and comparison with other state-of-the-art models go through the official paper. Whether for academic research, industry applications, or personal projects, YOLOv10 offers a powerful and efficient solution for exploring and innovating within the vast domain of computer vision.

Computer Vision | Deep Learning | Machine Learning | Object Detection | AI

Edit profile

Written by Sunidhi Ashtekar

101 Followers