Name: Helitha Nimnaka
GitHub: HelithaNimnaka/Vehicle-Detection-Using-CCTV-Camera

Vehicle Detection Using CCTV Camera

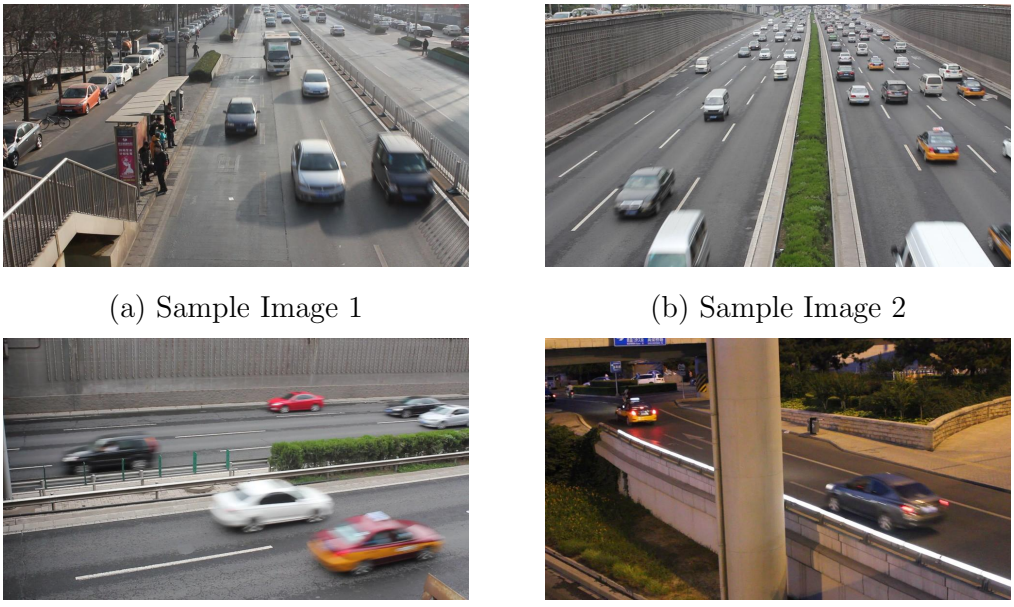
Objective

Develop an optimized vehicle detection model that can identify and count vehicles from a CCTV video feed with high accuracy and real-time performance.

1 Dataset Selection & Preparation

For this project, I have selected the **UA-DETRAC** dataset as the primary source for training and evaluating the vehicle detection model. This dataset comprises real-world traffic surveillance videos captured from CCTV cameras, making it highly suitable for developing vehicle detection models tailored for urban traffic scenarios.

- **Training images:** 83,791 samples
- **Validation images:** 28,170 samples (after test split)
- **Test images:** 28,170 samples (50% of the validation set)
- **Calibration images:** 500 samples (randomly selected from the validation set)
- **Image resolution:** 960 × 540 pixels (resized to 640 × 640)
- **Number of classes:** 0:`truck`, 1:`car`, 2:`van`, 3:`bus`



(a) Sample Image 1 (b) Sample Image 2

(c) Sample Image 3 (d) Sample Image 4

Figure 1: UA-DETRAC Dataset

2 Model Selection & Training

2.1 Ultralytics YOLO11

YOLO11 represents the latest advancement in the Ultralytics YOLO line of real-time object detectors, setting new standards in accuracy, speed, and efficiency. Building on the strong foundation of its predecessors, YOLO11 incorporates major improvements in both architecture and training techniques, positioning it as a flexible solution for diverse computer vision applications.

Out of the available YOLO11 versions, the YOLO11 medium (YOLO11m) model is selected for this project. Depending on the deployment requirements, models ranging from YOLO-Nano to YOLO-Extra-Large can be considered. These models are pretrained on the COCO dataset, which includes 80 object classes.

However, the UA-DETRAC dataset includes only 4 vehicle categories: `car`, `truck`, `van`, and `bus`. Therefore, the prediction head (i.e., the final layer) of the model is modified to match the number of classes in the target dataset.

Table 1: Performance and size metrics of the YOLO11m model

Model	Size (pixels)	$\text{mAP}_{50\text{-}95}$	Speed CPU ONNX (ms)	Speed T4 TensorRT (ms)	Params (M)	FLOPs (B)
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0

2.2 Key Features

- **Improved Feature Extraction:** YOLO11 utilizes an enhanced backbone and neck structure that boosts its ability to extract features, enabling more accurate detection and handling of complex tasks.
- **Designed for Speed and Efficiency:** With upgraded architectural elements and a streamlined training process, YOLO11 delivers faster inference times while striking a balance between precision and performance.
- **Higher Accuracy with Reduced Parameters:** The YOLO11m model achieves greater mean Average Precision (mAP) on the COCO dataset while using 22% fewer parameters compared to YOLOv8m, making it more computationally efficient without sacrificing accuracy.
- **Versatile Deployment:** YOLO11 supports deployment across a variety of platforms including edge devices, cloud environments, and NVIDIA GPU-enabled systems, ensuring broad compatibility.
- **Supports Multiple Vision Tasks:** YOLO11 is capable of tackling a wide array of computer vision problems such as object detection, instance segmentation, image classification, pose estimation, and oriented bounding box detection (OBB).

2.3 Model Architecture Overview

The YOLO11m model architecture is a highly optimized convolutional neural network designed for real-time object detection. It follows a modular design composed of the following key components:

- **Backbone:** The network begins with a series of convolutional layers using `Conv2d`, batch normalization, and SiLU activations. These layers extract low-level features while progressively downsampling the input.
- **C3k and C3k2 Modules:** These modules enhance feature representation through multi-branch convolutional paths and bottleneck layers. They incorporate skip connections and depth-wise convolutions for computational efficiency.

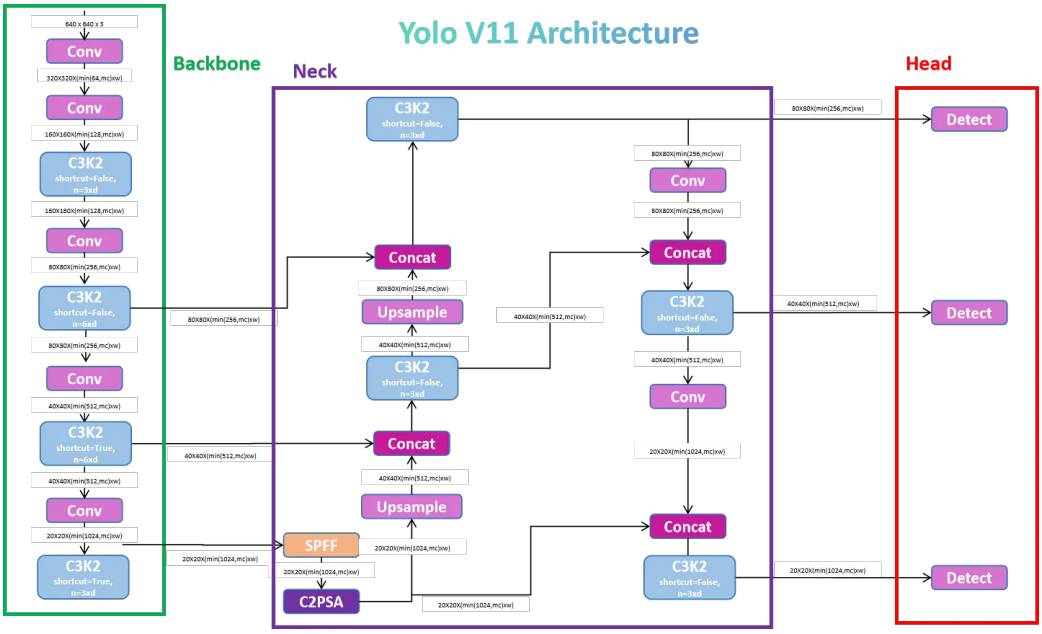


Figure 2: YOLOv11 Model Architecture

- **SPPF Block:** The Spatial Pyramid Pooling – Fast (SPPF) layer aggregates multi-scale spatial features, improving the model's ability to detect objects at various scales.
- **Attention Mechanism:** The C2PSA block integrates parallel self-attention (PSA) modules to enrich contextual information and refine features.
- **Neck and PANet Structure:** The architecture employs upsampling and concatenation operations to fuse multi-resolution features, enabling stronger localization and classification capabilities.
- **Detection Head:** A multi-scale detection head processes fused features from three different scales to detect objects of varying sizes. It includes convolutional blocks with depth-wise convolutions and a Distribution Focal Loss (DFL) layer for improved bounding box regression.

This hierarchical design enables YOLO11m to achieve state-of-the-art detection performance while maintaining real-time inference speeds, even on edge devices.

2.4 Training Configuration

The YOLO11m model is fine-tuned on the UA-DETRAC dataset using the Ultralytics YOLOv8+ framework. The training process is designed to optimize performance while applying a diverse range of augmentation techniques to enhance robustness and generalization. The key aspects of the training configuration are as follows:

- **Model:** Pretrained `YOLO11m` weights are used as the starting point to leverage transfer learning benefits.
- **Dataset:** The UA-DETRAC dataset is defined using a custom YAML file (`ua_detrac.yaml`) specifying training and validation splits.
- **Input Image Size:** All images are resized to `640×640` pixels during training.
- **Epochs and Batch Size:** The model is trained for **25 epochs** with a batch size of 16 and 4 data loading workers.

- **Device:** Training is performed on a GPU (`cuda:0`) for faster computation.
- **Early Stopping:** Training stops early if validation performance does not improve for 10 consecutive epochs (`patience=10`).
- **Layer Freezing:** The first 10 layers are frozen during training to retain low-level features from the pretrained model.

Data Augmentation Techniques: To improve the model's ability to generalize to various scenarios, several augmentation techniques are applied during training:

- **Geometric Augmentations:**
  - `degrees=10`: Random rotation up to 10 degrees.
  - `translate=0.1`: Image translation by up to 10% of width/height.
  - `scale=0.5`: Random scaling within a 50% range.
  - `shear=2`: Shearing transformation with a factor of 2.
  - `perspective=0.001`: Slight perspective distortion.
  - `flipud=0.5, fliplr=0.5`: 50% chance of vertical and horizontal flips.
- **Advanced Augmentations:**
  - `mosaic=1.0`: Always applies mosaic augmentation (combines 4 images into one).
  - `mixup=0.1`: Mixup augmentation with a 10% probability.
- **Color Space Augmentations:**
  - `hsv_h=0.015`: Hue variation of 1.5%.
  - `hsv_s=0.7`: Saturation variation of up to 70%.
  - `hsv_v=0.4`: Brightness (value) variation of up to 40%.

These augmentations allow the model to learn diverse representations of vehicles under various real-world conditions, including changes in lighting, scale, orientation, and appearance.

2.5 Hardware Specifications

All training and inference tasks are conducted on a high-performance desktop system equipped with the following hardware:

- **RAM:** 32 GB DDR4
- **CPU:** Intel® Core™ i9-9900K @ 3.60GHz
  - Architecture: `x86_64`
  - Cores: 8 cores, 16 threads (2 threads per core)
  - Base Clock: 3.60 GHz, Max Turbo: 5.00 GHz
  - L3 Cache: 16 MB
- **GPU:** NVIDIA GeForce RTX 2080 Ti
  - Memory: 11 GB GDDR6
  - CUDA Version: 12.7
  - Driver Version: 565.57.01

These specifications provide sufficient computational resources for both model training and real-time inference, enabling high-throughput performance while maintaining detection accuracy.

2.6 Model Training Progress and Results

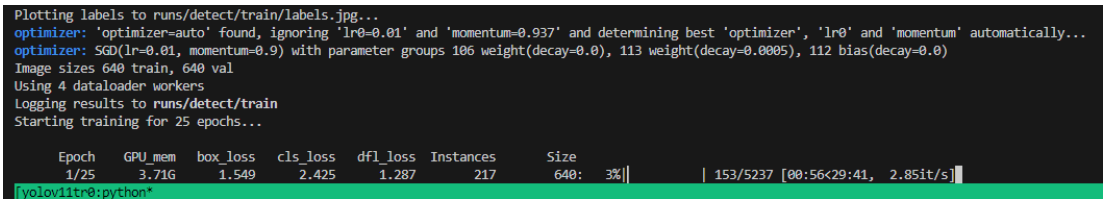


Figure 3: Training initialization: model configuration and start of epoch 1.

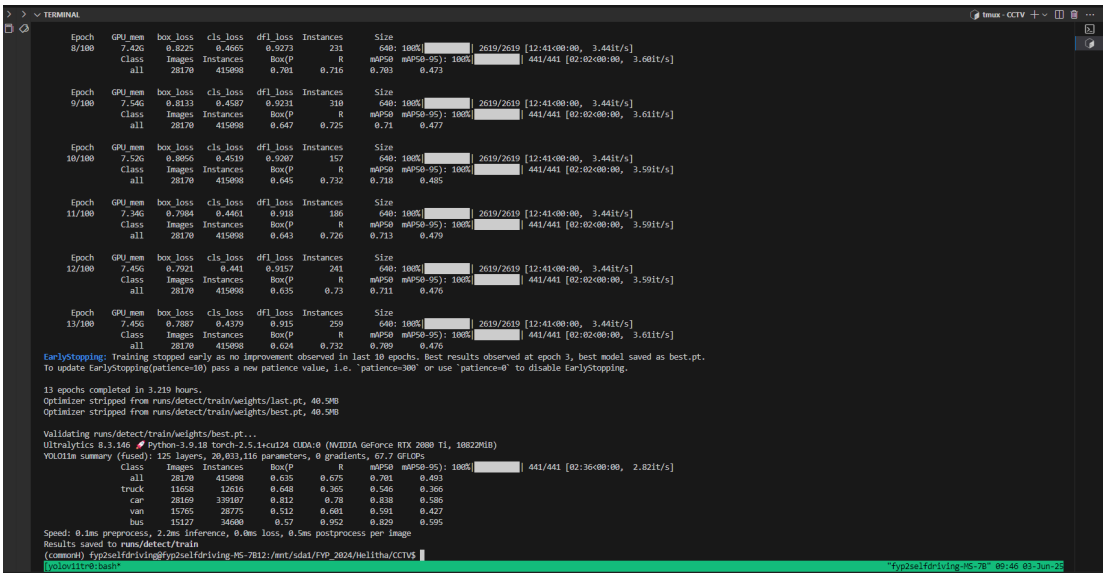


Figure 4: Training completion with early stopping and final evaluation metrics.

The training process was executed using YOLOv11m with a resolution of 640×640 and the SGD optimizer. As shown in Figure 3, the model begins training with 25 planned epochs. However, due to early stopping triggered by a lack of improvement, the training is stopped at epoch 13.

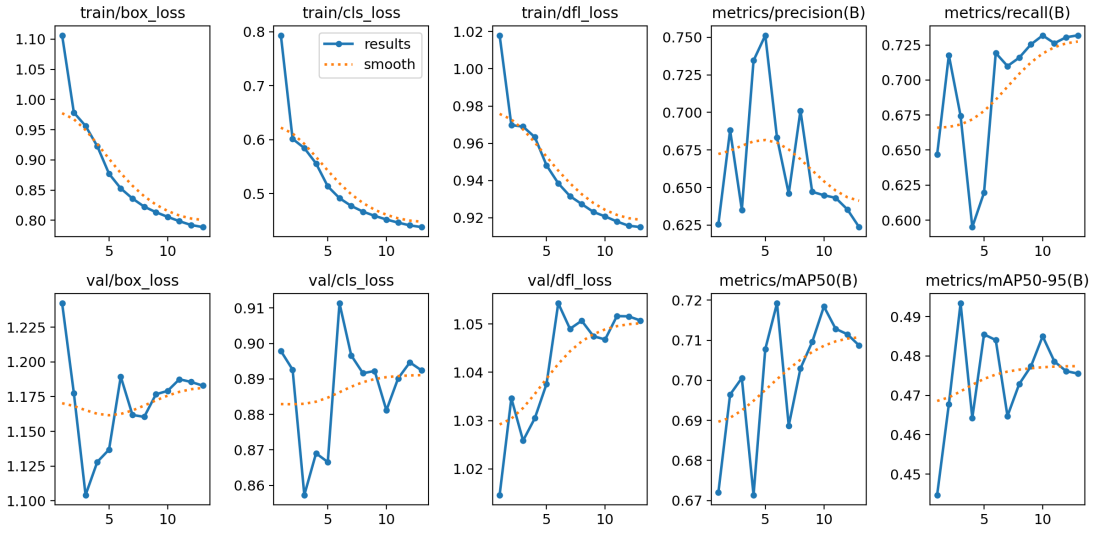
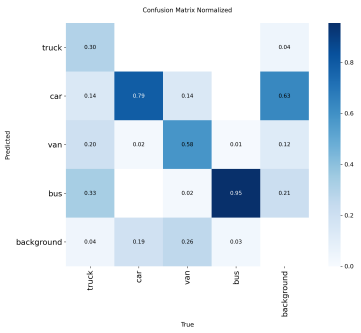
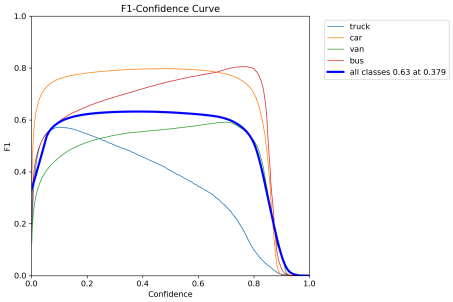


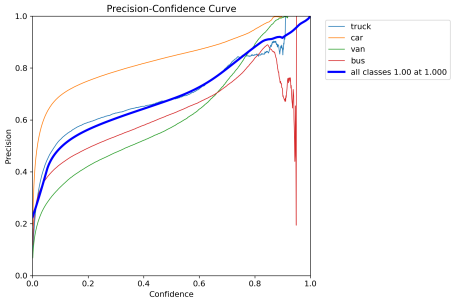
Figure 5: Training Results Metrics



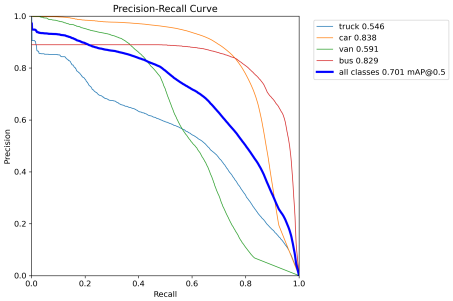
(a) Confusion Matrix (Normalized)



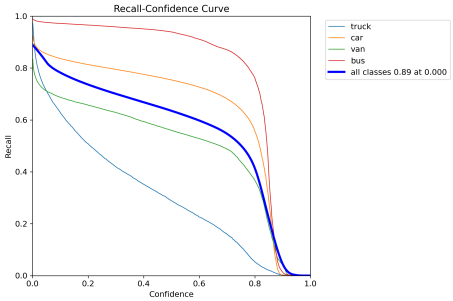
(b) F1 Score Curve



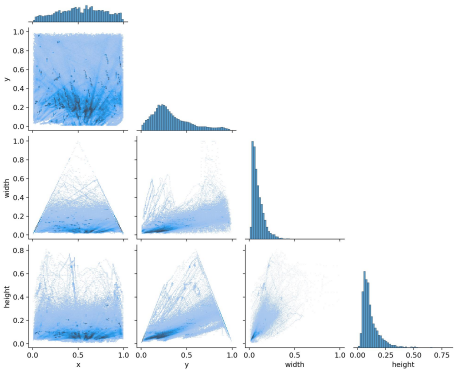
(c) Precision Curve



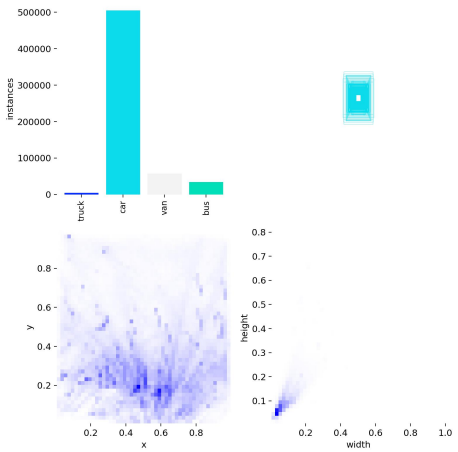
(d) Precision-Recall (PR) Curve



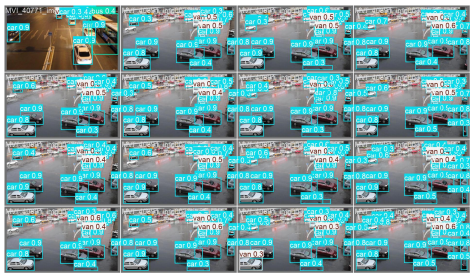
(e) Recall Curve



(f) Labels Correlogram



(g) Labels Distribution

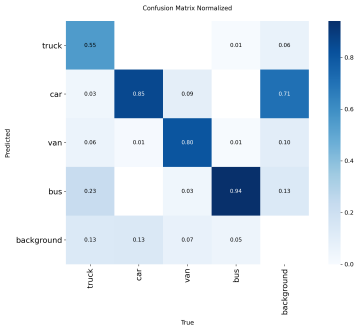


(h) Validation Batch0 Predictions

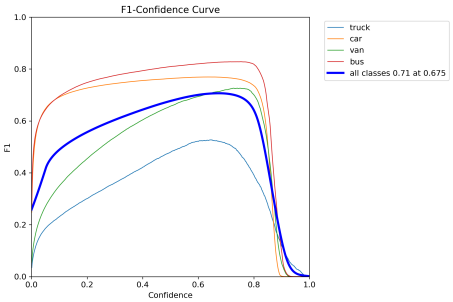
Figure 6: Visualizations of training results.

3 Model Evaluation

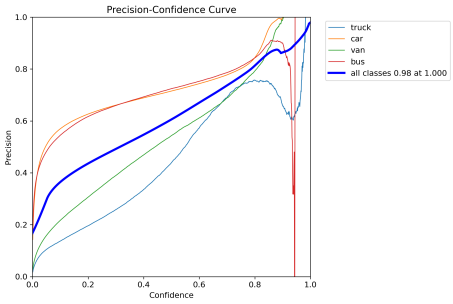
The following visualizations present the evaluation results of the fine-tuned YOLO model on the test dataset. Key performance metrics such as precision, recall, F1 score, and PR curves are included, along with visual outputs from inference on test samples.



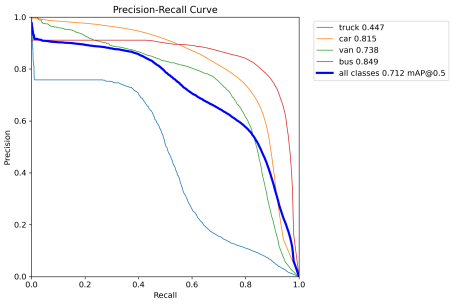
(a) Confusion Matrix (Normalized)



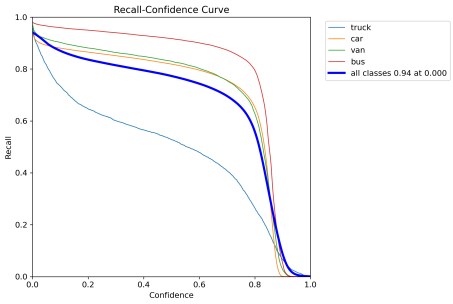
(b) F1 Score Curve



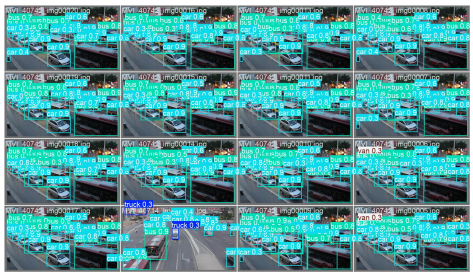
(c) Precision Curve



(d) Precision-Recall (PR) Curve



(e) Recall Curve



(f) Sample Predictions from Test Batch

Figure 7: Evaluation results on the test set.

The evaluation was conducted on a test set consisting of 28,170 images and 260,676 labeled instances. The Ultralytics evaluation tool reported a **fitness score**, reflecting a balanced trade-off between accuracy and speed.

The fine-tuned YOLOv11m model demonstrated strong overall performance, as summarized in below,.

- **185 frames per second (FPS)**
- Fitness Score = 0.528

Table 2: Class-wise Evaluation Metrics on Test Set

Class	Precision	Recall	mAP@50	mAP@50–95
`truck`	0.664	0.429	0.447	0.298
`car`	0.768	0.769	0.815	0.593
`van`	0.665	0.769	0.738	0.553
`bus`	0.777	0.882	0.849	0.586
Overall	**0.718**	**0.712**	**0.712**	**0.507**

Table 3: Per-image Latency Breakdown

Stage	Time (ms)
Preprocessing	0.1
Inference	4.8
Loss Computation	0.0
Postprocessing	0.5
Total	**5.4 ms**

4 ONNX Export for YOLOv11

ONNX (Open Neural Network Exchange) is an open-source format developed by Facebook and Microsoft to facilitate model interoperability among various deep learning frameworks. Exporting a model to the ONNX format enables deployment across diverse runtime environments such as ONNX Runtime, TensorRT, and OpenVINO.

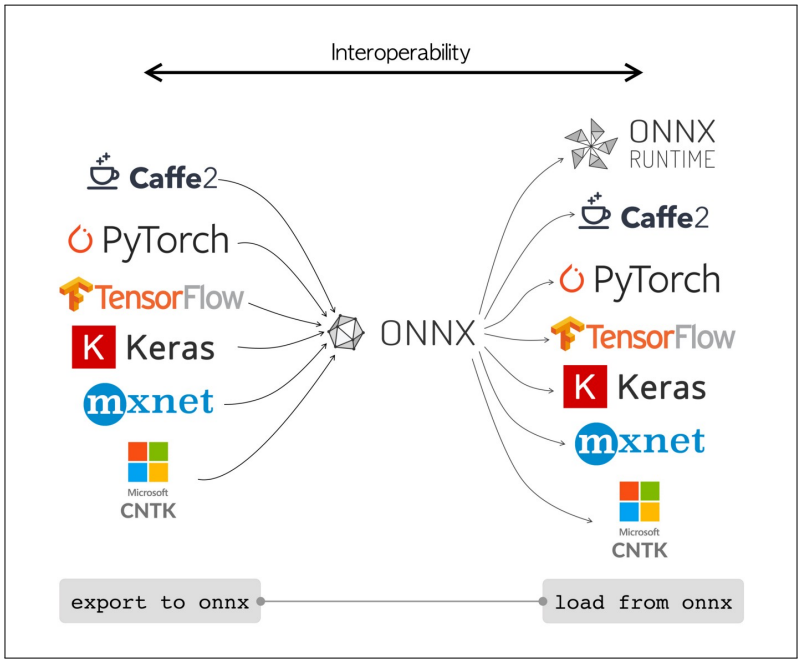


Figure 8: ONNX Model Representation

The fine-tuned YOLOv11m model was successfully converted from the PyTorch `.pt` format to the `.onnx` format using the Ultralytics export API. The exported ONNX model was validated with ONNX Runtime utilizing CUDA GPU acceleration, confirming compatibility with the following execution providers:

- `TensorrtExecutionProvider`
- `CUDAExecutionProvider`
- `AzureExecutionProvider`
- `CPUExecutionProvider`

The evaluation results, based on a test set comprising 28,170 images with 260,676 labeled instances, are summarized below:

- **65 frames per second (FPS)**
- Fitness Score: 0.508

Table 4: ONNX Model Evaluation Metrics (ONNX Runtime with CUDA)

Class	Precision	Recall	mAP@50	mAP@50–95
`truck`	0.639	0.429	0.438	0.278
`car`	0.760	0.767	0.806	0.564
`van`	0.692	0.761	0.742	0.533
`bus`	0.792	0.867	0.858	0.568
Overall	**0.721**	**0.706**	**0.711**	**0.486**

Table 5: ONNX Inference Latency Breakdown (CUDAExecutionProvider)

Stage	Time (ms)
Preprocessing	0.2
Inference	14.3
Loss Computation	0.0
Postprocessing	0.8
Total	**15.3 ms**

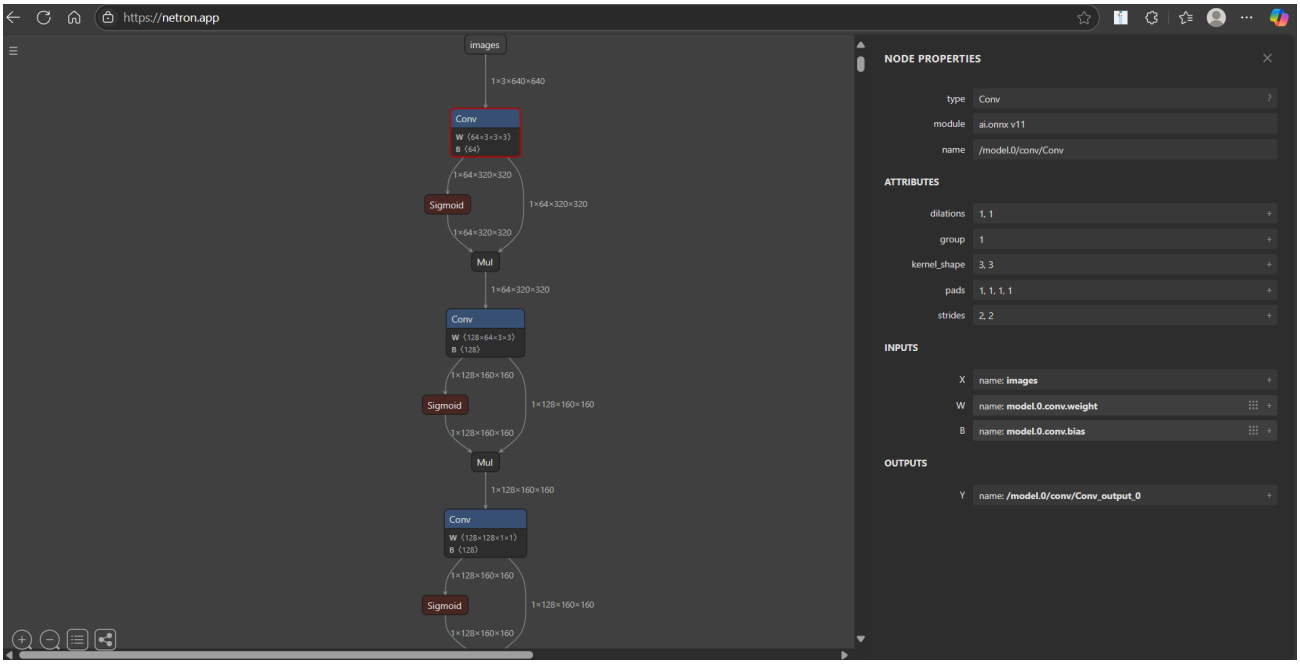


Figure 9: Netron visualization of the `.onnx` model. Visit https://netron.app/

5 TensorRT Export with INT8 Quantization

TensorRT is NVIDIA's deep learning inference library designed to accelerate neural network inference on GPUs. It employs advanced optimizations, including layer fusion, precision calibration (INT8/FP16), and efficient memory management. TensorRT supports various model formats, such as PyTorch, TensorFlow, and ONNX. For INT8 Post-Training Quantization (PTQ), TensorRT uses calibration with at least 500 representative images to accurately determine activation scaling factors. This approach ensures efficient real-time inference, particularly suited to object detection tasks.

To achieve high-performance inference suitable for real-time applications, the trained YOLOv11m model was exported to TensorRT format using INT8 quantization. This significantly reduces computational demands while maintaining robust detection accuracy.

- **Target Export Format:** TensorRT engine (`best.engine`)
- **Precision Mode:** INT8
- **Calibration Data:** 500 representative images specified in `calib.yaml`

Initially, the model was converted from PyTorch to ONNX format (`best.onnx`). TensorRT then executed several optimizations:

- Layer fusion
- Kernel auto-tuning
- Enhanced memory management

INT8 calibration was conducted using 250 batches of calibration images, measuring activation distributions and estimating optimal quantization scales. The resulting INT8-optimized TensorRT engine enables significantly accelerated inference with minimal impact on accuracy, making it ideal for NVIDIA GPU deployment.

Model performance was evaluated using TensorRT inference on the test set, as summarized in Table 6.

- **333 frames per second (FPS)**
- Fitness Score: 0.468

Table 6: TensorRT (INT8) Evaluation Metrics

Class	Precision	Recall	mAP@50	mAP@50–95
`truck`	0.616	0.436	0.429	0.279
`car`	0.744	0.809	0.763	0.534
`van`	0.596	0.789	0.625	0.450
`bus`	0.782	0.873	0.795	0.528
Overall	**0.684**	**0.727**	**0.653**	**0.448**

Inference performance metrics measured on an NVIDIA GeForce RTX 2080 Ti GPU are detailed in Table 7.

Table 7: TensorRT INT8 Inference Latency Breakdown

Stage	Time (ms)
Preprocessing	0.2
Inference	2
Loss Computation	0.0
Postprocessing	0.8
Total	**3 ms**

The optimized TensorRT engine achieves approximately **346 FPS**, greatly suitable for real-time deployments. Despite a modest reduction in mAP compared to the original model, the marked improvement in inference speed underscores the effectiveness of TensorRT INT8 optimization.

6 Real-Time Inference and Vehicle Counting

To evaluate the trained YOLOv11m models in a real-world scenario, a real-time inference pipeline was implemented using a CCTV video feed. The inference process utilizes GPU acceleration and integrates the **ByteTrack** multi-object tracking algorithm through the YOLOv11 tracking interface.

6.1 Inference Methods

- **Batch Mode (File-Based):** The entire video is passed to `model.track(source=...)`, which processes all frames internally, saves annotated video and label files, and then returns results. Unique track IDs are counted by parsing the generated label files.
- **Frame-by-Frame Mode:** Each frame is read via OpenCV's `VideoCapture` and passed individually to `model.track(frame, persist=True)`. Results are obtained immediately, allowing on-the-fly counting of unique IDs without post-processing label files.

6.2 Performance Comparison

PyTorch model with CUDA:

- Speed per image (384 × 640): 1.4 ms preprocessing, 9.3 ms inference, 1.0 ms postprocessing
- **85 FPS**
- **Total unique vehicles:** 609
  - `car`: 379
  - `van`: 343
  - `truck`: 150
  - `bus`: 104

ONNX Runtime (CUDAExecutionProvider):

- Speed per image (640 × 640): 1.8 ms preprocessing, 17.8 ms inference, 1.1 ms postprocessing
- **48 FPS**
- **Total unique vehicles:** 543
  - `car`: 343

  - `van`: 294
  - `truck`: 140
  - `bus`: 105

TensorRT INT8 with CUDA:

- Speed per image (384 × 640): 1.5 ms preprocessing, 2.4 ms inference, 1.1 ms postprocessing
- **200 FPS**
- **Total unique vehicles:** 609
  - `car`: 379
  - `van`: 343
  - `truck`: 150
  - `bus`: 104

This pipeline effectively demonstrates the capability of the model to perform accurate, real-time object detection and multi-object tracking, enabling consistent vehicle identification and counting for robust surveillance applications.

6.3 Multithreaded Tracking

Multithreaded tracking enables simultaneous object tracking across multiple video streams, which is particularly useful for applications such as multi-camera surveillance. By leveraging Python's `threading` module, the system achieves parallel processing of video inputs, enhancing efficiency and real-time performance.

The implementation uses a function `run_tracker_in_thread` that:

- Accepts a video file path, the model to use, and a file index as parameters.
- Reads video frames sequentially.
- Applies the object tracker to each frame.
- Displays the tracking results in real time.

Models Used

Two different models can be used, and each model processes a distinct video source.

Thread Management

- Threads are instantiated using `threading.Thread` with `daemon=True`, ensuring they terminate when the main program exits.
- The `start()` method initiates the threads.
- The `join()` method makes the main thread wait until all tracking threads have finished execution.

Post-Processing

After all threads complete, all OpenCV windows displaying the tracking outputs are closed using `cv2.destroyAllWindows()`.

Scalability

This approach is scalable and can be extended to multiple video files and models by creating additional threads and following the same logic.

7 YOLOv11 Model with OpenVINO INT8 Quantization

OpenVINO™ is an open-source software toolkit developed by Intel, designed to optimize and deploy deep learning models for high-performance inference across a variety of domains, including computer vision, natural language processing, automatic speech recognition, and generative AI. .

Key Features

- **Inference Optimization:** Boosts performance for deep learning models, enabling efficient execution of both small and large models in real-world applications.
- **Flexible Model Support:** Compatible with models trained in popular frameworks such as PyTorch, TensorFlow, ONNX, Keras, PaddlePaddle, and JAX/Flax. Also supports direct integration with models from the Hugging Face Hub via `Optimum Intel`.
- **Broad Platform Compatibility:** Reduces resource demands and supports deployment across a wide range of platforms, including:
  - CPUs (x86, ARM)
  - Intel integrated and discrete GPUs
  - Intel AI accelerators (e.g., Neural Processing Units)

7.1 Evaluation on Test Dataset

The evaluation was conducted on an **Intel® Core™ i9-9900K @ 3.60GHz** processor using 28,170 images and 260,676 labeled vehicle instances. Post-training quantization was performed using 500 calibration images.

- **18 frames per second (FPS)**
- Fitness Score = 0.498

Table 8: Class-wise Evaluation Metrics on Test Set (OpenVINO INT8)

Class	Precision	Recall	mAP@50	mAP@50–95
`truck`	0.604	0.414	0.409	0.258
`car`	0.749	0.772	0.798	0.557
`van`	0.694	0.751	0.726	0.522
`bus`	0.791	0.860	0.851	0.565
Overall	**0.709**	**0.699**	**0.696**	**0.476**

Table 9: Per-image Latency Breakdown (OpenVINO CPU Inference)

Stage	Time (ms)
Preprocessing	0.8
Inference	54.7
Loss Computation	0.0
Postprocessing	0.7
Total	**56.2 ms**

7.2 Real-Time Inference on CPU

- Speed per image (640 × 640): 2.4 ms preprocessing, 57.5 ms inference, 0.8 ms postprocessing
- **16 FPS**
- **Total unique vehicles:** 609
  - `car`: 379
  - `van`: 343
  - `truck`: 150
  - `bus`: 104

8 Further Improvements

- The model is trained to detect only four classes: truck, car, van, and bus. Since the UA-DETRAC dataset contains real CCTV footage, there is an imbalance in the number of instances per class. Alternatively, the model can be trained to detect vehicles in general rather than specific types, depending on the requirements.
- Although the dataset includes various locations and nighttime images, performance can be further improved by fine-tuning the model on additional datasets that include more diverse weather conditions such as snow.
- Optimization can be further enhanced if the target deployment device is known.

References

- **YOLOv11: An Overview of the Key Architectural Enhancements.** Available at: https://arxiv.org/html/2410.17725v1
- **YOLOv11 Architecture Explained: Next-Level Object Detection with Enhanced Speed and Accuracy.**
  Author: S. Nikhileswara Rao. Available at: https://medium.com/@nikhil-rao-20/yolov11-explained-next-level-object-detection-with-enhanced-speed-and-accuracy-2dbe2d376f71
- **Ultralytics YOLOv11**
  Authors: Glenn Jocher and Jing Qiu. Version 11.0.0, 2024.
  Available at: https://github.com/ultralytics/ultralytics
  ORCID: 0000-0001-5950-6979, 0000-0002-7603-6750, 0000-0003-3783-7069
  License: AGPL-3.
- **OpenVINO Documentation (2025)**
  Available at: https://docs.openvino.ai/2025/index.html