Final Project

Real-Time Object Recognition for Urban Driving Video

---

1. Overview

In this assignment, you will build a real-time object recognition system for urban street video, designed as a perception component for autonomous or driver-assistance applications. You will implement the pipeline entirely in a Jupyter notebook using Python, OpenCV, and pretrained deep learning detectors (e.g., YOLO family or comparable real-time models). The system must process live or recorded dash-cam footage, detect and label key road objects, and meet specified real-time constraints while reporting accuracy and latency metrics.

2. Learning Objectives

- Apply a pretrained, state-of-the-art object detector to video in real time within a Jupyter environment.
- Instrument and analyze latency, throughput (FPS), and accuracy trade-offs across model variants and inference settings.
- Perform principled evaluation: mAP@0.5, precision/recall, per-class metrics on a held-out validation split.
- Conduct error analysis to identify failure modes (occlusions, small objects, nighttime, adverse weather).
- Explore practical optimizations (image size, half-precision, confidence/NMS thresholds, model choice).

3. Prerequisites

Python programming, basic PyTorch usage, familiarity with convolutional neural networks, and comfort with Jupyter notebooks. Access to a CUDA-capable GPU is strongly recommended for real-time performance.

4. Dataset Specification

You will use a street-scene dataset appropriate for urban driving object detection. Choose ONE of the following options and adhere to the stated split and labeling policy:

1. **BDD100K (recommended):**

    - Use the detection subset and/or short driving videos. Classes of interest: vehicle (car, truck, bus), pedestrian, cyclist, traffic light, traffic sign. Curate a labeled subset of at least 2,000 images (or 10–15 minutes of labeled video frames) for validation/testing; the remainder may be used for exploratory tuning. Maintain a strict train/val/test separation. If only pretrained inference is used (no fine-tuning), reserve a clean val/test split for evaluation.

2. **Cityscapes / KITTI (alternative):**

   - Use image sequences and annotations for urban driving. Map labels into the assignment's detection classes. Ensure consistent train/val/test splits.

3. **Own dash-cam clip (with ethics & privacy):**

   - Record 5–10 minutes of urban driving video at 720p or higher. If annotations are unavailable, you may evaluate on a small manually labeled sample (≥ 500 frames) or an open labeled subset. Mask faces/plates as required by local privacy regulations.

Mandatory class set:

- vehicle (car, truck, bus)
- pedestrian
- cyclist
- traffic light
- traffic sign

Preprocessing:

Use RGB images with a working resolution around 640×640. Keep aspect ratio with letterboxing if required by the model. Normalize to the model's expected range. For video, decode with OpenCV and buffer frames for timing measurements.

Splits:

Define non-overlapping train/val/test sets. If you do not fine-tune, report metrics on a held-out validation or test split distinct from any tuning data.

Licensing & citation:

Respect dataset licenses. Cite the dataset and any pretrained model repositories you use. Ensure compliance with local privacy laws when recording your own video.

Real Test:

Record 90-120 secs. of street view video to use as a test. It should be a composite of three sections (around 40 secs. each one) showing typical street scenes, traffic, traffic signs, pedestrians, other cars, etc.

5. Tools, Libraries, and Runtime

Environment:

Jupyter Notebook (Python 3.10+). GPU with CUDA 11/12 strongly recommended for real-time.

Core libraries (choose one detector stack):

- PyTorch 2.x, torchvision, OpenCV-Python, NumPy, pandas, matplotlib.
- Ultralytics YOLO (v8/v9) for pretrained models and simple video inference utilities.
- Optional: ONNX Runtime or TensorRT for deployment-grade acceleration.
- Optional (tracking): DeepSORT or ByteTrack for multi-object tracking on top of detections.

Pretrained models (select ≥ 2 for comparison):

- YOLOv8n/s/m or YOLOv9-t/s (COCO-pretrained): speed/accuracy trade-offs; evaluate at least two variants.
- Optional alternatives: RT-DETR, EfficientDet-D0/D1, or a Detectron2 real-time baseline.

6. Real-Time Constraint and Measurement Protocol

Your notebook must demonstrate end-to-end real-time detection on a continuous video stream. Instrument the following metrics:

- • Throughput (FPS): average and 95th-percentile over ≥ 2 minutes of video.
- • End-to-end latency per frame (capture → preprocessing → inference → post-proc → render).
- • GPU/CPU utilization snapshot (if available).

Targets (guidelines, not hard fails):

- GPU: ≥ 25 FPS @ 720p, median latency ≤ 40 ms; CPU-only: ≥ 10 FPS @ 720p, median latency ≤ 100 ms.

Report exact hardware and software versions (GPU model, driver, CUDA, PyTorch), image size, and model variant. Use warm-up runs and exclude the first 50 frames from timing.

7. Evaluation Protocol (Accuracy)

Evaluate detection quality on a labeled validation/test split. Compute: mAP@0.5, precision, recall, F1, and per-class AP. Include PR curves if feasible. For video, you may subsample frames uniformly (e.g., 1 FPS) for labeling/evaluation.

If you add tracking, additionally report IDF1, MOTA/MOTP (if annotations support tracking) or a stability proxy (box IoU overlap across frames).

8. Required Tasks

4. 1) Data preparation: load the dataset, define class mappings, create train/val/test splits.
5. 2) Baseline inference: run a pretrained detector on video; overlay boxes, class labels, and confidence.

6. 3) Real-time instrumentation: measure FPS and latency with accurate timing and warming.
7. 4) Hyperparameter tuning: adjust input size, confidence threshold, NMS IoU, and model variant to meet real-time targets.
8. 5) Accuracy evaluation: compute metrics on the validation/test split; include per-class breakdown.
9. 6) Error analysis: identify at least three failure modes with annotated examples and short explanations.
10. 7) Ablations: compare at least two model variants and two image sizes; discuss trade-offs.

9. Deliverables

- • A single Jupyter notebook (*.ipynb) with all code, figures, and metric tables, runnable end-to-end.
- • A 3–5 minute screen-capture video or animated GIF demonstrating real-time detection on a city-street clip.
- • A short report section inside the notebook covering: setup details, metrics, error analysis, and lessons learned.
- • A requirements file and environment notes (e.g., conda YAML or pip requirements.txt).

10. Grading Rubric (100 pts)

- • Functionality & Real-Time Performance (25): meets or approaches FPS/latency targets; stable live demo.
- • Accuracy & Evaluation Rigor (25): correct metrics; per-class analysis; clear validation protocol.
- • Engineering & Optimization (20): clean notebook; sensible hyperparameter and model-variant exploration.
- • Error Analysis & Discussion (20): insightful examples, failure categorization, and proposed remedies.
- • Reproducibility & Documentation (10): environment details, dataset notes, and clear instructions.

11. Ethics, Safety, and Privacy

This assignment is for research/educational purposes. Do not deploy for surveillance or identification of individuals. Avoid face/plate identification; if present, blur or mask. When recording, comply with local laws and institutional review policies. Discuss potential biases (e.g., performance differences across lighting, weather, neighborhoods) and mitigation strategies.

12. Reproducibility Checklist

- • Random seeds, software versions, and exact pretrained weights specified.
- • Hardware details (GPU/CPU/RAM); CUDA and driver versions.
- • Exact image size, confidence/NMS IoU thresholds, and any pre/post-processing steps.

- • Links or citations for datasets and pretrained models.

Appendix A: Implementation Tips (Non-exhaustive)

- • Start with a small model (e.g., YOLOv8n) and 640-pixel input; measure, then scale up.
- • Use half-precision (FP16) on supported GPUs; pin memory; enable cudnn.benchmark where appropriate.
- • Batch size = 1 for live streaming; prefetch/queue frames separately from inference.
- • Tune confidence and NMS IoU to balance recall vs. duplicate boxes.
- • For tracking, reuse detection embeddings (if available) or use DeepSORT’s appearance model.