CS 4850 – Senior Project, Section 2

Spring 2026

Professor Perry

SP-104 Red: Real-Time Object Detection for Traffic Cameras

Final Report

Team Members: Tate York, Ryan Booth, Parsa Rajabi

Instructor: Sharon Perry

Date: April 20, 2026

Website: https://sp104-traffic-red.github.io/RealTimeTrafficDetection/

GitHub: https://github.com/SP104-Traffic-Red/RealTimeTrafficDetection

Stats and Status

Field	Value
Lines of Code (LOC)	774 (does not include libraries)
Components/Tools	6 – Python, PyTorch, YOLOv8 (Ultralytics), OpenCV, NumPy, CUDA
Hours Estimate	300
Hours Actual	240
Status	Project is 100% complete and working as designed

Table of Contents

1. Introduction

The SP-104 Red Traffic project is a machine learning-based software system designed to automatically detect and classify traffic-related objects in video footage captured by traffic cameras. The system utilizes a fine-tuned YOLOv8 (You Only Look Once) Convolutional Neural Network (CNN) implemented in PyTorch to process visual data and provide real-time analytical feedback.

The project scope includes dataset preparation, model training and evaluation, and deployment of a real-time inference demonstration using video input. The system architecture is designed to support pre-recorded video files. This project emphasizes practical application of deep learning techniques, performance evaluation, and system-level deployment considerations.

The system was developed iteratively across two major versions. Version 1 (V1) used the Kaggle Road Sign Detection dataset (877 images) with YOLOv5 and served as a proof-of-concept. After identifying dataset size and model architecture as the primary bottlenecks, Version 2 (V2) was built using the BDD100K dataset (10,000 images) with YOLOv8s. V2 was structured as a clean template rather than a patch on V1, ensuring architectural consistency and substantially improved detection quality across all metrics.

2. Challenges, Solutions, and Assumptions

2.1 Challenges and Solutions

The largest design decisions faced during development came from version handling and confidence training. The team initially planned to use YOLOv5 with a small Kaggle Road Sign Detection dataset containing only 877 images and 4 object classes (stop signs, crosswalk signs, speed limit signs, and traffic lights). This initial approach (V1) produced unsatisfactory detection results due to insufficient training data and frequent false positives on real-world traffic scenes.

To address this, the team made two critical pivots: first, upgrading the model architecture from YOLOv5 to YOLOv8, which promotes GPU-accelerated training and offers improved detection performance; and second, switching from the small Kaggle dataset to the BDD100K dataset from Berkeley DeepDrive, which contains 70,000 annotated driving images with 10 traffic-related object classes. Although, for the sake of decreasing training time, we only used about 15% of this dataset (10,000 images) which still took about 2.5 hours to train. Using this dataset required developing a new data conversion script (convert_bdd100k.py) to transform BDD100K's per-image JSON labels into YOLO-format .txt files. The V2 codebase was rebuilt from scratch as a clean implementation rather than patching V1.

Additionally, the model's confidence score was initially too low, causing incorrect detections where random objects were labeled as road signs. Through trial-and-error tuning, the confidence threshold was increased from 40% to 60%, which significantly reduced false positives and made results more reliable.

Another major challenge was finding suitable demo videos. Most publicly available traffic videos did not contain enough of the specific signs trained on, so the demo did not always showcase the full extent of the model's detection capability. The vastly more diverse dataset used in version 2 fixed this by simply detecting more object types. So, the demo video does not have to have very specific road signs. Any traffic camera or dashcam video is perfect for this demo.

2.2 Assumptions

- The system assumes input video is of sufficient resolution (at least 480p) to distinguish traffic objects.
- The runtime environment will have access to a CUDA-enabled GPU for optimal performance, though CPU inference is supported at lower frame rates.
- The BDD100K dataset labels are correctly annotated by Berkeley DeepDrive researchers.

- Training with 10,000 images (a subset of the full 70K) provides sufficient data for reasonable detection performance.
- Publicly available traffic video footage is representative of typical driving scenarios.

3. Requirements

3.1 Overview

This section defines the software requirements for the SP-104 Red Traffic object detection system.

3.2 Project Scope

The scope of this project is limited to the software implementation of an object detection pipeline. Key activities include: acquiring and preprocessing the BDD100K dataset; fine-tuning a pre-trained YOLOv8 model using PyTorch; developing a Python-based inference engine to process pre-recorded traffic video; and generating visual output with bounding boxes and class labels overlaid on the original footage.

3.3 Definitions and Acronyms

Term	Definition
mAP	Mean Average Precision – a standard metric for evaluating object detection accuracy
YOLOv8	You Only Look Once version 8 – a real-time object detection architecture by Ultralytics
BDD100K	Berkeley DeepDrive 100K – a large-scale driving dataset with 100K annotated frames
CNN	Convolutional Neural Network – a class of deep learning models for image analysis

NMS	Non-Maximum Suppression – post-processing to remove duplicate bounding boxes
FPS	Frames Per Second – rate at which video frames are processed
COCO	Common Objects in Context – a benchmark dataset with 80 general object classes
Inference	The process of using a trained model to make predictions on new, unseen data
Bounding Box	A rectangular box drawn around a detected object

3.4 Constraints

Environment: The system must be developed in Python 3.9+. The deep learning components must be implemented using PyTorch. Training requires NVIDIA GPU availability; the demo can run on standard laptop hardware.

System: The system must utilize the YOLOv8 architecture. Input data formats are restricted to standard video containers (.mp4, .avi) and image formats (.jpg, .png).

3.5 Functional Requirements

Req ID	Description
REQ-3.1.1	The system shall allow the user to input a filepath to a pre-recorded video file
REQ-3.1.2	The system shall parse video files into individual frames for processing
REQ-3.2.1	The system shall load the fine-tuned YOLOv8 weights (best.pt) upon application startup
REQ-3.2.2	The system shall detect the 10 BDD100K traffic object classes

REQ-3.2.3	The system shall calculate a confidence score (0.0 to 1.0) for each detected object
REQ-3.2.4	The system shall filter out detections with a confidence score below a user-defined threshold (default 0.60)
REQ-3.3.1	The system shall draw a bounding box around each detected object on the video frame
REQ-3.3.2	The system shall display the Class Name and Confidence Score adjacent to each bounding box
REQ-3.3.3	The system shall display the processed video in a window in real-time (or near real-time)
REQ-3.3.4	The system shall calculate and print mAP metrics after a validation run is completed

3.6 Non-Functional Requirements

The system should achieve an inference speed of at least 30 Frames Per Second (FPS) on a standard GPU (e.g., NVIDIA T4 or RTX series). The system should achieve a mean Average Precision (mAP@0.5) of at least 0.70 on the validation dataset. The visualization must clearly distinguish between different object classes using different colored bounding boxes.

3.7 User Interface

The interface is a Command Line Interface (CLI) for launching the script, followed by a graphical window (using OpenCV imshow) displaying the processed video feed.

3.8 Use Case: Run Detection on Video

Actor: User / Developer. **Precondition:** Model is trained; Video file exists.

Flow: (1) User executes the detection script in the terminal, specifying the video source. (2) System loads the YOLOv8 model. (3) System opens the video file. (4) System processes frames in a loop. (5) System displays frames with bounding boxes. **Postcondition:** Video finishes; system closes the window.

4. Analysis and Design

4.1 System Architecture

The system follows a pipeline architecture typical of computer vision applications. Data flows sequentially from the input source through preprocessing, inference, and post-processing steps.

Input Source: Raw video files or streams. **Preprocessing Module:** Resizes images to 640 x 640 pixels and normalizes pixel values (0-1) to match the YOLOv8 input requirements. **Inference Engine:** The core PyTorch model (YOLOv8s) processes the tensor and outputs raw predictions (coordinates, objectness score, class probabilities). **Post-Processing (NMS):** Non-Maximum Suppression removes duplicate bounding boxes and filters low-confidence detections. **Visualization Module:** Overlays the final bounding boxes onto the original frame and displays it to the user.

4.2 Technology Stack

Component	Technology
Language	Python 3.9
ML Framework	PyTorch 2.0+ (CUDA 12.1)
Computer Vision	OpenCV (cv2)
Model	YOLOv8s (Ultralytics)
Dataset	BDD100K (10,000 images)

4.3 Data Design

The system is trained on the BDD100K dataset from Berkeley DeepDrive. The data is organized in YOLO format: Images as .jpg or .png files, and Labels as .txt files corresponding to each image, with format: class_id x_center y_center width height (coordinates normalized between 0.0 and 1.0). The trained model weights are stored in best.pt, a binary file containing the learned parameters loaded into memory at runtime.

4.4 Component Design

The software is divided into the following primary Python modules:

1. **convert_bdd100k.py**: Converts BDD100K per-image JSON labels into YOLO-format .txt files. Handles the mapping of BDD100K category names to numeric class IDs and normalizes bounding box coordinates.
2. **train.py**: Responsible for training the model using the formatted dataset. Uses the Ultralytics YOLOv8 API with a pretrained model (yolov8s.pt). Training is run for 50 epochs with an image size of 640 on GPU. The dataset is referenced through a data.yaml file pointing to training and validation directories.
3. **detect.py**: The main execution script for inference. Uses OpenCV to read video files or webcam feeds, processes frame by frame, passes each frame to the YOLOv8 model, and draws bounding boxes with labels and confidence scores on each frame.
4. **evaluate.py**: Generates mAP metrics using the Ultralytics YOLOv8 val() method against the BDD100K validation set. Produces per-class precision, recall, and mAP@0.5 metrics.

4.5 User Interface Design

Users interact through CLI arguments: --source (path to video file), --conf (confidence threshold, e.g., 0.6), --weights (path to trained model file), --save (flag to save output video). The output window displays detected objects with colored bounding boxes and label tags showing class name and confidence percentage.

5. Development

5.1 System Integration and Workflow Overview

The program works through file management for data formatting and training. Once the YOLO model is tuned on the dataset, it is moved to the detection script where it can begin analyzing user-fed videos, labeling recognized elements as it parses each frame.

The system was developed iteratively across two major versions:

- **Version 1 (V1)** used the Kaggle Road Sign Detection dataset (877 images, 4 classes) with YOLOv5. This served as a proof-of-concept but produced unsatisfactory detection results due to insufficient training data. The model frequently missed detections and produced false positives on real-world traffic scenes.
- **Version 2 (V2)** was built using the BDD100K dataset (10,000 images, 10 classes) with YOLOv8s. The V2 codebase was structured as a clean template rather than a patch on V1, ensuring architectural consistency. V2 demonstrated substantial improvement across all detection metrics, validating the dataset switch and architecture upgrade.

5.2 Development Documentation

convert_bdd100k.py: This script converts the BDD100K dataset's per-image JSON label files into YOLO-format .txt files. It maps BDD100K category names to numeric class IDs and normalizes bounding box coordinates from pixel values to the 0.0-1.0 range expected by YOLOv8. The script creates the required folder structure for YOLO, including separate training and validation folders for both images and labels.

train.py: This script is responsible for training the model using the formatted dataset. It uses the Ultralytics YOLOv8 API and starts with a pretrained model (yolov8s.pt) to leverage transfer learning. Training was run for 50 epochs with an image size of 640, executed on GPU for speed (YOLOv8 promotes GPU usage compared to

CPU-dominant YOLOv5). The script generates performance graphs automatically, which helped track model improvement over time. One issue encountered was that the model would sometimes label random objects as road signs. To address this, the confidence threshold was increased from 40% to 60%, reducing false positives and improving reliability.

detect.py: This script handles running the trained model on video input. It uses OpenCV to read either a video file or a webcam feed and processes the video frame by frame. Each frame is passed into the YOLOv8 model, which returns predictions drawn directly onto the frame with bounding boxes, labels, and confidence scores. The script accepts input through command line arguments for flexibility. It also calculates the delay between frames based on the video's FPS so that playback runs at natural speed.

evaluate.py: This script generates mAP metrics using the Ultralytics YOLOv8 val() method on the BDD100K validation set. It produces per-class breakdown of precision, recall, and mAP@0.5 scores, providing quantitative evidence of the model's detection accuracy.

5.3 Database and External Connections

The program uses the BDD100K dataset from Berkeley DeepDrive (bdd-data.berkeley.edu) for external data access, specifically for fine-tuning the model. The dataset is downloaded by the user and placed into the appropriate file directory for convert_bdd100k.py to process. No traditional database connection is required, as all data is stored as local files (images, labels, and model weights). This is not applicable beyond dataset retrieval.

5.4 Project Setup and Environment Configuration

Prerequisites: Python 3.9+, CUDA-enabled PyTorch, YOLOv8 (Ultralytics), OpenCV, NumPy, NVIDIA GPU for CUDA utilization.

Installation Steps: (1) Clone the git repository. (2) Create the virtual environment (Python 3.9+ must be installed). (3) Activate the virtual environment. (4) Install PyTorch with CUDA for NVIDIA GPUs. (5) Install other dependencies (YOLOv8, OpenCV, NumPy) from requirements.txt. (6) Download and convert the BDD100K dataset. (7) Train the model or use provided best.pt weights. (8) Run the detection script on a video file using the command line.

Verification: In a successful run, the user will see bounding boxes drawn around detected traffic objects in any passed video, with labels and confidence scores displayed for each detection.

6. Test Plan

6.1 Purpose and Scope

The purpose of this section is to define the test plan and report testing results for the SP-104 Red real-time traffic detection system. The system under test is a YOLOv8-based object detection pipeline that processes video input and produces annotated output with bounding boxes around detected traffic objects.

Testing focused on the core functionality including verifying object detection using static images, video files, and live webcam input. Model training execution and data preprocessing were evaluated to ensure the full pipeline functions correctly. Performance during continuous input was also tested to confirm system stability over time.

Out of scope: cloud deployment, user interface testing (no formal UI), advanced ML optimization, and large-scale benchmarking.

6.2 Progression Test Objectives

Ref	Function	Test Objective	Evaluation Criteria	Priority
TC-01	Video Input	Verify system loads .mp4 files	Video opens, frames extracted without error	High
TC-02	Model Loading	Verify best.pt loads at startup	Model initializes without exceptions	High
TC-03	Object Detection	Verify detection of 10 BDD100K classes	mAP@0.5 >= 0.40 on validation set	High
TC-04	Confidence Scores	Verify confidence scores in [0.0, 1.0]	All displayed scores fall within valid range	Medium
TC-05	Confidence Threshold	Verify --conf flag filters detections	Raising threshold reduces displayed detections	Medium
TC-06	Bounding Boxes	Verify boxes drawn around detected objects	Visual overlay with colored rectangles on each detection	High
TC-07	Label Display	Verify class name + confidence displayed	Label text visible adjacent to each bounding box	High
TC-08	Real-Time Display	Verify video plays in near real-time	FPS >= 30 on GPU; no significant frame drops	High
TC-09	mAP Evaluation	Verify evaluate.py produces mAP metrics	mAP@0.5, precision, recall, per-class breakdown printed	High
TC-10	Save Output	Verify --save flag writes output video	Output .mp4 file created in runs/detect/output/	Medium

TC-11	Data Conversion	Verify convert_bdd100k.py output	YOLO .txt files created with correct normalized coordinates	High

6.3 Regression Test Objectives

Regression testing validates that the transition from V1 to V2 resolved known issues and did not introduce new defects.

Ref	Function	Test Objective	Evaluation Criteria
RT-01	V1 vs V2 Detection Quality	Verify V2 outperforms V1 on real traffic video	V2 detects significantly more objects with fewer false positives
RT-02	CLI Compatibility	Verify all CLI arguments still work after V2 refactor	--source, --conf, --weights, --save all function as documented
RT-03	Output Format	Verify bounding box and label rendering unchanged	Visual output format consistent with V1 design

6.4 Other Testing

Security: Not applicable. The system runs locally as a command-line application with no network services, user authentication, or data storage. Input is restricted to local video files and webcam feeds.

Stress and Volume Testing: Volume testing was performed by running inference on extended traffic video clips (5+ minutes) to verify consistent FPS without memory leaks or frame buffer overflows. The system processed over 9,000 frames without

degradation. Stress testing of the training pipeline was implicitly performed by training on 10,000 images over 50 epochs.

Connectivity Testing: Limited in scope. Pre-recorded video files were used for all formal test execution.

Unit Testing: Unit-level verification was performed on the data conversion pipeline (convert_bdd100k.py) by spot-checking output .txt label files against their source JSON annotations. Normalized bounding box coordinates were manually verified for correct conversion from pixel coordinates.

Integration Testing: Verified the end-to-end pipeline: raw BDD100K data is converted to YOLO format, the model is trained on the converted data, evaluation produces mAP metrics against the validation set, and inference runs on video input with bounding box overlays.

6.5 Test Strategy

Build Strategy: The system was developed iteratively across two major versions. V1 used the Kaggle Road Sign Detection dataset (877 images) with YOLOv5 and served as a proof-of-concept. After identifying dataset size as the primary bottleneck, V2 was built using BDD100K (10,000 images) with YOLOv8s. Testing followed each build increment, with model evaluation after every training run.

Test Environment: All testing was performed on personal development machines with NVIDIA GPU support. No shared test server or cloud infrastructure was required. Hardware: NVIDIA RTX 4080 Super GPU, Intel/AMD CPU, 16+ GB RAM, SSD with 50+ GB free. Software: Python 3.9+, PyTorch 2.0+ (CUDA 12.1), Ultralytics YOLOv8 8.x, OpenCV 4.x, Windows 10/11.

Testing Tools:

Process	Tool
Model Evaluation	evaluate.py (Ultralytics YOLOv8 val() method)
Inference Testing	detect.py (OpenCV + YOLOv8 predict)
Data Validation	Manual inspection + Python scripts
Test Case Tracking	Microsoft Word document
Defect Tracking	GitHub Issues
Version Control	Git / GitHub (SP104-Traffic-Red organization)

6.6 Test Execution Schedule

Test Phase	Activities	Timeline	Status
Data Pipeline	Verify conversion script output	March 2026	Complete
Model Training	Train and validate model weights	March–April 2026	Complete
Evaluation	Generate mAP and per-class metrics	April 2026	Complete
Demo Testing	Run inference on traffic video	April 2026	Complete
Regression (V1 vs V2)	Compare V1 and V2 on same video	April 2026	Complete

6.7 Assumptions and Dependencies

Assumptions: Input video resolution is at least 480p. BDD100K labels are correctly annotated. NVIDIA GPU driver and CUDA toolkit are properly installed. Training with

10,000 images provides sufficient performance. Publicly available traffic footage is representative of typical driving scenarios.

Dependencies: BDD100K dataset availability from bdd-data.berkeley.edu (requires free registration). Pre-trained YOLOv8s weights (COCO) from Ultralytics for transfer learning. CUDA 12.1 compatible GPU hardware. GitHub repository access for all team members.

6.8 Entry and Exit Criteria

Entry Criteria: BDD100K dataset downloaded and converted to YOLO format. Model training completed and best.pt weights available. All Python scripts execute without import errors. Test environment configured and verified.

Exit Criteria: All progression test cases (TC-01 through TC-12) executed and results documented. mAP@0.5 evaluation completed on full BDD100K validation set. Regression comparison between V1 and V2 performed and documented. Real-time inference demo runs without crashes on at least one traffic video. All critical and high-severity defects resolved.

7. Test Report

7.1 Test Results Summary

All progression test cases (TC-01 through TC-11) were executed successfully. The V2 system (BDD100K + YOLOv8s) demonstrated substantial improvement over V1 (Kaggle + YOLOv5) across all detection metrics. Regression testing confirmed that the architecture upgrade and dataset switch resolved the core detection quality issues identified in V1, including frequent missed detections and false positives.

Key results: The model achieves real-time inference at 30+ FPS on GPU hardware. The confidence threshold of 60% effectively filters false positives. All CLI arguments function

as documented. The data conversion pipeline correctly produces YOLO-format labels from BDD100K JSON annotations.

7.2 Regression Testing Results

V1 (877 images, Kaggle/Roboflow dataset, YOLOv5) failed to generalize to real-world traffic scenes, producing frequent missed detections and false positives. V2 (10,000 images, BDD100K, YOLOv8s) demonstrated substantial improvement across all metrics. The comparison validates that the dataset switch and architecture upgrade resolved the core detection quality issues.

7.3 Defect Management

Because of the relatively small scope of this project and a relatively small codebase (774 lines of code), the team kept personal communication to track bugs and defects as opposed to using GitHub issues. However, for a larger codebase where it's hard to track bugs, the team would use GitHub issues for defect management.

8. Version Control

All source code and documentation are maintained in a GitHub organization repository (SP104-Traffic-Red) at https://github.com/SP104-Traffic-Red/RealTimeTrafficDetection. Python dependencies are pinned in requirements.txt to ensure reproducible environments. Model weights are versioned by training run (stored in runs/detect/traffic_model/).

The team used Git for all version control operations throughout the project lifecycle. The repository includes the complete codebase, configuration files, and documentation.

9. Conclusion

The SP-104 Red Traffic project successfully developed a real-time object detection system capable of identifying traffic-related objects in video footage using a fine-tuned YOLOv8 model. The project progressed through two major iterations: Version 1 served

as a proof-of-concept using YOLOv5 with a small Kaggle dataset, while Version 2 delivered the final production system using YOLOv8 with the BDD100K dataset.

The V2 system detects 10 classes of traffic objects with real-time inference at 30+ FPS on GPU hardware. The confidence threshold tuning from 40% to 60% effectively reduced false positives. All functional requirements defined in the SRS were met, and comprehensive testing validated the system's accuracy, performance, and reliability.

Key accomplishments include: successful migration from YOLOv5 to YOLOv8 architecture; scaling from a 877-image dataset to a 10,000-image dataset with 10 object classes; achieving real-time detection with accurate bounding boxes and confidence scores; and establishing a complete ML pipeline from data conversion through training, evaluation, and inference.

The project demonstrates practical application of deep learning techniques for traffic analysis and provides a foundation for future enhancements such as expanding to the full BDD100K dataset, adding vehicle tracking across frames, and deploying to edge devices for real-world traffic camera integration.