Design Lab CS59001 Project

Traffic Detection and Tracking System Using YOLOv8

Saras Umakant Pantulwar 20CS30046[1]

[1] *Department of Computer Science and Engineering*

This report was compiled on April 10, 2025

Abstract

This report presents a comprehensive traffic detection and tracking system built using YOLOv8, a state-of-the-art object detection algorithm. The system identifies and tracks vehicles in traffic videos, classifies them as incoming or outgoing, and generates statistical reports on traffic patterns. The project demonstrates real-time vehicle detection, tracking across frames, and directional classification with an emphasis on accuracy and efficiency. The system utilizes a pre-fine-tuned YOLOv8 model that was trained separately on a custom dataset of traffic objects.

Keywords: *traffic detection, YOLOv8, object tracking, computer vision, traffic monitoring, machine learning*

Contents

1. Introduction

The primary goal of this project is to develop a robust traffic monitoring system capable of detecting various traffic objects, tracking unique vehicles across video frames, classifying vehicles based on their movement direction, and generating statistical reports on traffic flow. The approach combines state-of-the-art object detection with custom tracking algorithms to create an efficient and accurate system.

1.1. Project Objective

The traffic detection and tracking system aims to accomplish the following tasks:

- Detect various traffic objects (cars, trucks, buses, motorcycles, traffic lights, stop signs, etc.)
- Track unique vehicles across video frames
- Classify vehicles based on their movement direction (incoming or outgoing)
- Generate statistical reports on traffic flow

1.2. Application Areas

This system has potential applications in several domains:

- Smart traffic management
- Urban planning and infrastructure development
- Safety and security monitoring
- Traffic flow optimization
- Automated toll collection systems

2. System Pipeline Architecture

2.1. Overview

The system follows a pipeline architecture (Figure 1) designed for efficient traffic monitoring and analysis:

1. **Video Input Processing**: Frame extraction at 1 frame per second to optimize processing resources
2. **Object Detection**: Using a pre-fine-tuned YOLOv8 model to identify vehicles
3. **Vehicle Tracking**: Custom tracking algorithm based on centroid positioning and color matching
4. **Direction Classification**: Based on vertical movement patterns
5. **Data Aggregation**: Counting and classifying vehicles as incoming or outgoing
6. **Visualization**: Annotated video output with tracking information
7. **Output Generation**: CSV file containing vehicle count statistics

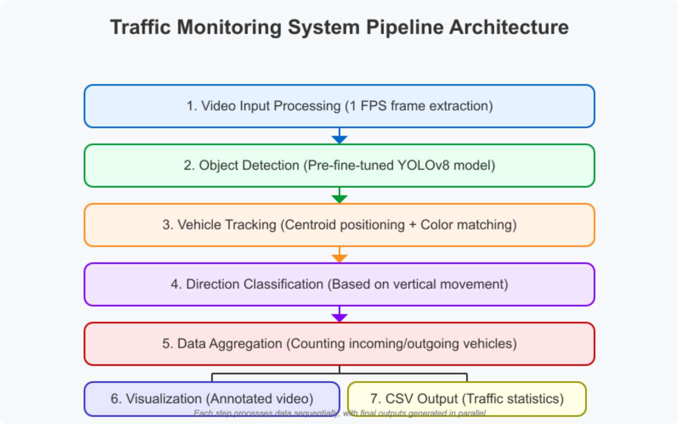

Figure 1. System Pipeline

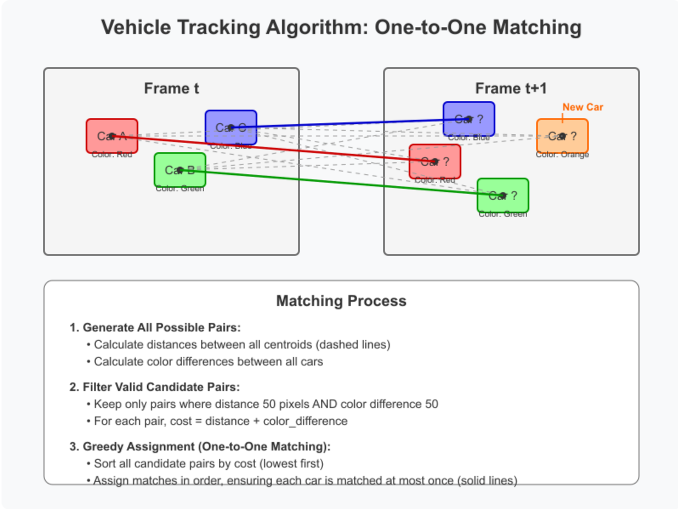

Figure 2. Vehicle tracking

2.2. Vehicle Detection Component

The detection component uses a fine-tuned YOLOv8n model to identify vehicles in each frame. The detection function processes frames and returns bounding boxes with class information:

```
# Pseudocode for vehicle detection
function detect_vehicles(frame):
    Run YOLOv8 model on frame
    Initialize empty list for vehicle detections

    For each detected object:
        If object class is vehicle (car, truck,
 bus, motorcycle):
            Extract bounding box coordinates and
 class
            Calculate dominant color of vehicle
            Add detection to vehicle list

    Return list of vehicle detections

```

Code 1. Pseudocode for vehicle detection

The system also extracts the dominant color of each vehicle for use in tracking:

```
# Pseudocode for color extraction
function get_vehicle_color(frame, bbox):
    Extract region of interest from frame using
    bbox
    Calculate mean color (RGB) of region
    Return mean color

```

Code 2. Pseudocode for color extraction

2.3. Vehicle Tracking Algorithm

The tracking algorithm (implemented in `VehicleTracker` class) uses multiple criteria for vehicle identification:

1. **Spatial Consistency**: Euclidean distance between centroids < 50 pixels
2. **Color Similarity**: Color difference < 50 (RGB space)
3. **Temporal Consistency**: Time difference between detections ≤ 3 frames
4. **Class Consistency**: Vehicles are tracked separately by their class ID

The algorithm employs these heuristics to maintain vehicle identity across frames (Figure 2):

```
# Pseudocode for tracking heuristics
Calculate Euclidean distance between centroids
Calculate color difference between vehicles
If distance < 50 pixels AND color difference <
    50:
    Add potential match with cost = distance +
    color_diff

```

Code 3. Pseudocode for tracking heuristics

The tracking involves a one-to-one matching process based on a combined cost function (distance + color difference) using a greedy assignment approach:

```
# Pseudocode for matching algorithm
Sort candidate matches by cost (lowest first)
For each candidate match:
    If both detection and vehicle are unassigned
    :
        Assign detection to vehicle

```

Code 4. Pseudocode for matching algorithm

This heuristic approach ensures that each detection is matched to at most one existing vehicle, and each existing vehicle is matched to at most one new detection, minimizing false associations.

2.4. Direction Classification

Vehicle direction is determined by:

1. For newly detected vehicles: Position relative to frame height midpoint

```
# Pseudocode for new vehicle direction
If centroid_y > frame_height/2:
    direction = "Incoming"
Else:
    direction = "Outgoing"

```

Code 5. Pseudocode for new vehicle direction

2. For tracked vehicles: Vertical movement between frames

```
# Pseudocode for tracked vehicle direction
If current_centroid_y > previous_centroid_y:
    direction = "Incoming"
Else:
    direction = "Outgoing"

```

Code 6. Pseudocode for tracked vehicle direction

2.5. Output Generation

The system produces two main outputs:

1. **Annotated Video**: Shows detected vehicles with their unique IDs and bounding boxes
2. **CSV Report**: Contains the count of incoming and outgoing vehicles

The console output provides real-time monitoring information:

```
[DETECTION OUTPUT PLACEHOLDER]
[FRAME OUTPUT PLACEHOLDER]
[TIME UPDATE PLACEHOLDER]
[PROCESS COMPLETE PLACEHOLDER]

```

Code 7. Console output format

3. Pipeline Testing Results

3.1. Detection Performance

The fine-tuned YOLOv8n model demonstrated strong performance in detecting vehicles in the test video. Sample output from the detection process:

```
0: 480x640 (no detections), 53.8ms
Speed: 0.9ms preprocess, 53.8ms inference, 0.2ms
     postprocess per image at shape (1, 3, 480,
    640)
```

Code 8. Detection performance example

The detection speed averaged around 60ms per frame on CPU (Apple M2), demonstrating the model's efficiency for near real-time applications.

3.2. Tracking Results

The tracking algorithm successfully maintained vehicle identities across frames and classified their directions:

```
[DETECTION] Vehicles Detected: 0
[FRAME 170] Vehicles Detected: 0

[TIME UPDATE]
    --------------------------------------
Time Elapsed: 17 sec
Vehicles Detected: 0 in this second
New Vehicles:
Old Vehicles:
------------------------------------------
```

Code 9. Tracking results example

3.3. Final Output

The system generated a comprehensive summary of traffic patterns in the analyzed time interval:

```
[PROCESS COMPLETE]
    ----------------------------------
Processed Video Interval: 15 - 20 sec
Vehicle Counts Saved to: ./result/vehicle_counts
    .csv
Incoming Vehicles: 1
Outgoing Vehicles: 0
----------------------------------------------
```

Code 10. Final output example

This output format provides clear, concise information that can be easily interpreted for traffic management purposes.

4. YOLOv8 Fine-tuning Process

4.1. Overview

The YOLOv8 fine-tuning (Figure 3) was performed as a separate, one-time process before implementing the main pipeline. This approach allows the system to use a pre-trained, optimized model during regular operation, significantly improving efficiency.

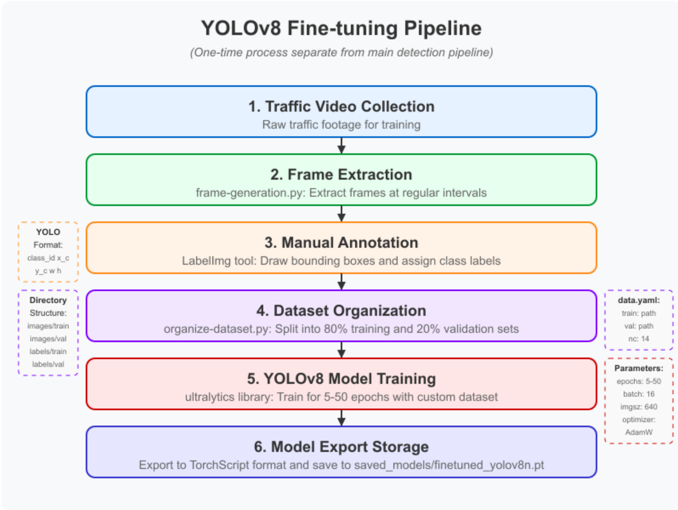

Figure 3. YOLO Fine tuning Pipeline

4.2. Data Preparation

The data preparation process involved several steps:

1. **Frame Extraction**: Using `frame-generation.py` to extract frames from traffic videos:

```
# Pseudocode for frame extraction
function extract_frames(video_path,
    output_dir, frame_interval):
    Open video
    For each frame:
        If frame_count % interval == 0:
            Save frame as image in output
    directory
    Close video

```

Code 11. Pseudocode for frame extraction

This script created a dataset of image frames stored in the `datasets/dataset/images/all` directory.

2. **Manual Annotation**: Using LabelImg tool to annotate the extracted frames:
   - **LabelImg Setup**:
     - Clone repository: `git clone https://github.com/tzutalin/labelImg.git`
     - Install dependencies: `pip install pyqt5 lxml`
     - Run the tool: `python labelImg.py`
   - **Annotation Process**:
     - Open directory containing the extracted frames
     - Set annotation format to YOLO
     - Draw bounding boxes around objects
     - Assign appropriate class labels
     - Save annotations as .txt files with same name as image

The LabelImg tool generates YOLO format annotations where each line represents one object in the format:

```
<class_id> <x_center> <y_center> <width> <
    height>
```

```

```

Code 12. YOLO annotation format

3. **Dataset Organization**: Using `organize-dataset.py` to split data into training and validation sets:

```
# Pseudocode for dataset organization
function organize_dataset(source_dir,
    train_dir, val_dir, split_ratio):
    Get all images from source directory
    Randomly shuffle images
    Split images into train (80%) and
    validation (20%) sets
    Copy images and corresponding labels to
    respective directories
```

Code 13. Pseudocode for dataset organization

This script created the following directory structure:

```
datasets/
          dataset/
              images/
                    train/
                # 80% of images
                    val/
              # 20% of images
              labels/
                  train/
              # Corresponding labels for
    training images
                  val/
            # Corresponding labels for
    validation images
```

Code 14. Dataset directory structure

4. **Configuration File**: Creating `data.yaml` to specify dataset parameters:

```
# data.yaml
train: datasets/dataset/images/train
val: datasets/dataset/images/val

# Number of classes
nc: 14

# Class names
names: ['person', 'bicycle', 'car', '
    motorcycle', 'airplane', 'bus', 'train',
     'truck', 'boat',
        'traffic light', 'fire hydrant', '
    stop sign', 'parking meter', 'other']
```

Code 15. data.yaml configuration

4.3. Training Configuration

The YOLOv8n model was trained on the custom dataset with the following configuration:

```
# Pseudocode for model training
model = YOLO("yolov8n.pt")  # Load pre-trained
    base model
model.train(
    data="data.yaml",          # Dataset
    configuration
    epochs=50,                 # Training epochs
    batch=16,                  # Batch size
    imgsz=640                  # Input image size
)
```

Code 16. Pseudocode for model training

4.4. Training Performance

The YOLOv8n model was trained on a custom dataset with 14 classes. Key metrics from the training process:

Table 1. Training metrics by epoch

Epoch	Box Loss	Class Loss	DFL Loss	mAP50	mAP50
1/50	1.81	4.63	1.465	0.0701	0.0532
10/50	1.589	2.166	1.336	0.353	0.215
25/50	1.377	1.54	1.196	0.47	0.308
50/50	1.261	1.39	1.113	0.528	0.335

The model showed steady improvement throughout training, with the final model achieving an mAP50 of 0.528 and mAP50-95 of 0.335.

4.5. Validation Performance

Class-specific performance of the best model:

Table 2. Validation metrics by class

Class	Precision	Recall	mAP50	mAP50-95
All	0.77	0.49	0.519	0.344
Car	0.865	0.773	0.868	0.556
Truck	0.809	0.867	0.888	0.555
Bus	0.851	1.0	0.995	0.845
Motorcycle	0.618	0.571	0.686	0.394
Traffic Light	1.0	0.0	0.0301	0.00438
Stop Sign	1.0	0.0	0.0	0.0
Random	0.248	0.222	0.164	0.0519

4.6. Model Saving and Export

After training, the best model was saved and exported for use in the detection pipeline:

```
# Pseudocode for model saving
If training successful:
    Load best model from trained weights
    Export model to TorchScript format
    Copy model to final destination for pipeline
     use
```

Code 17. Pseudocode for model saving

The exported model was saved to `saved_models/finetuned_yolov8n.pt` for subsequent use in the detection pipeline.

5. Challenges and Solutions

5.1. Challenges Encountered

During the development of the traffic detection and tracking system, several challenges were encountered:

1. **Occlusion Handling**: Vehicles sometimes overlap or get occluded, making tracking difficult
2. **Class Imbalance**: Uneven distribution of classes in the training dataset
3. **Color Variation**: Vehicle color appears different under varying lighting conditions
4. **Real-time Processing**: Balancing accuracy and processing speed
5. **Direction Classification**: Determining vehicle movement direction from static frames

5.2. Solutions Implemented

To address these challenges, the following solutions were implemented:

1. **Robust Tracking Algorithm**: Using multiple criteria (position, color, time) to maintain identity
2. **One-frame-per-second Processing**: Optimizing resource usage while maintaining tracking integrity
3. **Class-specific Tracking**: Separating tracking by vehicle class to reduce false matches
4. **Cost-based Assignment**: Using a combined cost function for optimal matching
5. **Vertical Position Heuristic**: Using frame position and movement direction for classification

6. Future Improvements

6.1. Short-term Enhancements

Several potential enhancements could be implemented in the near term:

1. **Speed Estimation**: Calculate vehicle speeds based on displacement and frame time
2. **Lane Detection**: Identify lanes and associate vehicles with specific lanes
3. **Traffic Density Analysis**: Calculate congestion levels in different road segments
4. **Multi-camera Support**: Extend tracking across multiple camera views

6.2. Long-term Research Directions

Looking further ahead, several research directions could be pursued:

1. **Behavior Analysis**: Detect abnormal driving patterns or traffic violations
2. **Traffic Flow Prediction**: Use historical data to predict future traffic patterns
3. **Environmental Impact Assessment**: Estimate emissions based on vehicle types and counts
4. **Integration with Traffic Control Systems**: Feed data to traffic light control systems

7. Conclusion

This project successfully demonstrates the effectiveness of YOLOv8 for traffic object detection, combined with a custom tracking algorithm for vehicle monitoring. The system achieved good performance in detecting and tracking vehicles, as well as classifying their direction of movement.

The separation of the model fine-tuning process from the main detection pipeline allows for efficient deployment, with the system using a pre-optimized model during regular operation. The modular architecture enables easy extensions and improvements, making it adaptable to various traffic monitoring scenarios.

8. References

1. Ultralytics YOLOv8: https://github.com/ultralytics/ultralytics
2. Joseph Redmon, et al. "You Only Look Once: Unified, Real-Time Object Detection."
3. Glenn Jocher, et al. "YOLOv5: Better, Faster, Stronger."
4. Wu, Z., et al. "Deep SORT: Simple Online and Realtime Tracking with a Deep Association Metric."
5. LabelImg: https://github.com/tzutalin/labelImg

Appendix A: Implementation Details

A.1 Project Structure

```
TrafficDetection/
          data.yaml
      # Dataset configuration
          frame-generation.py
      # Script to extract frames from videos
          organize-dataset.py
      # Script to organize dataset
          TrafficDetection.ipynb
      # Main notebook for detection & tracking
          saved_models/
         # Directory for saved models
                finetuned_yolov8n.pt
          # Fine-tuned YOLOv8 model
          datasets/
         # Training datasets
                dataset/
                    images/
                          all/
                       # All extracted frames
                          train/
                       # Training images
                          val/
                     # Validation images
                    labels/
                        train/
                     # Training labels
                        val/
                   # Validation labels
          frames/
         # Directory for input video frames
                output739864687.mp4
         # Sample input video
          result/
       # Results of detection and tracking
              output_lane_detected.mp4
          # Processed video with annotations
              vehicle_counts.csv
        # Traffic statistics in CSV format
```

Code 18. Project directory structure

A.2 Key Functions

Video Processing Function

The main video processing function handles the complete pipeline from video input to statistical output:

```
# Pseudocode for video processing
function process_video(input_video, output_video
    , output_csv, start_time, end_time):
    Open input video
    Initialize vehicle tracker
    Set up output video writer

    For each frame in the specified time range:
        If frame is at 1-second interval:
            Detect vehicles in frame
            Update tracker with detected
    vehicles
            Log progress information
            Display and save annotated frame
        Else:
            Save original frame to output video

    Calculate final statistics (incoming and
    outgoing vehicles)
    Save statistics to CSV file
    Display final summary
```

Code 19. Pseudocode for video processing

Vehicle Tracker Class

The `VehicleTracker` class implements the core tracking logic:

```
# Pseudocode for vehicle tracker
class VehicleTracker:
    Initialize empty vehicle dictionary

    function update(detected_vehicles, frame_idx
    , frame, fps):
        Group detections by vehicle class

        For each vehicle class:
            Find potential matches between new
    detections and existing vehicles
            Calculate matching cost based on
    distance and color
            Assign matches greedily to ensure
    one-to-one mapping

            For each matched detection:
                Update vehicle record
                Determine direction based on
    movement
                Draw bounding box and label

            For each unmatched detection:
                Create new vehicle entry
                Determine initial direction
    based on position
                Draw bounding box and label

        Return lists of new and updated vehicles
    , and annotated frame
```

Code 20. Pseudocode for vehicle tracker

A.3 Output Format

The system generates two primary outputs:

1. **Annotated Video**: The processed video with bounding boxes around detected vehicles, each labeled with its unique ID.
2. **CSV File**: A simple CSV file containing counts of incoming and outgoing vehicles:

```
Incoming Vehicles,Outgoing Vehicles
1,0
```

Code 21. CSV output format

3. **Console Output**: Detailed logging information to monitor the system's operation:

```
0: 480x640 (no detections), 53.8ms
Speed: 0.9ms preprocess, 53.8ms inference,
    0.2ms postprocess per image at shape (1,
     3, 480, 640)

[DETECTION] Vehicles Detected: 0
[FRAME 170] Vehicles Detected: 0

[TIME UPDATE]
    -------------------------------------
Time Elapsed: 17 sec
Vehicles Detected: 0 in this second
New Vehicles:
Old Vehicles:
----------------------------------

[PROCESS COMPLETE]
    -------------------------------
Processed Video Interval: 15 - 20 sec
Vehicle Counts Saved to: ./result/
    vehicle_counts.csv
Incoming Vehicles: 1
Outgoing Vehicles: 0
```

```
---------------------------------
```

Code 22. Console output example

This multi-faceted output approach provides both visual confirmation of the system's operation and statistical data for further analysis.