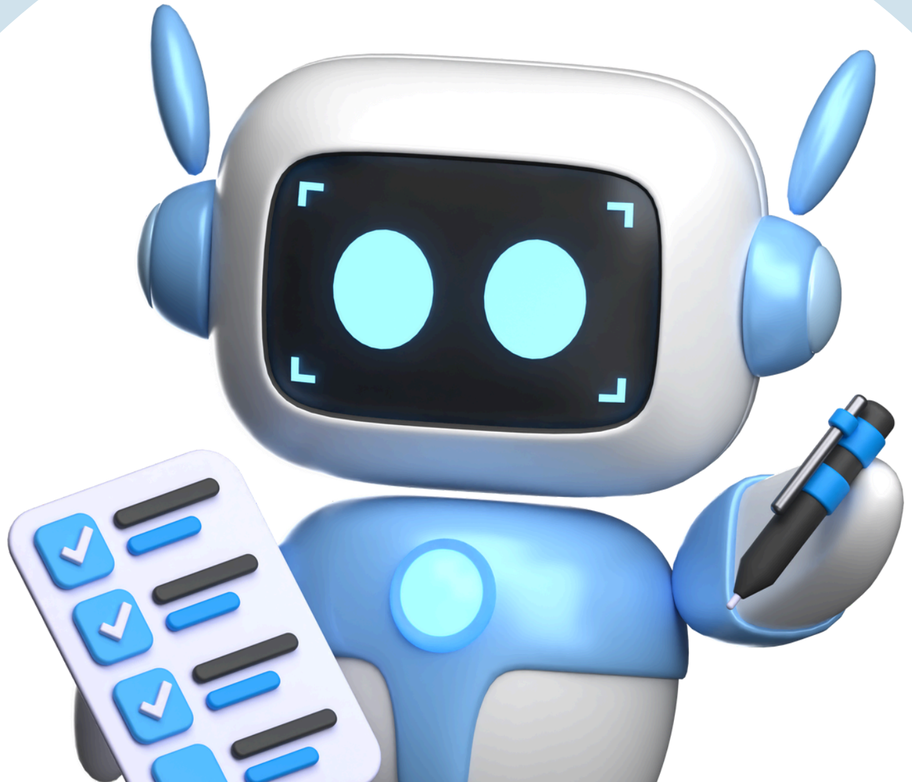
Introduction to Digital Image Processing

Final Project Presentation

Instructor - Dr. Pham Van Huy

Group Members
Thiha Aung - 523K0073
Thin Lei Sandi - 523K0075

YOLO IN COMPUTER VISION: REVOLUTIONIZING REAL-TIME OBJECT DETECTION



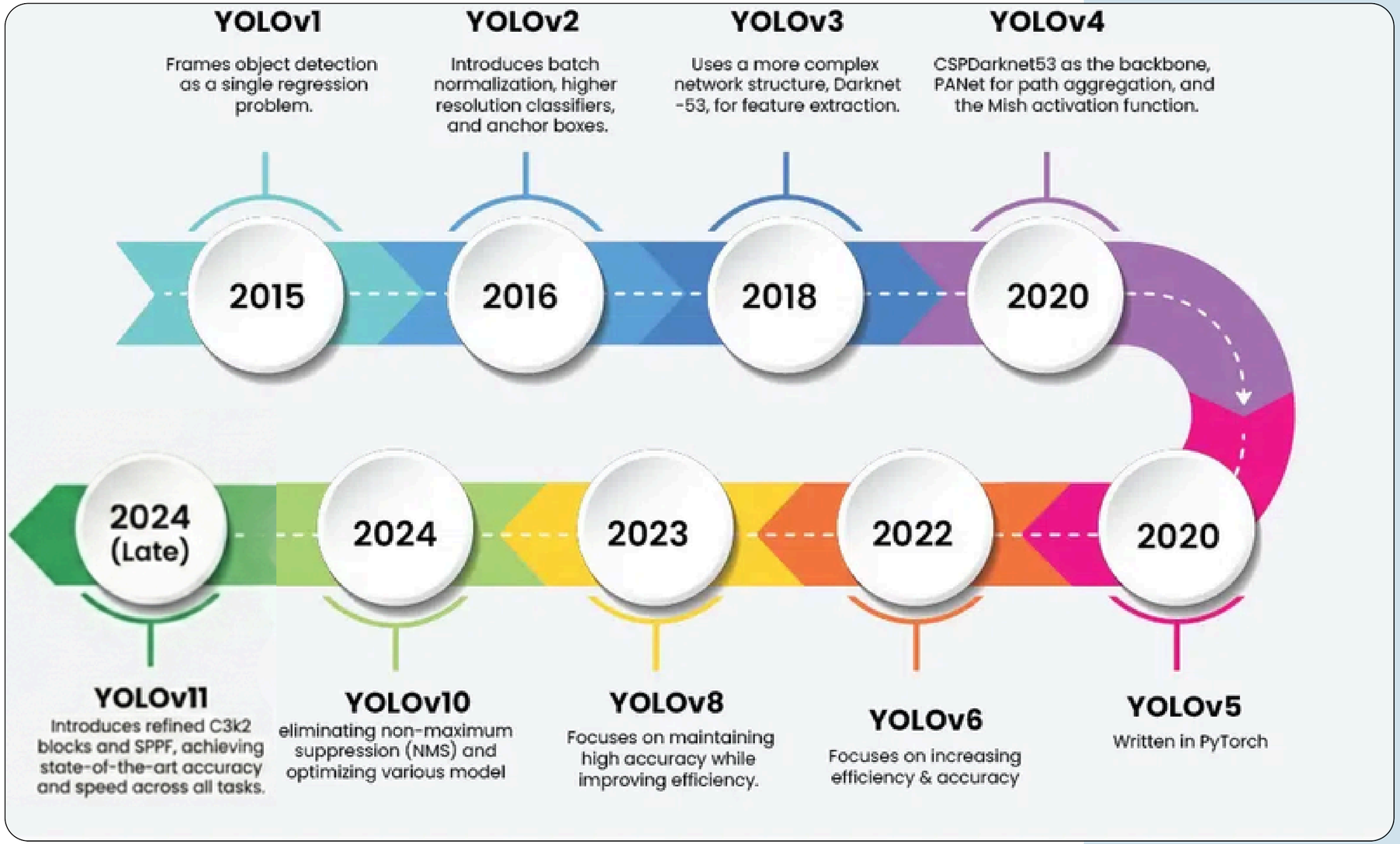
What is YOLO?

- YOLO, short for "**You Only Look Once**," is a groundbreaking real-time object detection algorithm first introduced in 2015 by Joseph Redmon and his team. Its core innovation lies in its approach to detection.

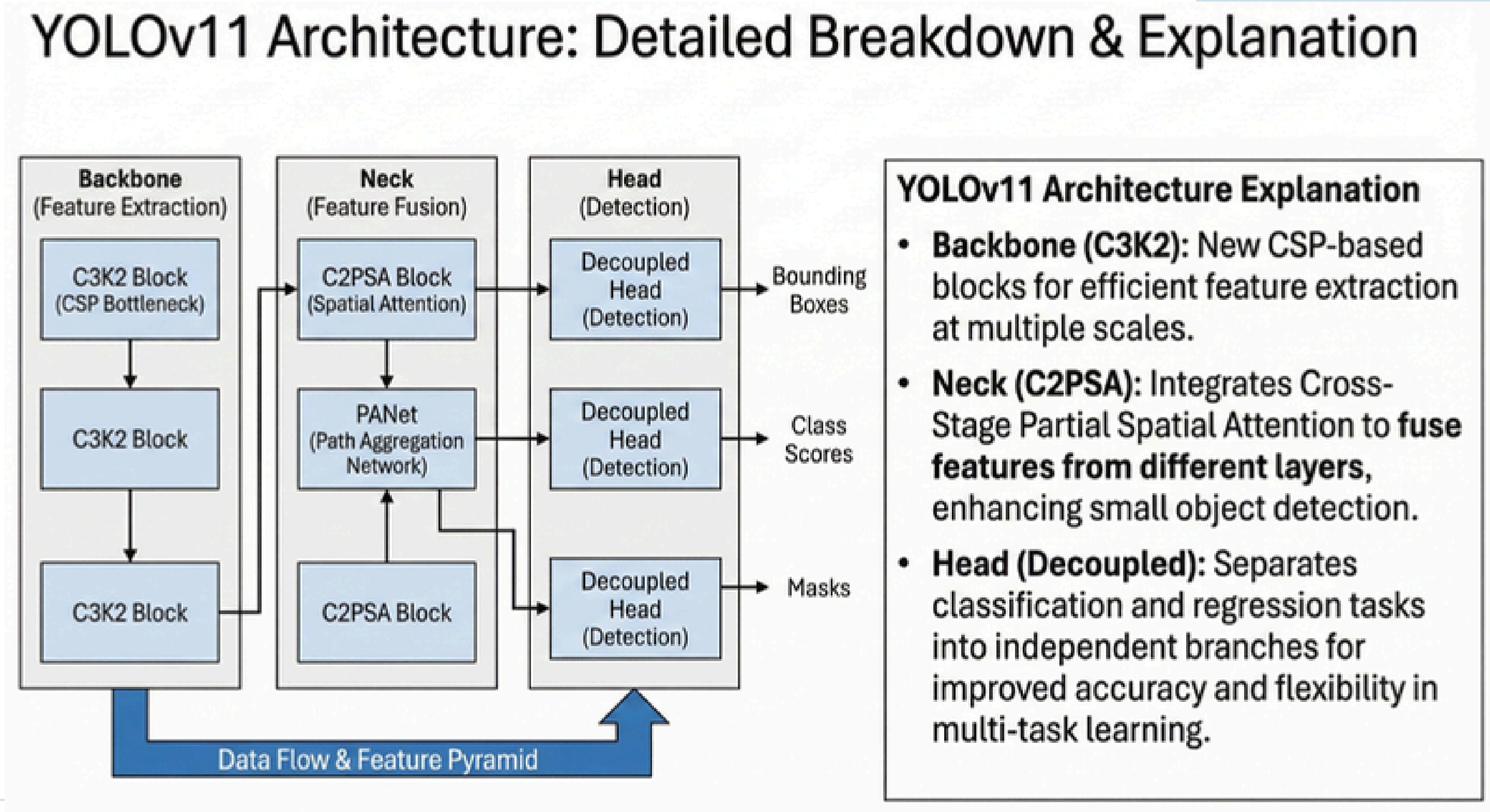
- Unlike older, multi-stage methods, YOLO frames object detection as a **single regression problem**. This means it predicts bounding boxes and class probabilities simultaneously in one pass, drastically accelerating the process.

- This unified network design allows for **lightning-fast processing** while maintaining impressive accuracy, making it ideal for applications where speed is paramount.

Evolution of YOLO: From v1 to v11



YOLO Architecture Overview



How YOLO Works: The Core Algorithm

1. GRID DIVISION

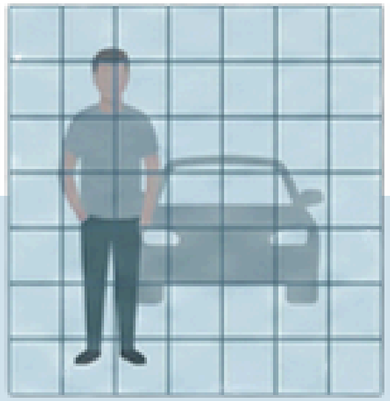
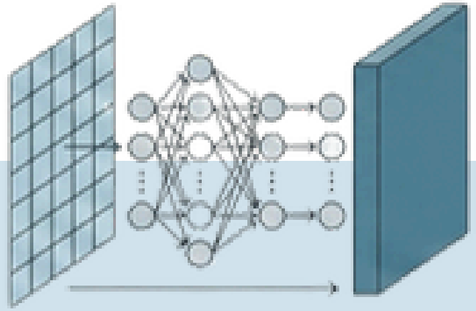


Image split into a grid. Each cell predicts objects.

→

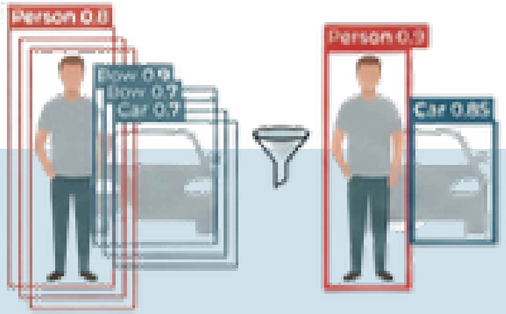
2. SINGLE CNN PASS



One neural network processes the whole image at once.

→

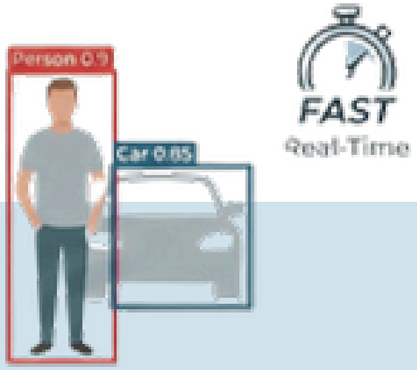
3. NON-MAXIMUM SUPPRESSION (NMS)



Filters redundant boxes, keeps the best predictions.

→

4. HIGH-SPEED OUTPUT



Delivers instant, accurate detections.

Why YOLO is a Game-Changer

Unmatched Speed

YOLO processes images at an incredible 45-91 FPS, consistently outperforming many traditional and modern object detection competitors.

High Accuracy

Achieves a Mean Average Precision (mAP) more than double that of other real-time detectors, ensuring reliable and precise object identification.

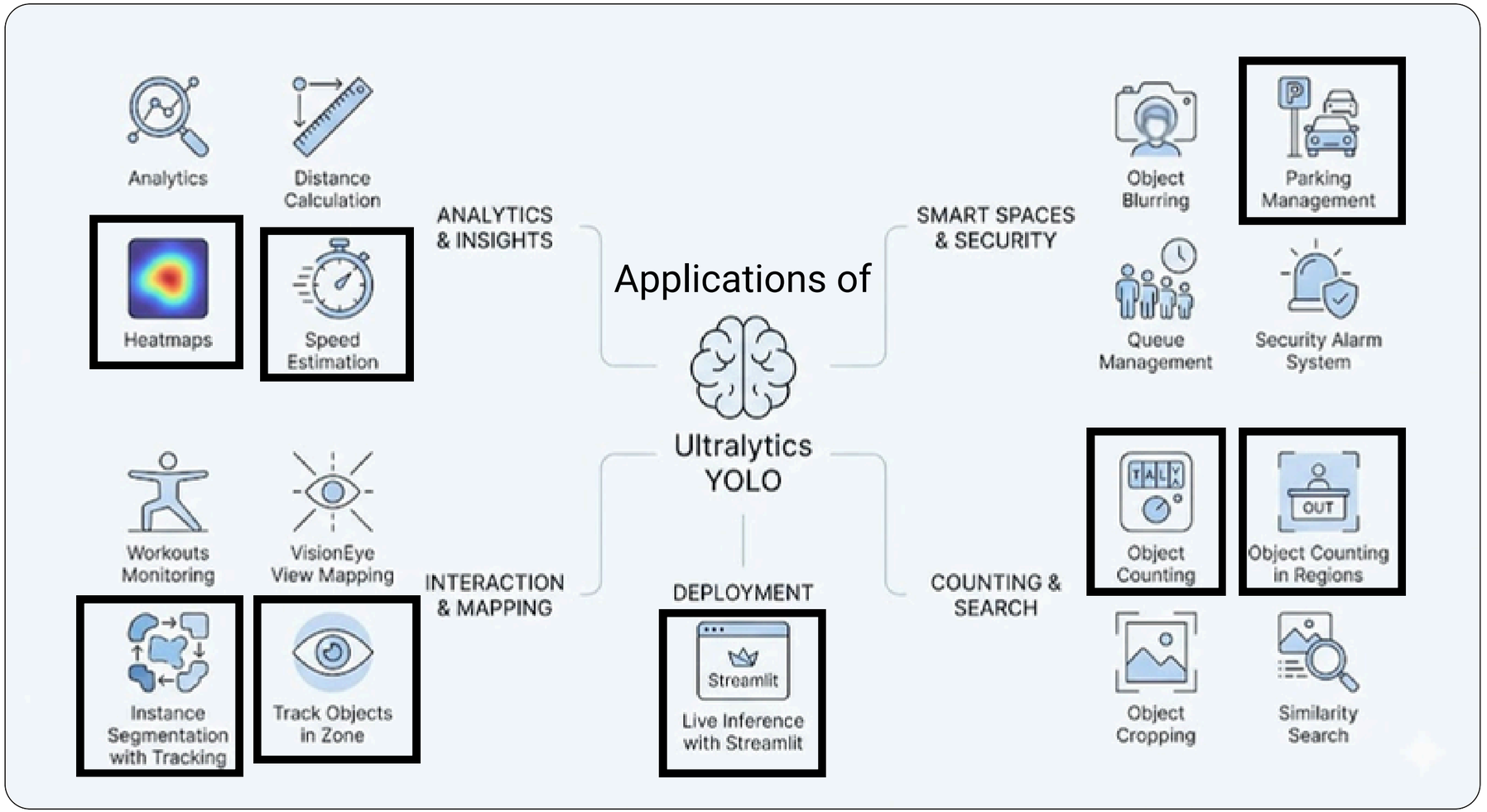
Strong Generalization

Demonstrates robust performance across a wide array of environments and diverse object types, adaptable to various real-world scenarios.

Open Source Power

A thriving open-source community continually contributes to its development, fostering rapid innovation and widespread adoption across industries.

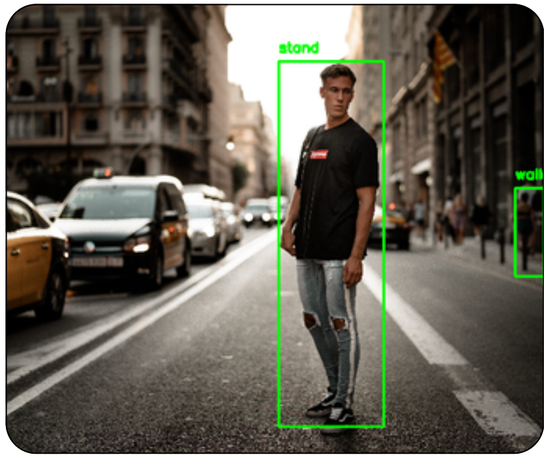
Ultralytics Solutions



Human Activity Detection

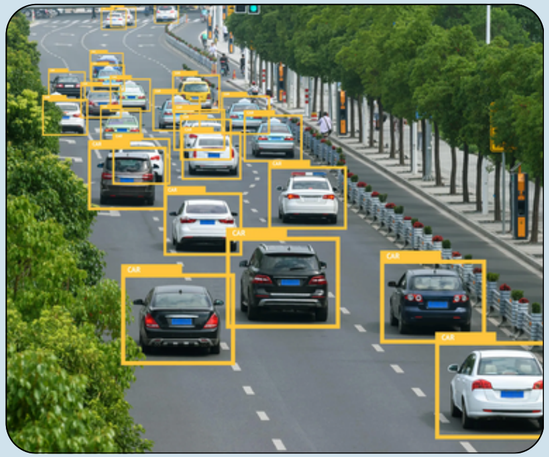
Technology Stack: Python,

OpenCV, Ultralytics YOLO



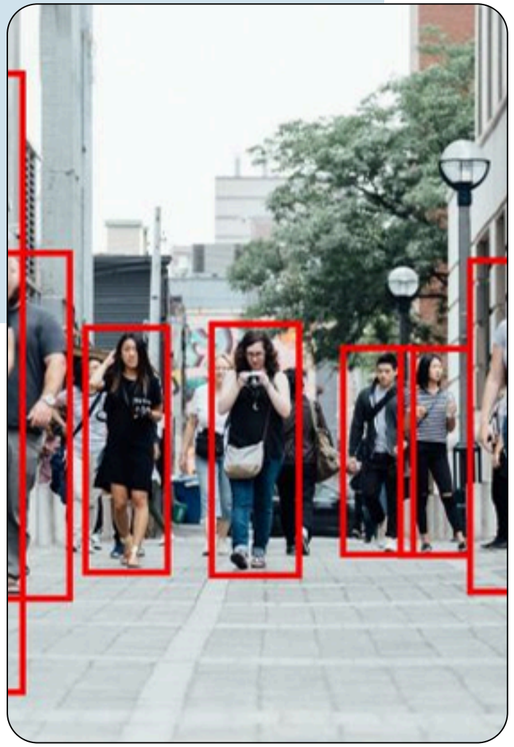
- Object Tracking
- Heatmap Visualization
- Object Counting
- Object Counting in Regions
- Speed Estimation

Vehicle Activity Detection



- Object Tracking
- Heatmap Visualization
- Object Counting
- Speed Estimation
- Live Inference with Streamlit Application

System Architecture & Technologies Used



Processing Pipeline

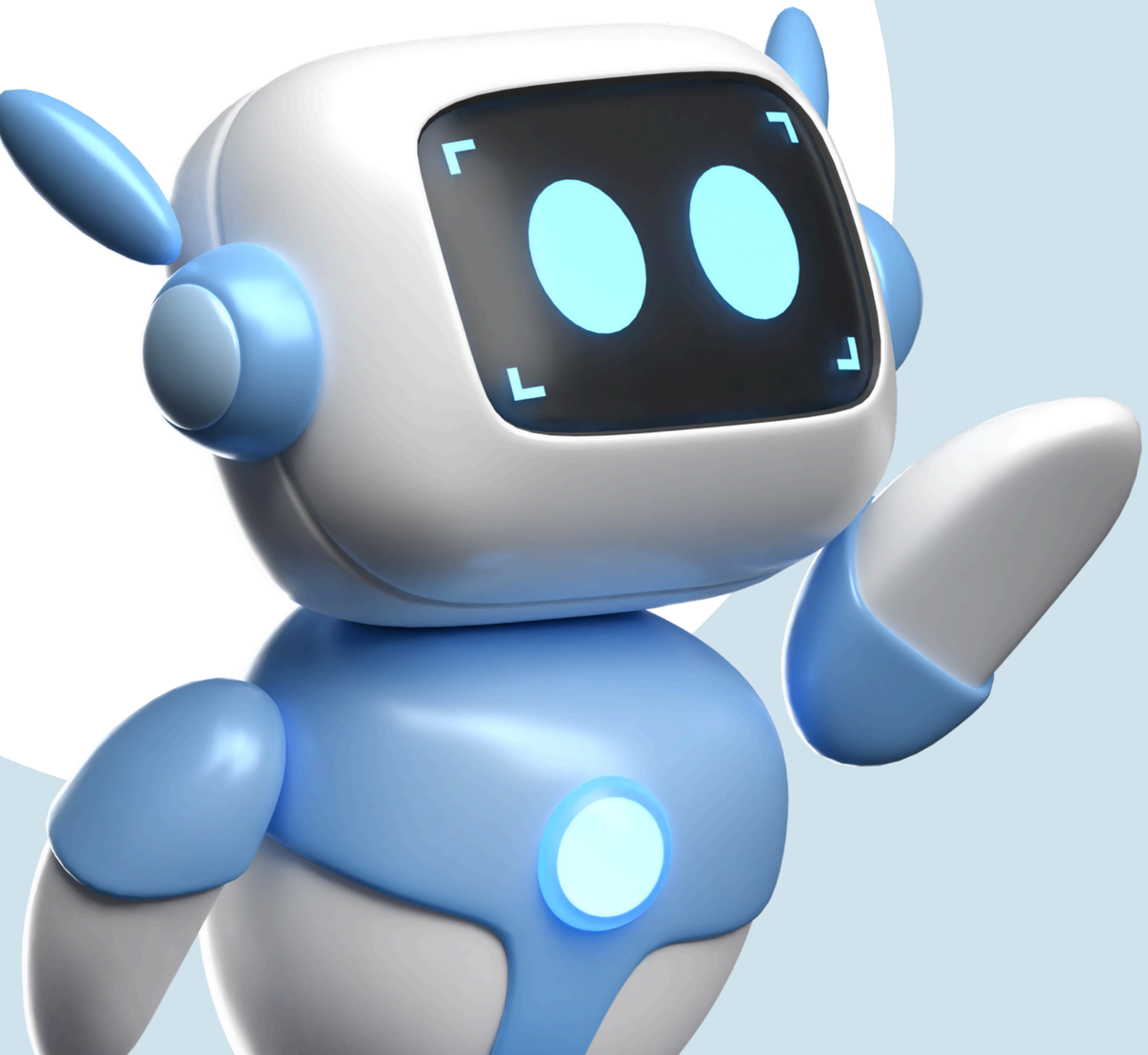
1. Video input (file or live stream)
2. YOLO detection
3. BoT-SORT object tracking
4. Shared tracking results
5. Activity analytics:
    a. Heatmap
    b. Counting
    c. Speed estimation
6. Visualization and output

Core Libraries

- Ultralytics YOLO
    - Detection, tracking, analytics solutions
- OpenCV
    - Video I/O and visualization
- BoT-SORT Tracker
    - Robust multi-object tracking
- Python
    - Modular, real-time processing

ultralytics

YOLO11



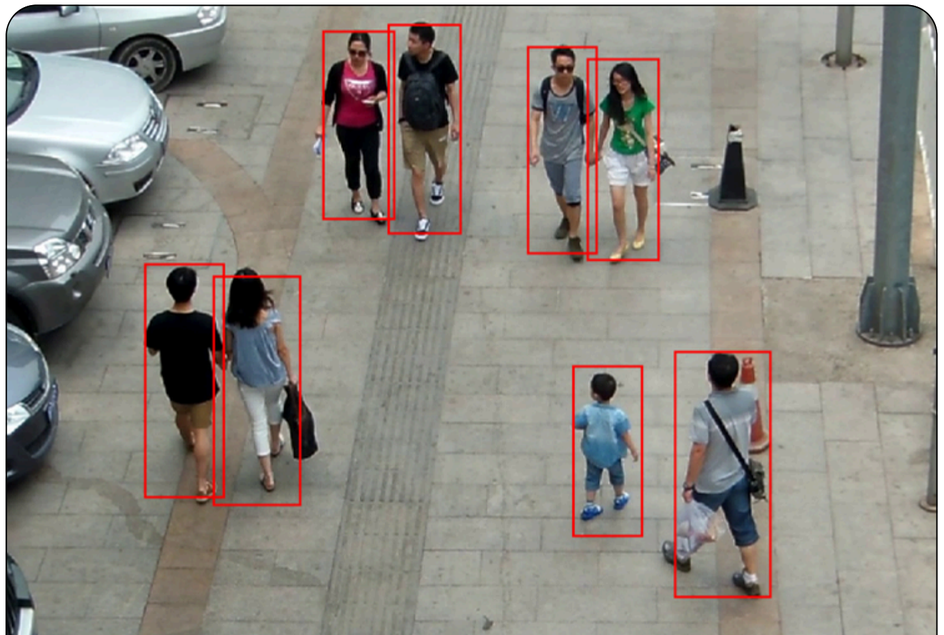
HUMAN ACTIVITY DETECTION

Problem Statement

- Need to understand how human move in a space
- Applications:
  - Crowd monitoring
  - Public safety
  - Retail analytics
  - Smart surveillance
- Key challenges:
  - Real-time performance
  - Accurate tracking across frames
  - Meaningful visual insights, not just bounding boxes

Objectives

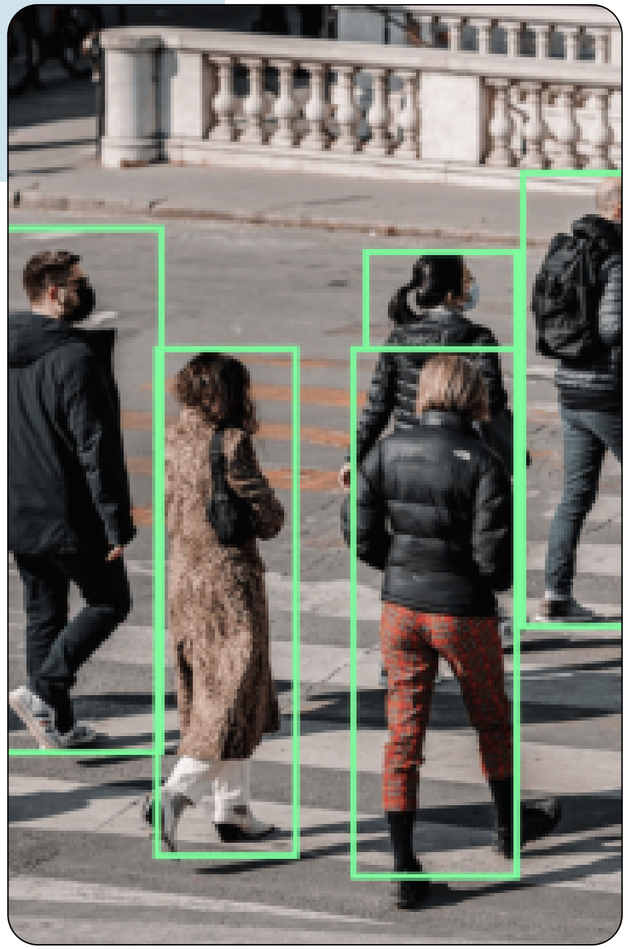
- Detect human in a video stream
- Track individuals consistently across frames
- Visualize movement intensity using a heatmap
- Count people crossing boundary
- Estimate walking speed in real-world units

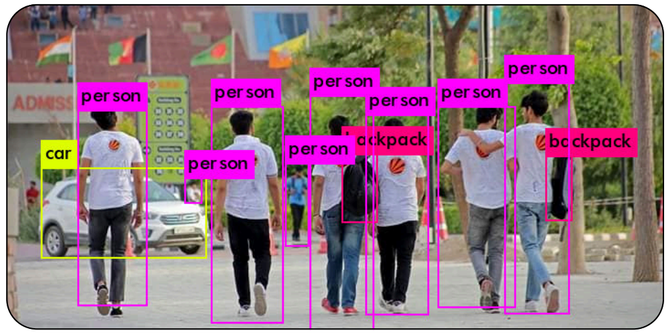


Human Activity Detection

Object Detection

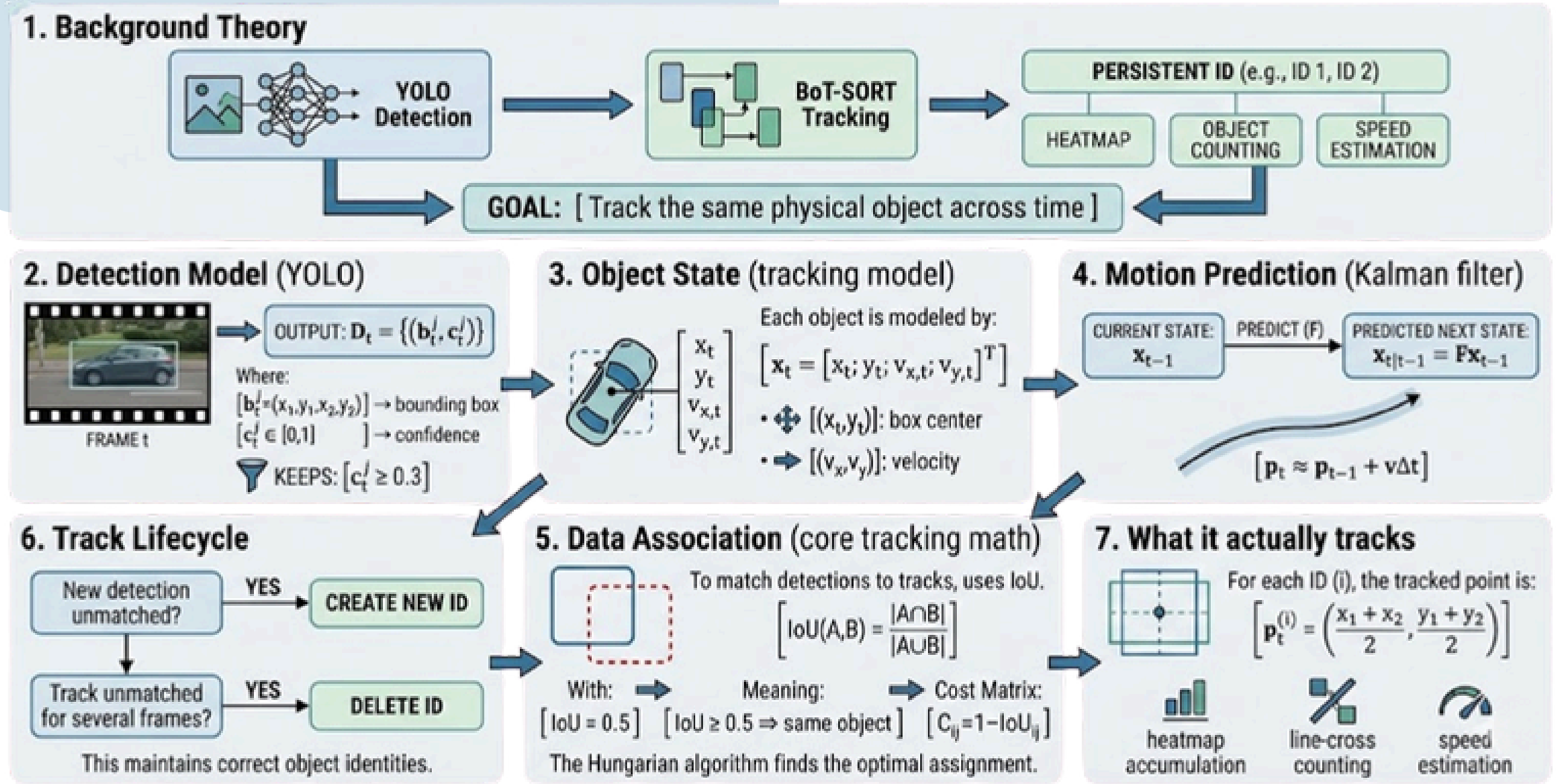
- Purpose: Locate humans in each video frame
- Model: YOLO11n (yolo11n.pt)
- Class Filter: Person (class 0)
- Thresholds:
- Confidence = 0.3
- IoU = 0.5
- Output: Bounding boxes and confidence scores
- Optimization: Real-time inference





Object Tracking

- Purpose: Maintain consistent human identities across frames
- Tracker: BoT-SORT (botsort.yaml)
- Logic: Assigns persistent Track IDs based on motion and appearance
- Usage: Shared tracking results reused across all solutions


1. Background Theory
YOLO Detection
BoT-SORT Tracking
PERSISTENT ID (e.g., ID 1, ID 2)
HEATMAP
OBJECT COUNTING
SPEED ESTIMATION
GOAL: [Track the same physical object across time]
2. Detection Model (YOLO)
FRAME t
OUTPUT: $D_t = \{(b_t^j, c_t^j)\}$
Where:
$[b_t^j=(x_1,y_1,x_2,y_2)]$ → bounding box
$[c_t^j \in [0,1]$] → confidence
KEEPS: $[c_t^j \geq 0.3]$
3. Object State (tracking model)
Each object is modeled by:
$[x_t = [x_t; y_t; v_{x,t}; v_{y,t}]^T]$
• $[(x_t,y_t)]$: box center
• $[(v_x,v_y)]$: velocity
4. Motion Prediction (Kalman filter)
CURRENT STATE: x_{t-1}
PREDICT (F)
PREDICTED NEXT STATE: $x_{t|t-1} = Fx_{t-1}$
$[p_t \approx p_{t-1} + v\Delta t]$
6. Track Lifecycle
New detection unmatched?
YES
CREATE NEW ID
Track unmatched for several frames?
YES
DELETE ID
This maintains correct object identities.
5. Data Association (core tracking math)
To match detections to tracks, uses IoU.
$[IoU(A,B) = \frac{|A \cap B|}{|A \cup B|}]$
With: [IoU = 0.5]
Meaning: [IoU ≥ 0.5 ⇒ same object]
Cost Matrix: $[C_{ij}=1-IoU_{ij}]$
The Hungarian algorithm finds the optimal assignment.
7. What it actually tracks
For each ID (i), the tracked point is:
$[p_t^{(i)} = \left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)]$
heatmap accumulation
line-cross counting
speed estimation

Human Activity Detection

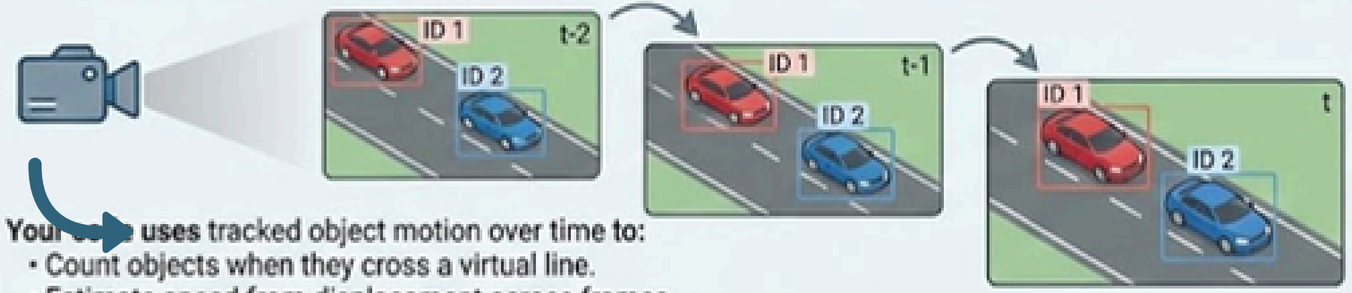
Speed Calculation

- Purpose: Measure human movement over time
- Method: Frame-to-frame displacement
- Conversion: Pixel distance → kilometers
- Stabilization: Speed locked after a motion window
- Classification:
- Walking — speed above threshold
- Stopped — speed near zero

Object Counting

- Purpose: Count humans crossing a defined region
- Method: Line-crossing based counting
- Region: Vertical line at frame center
- Counts:
  - Left — crossings in one direction
  - Right — crossings in opposite direction
- Logic: Uses unique Track IDs to prevent double counting

1. Background Theory

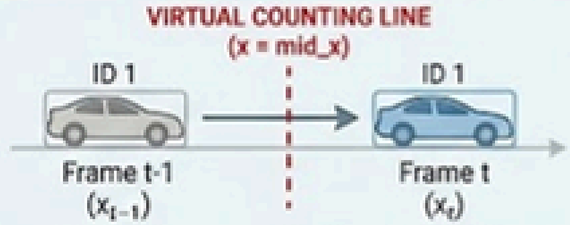


Your [illegible] uses tracked object motion over time to:

- Count objects when they cross a virtual line.
- Estimate speed from displacement across frames.

→ **Tracking ensures:** [same object ⇒ same ID over time]

2. Object Counting (line-crossing theory)



Concept: A virtual counting line divides the scene. An object is counted once when it crosses the line.

Mathematical condition for crossing:

$$[\,(x_{t-1} - \text{mid}_x)(x_t - \text{mid}_x) < 0\,]$$

Direction: → Left → Right → IN
← Right → Left → OUT

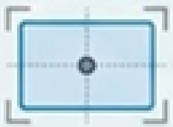
Counts: [Total = IN + OUT]

3. Speed Calculation (motion theory)

Physical principle:

$$\left[\text{Speed} = \frac{\text{Distance}}{\text{Time}}\right]$$

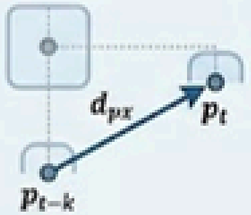
Step 1: Object position



Tracked object center:

$$\left[p_t = \left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)\right]$$

Step 2: Pixel displacement



Using a motion window of k frames (MOTION_WINDOW = 8):

$$[\,d_{px} = |p_t - p_{t-k}|\,]$$

Step 3: Pixel → meter conversion

PIXELS_PER_METER = 50.

$$\left[d_m = d_{px} \cdot \frac{1}{50}\right]$$

Step 4: Time elapsed

At fps frames per second:

$$\left[\Delta t = \frac{k}{\text{fps}}\right]$$

Final speed formula

$$\left[v = \frac{d_m}{\Delta t} = d_{px} \cdot \frac{1}{\text{PIXELS\_PER\_METER}} \cdot \frac{\text{fps}}{k}\right]$$

Speed is clipped: $[\,v = \min(v, 180)\,]$

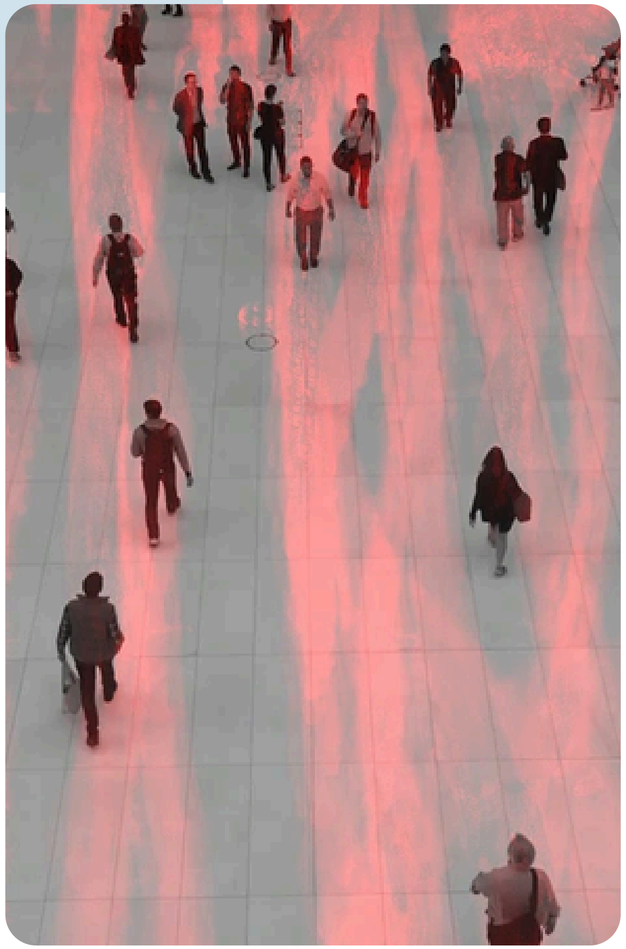
4. Key assumptions (important theory)

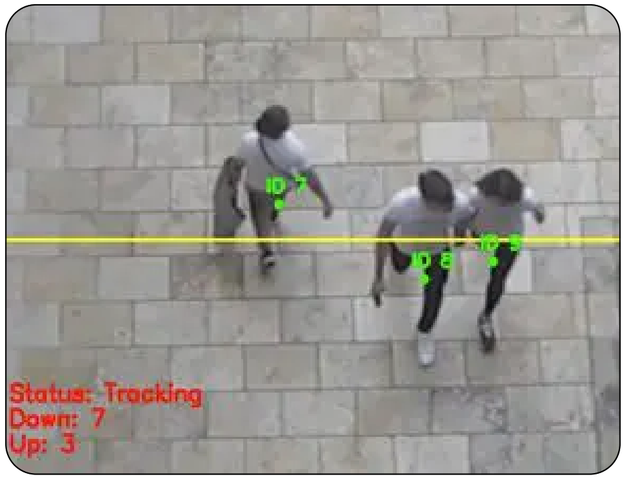
- Objects move smoothly
- Camera is static
- Fixed pixel-to-meter scale
- Tracking IDs are stable

Human Activity Detection

Heatmap Visualization

- Purpose: Visualize spatial density of human activity
- Method: Accumulates tracked human positions over time
- Colormap: TURBO
- Intensity:
  - Low activity → cool/dark colors
  - High activity → warm/bright colors
- Output: Activity heatmap overlay on video

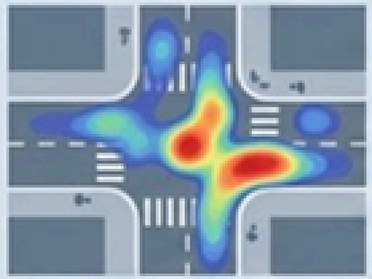




Line-Based Counting

- Virtual vertical line at frame center
- Counts:
  - Left crossings
  - Right crossings
- Displays total count in real time

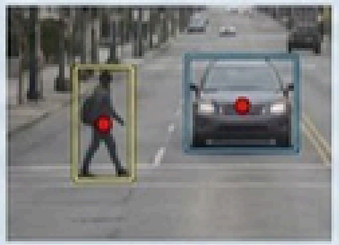
1. BACKGROUND THEORY (SPATIO-TEMPORAL DENSITY)



Idea: Regions with **frequent tracking** become "hotter".

Represents **Spatio-Temporal Density Estimation.**

2. INPUT FROM TRACKING (OBJECT CENTERS)

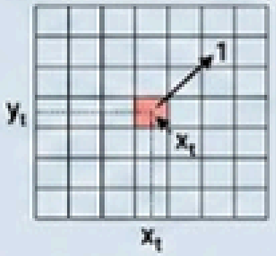


Uses tracking output. Extracts center of bounding box for each object ID (i) at frame (t).

$$p_t^{(i)} = \left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)$$

3. MATHEMATICAL MODEL (ACCUMULATION MATRIX)

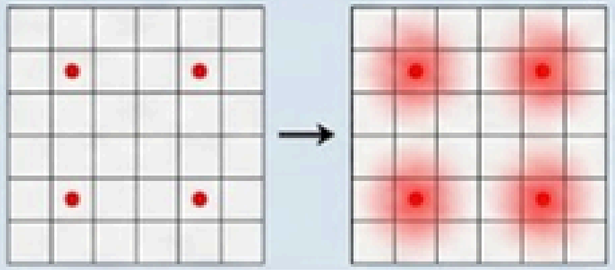
2D Accumulation Matrix H(x,y)



$$H(x_t, y_t) \leftarrow H(x_t, y_t) + 1$$

Update rule: Each visit of an object center increases intensity at that location.

4. SMOOTHING (GAUSSIAN SPREAD)



$$H(u,v) \leftarrow H(u,v) + \exp\left(-\frac{(u-x_t)^2 + (v-y_t)^2}{2\sigma^2}\right)$$

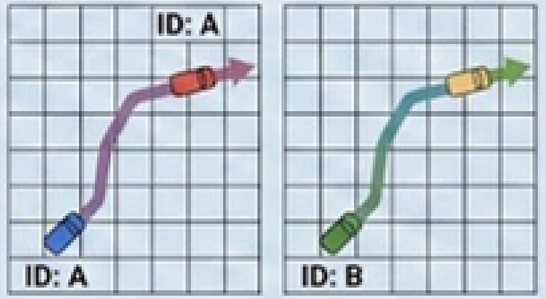
Applies spatial smoothing to avoid sharp dots and produce continuous "heat".

5. NORMALIZATION & VISUALIZATION

RAW $H(x,y)$ → Normalization: $\hat{H} = \frac{H - \min(H)}{\max(H) - \min(H)}$ → COLOR MAP APPLIED (e.g., TURBO)

LOW ACTIVITY — HIGH ACTIVITY

6. WHY TRACKING IDs MATTER



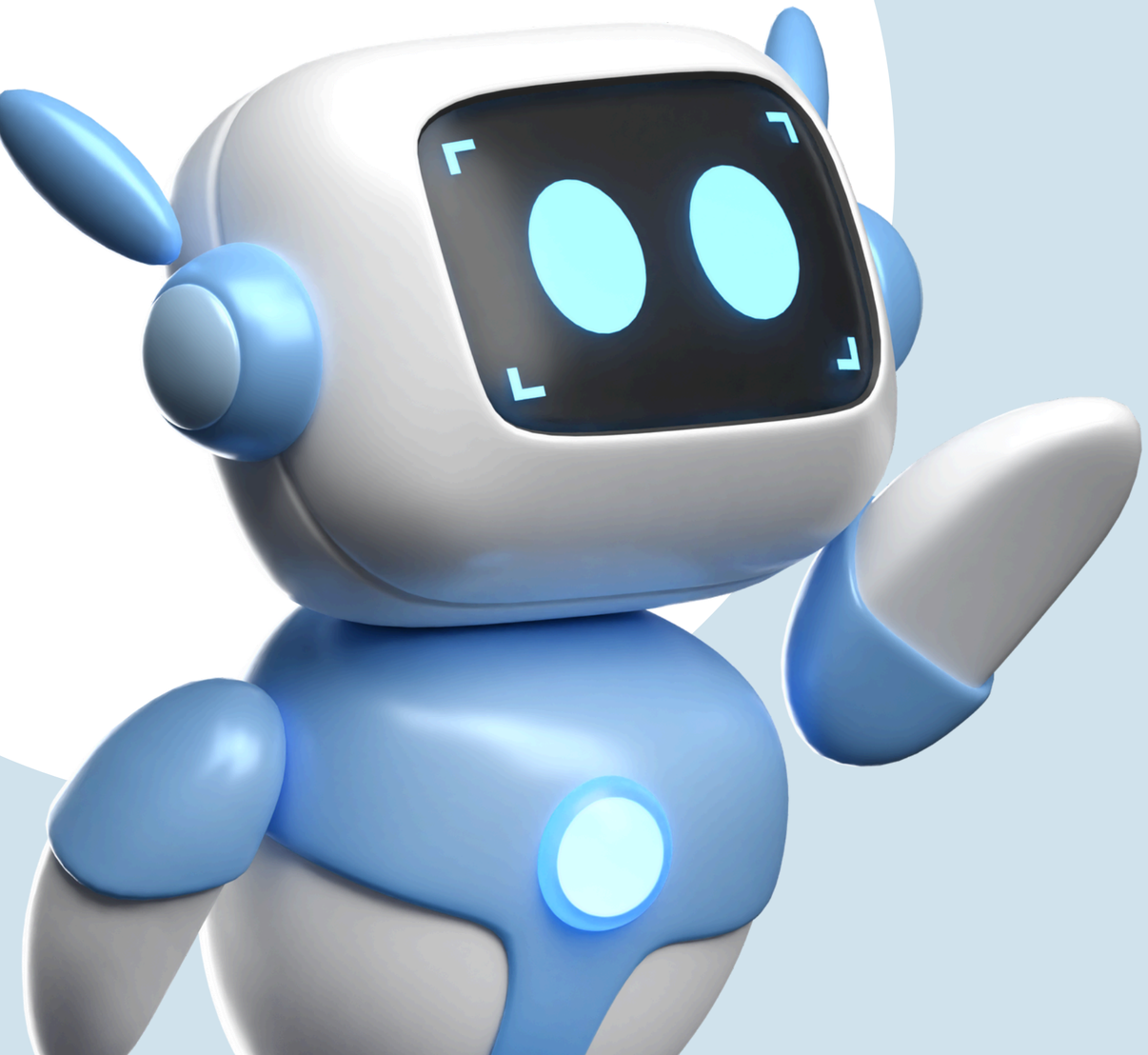
Ensures each object contributes once per frame. No duplicate counting. Reflects true motion patterns.

7. KEY ASSUMPTIONS

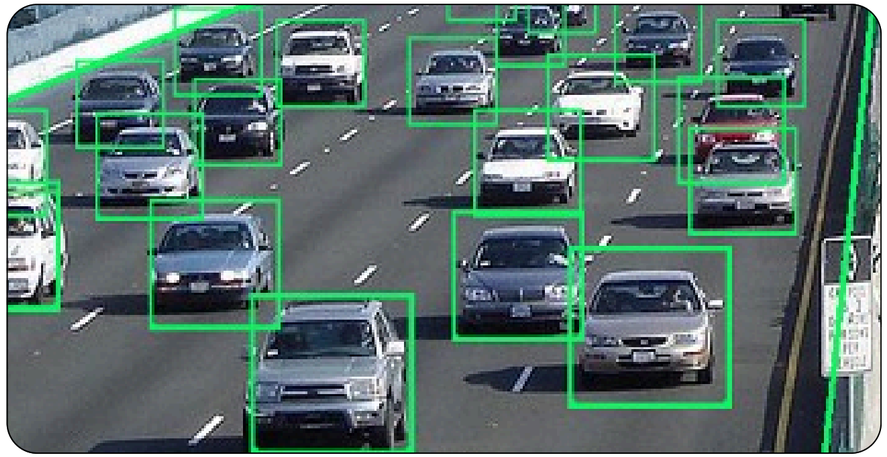
- Object centers represent location
- Camera is static
- Frequency ≈ importance of region

Key Parameters For Human Activity

Parameter(s)	What They Control	Why They Matter	Risk if Mis-tuned
CONF, IOU, TRACKER	Detection confidence, overlap matching, and ID tracking	Directly affect detection quality and track stability	ID switches → noisy heatmaps and incorrect speed estimates
PIXELS_PER_METER, MOTION_WINDOW	Conversion from pixel displacement to speed	Provide approximate real-world speed estimation	Uncalibrated scale → inaccurate speeds
--stop-speed (m/s)	Threshold for classifying stopped vs moving	Determines when motion is considered “stopped”	Too high → slow motion mislabeled as stopped



VEHICLE ACTIVITY DETECTION

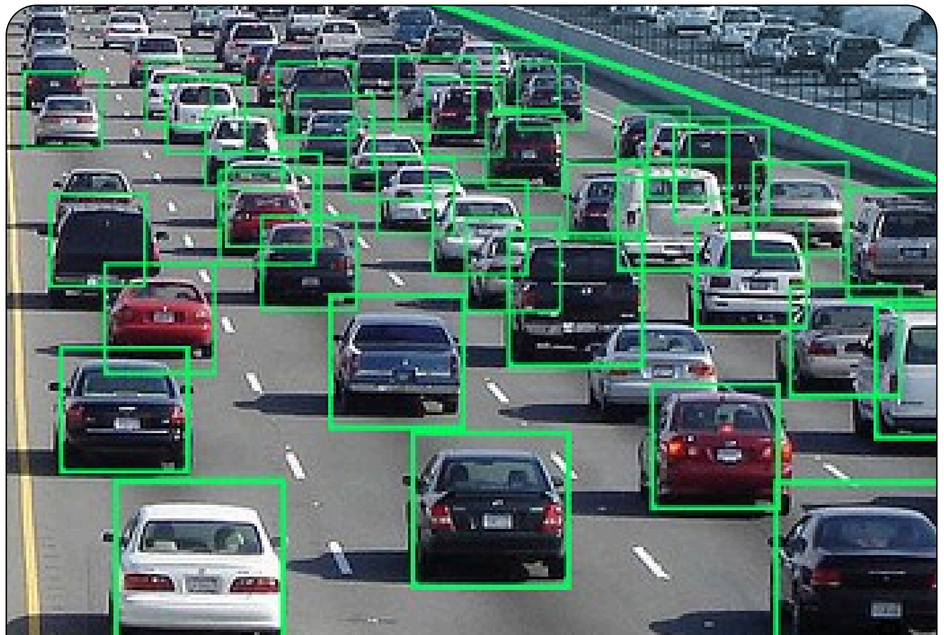


Problem Statement

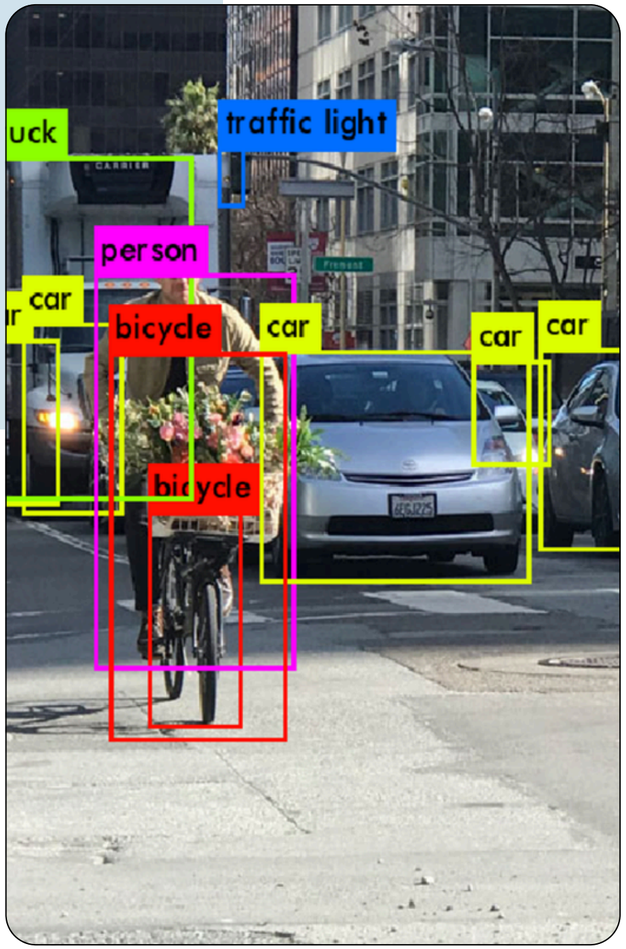
- Increasing traffic demands real-time vehicle monitoring
- Traditional sensor-based systems are costly and limited
- Manual monitoring is inefficient and error-prone
- Need an automated vision-based system to:
  - Detect and track vehicles
  - Estimate speed and traffic flow
  - Identify congestion
- Enables smarter and efficient traffic management systems

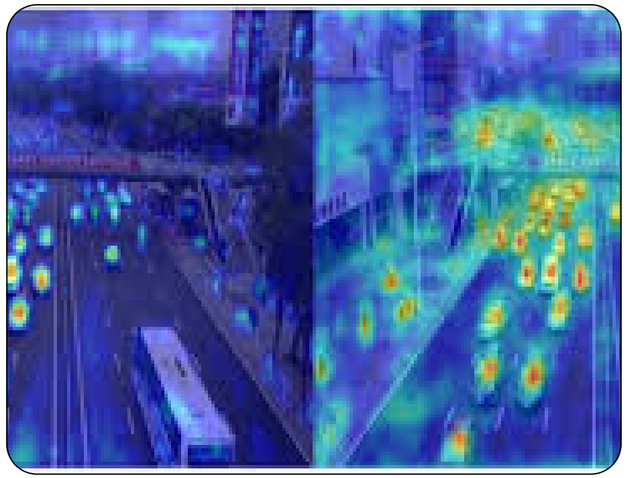
Objectives

- Detect vehicles from live streams
- Track motion behavior
- Identify congestion
- Measure speed
- Generate traffic heatmaps



Vehicle Activity Detection





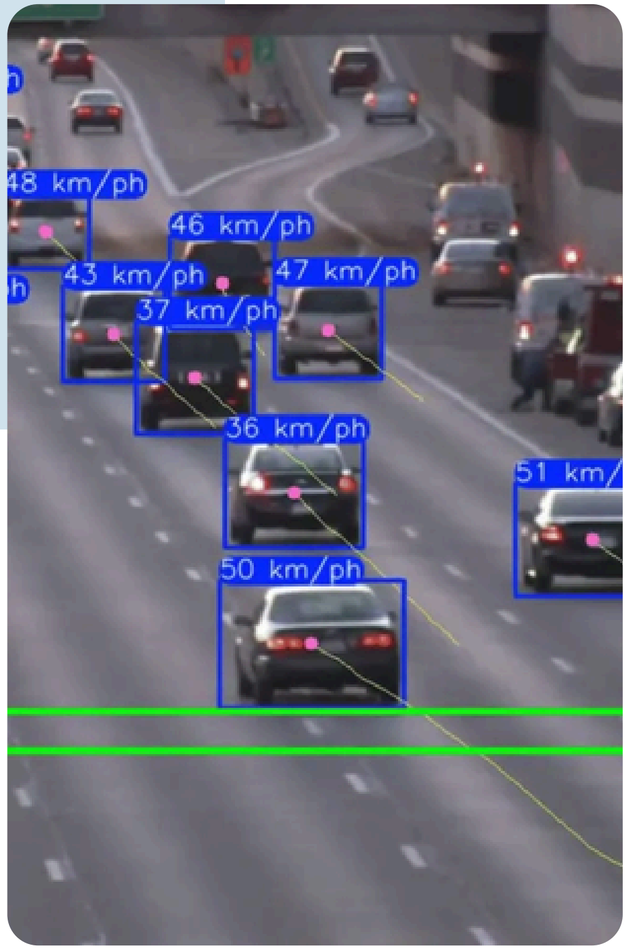
Vehicle Detection

- YOLO model: yolo11n.pt
- Vehicle Classes
  - Car
  - Motorcycle
  - Bus
  - Truck
  - Bicycle

Vehicle Heatmap

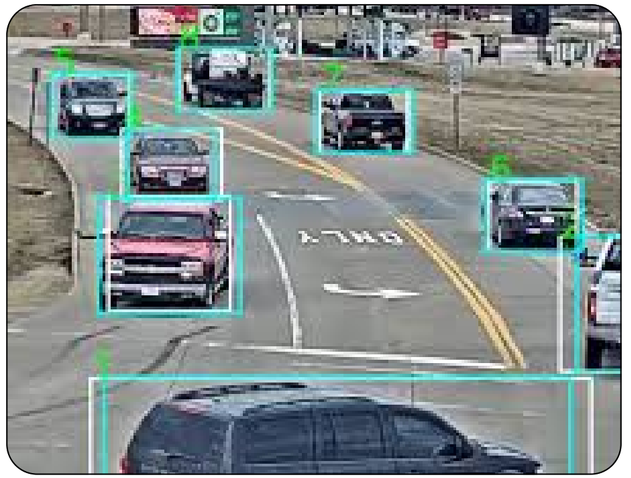
- Traffic Heatmap
- Highlights dense traffic zones
- Heatmap decay over time

Vehicle Activity Detection



Live Speed Estimation

- Continuous speed updates
- Pixel-to-meter conversion
- Speed displayed in km/h
- Used to classify:
  - Moving vehicles
  - Stopped vehicles



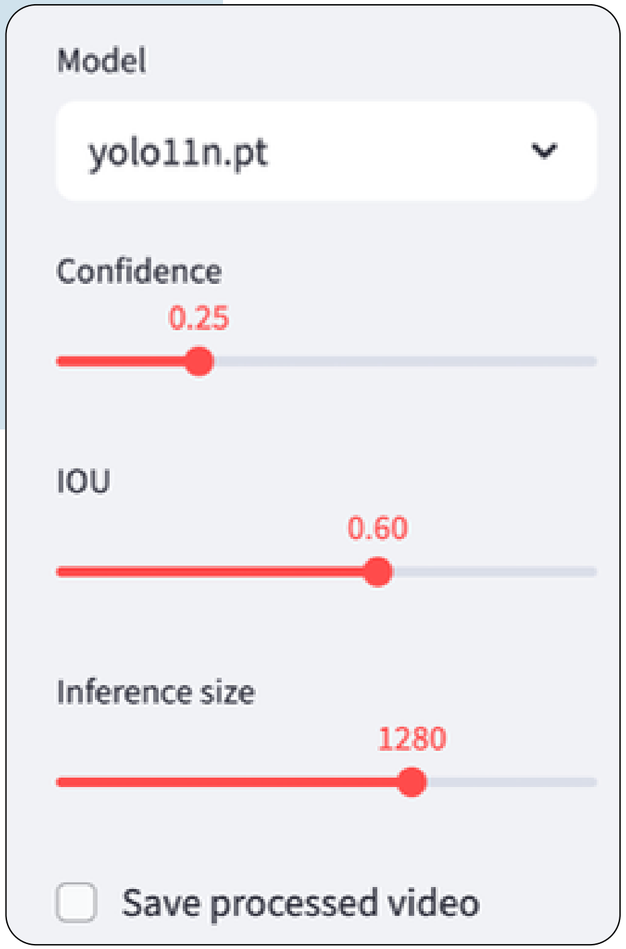
Vechicle Counting

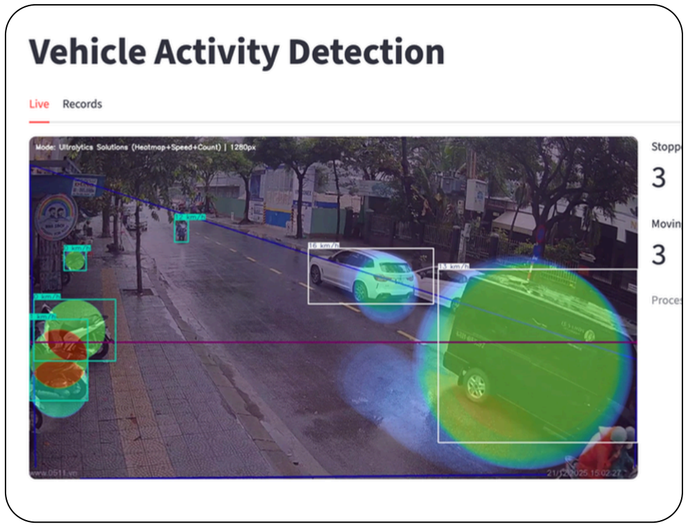
- Counts vehicles passing on live stream
- Measures traffic volume
- Supports for daily records

Vehicle Activity Detection

Live Streaming & Dashboard

- Live YouTube traffic analysis
- Adjustable parameters:
  - Confidence
  - IOU
  - Inference size
- Real-time statistics
- CSV export for records





Performance & Efficiency

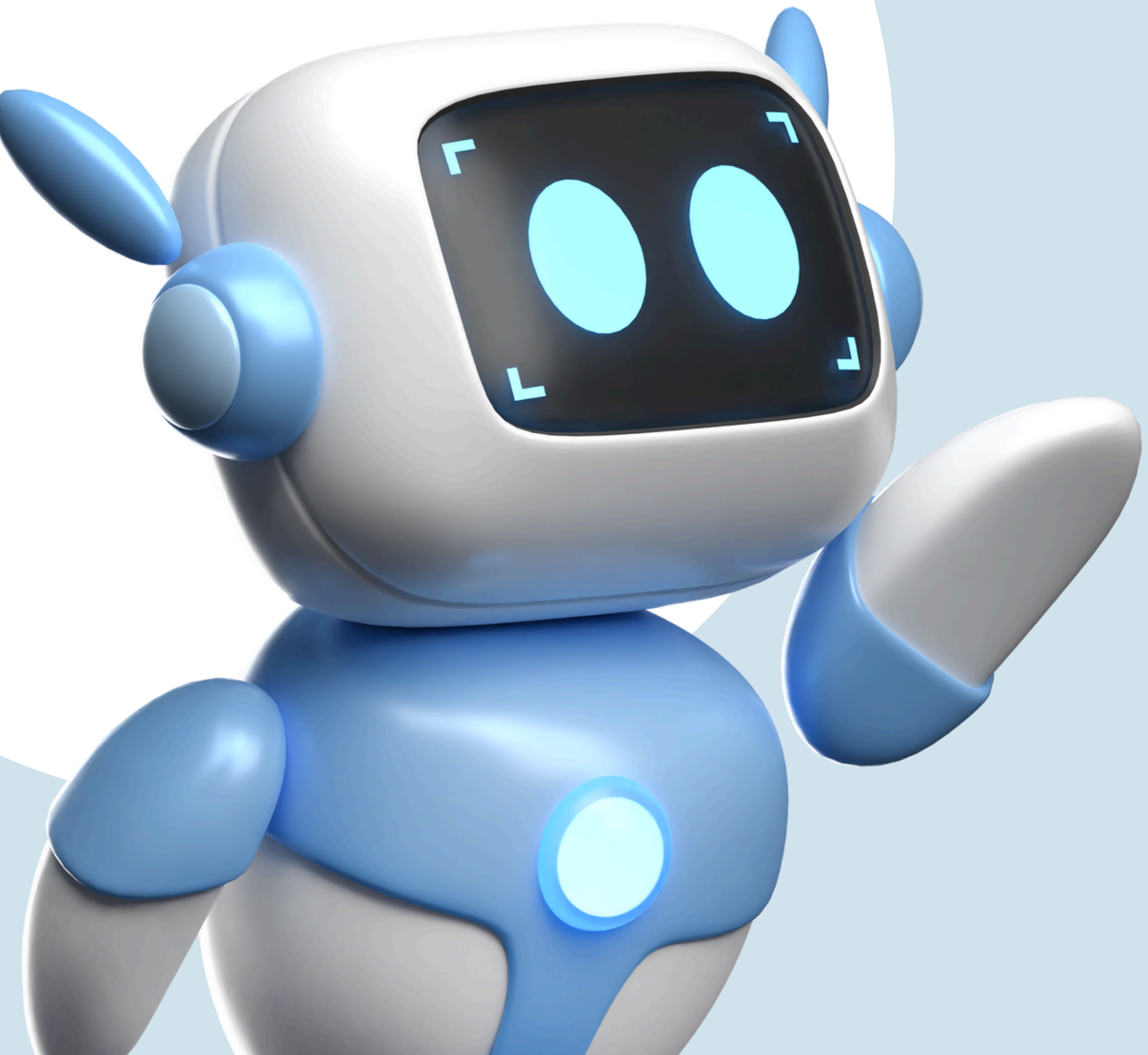
- Single inference per frame
- Near real-time FPS
- Efficient on standard hardware
- Modular Ultralytics solutions

Key Parameters For Vehicles Activity

Parameter	What it Controls	Why It Matters	Common Failure if Mis-tuned
VEHICLE_CLASSES	Which object classes are tracked (IDs)	Ensures only relevant vehicles are counted and visualized	Irrelevant counts and noisy heatmaps
STOP_SPEED_THRESHOLD	Threshold for stopped vs moving classification	Highly sensitive to tracking jitter and camera perspective	Slow vehicles marked as stopped or jitter marked as motion
CONGESTION_THRESHOLD	Sensitivity for labeling an area as congested	Determines when traffic is considered a “jam”	False congestion or missed real jams
HEATMAP_DECAY	How long activity persists in heatmaps	Controls temporal stability and interpretability	Lingering heat after traffic clears or flickering maps

Applications , Limitations & Future Improvements

Real-World Use Cases	Current Constraints	Enhancements
Smart surveillance systems	Speed is approximate	Camera calibration
Crowd monitoring	Camera calibration required for accuracy	Multi-camera tracking
Traffic congestion analysis	Occlusion affects dense scenes	Behavior classification
Smart cities	Dependent on camera angle	Anomaly detection
Retail footfall analysis		Cloud dashboard integration
Road safety analytics		Edge deployment



INSTANCE SEGMENTATION WITH OBJECT TRACKING

Instance Segmentation with Object Tracking

Object Tracking (ByteTrack)

ByteTrack is a "Tracking-by-Detection" algorithm. It uses a **Kalman** Filter to predict where an object will be in the next frame based on its current velocity.

Object Detection: Finds the "box" (Where is the car?).

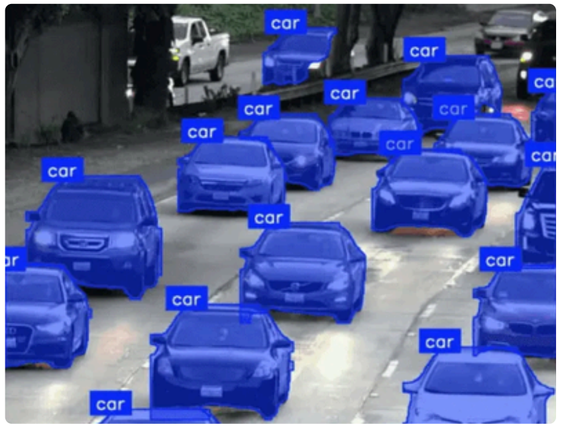
- Goal: Locate moving objects over time (temporal consistency).
- Method: Analyzes spatial location and visual features across frames.
- Logic: Assigns IDs by minimizing the cost function:
- Cost = 1 - IoU(Track_predicted, Box_detected).
- If Object overlaps significantly with Object A_t-1, assign the same Unique ID.

Instance Segmentation with Object Tracking

Instance Segmentation

While object detection draws boxes, instance segmentation identifies the exact shape of each object by creating a pixel-wise mask.

Instance Segmentation: Distinguishes individuals (e.g., "Car A" vs. "Car B") and assigns unique masks.



Mathematical Loss Function

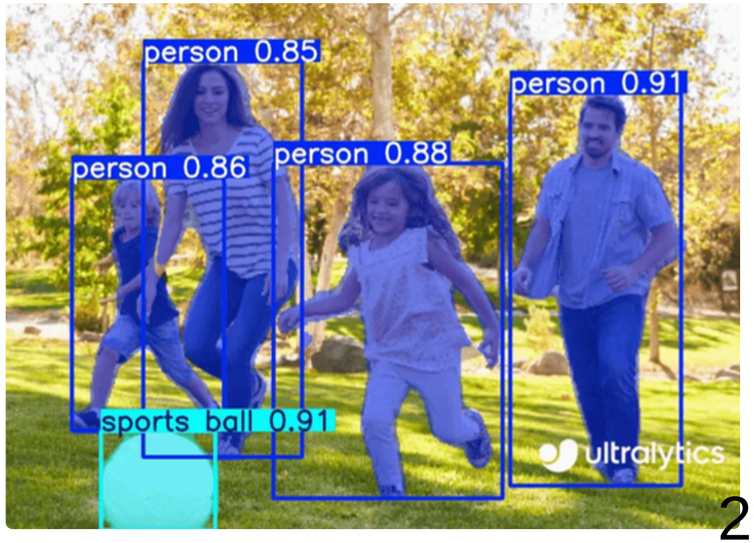
Instance segmentation in YOLOv11 uses a multi-part loss function:

Localization Loss (L_box): Measures how well the bounding box fits.

Classification Loss (L_cls): Uses Binary Cross-Entropy to verify the object class.

Mask Loss (L_mask): Specifically for segmentation, it calculates the difference between the predicted mask and the ground truth pixels.

$$L_{total} = \lambda_1 L_{box} + \lambda_2 L_{cls} + \lambda_3 L_{mask}$$




```
isegment = solutions.InstanceSegmentation( model="yolo11n-seg.pt")
```

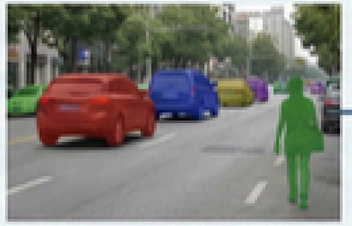
Instance Segmentation with Object Tracking: YOLOv11 & ByteTrack

1. SYSTEM INPUTS (VIDEO STREAM)

Video Stream

Input live video stream for processing.

2. INSTANCE SEGMENTATION MODEL (YOLOv11-SEG)



YOLOv11-seg Model

YOLOv11-seg model predicts pixel-level masks for each detected object instance.

3. OBJECT TRACKING (BYTETRACK)

Frame T

Frame T+1

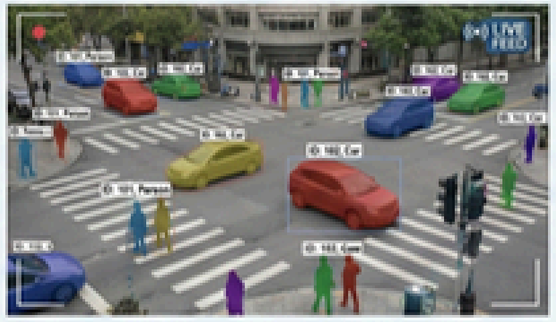
ByteTrack Tracker

ByteTrack associates detections across frames to assign a unique ID to each object.

4. COMBINED OUTPUT (MASK + ID)

Each object is represented by its precise segmentation mask and a persistent unique ID.

5. VISUALIZATION & OUTPUT

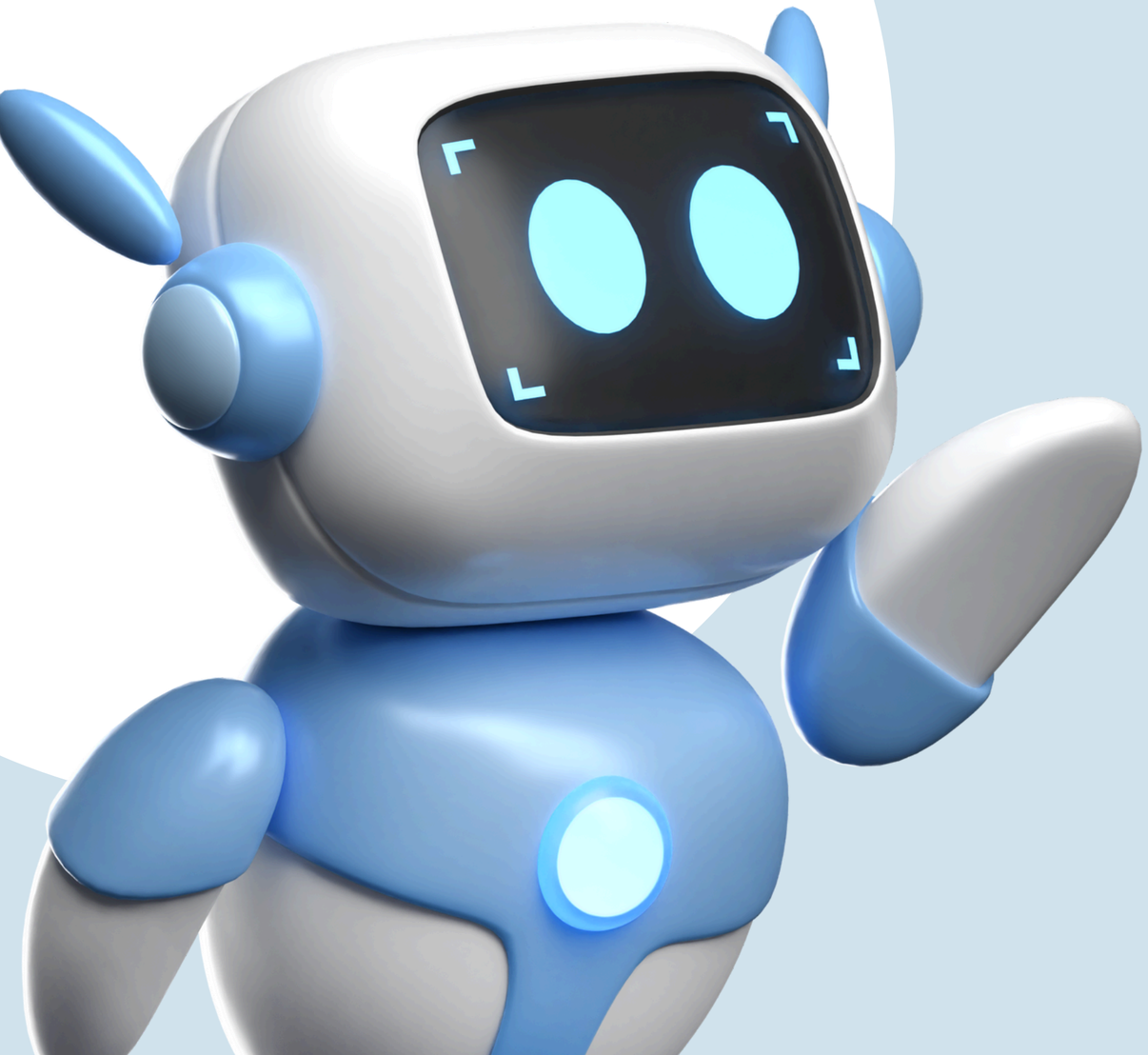


Real-time video output with overlaid segmentation masks and tracking IDs.

6. KEY APPLICATIONS & BENEFITS

- Precise Shape Analysis
- Multi-Object Tracking
- Autonomous Driving
- Video Analytics

Enables detailed understanding of object shapes and movement over time.

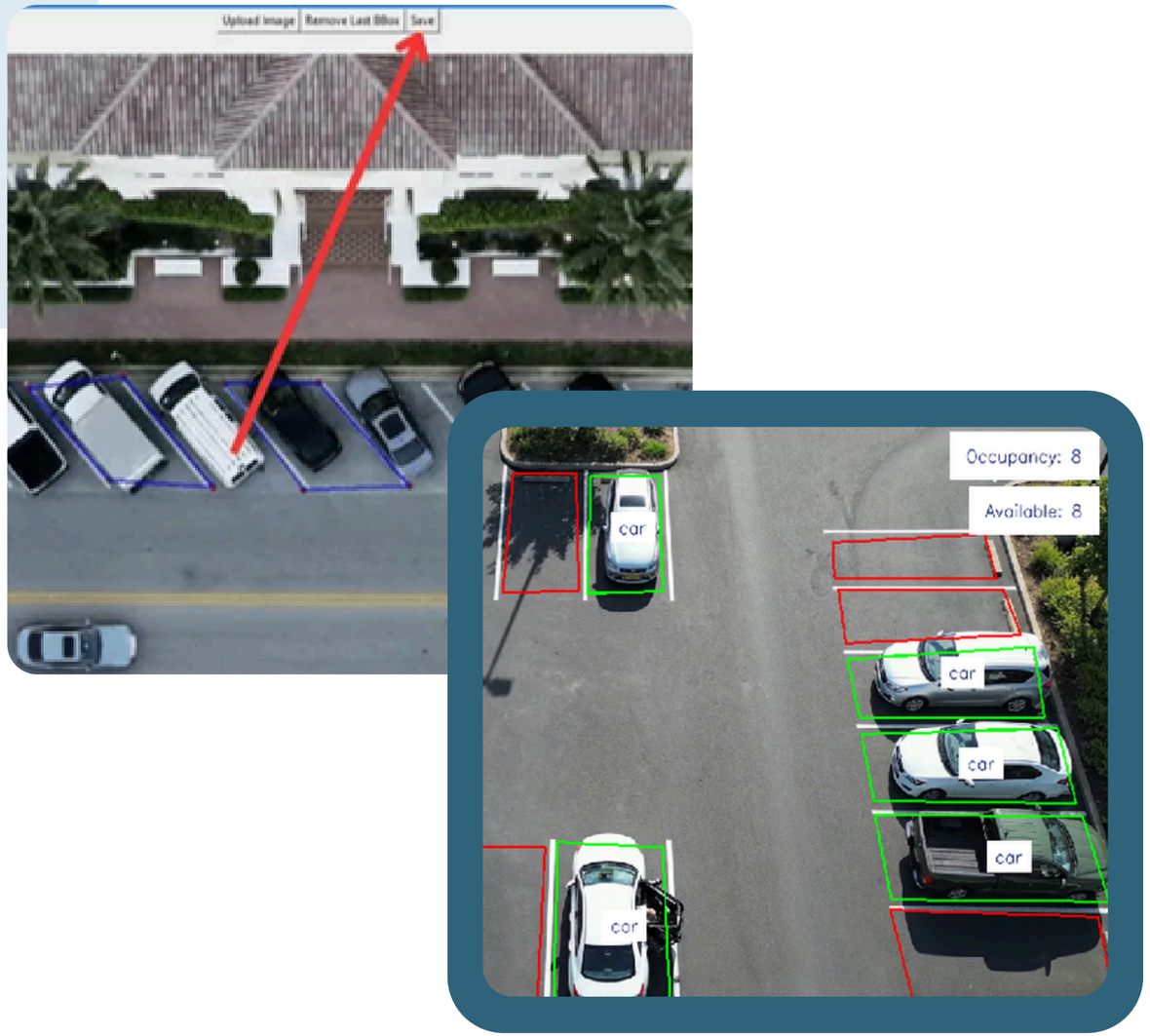


PARKING MANAGEMENT

Parking Management

Parking management is a "Region-of-Interest" (ROI) problem, moving from global object detection to local spatial state estimation.

- Polygon Space: Geometrically defined boundaries P = {v_1, v_2, ..., v_n} stored in the bounding_boxes.json file.
- Each parking slot is treated as a finite state machine with two states: Vacant (0) or Occupied (1).
- Region-Based Counting: Instead of counting all objects in a frame, the system monitors specific Regions of Interest (ROIs) defined in a bounding_boxes.json file.



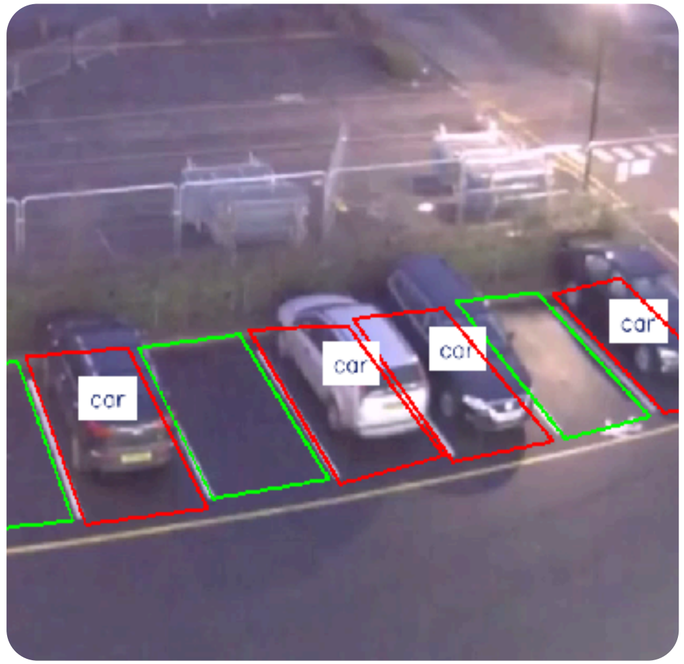
Mathematical Logic & Implementation

- Centroid Reduction: The bounding box B is reduced to a single point to simplify calculations:

$$C_x = x_{min} + \frac{width}{2}, \quad C_y = y_{min} + \frac{height}{2}$$

- The Point-in-Polygon (PIP) Algorithm: The system uses Ray Casting. A horizontal ray is projected from C: if it intersects the polygon edges an odd number of times, the point is inside.
- Area-based Occupancy (IoU): Alternatively, the system calculates the ratio of the overlap area to the slot area:

$$\text{Occupancy Ratio} = \frac{Area(B \cap P)}{Area(P)}$$



- Perspective Normalization: Polygons in the JSON are manually adjusted for Perspective Distortion, ensuring the digital boundaries match the real-world ground plane despite the camera angle.

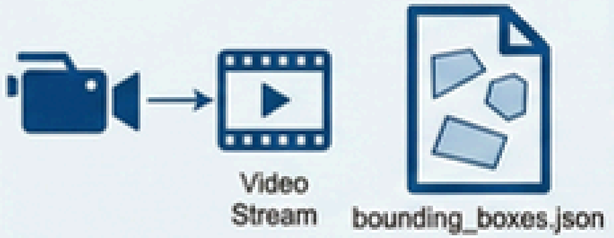
```
parking_manager = solutions.ParkingManagement(
model="yolo11m.pt",
 json_file="boxes.json",
tracker="bytetrack.yaml",
classes=[2],
Index 2: Cars
conf=0.15
```

Code Explanation – Parameter Tuning

Parameter	Recommended Value	Theoretical Justification
model	yolo11n.pt	Maximizes FPS to ensure real-time response to occupancy changes.
json_file	*.json	**Geometric Foundation:** Acts as the ground-truth map. Every pixel (x,y) of the car is checked against these polygons using the Ray Casting algorithm.
tracker	bytetrack.yaml	**Temporal Consistency:** Prevents "flicker." If a car is partially occluded, ByteTrack uses a Kalman Filter to predict its position, maintaining the "Occupied" status.
conf	0.15 to 0.25	**Sensitivity:** A lower threshold is necessary for parking lots where cars in the background may be small or poorly lit.
iou	0.5 to 0.7	**NMS Logic:** Sets the "Intersection over Union" limit. High values prevent the model from detecting multiple boxes for the same vehicle.
classes	[2]	**Target Filtering:** Specifically targets the "Car" class in the COCO dataset, ignoring pedestrians or other noise in the parking area.

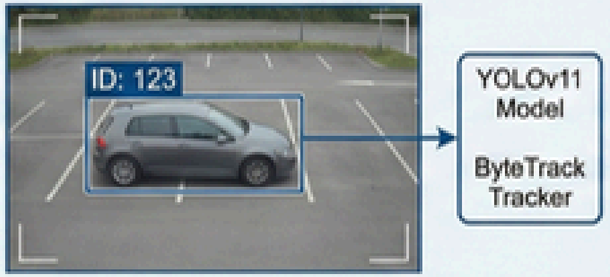
Parking Management System: YOLOv11 & Spatial Logic

1. SYSTEM INPUTS (VIDEO & MAP)



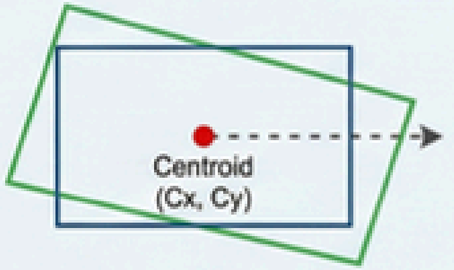
Input live video and predefined parking slot coordinates (polygons).

2. DETECTION & TRACKING (YOLOv11 & BYTETRACK)



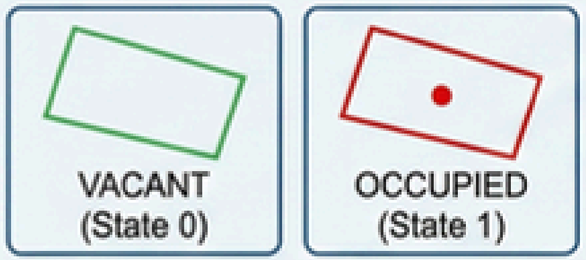
YOLOv11 detects 'Car' class; ByteTrack maintains unique ID across frames.

3. SPATIAL LOGIC (POINT-IN-POLYGON TEST)



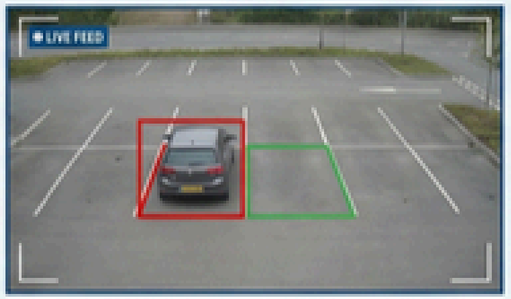
Calculates car's geometric centroid. Checks if centroid is inside polygon using Ray Casting algorithm.

4. STATE MACHINE (VACANT vs. OCCUPIED)



Boolean check: Is centroid inside? Update slot status accordingly.

5. VISUALIZATION & OUTPUT



Real-time overlay of Occupied (Red) and Vacant (Green) status on video.

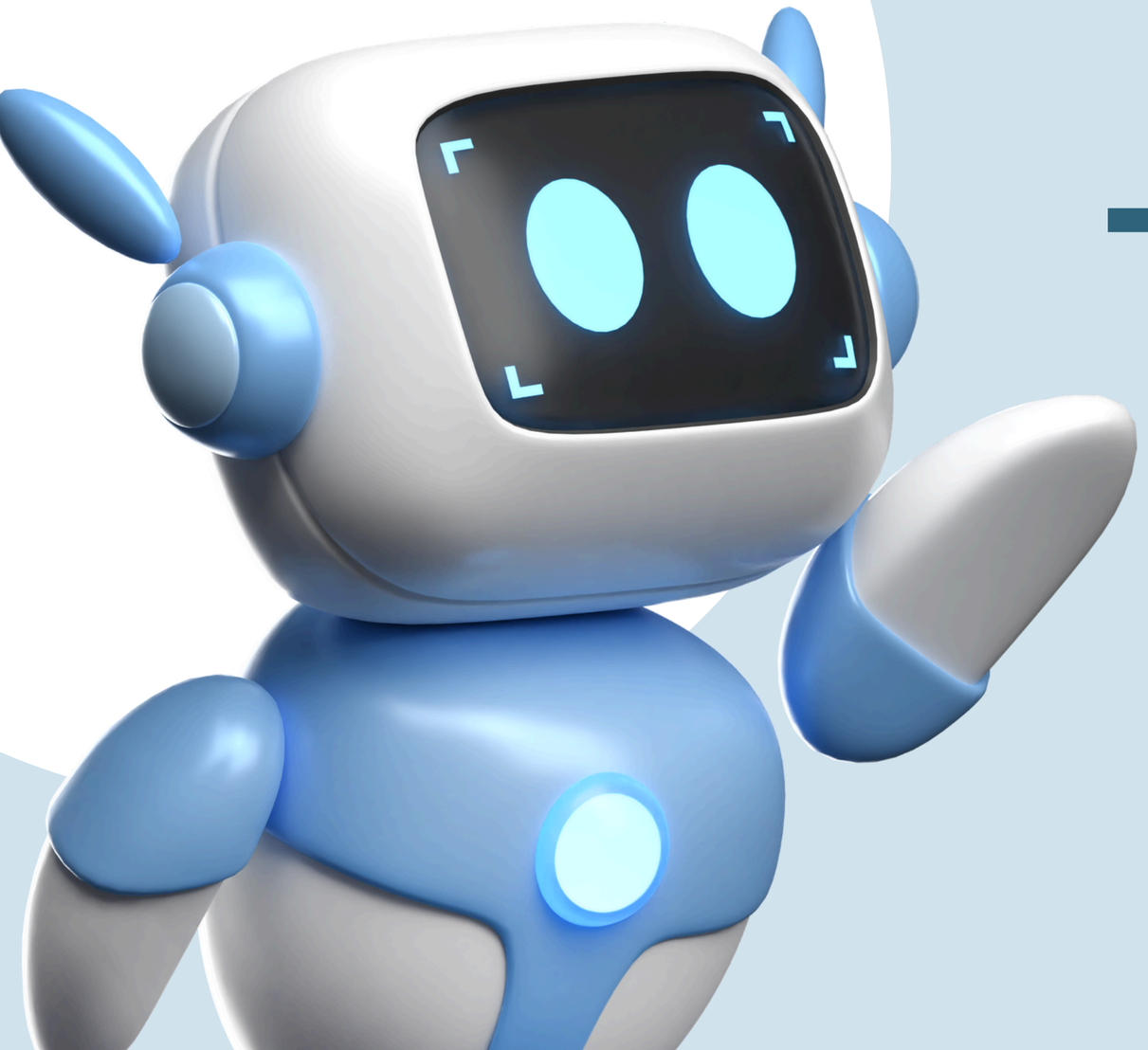
6. KEY ASSUMPTIONS & BENEFITS

- Static Camera Position
- Accurate Polygon Definition
- Automated Monitoring
- Real-time Analytics

Ensures reliable counting and status updates.

Limitations, Applications and Future Work

SYSTEM LIMITATIONS	PRACTICAL APPLICATIONS	FUTURE RESEARCH
Occlusion Vulnerability: Heavy or prolonged blocking of vehicles can lead to tracking ID "swaps" or lost detections.	**Smart City Infrastructure:** Provides precise lane occupancy and traffic density data for urban planning.	**Oriented Bounding Boxes (OBB):** Implementing OBB to better detect cars parked at diagonal angles.
Camera Dependency: The parking logic requires a static camera; any movement breaks the JSON polygon alignment.	**Automated Parking Guidance:** Real-time occupancy mapping for mobile apps to reduce driver search time.	**Hardware Optimization:** Exporting models to TensorRT with INT8 quantization for faster edge performance.
Environmental Sensitivity: Extreme shadows or low-light conditions can drop detection confidence below thresholds.	**Autonomous Perception:** Precise pixel-level footprinting for self-driving car obstacle avoidance.	**Behavioral Analytics:** Integrating Pose Estimation to detect accidents or suspicious human activity in lots.



THANK YOU!