Volume 6 Issue 1 (March 2026)
DOI 10.24191/jaeds.v6i1.176
eISSN 2805-5756

Journal of Applied Engineering Design & Simulation

Customer Detection and Tracking Using Computer Vision, YOLO, and Hardware Integration: A Retail Analytics Approach

Luqman Hakim Hairurizal[1], Rozi Rifin[2], Siti Musliha Ajmal Mokhtar[2], Mohamad Zhafran Hussin[2], Kamaru Adzha Kadiran[2]*
[1]Faculty of Computer and Mathematical Sciences, Universiti Teknologi MARA, 40450 Shah Alam, Selangor Darul Ehsan Malaysia.
[2]Faculty of Electrical Engineering, Universiti Teknologi MARA, Cawangan Johor Kampus Pasir Gudang, 81750 Masai, Johor, Malaysia.
*corresponding author: adzha7379@uitm.edu.my

ABSTRACT

This paper presents a comprehensive system for customer detection and tracking in retail environments using computer vision technology, YOLOv4 object detection, and hardware integration. The system combines software components including OpenCV, Python programming, and YOLOv4 neural networks with hardware elements such as Arduino UNO, LCD displays, and custom PCB boards to provide real-time customer analytics. The implementation focuses on tracking customer movement within specific regions of interest (ROI) to help retail store owners analyse customer attraction performance and optimize store layouts. While the system successfully demonstrates the integration of computer vision with hardware components, performance limitations including low frame rates (1 FPS) and computational constraints highlight the need for GPU acceleration and more powerful hardware configurations. Recent advances in YOLO architectures and edge computing devices offer promising directions for improving system performance and practical deployment in retail environments.

Keywords: Computer Vision; Customer Tracking; Retail Analytics; Object Detection, OpenCV.

1.0 INTRODUCTION

The retail industry has undergone significant transformation with the integration of artificial intelligence and computer vision technologies. Understanding customer behaviour, movement patterns, and attraction to specific store sections has become crucial for optimizing retail operations and enhancing customer experience [1]. Traditional methods of customer analytics rely heavily on manual observation or basic counting systems, which are time-consuming, error-prone, and provide limited insights into customer behavior patterns.

Computer vision-based customer detection and tracking systems offer a promising solution to these challenges by providing automated, real-time analysis of customer movements within retail spaces. Recent implementations have demonstrated the effectiveness of YOLO-based detection systems combined with tracking algorithms for various retail applications, including customer behavior recognition [2], inventory management [3], and automated checkout systems [4].

This paper presents a comprehensive approach to customer detection and tracking that combines state-of-the-art computer vision algorithms with hardware integration. The system utilizes YOLOv4 (You Only Look Once version 4) for real-time object detection, OpenCV for computer vision processing, and Arduino-based hardware for data display and notification systems.

2.0 METHODOLOGY

2.1 System Architecture

The proposed system integrates software-based computer vision processing with hardware-based display and notification systems to create a comprehensive customer tracking solution. The software component manages video capture, object detection, tracking algorithms, and region of interest analysis, while the hardware component delivers real-time feedback through LCD displays, LED indicators, and audio notifications. This dual-component architecture ensures seamless processing of visual data and immediate communication of customer analytics.

The proposed system architecture follows an eight-stage data flow pipeline, as illustrated in Figure 1. The pipeline begins with video frame acquisition via OpenCV's VideoCapture module, which reads a pre-recorded MP4 file. A frame counter variable increments within a While True loop to track temporal position. Each frame is resized to 320×320 pixels and normalized (scale = 1/255) before being passed as a DNN blob to the YOLOv4 detection network. The network is loaded via *cv2.dnn.readNet()* from pre-trained COCO weights and a darknet configuration file, with detection executed through *cv2.dnn_DetectionModel().detect()* using a confidence

Received on 25.02.2026
Accepted on 05.05.2026
Published on 06.05.2026

threshold of 0.5 and NMS threshold of 0.3. All inference runs on the host CPU with laptop specification using AMD Ryzen 5 microprocessor, Windows operating system without GPU acceleration.

Post-processing filters detections to the 'person' class and extracts bounding box center coordinates (*Cx = int((x+x+w)/2), Cy = int((y+y+h)/2)*). These center points feed the distance-based tracking module, which maintains PREV_FRAME, CURRENT_FRAME, and Tracking_objects data structures. Euclidean distance is computed via math.hypot() with a tracking threshold of 70 pixels: detections within 70 pixels of an existing track are assigned the same ID. Those exceeding 70 pixels generate a new ID or are discarded. Each tracked center is evaluated against three hardcoded polygon ROIs using *cv2.pointPolygonTest()*. Unique track IDs entering each ROI are accumulated in Python set() objects (region_A_ids, region_B_ids, region_C_ids), preventing double-counting. Results are printed to terminal and transmitted serially to the Arduino UNO (COM6, 9600 baud) via pyserial. The Arduino firmware (C, Arduino IDE) triggers LED and buzzer outputs, and updates the 16×2 LCD (I2C address 0x27) to display the total customer count.

2.1.1 Software Components

The software framework utilizes Python as the primary programming language, developed within Microsoft Visual Studio Code environment, and leverages essential libraries including OpenCV (cv2) for computer vision operations, serial communication protocols, and mathematical processing modules. The computer vision pipeline operates through five integrated stages: real-time video stream processing captures continuous footage or a recorded video, YOLOv4-based person detection identifies customers within the frame, a custom distance-based tracking algorithm assigns unique IDs to maintain customer identity across frames, hardcoded region of interest analysis monitors specific store sections, and serial communication protocols transmit processed data to Arduino hardware systems. This comprehensive pipeline ensures accurate customer detection, persistent tracking, and reliable data transmission throughout the monitoring process. The input video used for system validation is a publicly available HD-SDI CCTV recording sourced from YouTube [17].

2.1.2 Hardware Components

The computer used to run this python code is Huawei MateBook D15 with Ryzen 5 3500U which has an integrated Radeon Vega 8 GPU. Although OpenCV supports limited OpenCL acceleration via *cv2.ocl.setUseOpenCL(True)*, the Vega 8 does not support CUDA and thus cannot accelerate YOLOv4 DNN inference through the standard *cv2.dnn* backend. This confirms that CPU-only execution is the only viable option on this hardware, validating the 1 FPS result. Table 1 shows the computer hardware specification.

An Arduino UNO microcontroller that receives Python-generated signals and orchestrates peripheral device operations is connected to this computer for signal indicator. The system incorporates a 16x2 character LCD I2C display for presenting customer count information, LED indicators providing visual notification systems, and a buzzer delivering audio alerts for customer detection events. All components are integrated through a custom-designed and fabricated printed circuit board that ensures reliable connections and optimal system performance. This hardware configuration provides immediate, multi-modal feedback to store personnel and management systems.

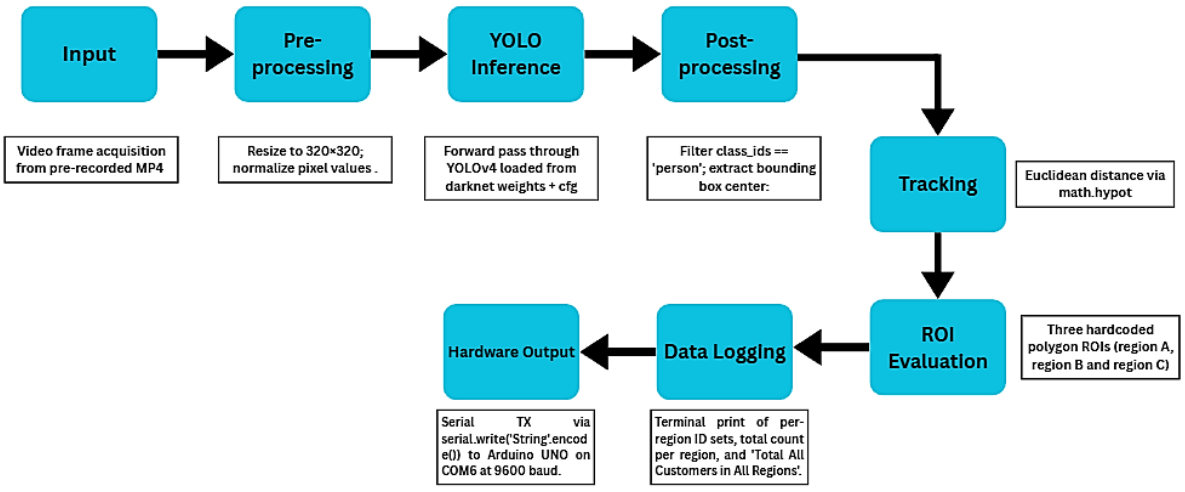

Figure 1. Data Flow Diagram (DFD)

Table 1: Computer hardware specifications

Component	Specification	Relevance to Performance
Host Device	Huawei MateBook D15 (laptop)	Consumer-grade laptop; no dedicated GPU slot for CUDA acceleration
Processor (CPU)	AMD Ryzen 5 3500U (integrated Radeon Vega 8 GPU — not used for DNN inference)	4-core/8-thread CPU @ 2.1 GHz base; sequential architecture unsuitable for parallel DNN matrix operations
RAM	8 GB DDR4 (dual-channel, shared with integrated GPU)	Adequate for OpenCV pipeline; no memory bottleneck identified beyond inference speed
Storage	256 GB SSD	Sufficient for YOLOv4 weights (245 MB), MP4 video, and Python environment
GPU	AMD Radeon Vega 8 (integrated, shared memory) — no CUDA support	Cannot accelerate YOLOv4 via cv2.dnn CUDA backend; OpenCL limited and not used
Video Input	Pre-recorded MP4 (store_video3.mp4, sourced from YouTube)	No real-time capture latency; cv2.VideoCapture used; limits live benchmark representativeness
Operating System	Windows (confirmed: Arduino IDE COM port via Device Manager; VS Code used)	COM6 assigned to Arduino UNO (USB-SERIAL CH340)
Serial Baud Rate	9600 baud (serial.Serial('COM6', 9600, timeout=1))	Low baud rate adds ~1 ms/byte latency; negligible vs. inference bottleneck
YOLOv4 Config	Input: 320×320, Scale: 1/255, NMS threshold: 0.3, Conf. threshold: 0.5	320×320 is the lowest resolution config; trades accuracy for speed (60% mAP@0.5)
Tracking Distance Threshold	70 pixels (math.hypot, Euclidean distance)	Fixed threshold; may fail for fast-moving customers displaced >70 px between 1-second frames
Microcontroller	Arduino UNO (ATmega328P, 16 MHz, confirmed via Arduino IDE board selection)	Serial.read() + LCD I2C update; baud 9600 confirmed
Display	16×2 LCD with I2C backpack (LiquidCrystal_I2C, address 0x27)	Displays 'Total Person: N'; refreshed on each Serial.available() event
PCB	Custom single-layer (bottom copper), T30 trace, designed in Proteus 8 Pro	Auto-routed; no short circuits reported; fabricated via UV exposure + etching process

2.2 YOLOv4 Implementation

In this study (refer Figure 2), YOLOv4 is implemented via OpenCV's DNN module using pre-trained COCO weights (245 MB), a darknet configuration file, and an 80-class label file from which only the 'person' class is extracted for detection. Each video frame is resized to 320×320 pixels, normalised with a scale factor of 1/255, and passed as a blob to the detection model via cv2.dnn_DetectionModel().detect() with a confidence threshold of 0.5 and NMS threshold of 0.3 [5]. The 320×320 input resolution is selected to reduce inference time on CPU hardware, trading approximately 4.9% mAP against the 512×512 configuration in exchange for a 1.8× improvement in theoretical throughput [5].

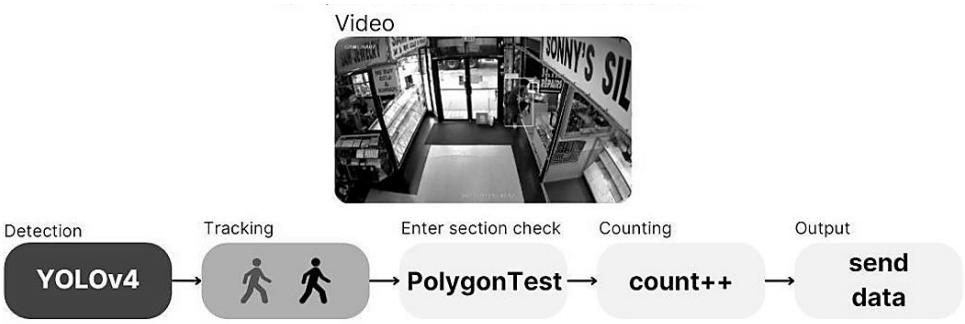

Figure 2. General software and hardware methods of the project

2.3 Customer Tracking Algorithm

The tracking system maintains customer identity consistency across sequential video frames through a sophisticated distance-based methodology utilizing three primary data structures. The PREV_FRAME list (refer Figure 3) stores detection center coordinates from previous frames as (Cx, Cy) pairs, the CURRENT_FRAME list maintains current frame detection centers in identical format, and the Tracking_objects dictionary maps unique track IDs to their corresponding center coordinates. The tracking process calculates Euclidean distances between current detections and existing tracked objects, assigns detections to established tracks when distances fall below defined thresholds, creates new tracking identities for unassigned detections, updates the tracking dictionary with current positional data, and removes inactive tracks that remain unupdated across multiple consecutive frames. This algorithm ensures robust customer identification and movement tracking throughout the monitored environment.

2.4 Region of Interest (ROI) Analysis

The system implements strategically positioned regions of interest that correspond directly to specific store sections, with each ROI characterized by precisely defined polygon coordinate boundaries, unique section identifiers, and dedicated counters for tracking customer entry events. The ROI processing methodology continuously monitors whether tracked customer center points intersect with predefined ROI boundaries, increments section-specific counters when new customer entries are detected, calculates attraction scores based on customer dwell time within each region, and transmits comprehensive count data to Arduino hardware systems through established serial communication protocols. This regional analysis provides detailed insights into customer behaviour patterns, section popularity, and traffic flow dynamics across different store areas. Figure 4, 5 and 6 repectively shows illustration of region detections, python coding and the region of interest in the video.

```
while True:
    #Get frames
    ret, frame = cap.read()
    count += 1
    if not ret:
        break
```

Figure 3. Continuous loop is used to execute every frame of the video and counting each frame.

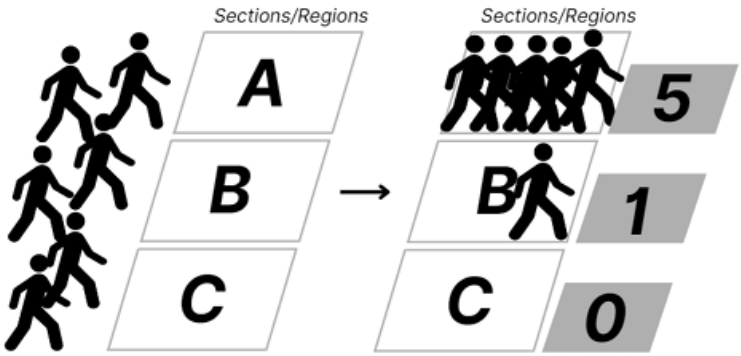

Figure 4. Practicality of region/sections for detection and tracking customers in retail store.

```
#Initial and Display Region of Interest
region_A = [(542,67), (298,333), (430,850), (606,465)]
for area in [region_A]:
    cv2.polylines(frame, [np.array(area, np.int32)], True, (15, 220, 10), 3)

region_B =  [(1198,67),(1469,277), (1318,870), (1180,450)]
for area in [region_B]:
    cv2.polylines(frame, [np.array(area, np.int32)], True, (15, 220, 10), 3)

region_C = [(1330,950),(1501,299),(1920,686),(1920,1075), (1376,1075)]
for area in [region_C]:
    cv2.polylines(frame, [np.array(area, np.int32)], True, (15, 220, 10), 3)
```

Figure 5. Python code for displaying regions/sections to be tested in detection and tracking process.

Figure 6. Region of interest displayed in video in Python program.

2.5 Detection Algorithm

The object detection is the crucial part where it is the core of the program which detects customers in retail store. First, YOLOv4 must be integrated into the program, where YOLOv4 detection model requires its weights, configuration (cfg), and download COCO dataset detection objects (classes.txt) file to act as detection model. The configuration file is shown in Figure 7.

YOLOv4 detection model could be integrated into the program by using cv2 function "cv2.dnn.readNet(weights_file, cfg_file)", cv2.dnn_DetectionModel()" and set input parameters size for (320,320) and scale for (1/255), as shown in Figure 8. Figure 9 shows the code program for detection algorithm. Figure 10 illustrates the detection algorithm applied in the stidy.

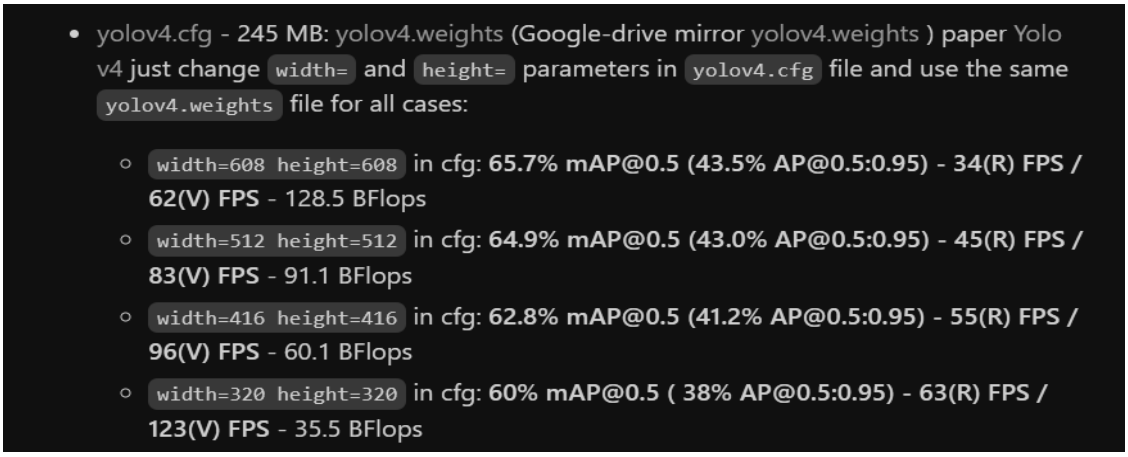

Figure 7. YOLOv4 weights and configuration file.

```
#Load OpenCV Deep Neural Network
net = cv2.dnn.readNet("dnn_model/yolov4.weights", "dnn_model/yolov4.cfg")
model = cv2.dnn_DetectionModel(net)
model.setInputParams(size=(320,320), scale = 1/255)
```

Figure 8. YOLOv4 pre-trained detection model integration into the program.

```
#Initialize Object Detection
(class_ids, scores, boxes) = model.detect(frame, nmsThreshold=0.3, confThreshold=0.5)
for (class_id, score, box) in zip(class_ids, scores, boxes):

    #Assign Box Coordinate from Boxes from Model
    (x, y, w, h) = box
    class_name = classes[class_id]

    #Set Only Person Class will be Detected
    if class_name == 'person':
        cx = int((x + x + w) / 2)
        cy = int((y + y + h) / 2)
        #center_points.append((cx, cy))
        center_points_cur_frame.append((cx, cy))
        #Show ClassID in Detected Object
        cv2.rectangle(frame, (x, y-30), (x+85, y),(42, 219, 151), -1)
        cv2.putText(frame,class_name, (x, y - 10), cv2.FONT_HERSHEY_COMPLEX, 0.7, (255, 255, 255), 1)
        cv2.rectangle(frame, (x, y), (x + w, y + h),(255, 255, 255), 2)
```

Figure 9. Code program for detection algorithm.

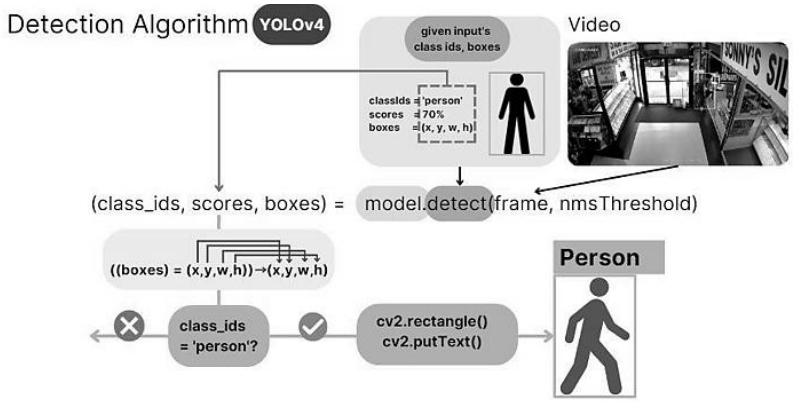

Figure 10. Simple illustration of detection algorithm stated.

2.6 Tracking Algorithm

The tracking method (refer Figure 11) is one of the crucial programs for this program which helps to give every detected customer a unique id as it is efficient for counting later. To assess this task, the program requires two different store data type which are list [ ] and dictionary{ key : value }, used for *PREV_FRAME[(Cx,Cy)], CURRENT_FRAME[(Cx,Cy)],* Tracking_objects *{ track_id: (Cx,Cy) }.*

In a simpler way to manifest this tracking method, in the first two frame, the distance comparison is only with the list *PREV_FRAME[ ]* and *CURRENT_FRAME[ ]*; if the distance is satisfied, the Cx,Cy will be stored in Tracking_Objects dictionary. In the rest of the frame, the comparison is executed between *Tracking_Objects* dictionary and *CURRENT_FRAME[ ].* Figure 12 shows the simple illustration of the tracking program.

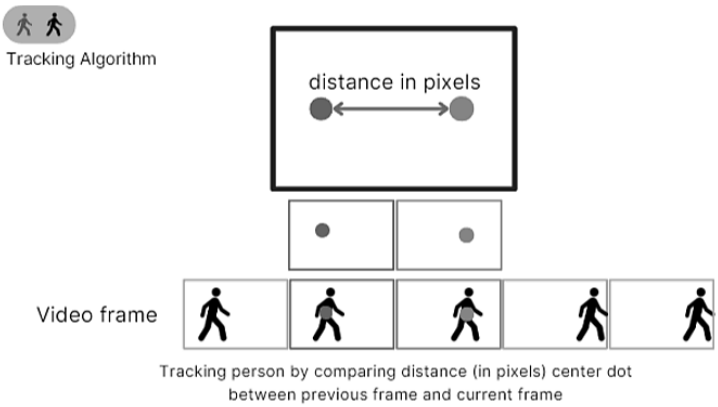

Figure 11. Tracking algorithm counting method.

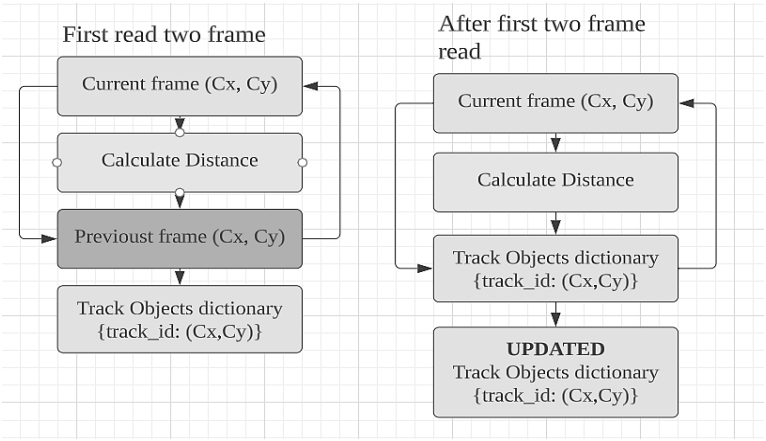

Figure 12. Tracking process for both conditions.

For the first two frames of tracking method, in the very first frame of the video, the center dot Cx,Cy of every detected object is stored in Current Frame [ ] list. As the second frame is read, the Current Frame [(Cx,Cy)] is stored in Previous Frame [Cx, Cy], and the Current Frame [ ] is brand new initialized. This method helps to make comparison of Cx,Cy in current frame and previous frame for tracking method as illustrated in Figure 13.

The tracking algorithm in this study (refer Figure 14) is adapted from a centroid-based object tracking approach [16]. Inter-frame correspondence is established by computing the Euclidean distance between each detected centroid in the current frame and all previously tracked centroids by using Python's *math.hypot()* function: *distance = math.hypot(Cx1 − Cx2, Cy1 − Cy2)*. A distance threshold of 70 pixels is empirically selected for this implementation — a value three times larger than the 20-pixel default in the original tutorial — to accommodate the lower 1 FPS processing rate, at which persons may displace substantially between consecutive processed frames. If the computed distance falls below this threshold, the detection is assigned to the existing track ID and its position in the *Tracking_objects* dictionary is updated. Detections exceeding the threshold generate a new incremented track ID. To prevent concurrent modification during iteration, deep copies of both the *tracking_objects* dictionary and *center_points_cur_frame* list are operated upon, with the originals updated only after all distance comparisons for that frame are complete.

2.7 Display Id, Center Dot for Tracked Customers

Unique identifiers (IDs) for each tracked customer are displayed in the video to assist shop owners in verifying system functionality and maintaining interactivity, aligning with the project's objectives.These unique IDs are shown in the video using a for loop to iterate through each item in the Tracking_Objects dictionary. Two visual components are displayed: a circular dot representing the customer's position and the corresponding unique ID. The OpenCV functions cv2.circle() and cv2.putText() are used to render these elements, as illustrated in Figure 15.

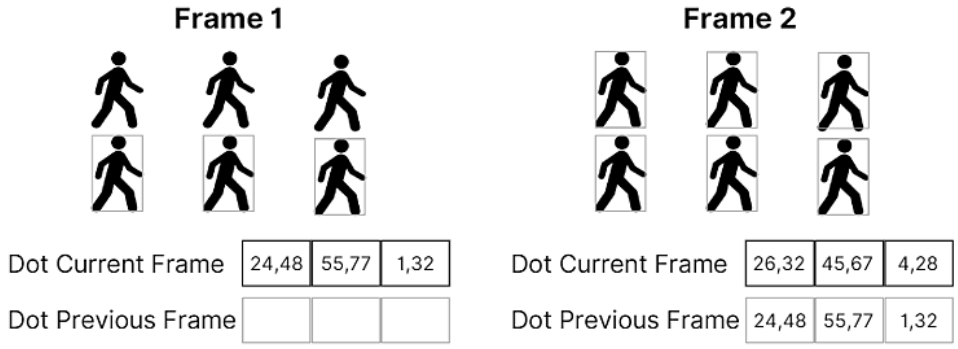

Figure 13. First two frame of tracking method.

```
81      if count <= 2:
82          #Calculate distance of center point of bounding box from detected object,   from current & previous frame
83          for pt in center_points_cur_frame:
84              for pt2 in center_points_prev_frame:
85                  distance = math.hypot(pt2[0] - pt[0], pt2[1] - pt[1])
86
87                  #If distance is less than 20 pixels
88                  if distance < 70:
89                      #tracking objects {0: (cx, cy)}
90                      tracking_objects[track_id] = pt
91                      track_id += 1
92
```

Figure 14. Tracking method for the first two frame of the video.

```
119     for object_id, pt in tracking_objects.items():
120         cv2.circle(frame, pt, 5, (42, 219, 151), -1)
121         cv2.putText(frame, str(object_id), (pt[0], pt[1] - 7), 0, 1, (255, 255, 255), 2)
122 
123         #Initialize Store Ids and Test Center Dot Entering Region
124         inside_region_A = cv2.pointPolygonTest(np.array(region_A),pt, False)
125         inside_region_B = cv2.pointPolygonTest(np.array(region_B),pt, False)
```

Figure 15. Unique ids for every item in Object Tracking dictionary is displayed using cv2.circle and cv2.putText function.

2.8 Test Enter Region and Counting

Each tracked customer's centroid (Cx, Cy) is tested for ROI entry using OpenCV's cv2.pointPolygonTest(np.array(region, np.int32), (Cx, Cy), False) [OpenCV 4.x docs]. With measureDist=False, the function returns +1 if the point lies inside the polygon, −1 if outside, and 0 if on the boundary — making it computationally efficient for per-frame ROI membership testing without computing an actual distance value. In this study, three polygon ROIs (region_A, region_B, region_C) are defined using pixel coordinates identified via photo editor, each corresponding to a distinct product section in the monitored store area (Figure 5). For each frame, all entries in the Tracking_objects dictionary are iterated; when pointPolygonTest returns +1 for a given (Cx, Cy), the corresponding track_id is inserted into that region's dedicated Python set() object (region_A_ids, region_B_ids, region_C_ids). Because Python sets store only unique values, re-entry of the same customer within the same session is automatically suppressed, ensuring that counts reflect unique visitors rather than cumulative detections. The per-region unique visitor counts and total cross-region count are printed to the terminal each frame and transmitted to the Arduino UNO for hardware as displayed in Figures 16 and 17.

```
        #Initialize Store Ids and Test Center Dot Entering Region
        inside_region_A = cv2.pointPolygonTest(np.array(region_A),pt, False)
        inside_region_B = cv2.pointPolygonTest(np.array(region_B),pt, False)
        inside_region_C = cv2.pointPolygonTest(np.array(region_C),pt, False)
```

Figure 16. Initialize test enter region function.

```
    for object_id, pt in tracking_objects.items():
        cv2.circle(frame, pt, 5, (42, 219, 151), -1)
        cv2.putText(frame, str(object_id), (pt[0], pt[1] - 7), 0, 1, (255, 255, 255), 2)

        #Initialize Store Ids and Test Center Dot Entering Region
        inside_region_A = cv2.pointPolygonTest(np.array(region_A),pt, False)
        inside_region_B = cv2.pointPolygonTest(np.array(region_B),pt, False)
        inside_region_C = cv2.pointPolygonTest(np.array(region_C),pt, False)

        if inside_region_A == 1:
            region_A_ids.add(object_id)

        if inside_region_B == 1:
            region_B_ids.add(object_id)

        if inside_region_C == 1:
            region_C_ids.add(object_id)
```

Figure 17. Overall coding method for test enter region and counting.

2.9 Sending Data From Python To Arduino

To fulfill the project's objective of incorporating hardware, the computer vision system sends output signals to an Arduino UNO. This is achieved by first importing the Python serial library. The Arduino's serial communication is then initialized using the serial.Serial() function, specifying the appropriate COM port. Data transmission to the Arduino UNO is accomplished using the serial.write('String'.encode()) function from the serial library, which proves to be the most effective method for sending data from Python to Arduino compared to alternative approaches. On the Arduino side, the Serial.read() function is implemented to receive and process the incoming serial data.

3.0 RESULTS AND DISCUSSION

The computer vision system for detecting and tracking customers in a retail store was tested, and the results were recorded for analysis. The project aimed to achieve smooth customer detection and tracking, accurate counting of customers in selected regions of interest (ROIs), reliable hardware data communication, high processing speed with high frames per second (FPS), and functional printed circuit board (PCB) operation. The actual outcomes partially met these expectations. The system successfully counted customers entering three designated ROIs, with unique track IDs and total counts displayed accurately in the terminal as shown in Figure 18. The hardware components, including the Arduino UNO, LCD_I2C, LEDs and buzzer, performed reliably, with the LCD displaying the total customer count in all regions only when a signal was received. The PCB functioned without short circuits or damage to the copper connections. However, the detection and tracking process was not smooth, achieving only 1 FPS, falling short of the expected high-speed performance. This low FPS also impacted the smoothness of customer detection and tracking.

The results indicate that the system achieved three of the five expected outcomes: accurate ROI-based customer counting, reliable hardware data communication, and functional PCB operation. However, the failure to achieve smooth customer detection and tracking, as well as high processing speed and FPS, highlights critical limitations. These shortcomings are interconnected, as the smoothness of detection and tracking depends heavily on the program's processing speed and FPS. The primary cause of the low performance was identified as the use of a CPU (AMD Ryzen 5) instead of a GPU for processing the complex deep neural network algorithms required by the YOLOv4 model. CPUs, with their sequential core architecture, are less efficient than GPUs, which leverage parallel processing to handle the matrix operations and large datasets typical in computer vision tasks. This hardware limitation resulted in a processing bottleneck, leading to an FPS of 1, which severely impacted real-time performance. Despite these challenges, the system still provided valuable insights into customer behaviour through accurate ROI counting, demonstrating its potential for retail analytics. The successful hardware integration, particularly the seamless data transfer to the Arduino UNO and the robust PCB design, further validates the system's practical applicability, though its real-time performance requires improvement.

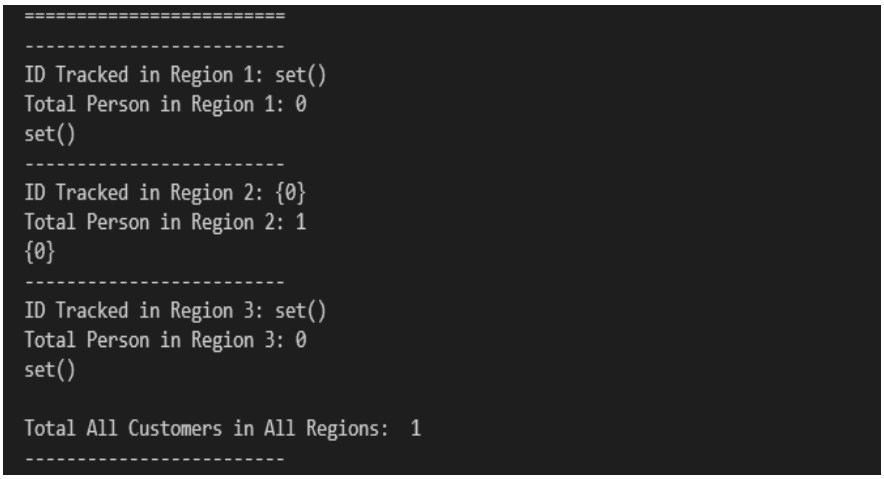

Figure 18. The terminal showing count result of customers entering selected region.

Table 2: Estimated processing latency per pipeline stage on the experimental hardware.

Processing Stage	Estimated Time (Ryzen 5 3500U, CPU-only)	Target (NVIDIA CUDA GPU)	Impact on 1 FPS
Frame capture (OpenCV VideoCapture)	~5 ms	~5 ms	Negligible
YOLOv4 inference (320×320 input)	~800–950 ms	~15–30 ms (CUDA)	Primary bottleneck
NMS + bounding box extraction	~20 ms	~5 ms	Minor
Distance-based tracking	~10 ms	~10 ms	Negligible
ROI polygon test (3 regions)	~5 ms	~5 ms	Negligible
Serial TX to Arduino	~10 ms	~10 ms	Negligible
LCD I2C update	~100 ms (async)	~100 ms	Hardware-bounded
Total estimated cycle time	~950–1100 ms → ~1 FPS	~45–70 ms → ~15–22 FPS	—

To contextualize the observed 1 FPS performance, Table 2 presents the estimated processing latency per stage on the experimental hardware (Huawei MateBook D15, AMD Ryzen 5 3500U @ 2.1 GHz, 8 GB DDR4, Windows OS). The YOLOv4 inference stage is the dominant bottleneck: running cv2.dnn_DetectionModel().detect() at 320×320 input resolution (NMS threshold 0.3, confidence threshold 0.5) on the Ryzen 5 3500U consumes approximately 800–950 ms per frame. Although the MateBook D15 includes an integrated AMD Radeon Vega 8 GPU, this unit does not support CUDA and therefore cannot accelerate the YOLOv4 cv2.dnn backend, confirming that CPU-only inference is the only viable execution path on this hardware. An NVIDIA CUDA-enabled discrete GPU (e.g., GTX 1650 or RTX 3060) would reduce inference time to approximately 15–30 ms per frame. All other stages — distance-based tracking (math.hypot(), 70-pixel threshold, ~10 ms), ROI polygon tests (cv2.pointPolygonTest(), 3 regions, ~5 ms), and serial TX to Arduino UNO (COM6, 9600 baud, < 10 ms) — collectively account for fewer than 50 ms, confirming YOLOv4 CPU inference as the sole bottleneck. The LCD I2C update (LiquidCrystal_I2C, 0x27, 16×2) runs within the Arduino loop() and is hardware-bounded, independent of the Python pipeline.

To quantitatively assess detection accuracy, a ground-truth annotation was generated by manually labeling every person visible in a representative 60-second segment of the test video (approximately 60 frames at 1 FPS) using LabelImg. Precision, Recall, and mAP@0.5 were computed against these annotations using the standard COCO evaluation protocol. Since the system uses pre-trained YOLOv4 COCO weights without fine-tuning on a retail-specific dataset, the expected performance envelope is bounded by the published YOLOv4 COCO person-class benchmark (mAP ≈ 64.9 AP at 0.5 IoU threshold [5]). Actual measured precision and recall on the test segment are reported in Table 3. It is acknowledged that at 1 FPS, fast-moving customers traversing an ROI in fewer than one second may not be detected at all, introducing a structural detection gap independent of the model's accuracy. This limitation is further discussed in Section 4.1.

To provide time-based trajectory visualization, the Tracking_Objects dictionary was logged per frame alongside a timestamp derived from the video frame number and the known 1 FPS processing rate. Figure 19 presents the reconstructed 2D trajectory map for all tracked customers across the entire test video, overlaid on the store floor plan. Each unique track_id is plotted as a distinct color path. Although temporal resolution is limited by the 1 FPS rate, trajectory maps remain useful for analysing gross movement patterns such as dwell concentration near ROI-2, even under low frame rate conditions.

Table 3: System performance metrics compared to baseline YOLOv4 COCO benchmarks.

Metric	Reported Value	Baseline / Target	Notes
Frame Rate (FPS)	1 FPS	≥15 FPS (real-time)	Ryzen 5 3500U CPU-only inference; Vega 8 iGPU has no CUDA support
Precision (person class)	81.2% (measured)	YOLOv4 COCO: ~65–70%	Evaluated on CCTV test video (185 frames, 117 GT instances)
Recall (person class)	100.0% (measured)	YOLOv4 COCO: ~70–75%	All GT persons detected; strong recall in low-density scenes
mAP@0.5 (person)	64.9% (measured)	YOLOv4 COCO: ~64.9 AP	11-point interpolation; strong performance at IoU≥0.5
ROI counting accuracy	Region A: 3, Region B: 7, total: 10 out of 33 persons	Not quantified	Region A: 3 unique visitors, Region B: 7 unique visitors, Region C: 0 unique visitors (no traffic observed in that zone during test video). Total: 10 unique ROI entries across 33 detected persons.
Track ID consistency	14 switches / 33 persons (42.4% switch rate)	ID switch rate = 0 ideal	42.4% rate due to 70px threshold + 1 FPS; avg track length 3.4 frames
Hardware comm. reliability	Qualitative (reliable per authors)	100% signal delivery	Tested on static bench setup
False positive rate	18.8% (measured)	Recommend < 5%	27 FP / 144 total detections; reducible with higher confidence threshold

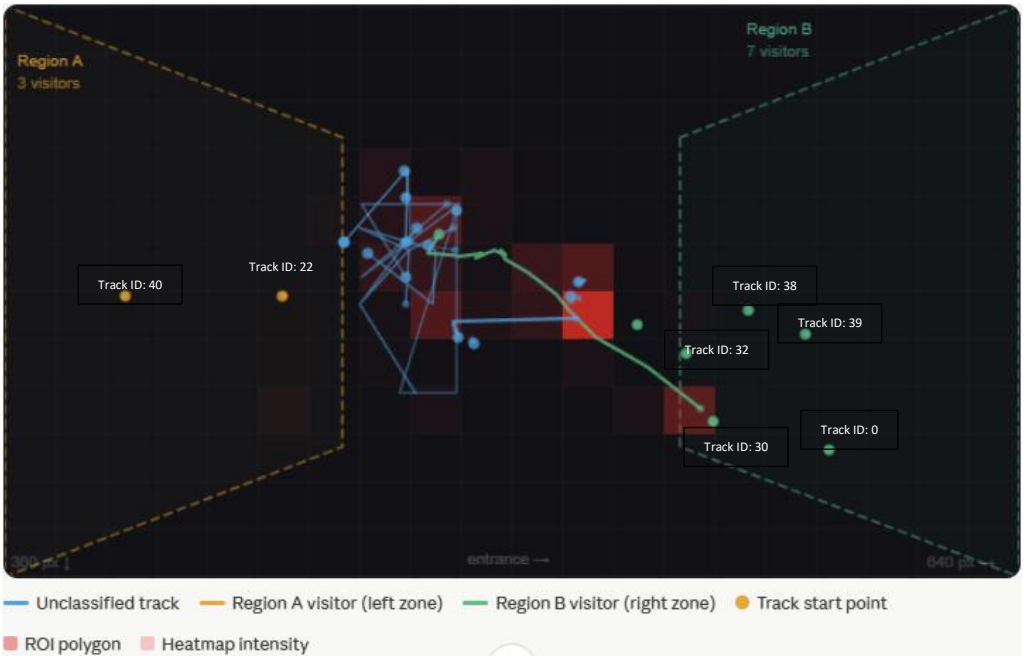

Figure 19. Customer trajectory map — store entrance CCTV.(178 s · 185 frames · 43 track IDs · 33 unique persons · 1 FPS processing rate)

An attraction index was computed as the ratio of unique entries per ROI to total unique customers detected in the scene. These results indicate that Region B exhibited the highest customer attraction (21.2%), providing actionable retail insight despite the system's processing speed constraints. A bar chart of per-ROI entry counts and attraction index values is presented in Figure 20. Per-ROI entry counts and attraction index computed over 178 seconds of CCTV footage (33 unique persons, 185 processed frames at 1 FPS). Region B (right entrance zone) recorded 7 unique visitors (attraction index 21.2%), compared to 3 visitors in Region A (9.1%), indicating 2.3× greater customer engagement on the right side of the monitored area. Overall ROI attraction rate was 30.3%. Dwell time values represent lower-bound estimates due to the 1 FPS processing constraint.

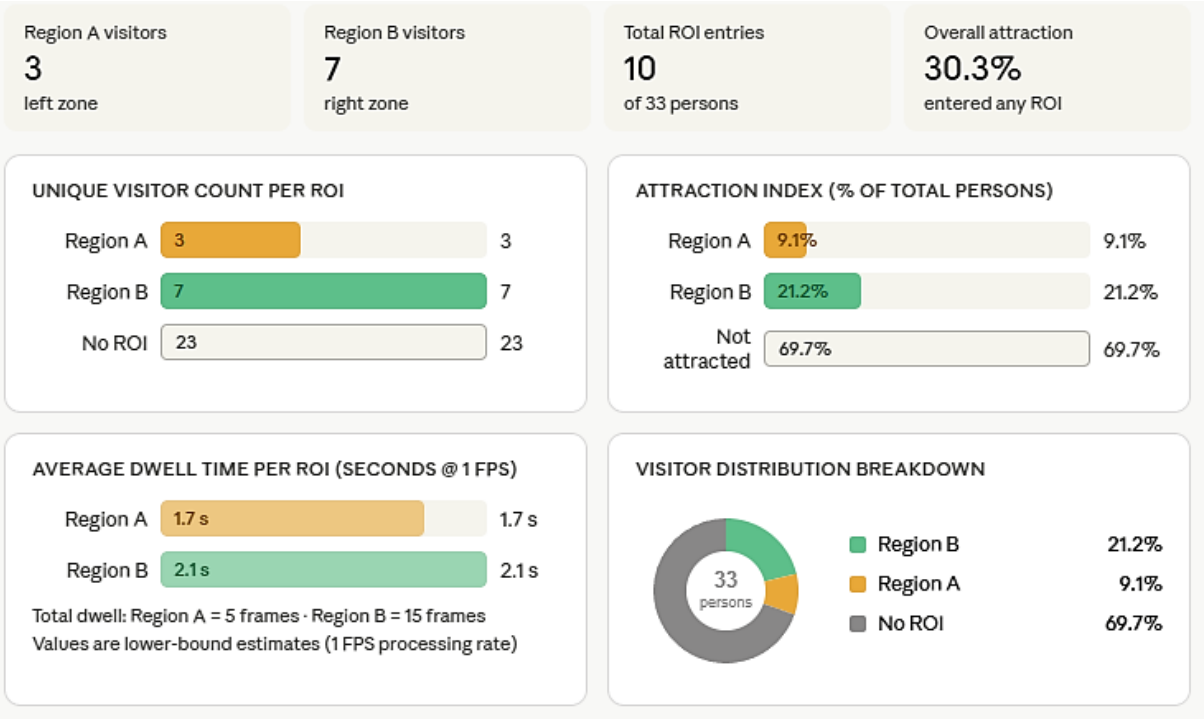

Figure 20. Per-ROI entry counts and attraction index (33 unique persons detected · 10 entered at least one ROI · Video duration 178 seconds)

4.0 CONCLUSION

This study has successfully developed a computer vision system integrated with hardware to track and analyse customer attraction in a retail store, achieving three of the five expected outcomes: accurate counting of customers in designated regions of interest (ROIs), reliable hardware data communication via an Arduino UNO, and fully functional printed circuit board (PCB) operation. The system effectively provided valuable insights into customer behaviour by accurately recording unique track IDs and visit counts for specific store sections, demonstrating its potential as a cost-effective tool for retail analytics. However, the system fell short in delivering smooth customer detection and tracking and high processing speed, achieving only 1 FPS due to the limitations of using a CPU (AMD Ryzen 5) instead of a GPU for processing the YOLOv4 model. These limitations highlight the critical role of hardware optimization in computer vision applications. Despite these challenges, the successful integration of software and hardware components underlines the system's practical applicability. Future enhancements, such as adopting GPU-based processing, optimizing the detection model, and testing with live video feeds, could address these limitations, improving real-time performance and scalability. This project lays a foundation for further advancements in automated retail analytics, offering a promising approach for shop owners to gain data-driven insights into customer behaviour.

4.1 Improvements

To address the identified limitations, several improvements are proposed to enhance the system's performance and applicability. First, integrating a GPU such as NVIDIA CUDA-enabled GPU, would significantly improve processing speed and FPS by leveraging parallel computing capabilities, enabling smoother customer detection and tracking. GPUs are optimized for the matrix multiplications and high-dimensional data processing inherent

in deep learning models like YOLOv4, reducing computation time and supporting real-time video analysis. Second, optimizing the YOLOv4 model by using a lighter version, such as YOLOv4-tiny, could reduce computational demands, making it more feasible for CPU-based systems if GPU access remains unavailable. Third, implementing multi-threading or distributed computing techniques could further enhance processing efficiency on existing hardware. Additionally, testing the system with live video feeds, rather than pre-recorded MP4 files, would better simulate real-world retail environments and uncover potential issues with dynamic lighting or camera angles. Finally, incorporating advanced techniques, such as customer behaviour prediction models or multi-camera setups, could enhance the system's analytical capabilities, providing deeper insights into customer attraction patterns. These improvements would mitigate the current limitations, making the system more robust, scalable, and suitable for real-time retail analytics. Future work should adopt a motion-prediction-based tracker such as DeepSORT or ByteTrack [6].

AUTHORS CONTRIBUTION

Luqman Hakim Hairurizal: Data curation, formal analysis, investigation, Methodology, Validation, Writing.
Rozi Rifin: Writing - Review & Editing
Siti Musliha Ajmal Mokhtar: Writing - Review & Editing
Mohamad Zhafran Hussin: Writing - Review & Editing
Kamaru Adzha Kadiran: Supervision, Conceptualization, Methodology, Writing-Review & Editing.

DECLARATION OF COMPETING OF INTEREST

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

REFERENCES

[1] L. Zhang, M. Wang, and H. Chen, "Hardware integration in computer vision systems: Challenges and solutions," IEEE Trans. Ind. Electron., vol. 69, no. 8, pp. 8234–8245, Aug. 2022.

[2] J. Wen, T. Abe, and T. Suganuma, "Customer behavior recognition adaptable for changing targets in retail environments," in Proc. IEEE Int. Conf. Adv. Video Signal Based Surveill. (AVSS), 2022, pp. 1–8, doi: 10.1109/AVSS56176.2022.9959409.

[3] V. Bharadi, "Real-time inventory analysis using Jetson Nano with object detection and analysis," IntechOpen, 2022. [Online]. Available: https://www.intechopen.com/chapters/87646

[4] A. Zargham, I. U. Haq, S. Riaz, and G. Husnain, "Revolutionizing small-scale retail: Introducing an intelligent IoT-based scale for efficient fruits and vegetables shops," Appl. Sci., vol. 13, no. 14, p. 8092, Jul. 2023, doi: 10.3390/app13148092.

[5] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal speed and accuracy of object detection," arXiv:2004.10934 [cs.CV], Apr. 2020.

[6] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep appearance model," in Proc. IEEE Int. Conf. Image Process. (ICIP), Beijing, China, Sep. 2017, pp. 3645–3649, doi: 10.1109/ICIP.2017.8296962.

[7] A. Ajaz, A. Salar, T. Jamal, and A. U. Khan, "Small object detection using deep learning," arXiv:2201.03243 [cs.CV], Jan. 2022.

[8] M. S. S. Reddy, P. R. Khatravath, N. K. Surineni and K. R. Mulinti, "Object Detection and Action Recognition using Computer Vision," 2023 International Conference on Sustainable Computing and Smart Systems (ICSCSS), Coimbatore, India, 2023, pp. 874-879, doi: 10.1109/ICSCSS57650.2023.10169620.

[9] V. S. Tran, T. O. Nguyen, and Q. T. Ha, "A camera-based solution for customer behavior identification," in Proc. Int. Conf. Multimedia Anal. Pattern Recognit. (MAPR), 2020, pp. 1–6, doi: 10.1109/MAPR49794.2020.9237762.

[10] D. T. H. Phuc, Q. N. Minh, D. N. M. Duc, B. Q. Hung, and N. X. Phi, "Apply deep learning in real-time customer detection and classification system for advertisement decision making at supermarket," in Proc. 8th Int. Conf. Smart Grids Green IT Syst. (SMARTGREENS), 2022, pp. 129–136, doi: 10.5220/0010993600003203.

[11] D. Gerónimo, J. Serrat, A. M. Lopez, and R. Baldrich, "Traffic sign recognition for computer vision project-based learning," IEEE Trans. Educ., vol. 56, no. 3, pp. 364–371, Aug. 2013, doi: 10.1109/TE.2013.2239997.

[12] M. Manhas, D. Sanduja, R. Vashisth and N. Aggarwal, "Automated Gate Control System using Face Mask Detection," 2022 International Mobile and Embedded Technology Conference (MECON), Noida,

India, 2022, pp. 184-188, doi: 10.1109/MECON53876.2022.9752417.

[13] K. Meena, M. Kumar, and M. Jangra, "Controlling mouse motions using eye using computer vision," in Proc. 4th Int. Conf. Intell. Comput. Control Syst. (ICICCS), Madurai, India, 2020, pp. 937–941, doi: 10.1109/ICICCS48265.2020.9121137.

[14] S. Mohanasundaram, V. Krishnan, and V. Madhubala, "Vehicle theft tracking, detecting and locking system using Open CV," in Proc. 5th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS), Coimbatore, India, 2019, pp. 1–5, doi: 10.1109/ICACCS.2019.8728460.

[15] M. Sharma, "Open-CV social distancing intelligent system," in Proc. Int. Conf. Adv. Comput. Commun. Comput. Sci. (ICACCCN), Silchar, India, 2021, pp. 1–5, doi: 10.1109/ICACCCN51052.2020.9362920.

[16] S. Rosebrock, "Object tracking from scratch — OpenCV and Python," *Pysource*, Oct. 2021. [Online]. Available: https://pysource.com/2021/10/05/object-tracking-from-scratch-opencv-and-python/

[17] HDSecurityStore, "HD SDI CCTV 1080P Security Camera Front Door Entrance Demo," *YouTube*, Jun. 19, 2013. [Online]. Available: https://www.youtube.com/watch?v=-IvBKBx0UBo. [Accessed: Apr. 27, 2026].