Project Report: AI-Powered Multi-Layered Surveillance System

Presented by -

Karan Sehgal (22BCE3939)

Vellore Institute of Technology , Vellore

1. Proposed Solution

- **Detailed Explanation of the Proposed Solution:** This project implements an end-to-end, intelligent surveillance system designed to automate the detection, tracking, and analysis of objects and behaviors in video streams. The system operates as a multi-layered pipeline that transforms raw video footage into actionable security insights.

    1. **Layer 1: Video Ingestion and Pre-processing:** The system is initialized with a robust `AvenueDatasetLoader` module that automatically discovers and categorizes video files from specified directories into training and testing sets. A `VideoProcessor` then handles the low-level tasks of reading video streams, extracting individual frames, and preparing them for analysis by converting color spaces (BGR to RGB) and resizing if necessary.
    2. **Layer 2: Object Detection:** At the core of the perception layer is a dynamic `ObjectDetector`. Each frame is fed into a deep learning model to identify and localize a wide range of objects (e.g., 'person', 'car', 'backpack'). The detector is engineered for resilience; it first attempts to load a locally stored **YOLOv3** model. If these files are not found, it seamlessly transitions to using the **YOLOv8** model, which it can download and initialize on-the-fly, ensuring the system remains operational even with missing local dependencies.
    3. **Layer 3: Object Tracking and State Management:** Detections from Layer 2 are passed to the `ObjectTracker`. This module is responsible for maintaining object persistence across frames. It assigns a unique ID to each new object and tracks its trajectory using a centroid-based algorithm. The tracker maintains a state for each object, including its current bounding box, its path history (a list of previous centroid coordinates), and a "disappeared" counter to manage objects that leave the frame or become occluded.
    4. **Layer 4: Anomaly Detection and Behavioral Analysis:** This is the system's primary intelligence layer. The `AnomalyDetector` module consumes the stateful data from the tracker to perform high-level behavioral analysis. It is not just looking at a single frame but at the history and context of each object's movement. It is specifically programmed to identify several critical anomaly types:

        - **Abandoned Objects:** Identifies portable items ('backpack', 'suitcase') that have become stationary for a predefined duration (`abandonment_time`) and are no longer in close proximity (`max_distance`) to any person.

        - **Unusual Crowd Behavior:** Detects the formation of dense crowds by calculating the average distance between all detected individuals. If the number of people exceeds a threshold and their average proximity is too close, it flags a crowd anomaly.
        - **Unauthorized or Suspicious Movement:** Analyzes the path history of individuals to detect running (by calculating speed based on pixel displacement over frames) and erratic or circling patterns (by analyzing the efficiency of movement and changes in direction).
        - **Sudden Scene Changes:** Monitors the overall motion level between consecutive frames to detect sudden bursts of activity, like a crowd dispersing in panic.
    5. **Layer 5: Visualization and Reporting:** The final output is a synthesized video stream where the original footage is augmented with rich visual information. This includes colored bounding boxes, unique object IDs, tracking paths, and prominent on-screen alerts that describe any detected anomalies in real-time. Concurrently, the system's `EnhancedLogger` generates a detailed, timestamped log of all detections and anomalies, saving them to structured JSON and summary text files for forensic analysis and reporting.

- **How It Addresses the Problem:** Standard surveillance infrastructure presents two major challenges: the inability of human operators to monitor numerous feeds effectively and the reactive nature of security responses. This system directly addresses these issues by:
    1. **Automating Vigilance:** It shifts the paradigm from manual, human-dependent monitoring to a proactive, automated system that tirelessly analyzes every frame for threats, eliminating human error and fatigue.
    2. **Providing Context-Rich Alerts:** Instead of a simple motion alert, the system provides high-level, descriptive alerts (e.g., "Abandoned backpack detected," "Person 5 running"). This allows security personnel to immediately understand the nature of the threat and respond more appropriately.
    3. **Enabling Proactive Security:** By identifying suspicious patterns *before* an incident escalates (e.g., detecting loitering or erratic movement), the system allows for early intervention.
    4. **Creating Immutable Forensic Records:** The detailed JSON and text logs provide a comprehensive, searchable record of all activity, which is invaluable for post-incident investigation, providing precise data on when and where an event occurred.
- **Innovation and Uniqueness of the Solution:** The innovation of this system is rooted in its architectural resilience, the depth of its analytical capabilities, and its modular design.
    1. **Resilient Hybrid Detection Engine:** The automatic fallback from YOLOv3 to YOLOv8 is a critical feature that ensures operational robustness. It decouples the system's functionality from the local availability of specific model files, making it more reliable and easier to deploy.

2. **Context-Aware Anomaly Detection:** The system's intelligence goes beyond simple object detection. By building the anomaly analysis on top of a persistent object tracker, it can understand the *temporal context* of actions. It knows not just that a bag is present, but that it was carried by a person and has now been stationary for 8 seconds, far from any individual. This context-awareness is what separates it from basic motion detectors.
3. **Extensible Modular Architecture:** The codebase is intentionally designed with a clear separation of concerns (`VideoProcessor`, `ObjectDetector`, `ObjectTracker`, `AnomalyDetector`, `EnhancedLogger`). This object-oriented design makes the system highly maintainable and extensible. For instance, adding a new anomaly type (e.g., "Vehicle driving on sidewalk") would simply require adding a new method to the `AnomalyDetector` class without needing to modify the detection or tracking logic.

2. Technical Approach

- **Technologies to be Used:**
  - **Programming Language: Python (3.8+)**
  - **Core Computer Vision: OpenCV (`opencv-python`)** is the cornerstone of the project, used for:
    - Video I/O (`cv2.VideoCapture`, `cv2.VideoWriter`).
    - Image processing (e.g., `cv2.cvtColor`, `cv2.resize`, `cv2.absdiff`).
    - DNN Module (`cv2.dnn.readNetFromDarknet`) for running the YOLOv3 model.
    - Visualization (`cv2.rectangle`, `cv2.putText`, `cv2.line`).
  - **Numerical Computation: NumPy** is used for all array manipulations and mathematical operations, particularly for efficient vector calculations like computing Euclidean distances between object centroids (`np.linalg.norm`).
  - **Object Detection Models:**
    - **YOLOv3:** A fast and accurate object detection model. The system uses the pre-trained `yolov3.weights` and `yolov3.cfg` files.
    - **Ultralytics YOLOv8:** A state-of-the-art model used as a fallback. The `ultralytics` library is leveraged for its ease of use and automatic model downloading capabilities.
  - **Data Structures:** The Python `collections` library, specifically `OrderedDict` and `deque`, is used for managing object histories and frame buffers efficiently.
- **Methodology and Process for Implementation:** The system's architecture is a sequential pipeline where the output of each module serves as the input for the next. The entire process is orchestrated by the `ObjectDetectionSystem` class.

- **System Initialization:** The main script instantiates the `ObjectDetectionSystem`, which in turn initializes the `AvenueDatasetLoader` to locate all video files.
- **Video Iteration:** The system processes one video at a time. For each video, a `VideoProcessor` is created, and an output `VideoWriter` is prepared.
- **Frame-by-Frame Processing Loop:** a. A frame is read from the video source. b. The frame is passed to `ObjectDetector.detect_objects()`. Internally, this involves creating a blob from the image, performing a forward pass through the YOLO network, and applying non-maxima suppression to filter out weak and overlapping detections. This yields a list of `detections`. c. This `detections` list is fed into `ObjectTracker.update()`. The tracker's algorithm works as follows: * It calculates a distance matrix (using `np.linalg.norm`) between the centroids of newly detected objects and the centroids of existing tracked objects. * It matches detections to existing objects by finding the pairs with the minimum Euclidean distance, ensuring no object is matched twice. * For matched pairs, it updates the object's bounding box, appends the new centroid to its path, and resets its `disappeared` counter. * Detections that could not be matched are registered as new objects with a unique ID. * Tracked objects that were not matched to any new detection have their `disappeared` counter incremented. If this counter exceeds `max_disappeared`, the object is deregistered. * The method returns an `OrderedDict` of all currently tracked `objects`. d. The `objects` dictionary is passed to `AnomalyDetector.detect_all_anomalies()`. This master function calls several sub-methods, each analyzing the tracked objects' data (paths, class names, positions) against its specific heuristics and thresholds to identify anomalies. e. The original frame, the `objects` dictionary, and the list of `anomalies` are passed to `draw_detections_with_anomalies()`. This function draws all the visual overlays. f. The annotated frame is written to the output video file and displayed on the screen. g. All relevant data is logged via the `EnhancedLogger`.
- **Finalization:** After processing all frames, the video files are closed, and the `save_log()` method is called to write the final JSON and text reports to disk.

3. Feasibility and Viability

- **Analysis of the Feasibility of the Idea:** The project is highly feasible and has been implemented as a functional prototype. Its feasibility is grounded in the use of mature, well-documented, and powerful open-source technologies. The core components—object detection with YOLO and centroid-based tracking—are standard and proven techniques in computer vision. The modular code structure confirms that the design is sound and allows for independent development and testing of each component. The inclusion of scripts to diagnose path issues (`check.py`) and download model dependencies (`download.py`) further enhances the project's practicality and ease of deployment.
- **Potential Challenges and Risks:**

- **Computational Bottlenecks:** The primary risk is performance. Real-time object detection is computationally expensive, as it requires a forward pass of a deep neural network for every single frame. Without a dedicated GPU, the system's processing speed (FPS) will be significantly lower than the video's native framerate, making true real-time analysis impossible. This is the most significant barrier to deploying the system in a live security environment.
- **Accuracy and False Positives in Anomaly Detection:** The anomaly detection logic is heuristic-based and relies on manually tuned thresholds (e.g., `running_threshold`, `crowd_threshold`). These fixed values are not universally applicable. A fast walker might be misclassified as "running," or a group of people waiting for a bus could be flagged as a "crowd." An abandoned object might be temporarily occluded, causing the tracker to lose it and fail to raise an alarm. This can lead to a high rate of false alarms or missed events.
- **Object Tracking Instability:** The centroid tracking algorithm is simple and fast but struggles in complex scenarios. In crowded scenes, when multiple objects cross paths (occlusion), the tracker can easily swap their IDs. It also lacks an appearance model, so if two similarly sized objects get close, it can confuse them based on position alone.
- **Environmental and Lighting Dependencies:** The performance of the underlying YOLO models is highly dependent on the quality of the video feed. Poor lighting, adverse weather (rain, snow), camera glare, or deep shadows can significantly degrade detection accuracy, leading to missed objects and, consequently, failed tracking and anomaly detection.

- **Strategies for Overcoming These Challenges:**
  - **Performance Enhancement:**
    - **GPU Acceleration:** The most effective solution is to offload computation to a GPU. This can be achieved in OpenCV by setting the preferable backend and target: `net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)` and `net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)`.
    - **Frame Subsampling:** Instead of processing every frame, the system can be modified to analyze every Nth frame (e.g., every 2nd or 3rd frame). This reduces the computational load proportionally and can often be done without significant loss of tracking accuracy for slower-moving objects.
    - **Model Optimization:** Use a lighter version of the YOLO model (e.g., YOLOv8n or TinyYOLO) which offers faster inference speeds at the cost of slightly lower accuracy.
  - **Improving Anomaly Detection Accuracy:**
    - **Adaptive Thresholding:** Implement a calibration phase where the system observes a period of "normal" activity to learn baseline statistics for

movement speed and crowd density in a specific scene. These learned statistics can then be used to set dynamic, scene-specific thresholds.

- **State Machines:** For more complex anomalies, implement finite-state machines. For example, an object's state could transition from 'carried' -> 'stationary' -> 'abandoned' based on a sequence of conditions, making the detection logic more robust than a simple timer.

- **Enhancing Tracking Robustness:**
  - **Kalman Filtering:** Integrate a Kalman filter into the tracker to predict an object's position in the next frame. This makes the tracker more resilient to short-term occlusions, as it can continue to estimate the object's location even when it's not detected.
  - **Advanced Tracking Algorithms:** Replace the centroid tracker with a more sophisticated algorithm like **DeepSORT**, which combines motion prediction with a deep learning-based appearance model. This allows the tracker to re-identify an object after a long occlusion based on its visual features, significantly reducing ID switches.
- **Addressing Environmental Challenges:**
  - **Image Pre-processing:** Before detection, apply image enhancement techniques like histogram equalization or gamma correction to improve contrast in poorly lit scenes.
  - **Model Fine-Tuning:** For a fixed-camera deployment, fine-tune the YOLO model on a custom dataset captured from that specific camera under various conditions (day, night, rain). This will make the detector highly specialized and robust for that particular environment.

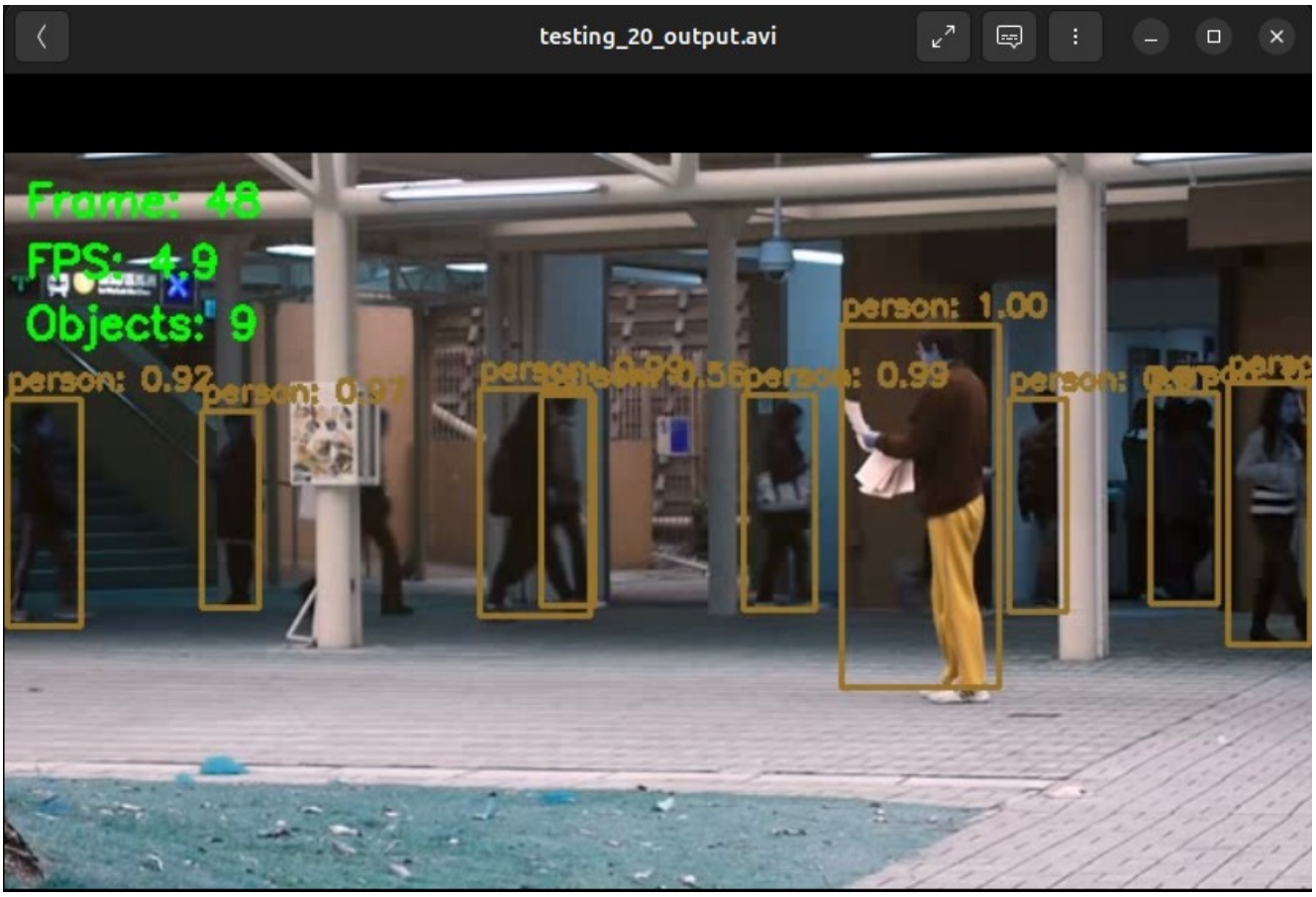
4. Research and References

- **Object Detection (YOLO Framework):**
  - Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). "You Only Look Once: Unified, Real-Time Object Detection." *CVPR*.
    - **Link:** https://arxiv.org/abs/1506.02640
  - Jocher, G., et al. (2023). "YOLOv8 by Ultralytics."
    - **Link:** https://github.com/ultralytics/ultralytics
- **Object Tracking Methodology:**
  - The implemented centroid tracking method is a foundational algorithm. A comprehensive practical guide is provided by A. Rosebrock.
    - **Link:** https://pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/
- **Dataset for Anomaly Detection:**

    - Lu, C., Shi, J., & Jia, J. (2013). "Abnormal Event Detection at 150 FPS in MATLAB." *ICCV*. The paper introduces the Avenue Dataset used for development and testing.
        - **Link:** http://www.cse.cuhk.edu.hk/leojia/projects/detectevents/dataset.html
- **Core Libraries and Tools:**
    - **OpenCV (Open Source Computer Vision Library):** The primary library for all vision-related tasks.
        - **Link:** https://opencv.org/
    - **NumPy (Numerical Python):** The fundamental package for scientific computing with Python.
        - **Link:** https://numpy.org/

Image samples -

1) Object Detection



2) Object Tracking

3) Anomaly detections

Important Links -

https://github.com/Karansehgal0611/surveillanceSystem *(Github repository including codes and documentation)*

https://karansehgal0611.github.io/surveillanceSystem/ *- Modules wise documentation (made from docstrings)*