Advanced Deep Learning
CT3-Mini Project
Inference Document
Intelligent Object Counting and Speed Monitoring System using YOLOv8

Name:S.Vanitha Reg.No. PA2312049010032

Aim:

To develop a real-time system for detecting, tracking, counting, and analysing object movement in surveillance footage using YOLOv8. The system should be capable of performing region-based counting and estimating object speed, and it is, deployed as a Flask-based web app containerized with Docker.

Procedure:

We can use the dataset that is publicly available like MOT20,VisDrone or custom surveillance/CCTV video. The focus is on objects such as pedestrians, vehicles, or bicycles in outdoor or indoor environments. In our system, we have used a publicly available outdoor video streams to detect the different types of vehicles, person, speed estimation and counting etc.

The application is designed to process the uploaded videos, detect and track objects, count the objects globally or per region, and estimate speeds. The final output is an annotated video, along with statistics related to object counts and speed.

Model Development:

The main components of this application are Yolov8, Flask, Docker. All these components are discussed in detail as follows:

1.**Yolov8:**

Here we use utilize the yolov8 pretrained model to develop this application. **YOLOv8** (You Only Look Once version 8) is the latest iteration of the **YOLO** (You Only Look Once) object detection framework, which has significantly advanced the state of real-time object detection.

YOLOv8 builds on the previous versions of YOLO by improving the model's accuracy, speed, and usability, particularly for real-time applications where speed is critical.

YOLOv8 can be used in a variety of real-world applications where fast and accurate object detection is crucial. Some common use cases include:

Autonomous Vehicles:

YOLOv8 can be used in self-driving cars to detect pedestrians, other vehicles, road signs, and obstacles, ensuring safety and proper navigation.

Security and Surveillance:

It can be used to detect intruders, monitor behaviour, or recognize specific individuals or objects.

Industrial Automation:

In manufacturing, YOLOv8 can help in quality control by identifying defective products or detecting anomalies in assembly lines.

Medical Imaging:

YOLOv8 can be applied to medical fields to detect abnormalities in X-rays, MRI scans, or other medical images, helping doctors with faster diagnosis.

Robotics:

YOLOv8 can be used in robotics for object manipulation, navigation, and environment mapping by detecting and tracking objects within the robot's workspace.

Retail and Inventory Management:

YOLOv8 can assist in monitoring shelves and inventory, detecting when stock levels are low or identifying misplaced items.

Agriculture:
In precision farming, YOLOv8 can be used to detect diseases, pests, and weeds in crops or to monitor livestock.

Augmented Reality (AR):
YOLOv8 can power object detection in AR applications, where real-time object recognition and interaction with virtual objects are required.

The **YOLOv8** model is designed to perform object detection, which involves identifying and classifying objects in images or video streams.

Working Principle:

Single Neural Network: YOLOv8, like its predecessors, uses a single convolutional neural network (CNN) that performs both object localization (bounding box prediction) and object classification in one go. This is different from earlier detection methods, which typically used separate stages for classification and localization.

Grid Division: The input image is divided into a grid, and each grid cell is responsible for predicting bounding boxes and class probabilities for objects that appear within that cell. YOLOv8 refines this process to improve detection accuracy and reduce errors, especially in crowded or complex scenes.

Bounding Box Prediction: Each grid cell predicts multiple bounding boxes with confidence scores (indicating the probability that an object is present in the box). YOLOv8 further refines these predictions using non-maximum suppression (NMS) to eliminate overlapping boxes and provide a final, accurate set of detections.

Anchor Boxes: YOLOv8 uses anchor boxes to predict bounding boxes more accurately. The anchor boxes help the model focus on detecting objects of different sizes and shapes.

Feature Pyramid Networks (FPN): YOLOv8 incorporates advanced techniques such as FPN, which improves its ability to detect objects at multiple scales. This is important for detecting both large and small objects effectively.

Backbone Network: YOLOv8 uses a deep neural network backbone (like **CSPDarkNet**), which extracts rich features from the input image to improve detection performance.

Prediction: After processing the image, YOLOv8 generates output in the form of class labels, confidence scores, and bounding box coordinates for each detected object.

Advantages of YOLOv8:
Speed, High Accuracy, Real-Time Detection, Compact and Efficient, Flexible, End-to-End Pipeline, Improved Generalization.

This **YOLOv8 Object Detection and Tracking Flask Web Application** provides a simple interface to upload videos, perform object detection, track objects across frames, count objects, and estimate speeds.

The results are presented with annotated bounding boxes, tracking IDs, and other statistical information.

YOLOv8 is a powerful and efficient object detection framework that balances speed and accuracy. Its ability to detect multiple objects in real-time, combined with improvements in accuracy and model efficiency, makes it an ideal choice for a wide range of applications. Whether you're developing self-driving cars, building a surveillance system, or working in healthcare, YOLOv8 provides a reliable solution for real-time object detection and tracking.

By integrating YOLOv8 into our projects, we can leverage its high-performance capabilities to solve complex detection problems efficiently.

2**.Flask web Interface:**
Flask is used for developing web applications using **python**, implemented on Werkzeug and Jinja2.

Advantages of using Flask framework are:

There is a built-in development server and a fast debugger provided.

- Lightweight
- Support Secure cookies
- Lightweight – Minimal core framework with modular extensions.
- Flexible Routing – Define routes with ease using decorators.
- Built-in Development Server – Includes debugging and error handling.
- Jinja2 Templating – Supports dynamic HTML generation.
- Extensible – Extensions available for authentication, database integration, and more.
- RESTful Support – Ideal for building APIs.
- Integrated Unit Testing – Helps in testing applications easily.

Flask is a powerful yet lightweight framework for developing web applications. Its flexibility makes it suitable projects, APIs, and even complex applications when integrated with extensions.

3. Docker:

- **Lightweight** – Minimal core framework with modular extensions.
- **Flexible Routing** – Define routes with ease using decorators.
- **Built-in Development Server** – Includes debugging and error handling.
- **Jinja2 Templating** – Supports dynamic HTML generation.
- **sExtensible** – Extensions available for authentication, database integration, and more.
- **RESTful Support** – Ideal for building APIs.
- **Integrated Unit Testing** – Helps in testing applications easily.

Advantages of Docker:

- Portability: Works seamlessly across different environments.
- Efficiency: Uses fewer resources compared to virtual machines.
- Scalability: Easily scalable using orchestration tools like Kubernetes.
- Isolation: Prevents conflicts between dependencies.
- Version Control: Simplifies application updates and rollbacks.

Docker Architecture:

Docker consists of the following core components:

Docker Client – Command-line tool to interact with Docker.
Docker Engine – Core service that runs containers.
Docker Registry – Stores Docker images (e.g., Docker Hub).
Docker Containers – Running instances of Docker images.

We will discuss about the implementation in detail in the following sections:

Procedure:

Step 1: Import all the important python libraries which is need for our application development.

import os - Provides functions to interact with the operating system (e.g., reading file paths, creating folders, etc.).

import cv2 - OpenCV library: Used for video processing, reading/writing frames, drawing bounding boxes, and manipulating video content.

import uuid - Used to generate unique identifiers (e.g., unique filenames for uploaded videos or processed files).

import shutil - Provides high-level file operations like copying or deleting directories/files.

from flask import Flask, render_template, request, redirect, url_for, send_from_directory.

Flask imports:

- **Flask:** The main Flask app class.
- render_template: Renders HTML templates (e.g., displaying the upload page).
- request: Accesses form data and uploaded files.
- redirect, url_for: For navigation between routes.
- send_from_directory: Serves files (e.g., processed video downloads).

from ultralytics import YOLO

- Imports the YOLOv8 class from the **Ultralytics** package. This is the core deep learning model used for object detection and tracking.

This import block is typically followed by:

- **Flask app initialization**
- **Route definitions for uploading, processing, and displaying videos**
- **YOLOv8 model loading**
- **Video processing logic (e.g., drawing boxes, counting, tracking)**

app = Flask(__name__)

This creates a new Flask web application instance.
__name__ is passed to let Flask determine the root path of the app (used for finding templates and static files).
app is the object through which you define routes and run the server.

model = YOLO("yolov8n.pt") # Choose any YOLOv8 variant.

This line **loads the YOLOv8 model** (specifically the **YOLOv8n** "nano" version).
yolov8n.pt is a pre-trained model file.

def process_video(filepath, output_path):
A function named process_video is declared.
Parameters:

- filepath: Path to the input video.
- output_path: Path where the processed video should be saved.

cap = cv2.VideoCapture(filepath)
Opens the video file for reading.
cap is a video capture object used to read frames one by one.

fourcc = cv2.VideoWriter_fourcc(*'mp4v')

cv2.VideoWriter_fourcc is a function in OpenCV that specifies the **codec** used to compress the frames of the output video.
*.'mp4v' unpacks the string into individual characters ('m', 'p', '4', 'v')—this is how OpenCV expects codec identifiers. 'mp4v' is a common codec for .mp4 video files (MPEG-4 format).
out = cv2.VideoWriter(output_path, fourcc, fps, (width, height))

This line creates a VideoWriter object named out that will be used to **write frames** to a video file.
results = model.track(frame, persist=True, tracker="bytetrack.yaml")

1. model.track(...)

- This method performs **object detection + tracking** in a single call.
- It runs YOLOv8 detection and links objects across frames using a tracking algorithm (e.g. ByteTrack).

2. frame:

- The input video frame (a NumPy array) on which detection and tracking will be performed.

3. persist=True

- Tells YOLO to **retain the same tracking IDs** across frames.
- This is essential for **multi-frame tracking** — it keeps object IDs consistent so you can count or analyze movement.

4. tracker="bytetrack.yaml"

- Specifies the use of the **ByteTrack** tracking algorithm.
- The YAML file contains config settings for the tracker (e.g., detection thresholds, buffer size).
- ByteTrack is a strong multi-object tracker, especially good for crowded scenes.

ids = results[0].boxes.id.cpu().numpy().astype(int)

Extracts the **tracking IDs** of detected objects from the YOLOv8 + ByteTrack results.

boxes = results[0].boxes.xyxy.cpu().numpy()

Gets the bounding box coordinates for each detected object in the form [x1, y1, x2, y2]:

- x1, y1: top-left corner
- x2, y2: bottom-right corner

for i, box in enumerate(boxes):

Loops over all detected bounding boxes.

x1, y1, x2, y2 = box.astype(int)

Extracts integer coordinates from each box.

obj_id = ids[i]

Retrieves the tracking ID of the current object (indexed to match the box).

cx, cy = int((x1 + x2)/2), int((y1 + y2)/2)

Computes the center point (cx, cy) of the bounding box.
Useful for region-based counting or path tracking.

counters['total'].add(obj_id)

Adds the current object's ID to a set that stores unique IDs.
counters['total'] is presumably a set() tracking all object IDs seen across all frames.
Because sets don't allow duplicates, this gives us the total count of unique objects.

if len(object_positions[obj_id]) > 1:

Checks if there's more than one recorded position for the object with obj_id.

object_positions is assumed to be a dictionary like:

```
object_positions = {
   obj_id1: [(x1, y1), (x2, y2), ...],
   obj_id2: [(x1, y1), ...],
   ...
}
```

dx = object_positions[obj_id][-1][0] - object_positions[obj_id][-2][0]
dy = object_positions[obj_id][-1][1] - object_positions[obj_id][-2][1]

- Calculates the difference in the x and y coordinates between the last two recorded positions of the object.
- This gives the pixel displacement along each axis.

speed = np.sqrt(dx2 + dy**2)**

- Applies the Euclidean distance formula to calculate the total pixel displacement (a simple approximation of speed).

This gives you the speed in pixels per frame.
It's a basic and relative speed estimate, useful for visual feedback (e.g., drawing speed on a video).
We can convert it to real-world units (e.g., meters/second) if you know:

- Real-world scale (pixels per meter)
- Frame rate (frames per second)

Draw a bounding box:
cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)

Draws a green rectangle ((0, 255, 0)) around the detected object.
(x1, y1) = top-left corner of the box
(x2, y2) = bottom-right corner
2 is the line thickness.

Display ID and speed as text:

cv2.putText(frame, f'ID: {obj_id} Spd:{speed:.1f}', (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 1)

Places a label above the bounding box.
Displays the tracking ID (obj_id) and speed, formatted to one decimal place.
(x1, y1 - 10) = position just above the box

cv2.FONT_HERSHEY_SIMPLEX = font style
0.5 = font scale (small)
(255, 0, 0) = text color (blue)
1 = line thickness

This is better for:

- **Visual tracking feedback**
- **Debugging speed estimates**
- **Creating rich annotations in surveillance, traffic, or pedestrian analysis systems.**

Use of HTML in this Application:

HTML allows users to upload video files (e.g., through a form) which will then be processed by the backend model.

After processing, HTML is used to show annotated video, object counts, and speed data to the user. Forms and buttons in HTML send requests (e.g. POST) to the Flask backend to start processing, download outputs, etc.

We can use HTML (along with CSS/JS) to show logs, statistics, and real-time updates like number of objects, speeds, or alerts.

HTML <video> tags can be used to display processed videos directly in the browser.
HTML is used to build the front-end interface that allows users to interact with the machine learning system easily — upload videos, start analysis, and view the output (annotated video and statistics).

Flask Code:

```
if __name__ == '__main__':
    app.run(debug=True)
```

It is a common Python pattern used to start your Flask web application when the script is run directly.

if __name__ == '__main__':

- This checks whether the script is being run directly (not imported as a module).
- If True, it runs the following code block — in this case, starts the Flask server.
- Prevents the app from unintentionally running if the script is imported elsewhere.

app.run(debug=True)

- Starts the Flask development server on localhost (127.0.0.1:5000) by default.
- debug=True enables:
  - Auto-reloading: Restarts the server when code changes.
  - Detailed error messages in the browser for easier debugging.

Dockerfile is used to containerize your Flask + YOLOv8 application.

FROM python:3.10-slim

Uses a lightweight Python 3.10 base image.
The -slim tag reduces image size by removing unnecessary build tools and files.

WORKDIR /app

- Sets the working directory inside the container to /app.
- All subsequent commands (like COPY or CMD) will run relative to this directory.

COPY . /app

Copies everything in your project folder (on the host machine) into the /app directory in the container.

RUN pip install --no-cache-dir ultralytics opencv-python Flask

Installs required Python packages:

- ultralytics → YOLOv8
- opencv-python → for video processing
- Flask → web framework

--no-cache-dir helps reduce Docker image size.

EXPOSE 5000

Informs Docker that the container will listen on port 5000 (default Flask port).

CMD ["flask", "run", "--host=0.0.0.0"]

This starts your Flask application and makes it accessible from any network interface (useful when running in Docker).

Important: Flask needs to know which app to run. We must set FLASK_APP environment variable or use app.py as our main script.

Now our Flask web interface (with YOLOv8 video processing) should be accessible at:
http://localhost:5000 **or** http://127.0.0.1:5000

Summary:
Launch the Flask Application
Running Using Docker
Build the Docker image:
Run the Docker container

Upload a Video for Object Detection and Tracking

The server will start running YOLOv8 inference on the video. The video will be processed frame by frame, with the following tasks being performed:

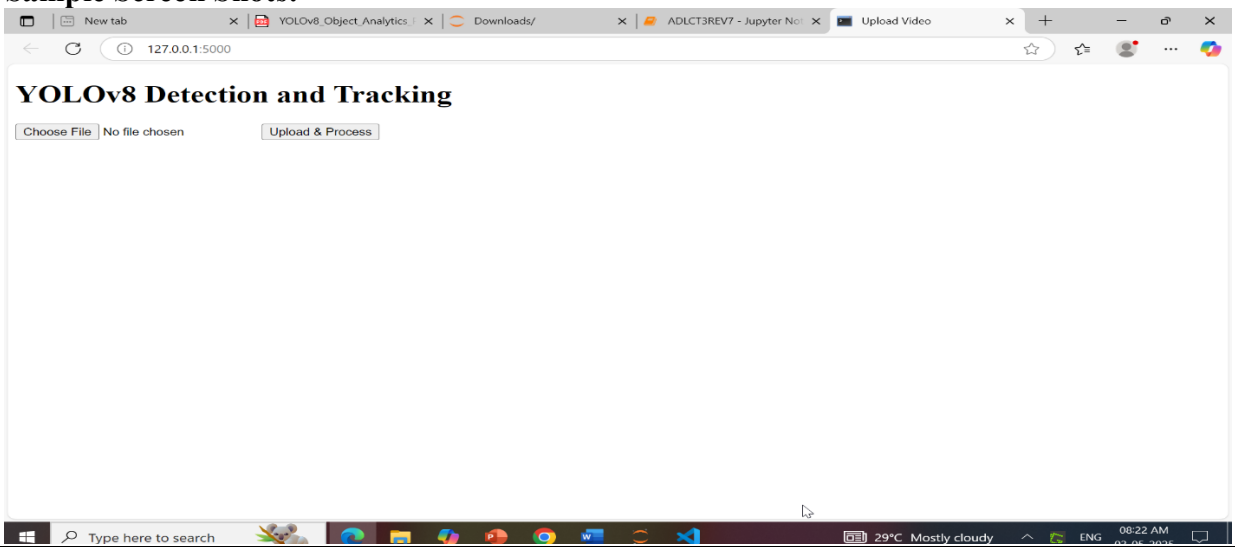
- **Object Detection**: Identifying objects within each frame.
- **Object Tracking**: Tracking objects across frames.
- **Object Counting**: Counting the total number of unique objects and objects per region (entry, exit, restricted areas).
- **Speed Estimation**: Calculating object speeds based on pixel displacement over time.

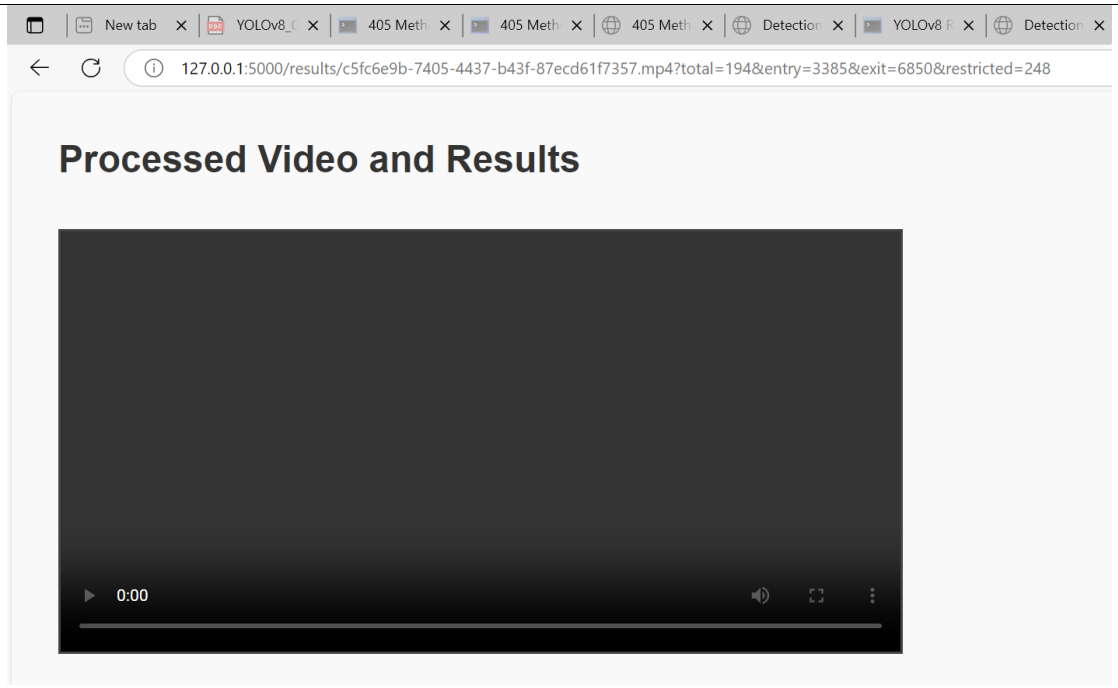
The output is available in this URL:
http://localhost:5000 **or** http://127.0.0.1:5000

Output:
Output Video:

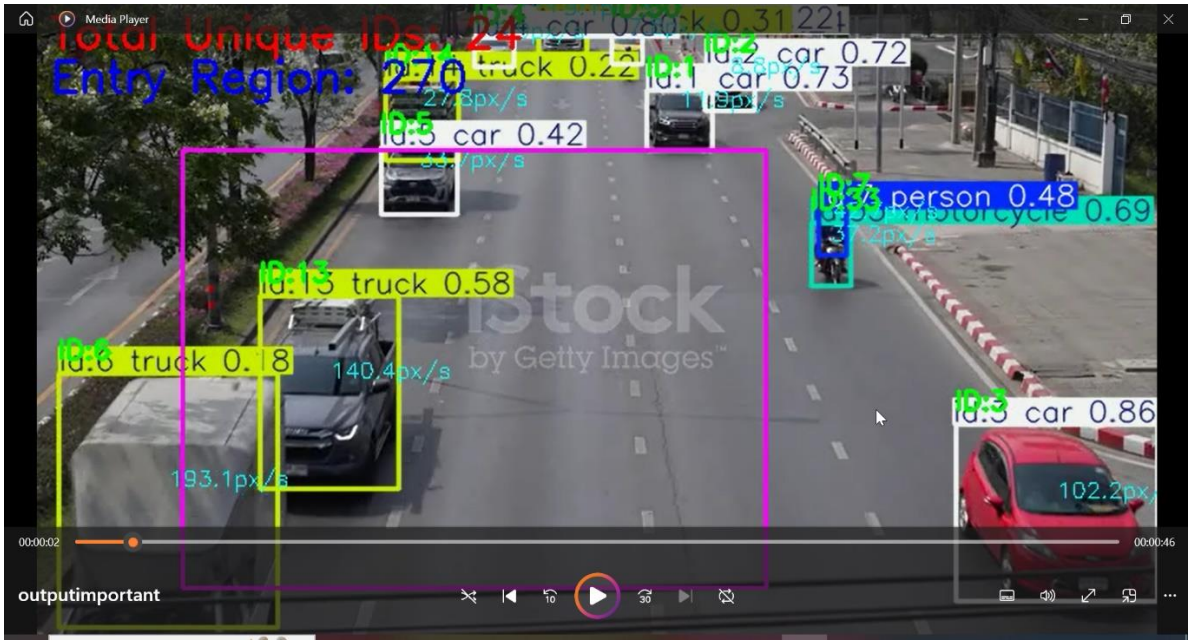
output.mp4

Sample Screen Shots:





Sample screenshot from the video:



Conclusion:

Thus the, “Intelligent Object Detection, Counting and Speed Monitoring System using YOLOv8 has been developed and implemented and verified successfully.