ChatGPT

Object Detection Pipeline for Video (Hospitals, Malls, Schools)

Video analytics for specific environments (hospitals, shopping malls, schools) can be built by running a deep learning object detector on each video frame to extract detected objects and their attributes. In practice, one processes the pre-recorded video with Python (e.g. using OpenCV) frame-by-frame, optionally resizing frames to balance speed and accuracy [1]. Modern detectors like YOLO (You Only Look Once) are often used because they can run in real time: for example, YOLOv3 processes images at ~30 FPS on a Titan X GPU [2] and is far faster than older R-CNNs [3]. Recent YOLO versions (e.g. YOLOv8) use an anchor-free architecture and optimized backbone to improve speed–accuracy trade-off [4]. To detect environment-specific objects (e.g. hospital beds or retail shelves), it's common to start with a COCO-pretrained model and **fine-tune** it on a smaller custom dataset of relevant classes [5]. For instance, a specialized Hospital Indoor Object Detection (HIOD) dataset (4,417 images, 56 categories) was created so that a model trained on it could accurately detect and classify common hospital objects [6]. Likewise, one could gather (or use existing) images of school and mall scenes, annotate the key objects, and fine-tune a pre-trained detector on those.

- **Step 1 – Frame Extraction:** Use OpenCV's `VideoCapture` (or similar) to read the video and extract frames at the desired rate. For example, reading 1 frame per second or per video-frame yields images for detection [7].
- **Step 2 – Preprocess / Resize:** Downscale or crop frames if needed for efficiency. For example, feeding YOLO a 640×640 pixel image is common; reducing resolution speeds up inference, although very low resolution can hurt small-object accuracy [1]. (Researchers observed that increasing YOLO's input from 224×224 to 448×448 raised mAP by ~4% [1], illustrating the speed–accuracy tradeoff.)
- **Step 3 – Object Detection:** Run each frame through a deep detection model. One-stage detectors like **YOLO** (v3–v8), **SSD**, or **EfficientDet** are popular for speed; two-stage detectors (Faster R-CNN, Mask R-CNN) are generally more accurate but slower. YOLOv8 (2023) in particular is designed for real-time detection with improved accuracy–speed balance [4]. In Python, the Ultralytics YOLOv8 library allows loading a pre-trained model (e.g. `yolov8n.pt`) and performing inference with simple commands [8]. As a baseline, a model pre-trained on COCO can detect general objects (persons, furniture, vehicles, etc.) [2] [9].
- **Step 4 – Extract Object Data:** For each frame, collect the model's detections. Typical outputs include: the **class label**, the **bounding box coordinates** (x, y, width, height), and a **confidence score** for each detected object [10] [7]. For example, PyTorch's object detection outputs a `boxes` tensor of shape [N,4] (coordinates) and a `labels` tensor of length N [10]. Google's Vertex AI docs similarly define a bounding box by its `[y_min,x_min,y_max,x_max]` coordinates and an associated `label` [11]. (Confidence scores are typically between 0–1, indicating model certainty.) One can easily store these properties (e.g. in JSON or CSV) for later analysis.
- **Step 5 – (Optional) Tracking:** In videos, you may wish to link detections across frames. Simple Online Realtime Tracking (SORT) or DeepSORT can take the frame-by-frame detections and assign persistent IDs, producing object trajectories [12]. This is not strictly required for per-frame analytics, but it enables counting how objects move or remain in scene.

Models and Training

A practical approach is to use a *pre-trained* object detector and fine-tune for domain-specific classes. For example, one might load a YOLOv8 model pre-trained on COCO (which knows ~80 common classes) and then **retrain** it on a smaller custom dataset of hospital/mall/school objects. Frameworks like PyTorch and its TorchVision provide tutorials for this: e.g. fine-tuning a Mask R-CNN on a custom dataset of pedestrians [5]. The training dataset should supply images with labeled bounding boxes and class IDs; PyTorch expects each training sample to include `boxes` (an [N,4] array of coordinates) and `labels` (an [N] array of class IDs) [10].

Using pre-trained weights dramatically reduces training time. In one study, starting from ImageNet/COCO weights led to much better results on a hospital dataset [6] than training from scratch. Similarly, if you collect even a few hundred labeled frames from your target environment, you can fine-tune a COCO-pretrained model to recognize specialized objects (e.g. medical equipment, campus signboards, etc.). Fine-tuning typically means lower learning rates and a smaller number of epochs, leveraging the general features already learned. Many tutorials and tools (Detectron2, YOLOv8, HuggingFace, etc.) support training on custom classes.

Detected Object Properties

For each detected object in the video, the key properties to record are:
- **Class label:** the object category (e.g. “person”, “car”, “bed”, “chair”, etc.).
- **Bounding box:** the object’s 2D location, given by coordinates (x_min, y_min, x_max, y_max) or (x, y, width, height) [10] [11].
- **Confidence score:** a probability or score for the detection, indicating how likely the box contains the object of that class [7].
- **(Optional) Other attributes:** from the cropped image within the box you can compute color histograms, aspect ratio, or pass the crop to an attribute classifier. If instance segmentation is needed, models like Mask R-CNN can add a **mask** for each object (pixel region), but this is more compute-intensive.

These properties can be output as structured data. For example, the Azure Video Indexer (via frame-level API) returns each object with its bounding rectangle and confidence [7]. With Python/OpenCV one could directly append detections to a list or file per frame. By aggregating this across frames, you obtain a dataset of “what objects appeared when and where,” which can support counts, presence statistics, or higher-level analysis later.

Practical Considerations

- **Resolution and Speed:** Larger input images improve detection accuracy, especially on small objects [1], but slow down processing. Choose a balanced resolution (e.g. around 640×360 up to 1280×720) that your hardware can handle in reasonable time. YOLOv8, for instance, uses 640×640 by default and offers “nano”/“small” variants for higher speed.
- **Hardware:** For video (many frames), use a GPU if possible. PyTorch and YOLO libraries will automatically use CUDA if available. This can turn a 10-second-per-image CPU detector into real-time speeds (tens of FPS) [2].

- **Library Choices:** In Python, common tools are OpenCV for video I/O, PyTorch or TensorFlow for model inference, and specialized APIs like Ultralytics YOLO or Detectron2 for ease of use. Ultralytics provides a simple API (e.g. `model = YOLO('yolov8n.pt'); results = model(frame)`) to run detection [8].
- **Environment Data:** If you have metadata (e.g. you know which video is hospital vs school), you could use a scene classification model to filter frames by scene type before detection. Otherwise, simply running one detector on all frames will still identify objects regardless of setting.

By following this approach, you can automate extraction of object-level data from video. The end result is, for each video frame, a list of detected objects with their types and coordinates. This data can then be analyzed or stored for whatever application (monitoring occupancy, inventory, safety, etc.) you need.

Sources: Modern object detection frameworks (e.g. YOLO) process full images at real-time speeds [2], and using pre-trained models greatly accelerates development [5]. Domain-specific datasets (like the HIOD for hospitals) demonstrate that training on relevant data yields accurate detections [6]. Standard APIs expect and output bounding-box coordinates and labels [10] [11], which we use as the primary "properties" of each object. Tracking algorithms (SORT/DeepSORT) can optionally maintain identities across frames [12]. All these elements together form a comprehensive pipeline for extracting objects and their attributes from video.

---

[1] YOLO Object Detection Explained: A Beginner's Guide | DataCamp
https://www.datacamp.com/blog/yolo-object-detection-explained

[2] [3] YOLO: Real-Time Object Detection
https://pjreddie.com/darknet/yolo/

[4] [8] Explore Ultralytics YOLOv8 - Ultralytics YOLO Docs
https://docs.ultralytics.com/models/yolov8/

[5] [10] TorchVision Object Detection Finetuning Tutorial — PyTorch Tutorials 2.10.0+cu130 documentation
https://docs.pytorch.org/tutorials/intermediate/torchvision_tutorial.html

[6] GitHub - Wangmmstar/Hospital_Scene_Data: The hosptal scene data wiith labels
https://github.com/Wangmmstar/Hospital_Scene_Data

[7] Help with Object Detection and Bounding Box Extraction Using Azure Content Understanding - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/2265403/help-with-object-detection-and-bounding-box-extrac

[9] YOLO object detection with OpenCV - PyImageSearch
https://pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/

[11] Bounding box detection | Generative AI on Vertex AI | Google Cloud Documentation
https://docs.cloud.google.com/vertex-ai/generative-ai/docs/bounding-box-detection

[12] SORT Explained: Real-Time Object Tracking in Python
https://blog.roboflow.com/sort-explained-real-time-object-tracking-in-python/