Project Report: YOLOv8 Live Webcam Object Detector

1. Cover Page

Field	Value
Project Title	YOLOv8 Live Webcam Object Detector
Course Name	Introduction to Problem Solving and Programming
Name	Yash Samadhan Jadhav
Registration No.	25BAI11472
Submission Date	24/11/2025

2. Introduction

The field of Computer Vision has advanced rapidly, driven by sophisticated deep learning models. Object detection, in particular, is a critical application. This project addresses the challenge of deploying a state-of-the-art object detection model, YOLOv8n, in a restrictive, remote computing environment: Google Colab. Standard real-time applications using local webcams fail in such environments. This project's core contribution is the development of a stable, custom bridge using Python's ipywidgets and injected JavaScript to capture, process, and display live webcam frames effectively, resulting in a robust and non-blinking demonstration of real-time AI.

3. Problem Statement

Developing real-time computer vision applications is a core skill in AI/ML engineering. However, theexecution environment often presents challenges. Specifically, when working within remote notebookenvironments like Google Colab, standard webcam access (cv2.VideoCapture(0)) and display methods(cv2.imshow())fail, hindering practical application and demonstration of model performance. Theproblem is to create a robust, fully self-contained, and functional real-time object detection application using a state-of-the-art model(YOLOv8n) that successfully navigates the technical limitations ofremote execution environments (i.e., accessing the user's local webcam via the browser) whilemaintaining a stable, non-blinking visual output.4. Functional Requirements (FR)This project is structured around three major functional modules:

FR No.	Module Name	Description
FR1	**Model Initialization**	The system must successfully load the pre-trained **YOLOv8n** model from the Ultralytics library upon execution.
FR2	**Frame Capture Bridge**	The system must implement a reliable, non-blocking bridge using embedded JavaScript and `google.colab.output.eval_js` to capture a frame from the user's local webcam and return it to the Python kernel as an OpenCV/NumPy array.
FR3	**Real-Time Inference Loop**	The system must run a continuous loop that takes the captured frame, performs YOLOv8 inference, annotates the image with bounding boxes and labels, and updates the display widget in place.
FR4	**Controlled Termination**	A dedicated user interface element (STOP button) must be implemented to cleanly and safely exit the inference loop, releasing all resources.

5. Non-functional Requirements (NFR)

NFR No.	Requirement	Description	Strategy Implemented
NFR1	**Usability**	The interface must provide clear user feedback and a single, obvious control for stopping the process.	Use `ipywidgets` for a dedicated "STOP Detection" button and an `HTML` widget for color-coded status messages (Orange=Capturing, Green=Processing).
NFR2	**Reliability**	The application must handle the high-latency camera capture process without producing corrupt (black) frames.	A `time.sleep(0.5)` delay is introduced between captures to allow the camera sensor sufficient time for initialization and exposure adjustment.
NFR3	**Performance**	The model chosen must ensure low-latency inference suitable for near-real-time applications.	Selection of the **YOLOv8n** (Nano) model, which is optimized for speed over absolute accuracy, running on the Colab's standard GPU/CPU runtime.
NFR4	**Maintainability**	The codebase must be modular, separating the Colab-specific webcam utilities from the main detection logic.	Functions are clearly separated (`get_webcam_frame` for JS/Colab interaction and `run_detector_from_webcam` for the main loop).

6. Design Diagrams

Use Case Diagram

Actor	Use Case	Description

User	Start Detection	Executes the script to initialize the application and begin the loop.
User	Grant Camera Access	Interacts with the browser security prompt to provide camera stream access.
User	Stop Detection	Clicks the dedicated button to cleanly terminate the process.
System	Capture Frame	Executes JavaScript to take a snapshot from the live stream.
System	Perform Inference	Runs the YOLOv8n model on the captured frame.
System	Display Results	Updates the `ipywidgets.Image` with the annotated frame.

Workflow Diagram (Process Flow)

The application follows a continuous polling loop, triggered and stopped by the user.

1. **Start:** User executes `run_detector_from_webcam()` .
2. **Initialization:** Widgets (Button, Status Label, Image) are displayed.
3. **Loop Condition:** Check `stop_detection` flag.
    - **IF True:** Go to **End**.
    - **IF False:** Continue.
4. **Capture Frame (JS):** Python calls `eval_js` , which runs JavaScript to open camera, capture photo (300ms delay), and return Base64 data.
5. **Process Frame (Python):** Base64 is decoded to NumPy array.
6. **Inference:** `model.predict()` is executed.
7. **Display:** Annotated frame is encoded to JPEG bytes and updates the `ipywidgets.Image` .
8. **Stabilization:** `time.sleep(0.5)` pause.
9. **Loop Back:** Return to **Loop Condition**.
10. **End:** Resources released.

Sequence Diagram

(Focusing on the critical Capture and Display sequence)

Object	Description
:User	Initiates and stops the process.
:Python Kernel	Main application logic and control flow.
:JS Bridge	Injected JavaScript code running in the browser.
:YOLOv8 Model	The object detection algorithm.
:IPy Widget	The visual display container (`frame_image`).

Class/Component Diagram

Given the minimal and sequential nature of this project, a formal Class Diagram is not required. However, the core Python components (modules/functions) and external dependencies are defined as follows:

Component Type	Component Name	Responsibility
External Library	`ultralytics.YOLO`	Loading the pre-trained model and running the inference.
External Library	`cv2` (OpenCV)	Image decoding, resizing, and encoding.
External Library	`ipywidgets`	Creating the user interface (Button, Image, HTML status).
Python Module	`realtime_detector.py`	Main entry point and control flow.
Python Function	`get_webcam_frame`	The critical bridge: calls JS, decodes base64 data.
Python Function	`run_detector_from_webcam`	The main execution loop and widget management.

ER Diagram (if storage used)

Not Applicable (N/A): This project does not use any persistent storage (database or local files). The data flow is transient: Webcam -> Memory -> YOLO Processing -> Display.

7. Design Decisions & Rationale

Design Decision	Rationale
Model Selection: YOLOv8n	Chosen for its high speed (fastest of the YOLOv8 family) and low resource consumption, making it ideal for the shared, constrained Colab environment where latency is a concern.
JS/ `ipywidgets` **for Webcam**	Standard OpenCV webcam capture fails in remote environments. The JS/Base64 bridge is the most robust workaround for accessing the browser's camera stream.
`ipywidgets.Image` **for Display**	Using an `ipywidgets.Image` object and updating its `.value` property *in place* prevents the constant output cell clearing (`clear_output`), which was the primary cause of the undesirable "blinking" effect.
Stabilization Delay (`time.sleep(0.5)` **)**	Addresses the "black frame" issue. The overhead of repeatedly opening and closing the camera stream via JS often led to frames being captured before the

sensor could properly expose. This small delay guarantees a well-lit, reliable frame capture.

8. Implementation Details

The entire solution is implemented in a single, modular Python file: realtime_detector.py(TOEFL)

Key Implementation Components:

1. JS Injection: The get_webcam_frame function employs a triple-quoted string containing asynchronous JavaScript. That code handles access to the camera, playing the video stream, drawing asingle frame to a canvas, stopping the stream, and returning the image data as a Base64 string.

2. Base64 Decoding: The returned Base64 string is decoded into raw bytes ("np.from b then processed by cv2.imdec ode to convert it into an OpenCV-compatible NumPy array.uffer)

3. UI and Loop Control: The run_detector_from_webcam function is built upon ipywidgets : The while loop is controlled by a boolean flag called stop_detection. The on_utton_clicbk handler flips this flag to terminate the process cleanly.

4. Update Display: The annotated OpenCV frame is converted back to JPEG bytes using cv2.imenc encoded in the ode, these bytes are assigned directly to the frame_image.value, making sure thatsmooth in-place update.10. Screenshots / Results

Figure	Description
Figure 1: Initial State	Screenshot showing the Colab output cell with the "STOP Detection" button and status label before the first frame is captured.
Figure 2: Detection Result	Screenshot of a typical frame showing the webcam feed with YOLOv8 bounding boxes and class labels correctly drawn on detected objects (e.g., person, chair, laptop).
Figure 3: Termination	Screenshot of the output cell after the STOP button is clicked, showing the "Detection Stopped" button and the cleanup messages.

10. Testing Approach

The project utilized an **Iterative Validation** approach, focused heavily on overcoming environmental constraints.

Test Case	Objective	Expected Result	Actual Result	Status
TC1: Model Load	Verify `yolov8n.pt` loads without network or file errors.	Model object is initialized successfully.	Success.	Pass
TC2: Camera Access	Verify the browser successfully prompts for and grants camera access.	The `get_webcam_frame` function returns a valid NumPy array.	Success.	Pass

TC3: Black Frame Fix	Verify the 300ms JS delay and 0.5s Python sleep prevent underexposed (black) frames.	All displayed frames are well-lit and clearly visible.	Success.	Pass
TC4: No Blinking	Verify the use of `ipywidgets.Image` prevents the entire output cell from blinking.	Only the image widget updates; the button and status text remain static.	Success.	Pass
TC5: Graceful Exit	Verify clicking the "STOP Detection" button terminates the loop and executes the `finally` block.	Application prints "Cleanup complete." and exits without errors.	Success.	Pass
TC6: Inference Integrity	Verify the YOLOv8 model correctly identifies and annotates common objects (e.g., person, keyboard).	Bounding boxes are accurately drawn and labeled on the detected objects.	Success.	Pass

11. Learnings & Key Takeaways

- **Environmental Constraints: The project showed that, in general, successful deployment depends more on overcoming environmental I/O limitations - such as remote webcam access - than complex model architecture.**
- **The power of ipywidgets: Learned how to leverage interactive widgets not just for aesthetic controls but as an important tool in creating stable, low-latency display updates in notebook environments.**
- **Full-stack thinking in ML: Real-time ML deployment integrates three domains: Deep Learning YOLOv8, Systems/OS interaction via JS for webcam access, and Frontend UI via ipywidgets.**

12. Future Enhancements

1. **Performance Optimisation: for noncritical applications, implement frame skipping techniques (e.g., process every Nth frame), or use a larger and faster model, such as YOLOv8s, if stronger GPU runtime is available.**
2. **Tracking & State: Apply a multi-object tracking algorithm-e.g., DeepSORT-to assign unique IDs to detected objects across successive frames.**
3. **Data Logging: Add functionality to log the events of detection - timestamp, object class, and bounding box coordinates - to a local file or a cloud database like Firebase/Firestore.**

13. References

1. **Ultralytics YOLOv8 Documentation:** https://docs.ultralytics.com/
2. **OpenCV (cv2) Library:** https://opencv.org/
3. **Google Colab Utilities Documentation:** For reference on `google.colab.output.eval_js` .
4. **IPyWidgets Documentation:** For reference on interactive display elements.