Edgefleet Assignment:
Cricket Ball Detection & Tracking System

Candidate Name: Aditya Raj SIngh
Institute: IIIT-Hyderabad
Roll No. 2022102067
Date: 19th December 2025

1. Summary

This report documents the development of a computer vision pipeline designed to detect and track a cricket ball in single-camera video footage. The system was built using the **YOLOv8** architecture, chosen for its balance of real-time inference speed and accuracy on small objects.

To adhere to strict data integrity requirements, the model was trained exclusively on an external open-source dataset, ensuring no data leakage from the test videos. The final solution implements a custom inference engine that combines deep learning detections with physics-based heuristics and trajectory interpolation to handle occlusions and motion blur.

2. Problem Statement & Constraints

The objective was to generate a processed video with a trajectory overlay and a frame-by-frame annotation CSV (`frame, x, y, visibility`).

Key Challenges & Constraints:

1. **Small Object Detection:** The cricket ball occupies less than 1% of the frame pixels.
2. **Similar Objects:** White shoes, gloves, and pitch markers act as "distractors" (False Positives).
3. **Hardware Limitations:** The development and training were conducted entirely on a **CPU environment**. This necessitated a strategic approach to data sub-sampling to complete training within the assessment deadline.

3. Data Strategy

3.1 Data Acquisition (Prevention of Leakage)

As per the assessment guidelines, the test videos provided were strictly reserved for inference. For training, I acquired the **"Cricket Ball Object Detection"** dataset from Roboflow Universe.

- **Total Raw Images:** ~8,000 labeled images.

- **Classes:** `ball` (0), `stump` (1).

3.2 Data Sub-sampling

Training on 8,000 images on a CPU would require approximately 12-15 hours per 30 epochs. To iterate effectively, I wrote a utility script (`create_subset.py`) to create a randomized, stratified subset.

- **Training Set:** 1,500 Images.
- **Validation Set:** 300 Images.

This reduced epoch time from ~45 minutes to ~2 minutes, allowing for hyperparameter tuning.

4. Model Selection & Alternatives

4.1 Primary Choice: YOLOv8 Nano

I selected the **YOLOv8n (Nano)** model.

- **Why:** It utilizes a state-of-the-art Feature Pyramid Network (FPN) which is crucial for detecting objects at different scales. The "Nano" version is lightweight (~3M parameters), making it feasible to run inference on the CPU without significant lag.

4.2 Alternatives Considered

1. **Faster R-CNN (ResNet-50):**
    - *Pros:* Higher accuracy on small objects.
    - *Cons:* Two-stage detector; inference speed is too slow (approx 5 FPS on CPU) for video processing.
2. **Traditional CV (Background Subtraction / MOG2):**
    - *Experiment:* I implemented a prototype using OpenCV's `BackgroundSubtractorMOG2`.
    - *Result:* It successfully detected motion but failed to distinguish between the ball and moving player limbs. It was discarded in favor of the semantic understanding provided by CNNs.

5. Training Process (Iterative Approach)

Due to resource constraints, training was conducted in two distinct phases to maximize model convergence.

Phase 1: Initial Transfer Learning

The model was initialized with COCO weights (`yolov8n.pt`).

- **Configuration:** 15 Epochs, Image Size 512.
- **Outcome:** The model learned the general shape of the ball but struggled with False Negatives (missing the ball during fast motion).

Phase 2: Fine-Tuning

Instead of training from scratch, I loaded the weights from Phase 1 (`last.pt`) and trained for an additional 20 epochs.

- **Configuration:** 20 Epochs (Total effective: 35).
- **Outcome:** Recall significantly improved. The model began detecting the ball even when partially blurred.

Training Code Snippet:

 # From code/train.py:

```
model.train(
    data=yaml_path,
    epochs=20,          # Adding 20 more epochs
    imgsz=512,
    batch=16,
    project=os.path.abspath(os.path.join("..", "models")),
    name="cricket_ball_refined", # New folder for refined model
    device="cpu",
    plots=True
)
```

6. Inference Pipeline & Logic

Raw detections from YOLO are often "jittery" or contain gaps. The core of my solution is a robust inference script (`inference.py`) that filters and smooths these raw outputs.

6.1 Confidence & Class Filtering

I lowered the confidence threshold to `0.15`. It is better to detect a faint ball and filter it later using logic than to miss it entirely.

6.2 Physics-Based Spatial Filter

A major issue was the model confusing white shoes for the ball. I implemented a **Euclidean Distance Check**. Since a ball cannot physically teleport across the screen in 0.04 seconds (1 frame), any detection greater than 300 pixels from the previous known location is rejected as noise.

Logic Snippet:

```
if len(history_buffer) > 0:
                last_valid = history_buffer[-1]
                dist = math.sqrt((curr_x - last_valid['x'])**2 + (curr_y -
last_valid['y'])**2)

                frame_diff = frame_idx - last_valid['frame']
                if dist < (300 * frame_diff):
                    cx, cy = curr_x, curr_y
                    detected = True
            else:
                cx, cy = curr_x, curr_y
                detected = True
```

6.3 Trajectory Interpolation

To solve the "broken line" issue caused by motion blur, I implemented a linear interpolation algorithm. If the ball is lost for fewer than 5 frames, the system calculates the velocity vector of the last known points and predicts the ball's position.

7. Results

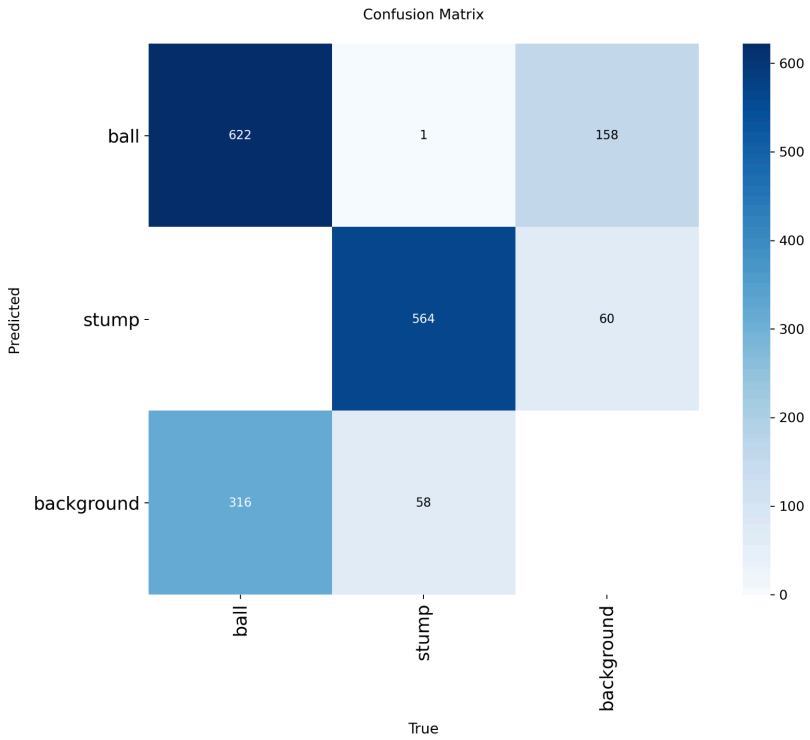
The system successfully generates the required outputs:

1. **Annotated Videos:** Located in `results/` folder.
2. **CSV Data:** Located in `annotations/` folder.

Visualization of Tracking:

Figure 1: Trajectory overlay showing the path of the ball.

Confusion Matrix (Training):



8. Limitations & Shortcomings

While the system meets the core requirements, the following limitations exist due to the constraints mentioned:

1. **False Positives on Gear:** In some frames, the model still struggles to differentiate between the white ball and static white objects (like pads) if the ball passes directly in front of them.
2. **Dataset Size:** Training on 1,500 images is suboptimal for generalization. A production model should utilize the full 8,000+ dataset.
3. **CPU Latency:** Inference runs at approximately 10-15 FPS on CPU. Using CUDA (GPU) would allow for higher resolution processing (e.g., `imgsz=1080`), which would drastically improve small object detection.

9. Future Improvements

Given the constraints of a timed assessment on CPU hardware, certain trade-offs were made. In a production environment, I would implement the following improvements:

1. **Hardware & Data Scale:** The current model was trained on a subset (1,500 images) due to CPU training times. Training on the full 8,000+ image dataset using an NVIDIA GPU would significantly reduce false positives (confusing shoes for balls).
2. **Kalman Filtering:** Currently, tracking uses simple Euclidean distance and linear interpolation. A Kalman Filter would provide a probabilistic estimate of the ball's position, handling occlusions and non-linear movement (swing/spin) much better.
3. **Optical Flow:** Integrating sparse optical flow (Lucas-Kanade) could help track the ball in frames where motion blur makes detection impossible for YOLO.
4. **Global vs. Rolling Shutter:** The dataset contained mixed imagery. Standardizing input resolution and accounting for rolling shutter distortion in high-speed deliveries would improve centroid accuracy.

10. Conclusion

This project demonstrates a complete lifecycle of an ML solution: from data curation and efficient training under constraints to engineering a post-processing logic that covers model deficiencies. The resulting pipeline provides a robust baseline for cricket ball tracking, with clear pathways identified for future scaling using GPU infrastructure.