Cricket Ball Detection and Trajectory Estimation Using YOLOv8

K Priyanka

Abstract

This report presents the design, implementation, and evaluation of a cricket ball detection and trajectory estimation pipeline based on the YOLOv8 object detection model. The pipeline processes raw cricket videos, detects the ball in each frame, computes the centroid, and overlays the predicted trajectory. Transfer learning is applied to fine-tune the YOLO model on a custom cricket ball dataset. The report summarizes the modelling decisions, fallback logic, assumptions, dataset challenges, error analysis, and improvements made to enhance model performance.

1 Introduction

Detecting a cricket ball in broadcast or mobile-recorded videos is challenging due to its small size, high speed, motion blur, and varying lighting conditions. Traditional detection methods fail when the background is cluttered or when the ball blends with the pitch. Recent advances in deep learning and one-stage detectors such as YOLOv8 offer strong real-time performance, making them suitable for sports analytics.

This work focuses on building a modular detection pipeline capable of:

- performing ball detection via YOLOv8,
- computing ball centroids,
- maintaining a simple trajectory curve,
- exporting per-frame annotations,
- processing multiple videos in batch.

Only the detection and trajectory computation modules are implemented in this phase, but the system is designed to integrate future modules such as speed estimation, 3D ball tracking, and camera-invariant depth estimation.

2 Dataset and Preprocessing

The dataset consists of cricket-rally video frames annotated with bounding boxes around the ball. The dataset is split into train, validation, and test sets using a stratified partition to preserve ball-visibility diversity.

2.1 Challenges Observed

- **Small object problem:** The ball often occupies fewer than 20 pixels.
- **Motion blur:** High-speed deliveries create streaked appearances.
- **Background clutter:** The ball merges with pitch colors and crowd.
- **Low class imbalance:** Only one class (ball) leads to overfitting.

2.2 Fixes Applied

- Training images were resized to 640×640 to preserve ball features.
- Augmentation such as motion blur, brightness shifts, and random crops were applied using YOLO's built-in augmentation pipeline.
- The dataset folder was cleaned for mislabeled frames and incorrect boxes.
- Bounding box padding was increased for partially visible balls.

3 Modeling Decisions

3.1 Choice of YOLOv8

YOLOv8 was selected due to:

- excellent support for small-object detection,
- simple transfer learning workflow,

- strong inference speed on CPU,
- built-in training optimizations.

The **YOLOv8n** model (nano variant) was used initially due to CPU constraints. Later experiments validated that **YOLOv8s** slightly improved detection accuracy.

3.2 Transfer Learning Strategy

We initialized the model with pretrained COCO weights:

$$\text{model} \leftarrow \text{YOLOv8n.pt pretrained on COCO}$$

Fine-tuning was done for 50 epochs with:

- batch size = 16,
- learning rate = 0.001,
- SGD optimizer.

Backbone layers were frozen for the first 10 epochs to prevent catastrophic forgetting.

4 Fallback Logic

Due to temporary detection failures (e.g., occlusion or motion blur), direct centroid extraction often breaks the trajectory. Fallback rules were added:

1. **If detection is missing for 1–2 frames:** Use last known centroid.
2. **If confidence** < 0.20**:** Reject detection to avoid noisy jumps.
3. **If bounding box size is below a threshold:** Mark detection invalid.
4. **If multiple boxes appear:** Choose the one closest to previous centroid.

$$\hat{c}_t = \begin{cases} c_t, & \text{if detection valid} \\ \hat{c}_{t-1}, & \text{fallback (1–2 missing frames)} \\ \text{NULL}, & \text{if missing ¿ 3 frames} \end{cases}$$

This significantly stabilized the trajectory plots.

5 Trajectory Computation

Given a bounding box (x_1, y_1, x_2, y_2), the centroid is computed as:

$$c_x = \frac{x_1 + x_2}{2}, \quad c_y = \frac{y_1 + y_2}{2}$$

The tracker stores the time-ordered centroids:

$$T = \{c_1, c_2, \ldots, c_n\}$$

A simple polyline is drawn using OpenCV. For smoother trajectories, a future extension would use a Kalman filter or spline interpolation.

6 System Architecture

- **model/** — YOLOv8 detection model.
- **tracking/** — simple centroid tracker.
- **code/** — training and batch inference scripts.
- **annotations/** — per-frame CSV files.
- **results/** — processed videos with trajectory overlays.

Batch processing automatically generates outputs for videos named 1.mp4 to 15.mp4.

7 Example Outputs

7.1 Detection Example

Bounding boxes are drawn around the ball with a red centroid point.

7.2 Trajectory Example

A polyline follows the centroid over multiple frames.

8 Performance Evaluation

- Precision improved from 0.62 to 0.78 after data cleaning.
- Recall improved from 0.55 to 0.74 after applying augmentation.
- FPS on CPU: 15–22 fps depending on video resolution.

9 Improvements Made

1. Resolved dataset path errors by restructuring YAML and folders.
2. Removed corrupted frames and fixed annotation inconsistencies.
3. Adjusted model confidence thresholds to reduce false positives.
4. Added fallback logic to stabilize centroid tracking.
5. Implemented batch processing for 15+ videos automatically.

10 Limitations and Future Work

- Depth estimation module not yet implemented.
- No 3D ball trajectory reconstruction.
- No camera calibration or pitch dimension normalization.
- Temporal smoothing can be further improved.

11 Conclusion

This project successfully developed a robust detection and trajectory pipeline for cricket ball tracking using YOLOv8. The system provides clean visualizations, per-frame annotations, and stable motion trajectories. The modular approach enables seamless integration of additional stages such as speed estimation, camera-invariant depth prediction, and real-time deployment.