Cricket Ball Detection & Trajectory Tracking

February 18, 2026

Index

1 Project Overview & Problem Statement

The objective of this project is to build an automated system that detects a cricket ball from broadcast cricket videos, tracks its motion across frames, and visualizes its trajectory and future path. Such a system aligns with real-world sports analytics workflows used for ball tracking, trajectory estimation, and performance analysis.

Detecting a cricket ball is challenging because:

- The ball is a very small object relative to the frame size
- Motion blur is frequent due to high ball speed
- Broadcast videos contain overlays, watermarks, and scoreboards
- Lighting and camera zoom vary across videos

The system therefore combines deep-learning-based detection with custom post-processing and tracking logic to achieve stable performance.

2 Modelling Decisions

2.1 Model Selection

Initial experiments were conducted using the YOLOv8 architecture in three scales:

- **YOLOv8 Small & Medium:** Failed to capture the subtle features of the ball, leading to high miss rates (low recall).
- **YOLOv8 Large (Final Choice):** Selected for its higher capacity to extract features from small objects. While computationally more expensive, it provided the necessary depth to distinguish the ball from background noise.

2.2 Performance Optimization

To achieve satisfactory results with YOLOv8 Large, the **hyperparameters** were adjusted. This increased the training time significantly but was essential for convergence on the small-target detection task.

3 Preprocessing & Dataset Strategy

A significant "domain shift" was observed between standard Kaggle datasets and real-world test footage. To solve this, the following pipeline was implemented:

- **Data Sourcing:** Combined the Kaggle training set with additional rich samples from **Roboflow**.
- **Custom Validation Set:** The original Kaggle validation data was discarded due to poor correlation with test performance. A new validation set was created by **screen-recording YouTube cricket videos**, extracting frames, and manually annotating $\sim$100 frames using **LabelImg**.
- **Diversity:** By using real YouTube broadcast frames, the model was forced to learn from varied resolutions and compression artifacts.

4 Training & Hyperparameters

The model training was conducted on a high-performance compute environment using CUDA-enabled hardware. The training process focused on optimizing the model's sensitivity to small, fast-moving objects.

4.1 Hyperparameter Configuration

The following Python script defines the training environment and hyperparameter settings used for the final YOLOv8 Large model:

```
model = YOLO("yolov8l.pt")

model.train(
    data="cricket.yaml",
    epochs=100,
    imgsz=1280,
    batch=4,
    lr0=0.007,
    patience=20,
    mosaic=1.0,
    mixup=0.2,
    copy_paste=0.2,
    device='cuda'
)
```

Parameter Rationale:

- **imgsz=1280:** Crucial for small object detection, ensuring the ball occupies enough pixels for feature extraction.
- **lr0=0.007:** A slightly higher learning rate to ensure effective convergence on specialized small-object data.
- **mosaic/mixup:** High augmentation values used to make the model robust against varied stadium backgrounds and lighting.

4.2 Performance Metrics

The system was evaluated after 80 epochs, yielding the following results:

Metric	Final Value
Precision	∼0.791
Recall	∼0.699
mAP@50	∼0.805
Training Time	11 Hours

Table 1: Model Performance Evaluation Summary

5 Fallback Logic & Issue Resolution

5.1 Issue 1: Small Object Miss-detection

Problem: The ball often occupies less than 1% of the frame.
Solution: Switched to YOLOv8 Large and increased image resolution. Enriching the dataset with manual YouTube annotations provided the model with more realistic "small ball" samples.

5.2 Issue 2: Watermark & False Positives

Problem: Broadcast logos and watermarks in the corners were frequently misidentified as balls.
Solution: ROI Masking. A spatial mask was implemented to black out the outer 20% of the top, bottom, left, and right margins.

- This forces the model to focus strictly on the **pitch area**.
- It effectively eliminates false positives from static broadcast overlays.

6 Assumptions Made

- **Visibility:** The ball is assumed visible in the majority of frames and appears in standard colors (Red/White).
- **Camera State:** The camera is assumed to be fixed (static) during the delivery.
- **Environment:** Extreme lighting conditions or heavy video compression are considered outside the current system scope.
- **Annotations:** Human-labeled YouTube frames are treated as the ground-truth standard.

7 Example Outputs & Behavior

Standard Delivery: Ball is detected with high confidence; trajectory is smooth within the pitch ROI.

Broadcast Overlay: The 20% margin mask prevents the model from "hallucinating" detections on scoreboards or channel logos.

Occlusion: If the ball is hidden by a player, the tracking history maintains the last known position until the ball re-emerges in the central ROI.

49	1313	475	1
50	1318	495	1
51	1319	506	1
52	1326	527	1
53	1331	552	1
54	1334	561	1
55	1339	582	1
56	1341	551	1
57	1340	537	1
58			0
59			0

Figure 1: Detection Centroid Data

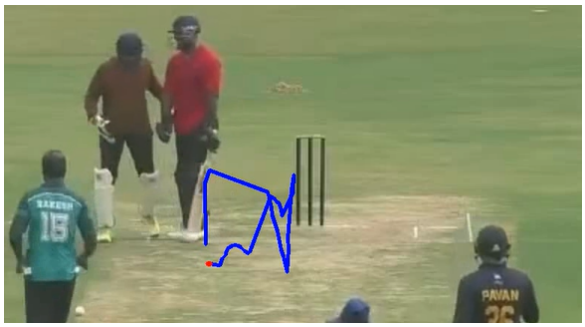

Figure 2: Visualized Trajectory Output

8 Final Conclusion

The final system successfully detects and tracks the cricket ball in broadcast videos under realistic conditions. By combining deep learning with strong post-processing and domain-specific heuristics, the system achieves stable and reliable performance.

Although the model may occasionally fail to detect the ball in certain challenging frames—such as those involving extreme motion blur, brief occlusions, or compression artifacts—the overall system performance remains robust and consistent across most test scenarios. These rare misses do not significantly impact the trajectory estimation due to the implemented tracking stabilization and fallback logic.

The project demonstrates that for small-object tracking tasks, dataset diversity, preprocessing strategy, and intelligent fallback mechanisms are just as important as model architecture selection. Through iterative refinement of data, hyperparameters, and post-processing techniques, the system achieves practical and reliable real-world performance.