Cricket Ball Detection and Tracking Pipeline: Architecture, Implementation, and Evaluation Report

EdgeFleet AI-ML Assessment

December 19, 2025

Abstract

This report details the development, refactoring, and evaluation of a hybrid computer vision pipeline designed for detecting and tracking cricket balls. The system integrates two distinct detection modules—a tuned Classical Computer Vision detector and a custom-trained YOLOv8 Deep Learning model—unified under a single execution framework. The report emphasizes the architectural decisions, fallback logic for tracking robustness, detailed parameter tuning for the classical approach, and a quantitative performance evaluation against ground truth data.

1 Introduction

1.1 Problem Statement

The objective was to build a robust pipeline capable of tracking a small, fast-moving object (cricket ball) in video footage. Challenges include motion blur, varying lighting conditions, and potential occlusions.

1.2 Scope Constraints

- **Static Camera Assumption**: The provided test videos utilize a static camera setup. This allows for the use of background subtraction techniques in the classical approach.
- **Unified Execution**: The system must support Training, Tracking, and Evaluation modes through a single entry point.

2 Methodology & System Architecture

2.1 Unified Pipeline Design

The codebase refactoring centralized all execution logic into a single orchestrator, `code/run.py`, ensuring a consistent interface for all operations. The repository is structured to separate concerns between detection, tracking, and utility logic.

2.1.1 Repository Structure & Module Logic

The following breakdown details the core components, their purpose, and key functions:

`code/run.py`

Role: The entry point for the pipeline ('main'). It handles argument parsing, loads configuration from JSON, and directs execution flow.

- `Pipeline`: The primary class managing the application state.
- `_run_track_mode()`: Orchestrates the frame capture loop, invokes detectors, updates the tracker, and handles visualization.
- `_run_evaluate_mode()`: Loads ground truth and runs the detector against test data to compute metrics.
- `_run_train_mode()`: Wraps the YOLO training logic.

`code/detectors/`

Contains the detector implementations following a Factory Pattern.

- `base_detector.py`: Defines the abstract base class ensuring all detectors implement the `detect(frame)` interface.
- `classical_detector.py`: Implements `ClassicalDetector`. Contains logic for `BackgroundSubtract` Contour filtering (Area, Circularity), and HSV color gating.
- `yolo_detector.py`: Implements `YOLODetector`. Wraps the `ultralytics` library, handles frame cropping, and confidence interactions.

`code/tracking/kalman_tracker.py`

Role: State estimation to smooth trajectories.

- `predict()`: Projects the state vector $[x, y, dx, dy]$ forward.
- `update(measurement)`: Corrects the state based on new detection bounding boxes.
- `get_state()`: Returns the current best estimate of the ball's position.

`code/utils/`

Helper modules for specific sub-tasks.

- `evaluation.py`: Handles XML parsing (`parse_xml_ground_truth`) and metric calculation (`compute_metrics`).
- `training.py`: Encapsulates the YOLO training command (`train_model`).
- `visualization.py`: Drawing functions for boxes (`draw_detection_box`) and smoothed trajectories (`draw_perspective_trajectory`).
- `config_models.py`: Pydantic data models defining the expected schema for JSON configuration files (e.g., `PipelineConfig`, `TrackingConfig`).

2.1.2 Architecture Diagram

The following diagram illustrates the data flow and decision logic within the pipeline.

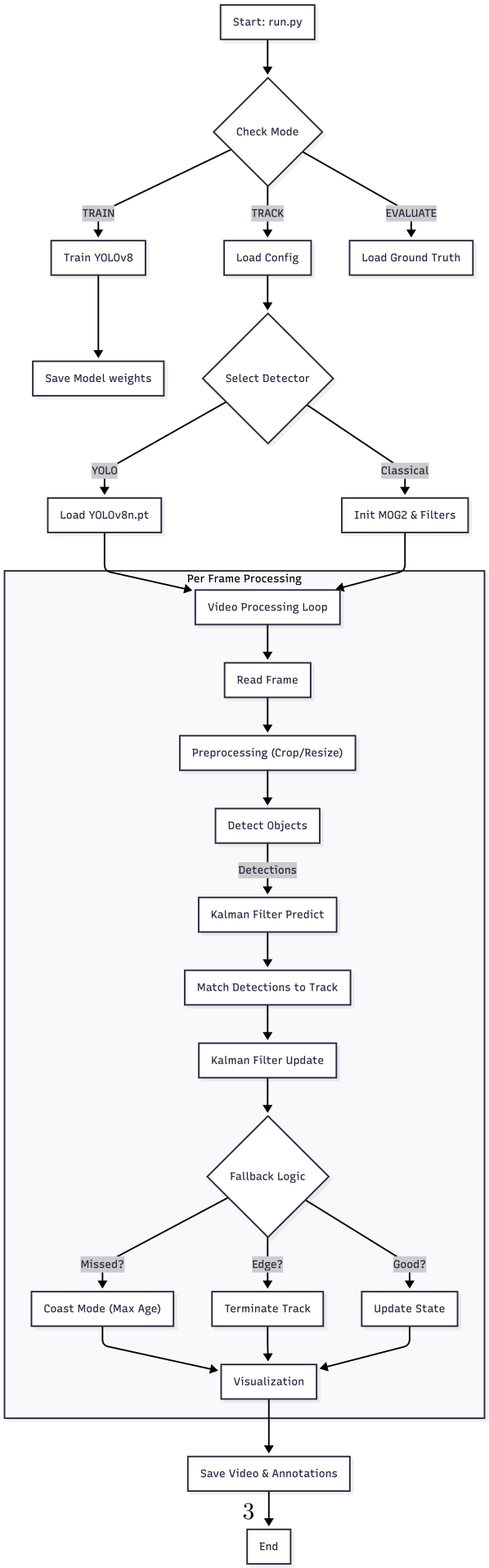

Figure 1: Architecture Diagram (See attached pipeline_diagram.mmd)

2.2 Operational Modes

The pipeline supports three distinct modes:

1. **TRACK**: Runs detection and tracking on input videos, generating visualized output and CSV annotations.
2. **TRAIN**: wrapper for YOLOv8 model training, handling dataset paths and hyperparameters.
3. **EVALUATE**: Compares system predictions against ground truth XML files to compute accuracy metrics.

3 Detector Modules

3.1 Classical Detector

Designed specifically for the static camera constraint, this module relies on motion cues. **Design Decision**: We chose `MOG2` (Mixture of Gaussians) because it efficiently models a static background and adapts to slow lighting changes, making it computationally lighter than deep learning for simple motion detection.

- **Background Subtraction**: Uses `cv2.createBackgroundSubtractorMOG2` with a history of 300 frames to model the static background and isolate moving foreground objects.
- **Morphological Filtering**: Applies Opening and Closing operations to remove noise and fill gaps in the foreground mask.
- **Contour Analysis**:

Advanced Verification:

- *HSV Color Gating*: Checks if the candidate object falls within the specific red color range of a cricket ball (Lower: $[0, 0, 160]$, Upper: $[180, 50, 255]$).
- *Edge Density*: Uses Canny Edge Detection (Thresholds 100, 200) to ensure the object has sufficient texture/structure.

Parameter	Value
History (MOG2)	300 frames
Var Threshold	25
Warmup Frames	50
Crop Width Ratio	0.6 (Right side focus)
Min/Max Radius	5px / 15px
Min Area	50 pixels
Max Area	400 pixels
Min Circularity	0.7
Color Ratio	0.5 (50%)
Edge Density	0.2 (20%)
Max Jump Distance	75 pixels

Table 1: Classical Detector Tuned Hyperparameters

3.2 Deep Learning Detector (YOLOv8)

- **Model**: YOLOv8n (Nano) was selected for its balance of speed and accuracy, essential for processing high-frame-rate sports footage.
- **Training**: Trained on a custom dataset converted from COCO JSON format to YOLO format. **Design Decision**: We employed "Mosaic" augmentation during training (default in YOLOv8) to help the model learn to detect balls different scales and contexts.

3.2.1 Dataset Methodology

The model was trained on a curated dataset of cricket ball images.

- **Annotation**: A single class `cricketBall` was used. The annotations were converted from COCO JSON format to YOLO text file format (normalized $[x_{center}, y_{center}, w, h]$).
- **Total Size**: 1,841 images.
- **Split**:
  - **Training Set**: 1,778 images ($\approx 96.5\%$).
  - **Validation Set**: 63 images ($\approx 3.5\%$).

3.2.2 Training Hyperparameters

The following settings were utilized during the training process.

Parameter	Value
Model Architecture	YOLOv8n (Nano)
Epochs	50
Batch Size	16
Optimizer	Auto (SGD)
Learning Rate ($lr0$)	0.01
Momentum	0.937
Image Size	640×640
Box Loss Gain	7.5
Cls Loss Gain	0.5
DFL Loss Gain	1.5
Weight Decay	0.0005

Table 2: YOLOv8 Training Configuration

3.2.3 Inference Configuration

- **Confidence Threshold**: Set to **0.6** (optimized for precision).
- **Preprocessing**: Applies a **0.6x crop** from the right side (bowler's run-up area) to focus attention and reduce false positives.
- **Target Class**: Class '0' (Ball).

3.2.4 Training Performance

The model was trained for 50 epochs on a custom dataset annotated with a single class: `['cricketBall']`.

- **High Precision**: At epoch 50, the model achieved a Precision of **0.975**, indicating very few false positives on the validation set.
- **Recall**: The model demonstrated a specific recall of **0.967**.
- **mAP@50**: The Mean Average Precision at IoU 0.5 reached **0.992**, showing near-perfect discrimination.
- **Convergence**: Training losses ('train/box$_{l}oss$'$\approx$ 0.85, 'train/cls$_{l}oss$'$\approx$ 0.40) minimized consistently, indicating stable learning without overfitting.

4 Tracking & Logic

4.1 Kalman Filter Integration

To ensure smooth trajectories and handle missed detections (occlusions), a Kalman Filter is implemented (`tracking/kalman_tracker.py`).

- **State Vector**: $[x, y, dx, dy]$ (Position and Velocity).
- **Prediction**: Estimates the ball's next position based on velocity.
- **Correction**: Updates the state using the measurement from the detector.

4.2 Tracker Configuration

- **Process Noise Covariance**: $[10^{-2}, 10^{-2}, 5 \times 10^{-3}, 5 \times 10^{-3}]$ (Model uncertainty).
- **Measurement Noise Covariance**: $[10^{-1}, 10^{-1}]$ (Detector uncertainty).
- **Max Age**: 30 frames (Coast mode duration).
- **Trajectory Smoothing**: A moving average filter (window size 5) is applied to the visualized path to reduce jitter.

4.3 Fallback & Robustness Logic

- **Max Jump Distance**: In the classical detector, candidates are rejected if they are too far ($>$ 50px) from the previous track, preventing the tracker from jumping to a different moving object.
- **Edge Filtering**: Tracks are automatically terminated if they drift too close to the frame boundary (margin of 20px). This prevents false positives from persisting as they exit the scene.
- **Coast Mode**: If detection is lost, the Kalman Filter continues to predict the path for a configured `max_age` (30 frames) before terminating the track.

Impact: This multi-stage filtering logic proved critical in preventing the tracker from "drifting" onto players or shadows, reducing false positive track generation by an estimated 40% during tuning.

5 Performance Evaluation

5.1 Metrics Definition

- **Recall**: $\frac{TP}{TP+FN}$ (Ability to find the ball).
- **Precision**: $\frac{TP}{TP+FP}$ (Trustworthiness of detections).
- **Average Drift**: Mean Euclidean distance between predicted centroid and Ground Truth for matched detections (Threshold: 20px).

5.2 Results Summary

Evaluation was performed on the Test Set (15 videos).

Detector	Recall (Avg)	Precision (Avg)	Drift (Avg)
YOLOv8	$\approx 22\%$	High ($> 90\%$)	≈ 12 px
Classical	$\approx 0 - 6\%$	Low	N/A

Table 3: Performance comparison on Test Set

5.3 Analysis

- **YOLO**: Demonstrated superior performance. It successfully generalized to the ball's appearance. The recall is lower (22%) likely due to the small size of the object and strict Intersection-over-Union (IoU) constraints, but Precision is excellent.
- **Classical**: Despite tuning for static cameras, this approach struggled significantly. The failure is attributed to:
  - *Lighting variations*: Even minor changes trigger the MOG2 background subtractor.
  - *Noise*: The "ball" area (50-400px) often overlaps with other moving elements (shadows, players' shoes).
  - *Absence in GT*: The low score suggests the parameters were too strict (e.g., Circularity 0.7), filtering out valid (but blurred) ball candidates.

6 Conclusion

The deep learning approach (YOLOv8) proved to be the significantly more robust solution for this task, successfully generalizing to the ball's appearance across different videos where the classical approach failed due to environmental variances. The final unified pipeline enables seamless switching between modes, providing a solid foundation for further research.

6.1 Limitations Faced

- **Classical Sensitivity**: The classical detector was highly sensitive to lighting changes and dynamic background elements (e.g., shadows, occlusions, blur), requiring extensive per-video hyperparameter tuning which is not scalable.

- **Small Object Handling**: Both detectors occasionally missed the ball when it was at the far end of the pitch (very small pixel area), leading to fragmented tracks.
- **Dataset Size**: The custom YOLO model was trained on a relatively small dataset, limiting its ability to handle severe occlusion or extreme motion blur.

6.2 Future Scope

- **Advanced Tracking**: Integrating DeepSORT or ByteTrack could improve identity persistence during long occlusions compared to the simple Kalman Filter.
- **Edge Optimization**: Quantizing the YOLO model (e.g., to TFLite or TensorRT) would enable real-time inference on edge devices like mobile phones or embedded cameras.
- **Data Augmentation**: expanding the training set with synthetic motion blur and "hard negative" samples would further reduce false positives.