Multi-Object Tracking System

Machine Learning for Visual Object Tracking - Practical Work Report

Yahya Ahachim

November 27, 2025

Contents

1 Introduction

This report presents the development and evaluation of object tracking systems implemented as part of the Machine Learning for Visual Object Tracking practical work. The project is divided into two main parts: (1) Single Object Tracking (SOT) that we correct using Kalman Filter, and (2) Multi-Object Tracking (MOT) with progressively sophisticated approaches including IoU-based tracking, Kalman Filter integration, and appearance-aware tracking using deep learning-based feature extraction for re-identification. Additionally, an end-to-end pipeline was developed using YOLO11x for detection. The MOT system was evaluated on the ADL-Rundle-6 sequence using standard metrics including HOTA, IDF1, and ID Switches.

2 Part 1: Single Object Tracking with Kalman Filter

2.1 Objective

Implement object tracking in 2D using a pre-existing object detection algorithm and integrate a Kalman Filter for smooth and accurate tracking. The object is represented as a point (centroid) in a Single Object Tracking (SOT) scenario.

2.2 Project Structure

The SOT implementation is located in the `2D_Kalman-Filter_TP1/` directory:

File	Description
`KalmanFilter.py`	Kalman Filter class implementation
`objTracking.py`	Main tracking script(also runs visualization)
`Detector.py`	Object detection using Canny edge detection
`video/randomball.avi`	Input video with object to track

Table 1: Single Object Tracking files

2.3 Kalman Filter Implementation

The `KalmanFilter.py` file contains the `Kalmanfilter` class with three main functions:
`__init__`, `predict` and `update`. And all three were simply implemented with the definitions provided in the assignment.

2.4 Object Tracking Script (objTracking.py)

The main tracking script performs:

1. **Initialization**: Create Kalman Filter with parameters:
   `dt=0.1, u_x=1, u_y=1, std_acc=1, x_std_meas=0.1, y_std_meas=0.1`
2. **Video Capture**: Load the input video sequence
3. **Tracking Loop**: For each frame:
   - Call `detect()` function to find object centroids using Canny edge detection
   - Call `predict()` to get predicted position using the Kalman Filter's defined model (constant and positive acceleration)
   - Call `update()` with detected centroid, giving us the corrected estimation and updating our error covariance matrix

- Store estimated position in trajectory

4. **Visualization**:
   - Green circle: Detected position
   - Blue rectangle: Predicted position
   - Red rectangle: Estimated (filtered) position
   - Yellow line: Trajectory path

2.5 Observed results

The visualization shows that the 2 rectangles correctly track the object. Also, The blue rectangle (prediction) is sometimes a bit off the position compared to the red rectangle (estimated position), which is exactly the expected result, as the estimation is the combination of prediction and the measurement, and so expected to correct their noise.

3 Part 2: Multi-Object Tracking

3.1 Multi-Object Tracking - Project Structure

The MOT project is organized in the `ADL-Rundle-6` directory with the following structure:

3.1.1 Python Scripts

File	Description
`IoU_based_tracker.py`	Basic tracker using IoU matching with Hungarian algorithm
`IoU_KF_tracker.py`	IoU tracker enhanced with Kalman Filter
`appearance_aware.py`	Tracker with ReID features + IoU + Kalman Filter
`end_to_end.py`	Complete pipeline: YOLO11x detection + appearance tracking
`detection_w_yolo11x.py`	Standalone YOLO11x detection script
`visualize_tracking.py`	Visualization and video generation tool (for all the above trackers)

Table 2: Python scripts and their purposes

- `appearance_aware.py` takes both Yolo5l and Yolo11x detections and generates both their result files
- `detection_w_yolo11x.py` can be used so that detection can be executed in "batch" mode, and generates the input for `appearance_aware.py` if used for Yolo11x

3.1.2 Detection Files

The used detections are:

- `det/Yolov5l/det.txt` - YOLOv5-large detections (given)
- `det/yolo11x/det.txt` - YOLO11x detections (generated)

3.1.3 Output Files

Results File	Video Output	Generator script
`tracking_results.txt`	`tracking_output.mp4`	`IoU_based_tracker.py`
`filtered_tracking_results.txt`	`filtered_tracking_output.mp4`	`IoU_KF_tracker.py`
`appearance_tracking_results.txt`	`appearance_tracking_output.mp4`	`appearance_aware.py`
`appearance_yolo11x_results.txt`	`appearance_yolo11x_output.mp4`	`appearance_aware.py`
`end_to_end_results.txt`	`end_to_end_output.mp4`	`end_to_end.py`

Table 3: Tracking results, corresponding visualization videos and scripts that generate them

3.2 Multi-Object Tracking - Methodology

3.2.1 Detection Format

All detection and tracking files follow the MOTChallenge format:

```
<frame>, <id>, <bb_left>, <bb_top>, <bb_width>, <bb_height>, <conf>, <x>, <y>, <z>
```

Where the last three values are set to -1 for 2D tracking (unused 3D coordinates).

3.2.2 IoU-Based Tracker

The baseline tracker (`IoU_based_tracker.py`) implements:

1. **Detection Loading**: Parse detections from text files and make a dictionary (frame, detections). And initialize current tracks list with the detection of the first frame with initial incremental object instance IDs
2. **Similarity Matrix**: For every new frame's detections, compute Intersection over Union (IoU) between current tracks and detections:

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

3. **Hungarian Algorithm**: Use `scipy.optimize.linear_sum_assignment` to find optimal matching that minimizes total Jaccard Index (1 - IoU)
4. **Track Management**:
   - Matched detections inherit the track ID
   - Unmatched detections create new tracks with incremented IDs
   - Unmatched tracks are terminated
   - The formatted current list of tracks is appended to the output

3.2.3 Kalman Filter Integration

The enhanced tracker (`IoU_KF_tracker.py`) adds motion prediction using a 2D Kalman Filter:

Process: Make a dictionary of Kalman Filters with keys being object instance IDs, initialized with the tracks of the first frame. Then for each new frame starting from the second:

1. **Predict**: Before matching, predict next position for all active tracks

2. **Match**: Use IoU between predicted positions and new detections
3. **Update**: For matched tracks, update Kalman state with measurement
4. **Smooth**: Use estimated (smoothed) position instead of raw detection

This rest of the tracker implementaion is similar to the previous one

3.2.4 Appearance-Aware Tracker

The most sophisticated tracker (`appearance_aware.py`) combines:

Similarity Score:

$$S = \alpha \cdot \text{IoU} + \beta \cdot \text{ReID\_similarity} \tag{2}$$

We chose $\alpha = \beta = 0.5$ (equal weighting).

Re-Identification Features:

- Model: OSNet (x0.25) trained on Market1501 dataset (given)
- Process: Extract 512-dimensional feature vectors from cropped bounding boxes
- Similarity: Chose function is cosine similarity between feature vectors

Preprocessing Pipeline: Crop bounding box from frame, then the given preprocessing steps which are:

1. Resize to 64×128 (width $\times$ height)
2. Normalize with ImageNet mean/std: $\mu = [0.485, 0.456, 0.406]$, $\sigma = [0.229, 0.224, 0.225]$
3. Convert to CHW format for ONNX inference

Processing: Identical to the previous tracker, except the added function `match_detections` that computes the IoU (not Jaccard Index this time) and the normalized_similarity of the re-identification features. Then combines them into the similarity score defined above and solves the linear sum assignment by maximizing the score this time. Which is then used for the matching between tracks and detections, hence taking appearance into consideration as well

3.2.5 End-to-End Pipeline

The complete pipeline (`end_to_end.py`) performs both detection and tracking:

Detection with YOLO11x:

- Model: YOLO11x converted to ONNX format
- Input: 640×640 with letterbox padding
- Class Filter: Only person class (COCO class 0)
- Post-processing: Non-Maximum Suppression (NMS) with IoU threshold 0.45

YOLO Post-processing:

1. Parse raw output: $(1, 84, 8400) \rightarrow (8400, 84)$
2. Extract box coordinates (center format) and class scores
3. Filter by confidence threshold (0.5) and person class
4. Apply NMS to remove overlapping detections
5. Convert to top-left corner format

4 Challenges and Solutions

4.1 Identity Fragmentation Problem

A key observation during development was the significant difference in the number of unique object IDs generated by each tracker compared to the ground truth. The ground truth contains only 24 unique pedestrians, but our trackers generated significantly more:

- **IoU-based Tracker**: ∼70 unique IDs
- **IoU + Kalman Filter**: ∼140+ unique IDs
- **Appearance-aware Tracker (baseline)**: 142 unique IDs

Interestingly, adding the Kalman Filter *increased* fragmentation rather than reducing it. This is because the Kalman Filter's motion predictions can diverge from actual positions during rapid movements or direction changes, causing the IoU between predicted and detected boxes to fall below the matching threshold.

4.2 Visual Analysis of Tracking Failures

By examining the generated tracking videos, we observed that re-identification failures primarily occur in two scenarios:

1. **Occlusions**: When pedestrians pass behind objects or each other, the tracker loses the association and assigns a new ID when the person reappears.
2. **Pose Changes**: Significant changes in body orientation alter the appearance features enough that the cosine similarity drops, breaking the identity link (like the woman with a child who gets down to pick up an object).

4.3 Improvement Experiments

To address identity fragmentation, we conducted iterative experiments with the appearance-aware tracker using YOLO11x detections:

4.3.1 Test 1: Baseline Configuration

- $\alpha = 0.5$, $\beta = 0.5$ (equal IoU and appearance weighting)
- No matching threshold
- No track persistence (unmatched tracks immediately terminated)

4.3.2 Test 2: Aggressive Track Persistence

- $\alpha = 0.3$, $\beta = 0.7$ (favor appearance)
- Matching threshold = 0.3
- Track persistence: MAX_AGE = 30 frames
- Lost tracks kept alive and matched using Kalman-predicted positions

4.3.3 Test 3: Balanced Configuration

- $\alpha = 0.5$, $\beta = 0.5$ (balanced weighting)
- Matching threshold = 0.2 (lowered)
- Track persistence: MAX_AGE = 10 frames (reduced)
- Raw detection coordinates used for re-found tracks (avoid Kalman divergence)

Visual Observations from Test 2 and Test 3: The aggressive settings caused flickering "ghost" bounding boxes moving on screen not bound to any object. Some bounding boxes were also less tightly fitted. This was caused by keeping lost tracks too long and using diverged Kalman predictions. But the problem persisted with the third test, although less severely.

5 Evaluation Results

All experiments were evaluated using TrackEval on the ADL-Rundle-6 sequence with HOTA, CLEAR, and Identity metrics.

5.1 Comparison of Results

Metric	Test 1 (Baseline)	Test 2	Test 3
HOTA	38.90	33.78	36.90
DetA	46.19	43.35	45.68
AssA	33.82	26.71	30.70
LocA	79.97	79.45	79.70
MOTA	50.01	44.48	48.51
MOTP	76.88	76.83	76.64
ID Switches	88	55	69
Fragmentations	104	101	118
IDF1	46.57	44.40	48.43
IDP	50.03	47.70	52.03
IDR	43.56	41.53	45.30
Unique IDs	142	32	52
Ground Truth IDs	24	24	24

Table 4: Comparison of tracking metrics across three test configurations

5.2 Analysis

5.2.1 Test 1 → Test 2: Aggressive Track Persistence

Positive outcomes:

- Unique IDs dropped dramatically: 142 → 32 (now only 1.3× ground truth instead of 6×)
- ID Switches reduced: 88 → 55 (37% improvement)

Negative outcomes:

- HOTA decreased: 38.90 → 33.78 (-13%)

- AssA decreased: 33.82 → 26.71 (-21%)
- MOTA decreased: 50.01 → 44.48 (-11%)

Interpretation: The tracker maintained identities too aggressively, holding onto incorrect associations for too long. The matching threshold (0.3) prevented creating new tracks even when appropriate, and the 30-frame track persistence caused "ghost" tracks with diverged Kalman predictions.

5.2.2 Test 2 → Test 3: Balanced Configuration

Changes made:

- Reduced MAX_AGE from 30 to 10 frames
- Lowered matching threshold from 0.3 to 0.2
- Reverted to balanced $\alpha = \beta = 0.5$
- Used raw detection coordinates for re-found tracks (because as we said earlier, since it restimates the position, it assumingly makes it harder to match a track with a detection. So we only apply it on matched tracks that weren't lost. The idea is that lost tracks may have moved very differently than what the Kalman filter is expecting during the frames it wasn't detected and the Kalman state updated).

Results:

- HOTA improved: 33.78 → 36.90 (+9%)
- IDF1 improved: 44.40 → 48.43 (+9%) — **best across all tests**
- ID Switches: 55 → 69 (slight increase, but still better than baseline's 88)
- Unique IDs: 32 → 52 (still much better than baseline's 142)

Interpretation: Test 3 achieved the best balance. While not recovering all metrics to baseline levels, it achieved the **highest IDF1** (48.43%) indicating best identity preservation, while keeping ID fragmentation under control (52 IDs vs 142 baseline).

5.3 Key Findings

1. **Trade-off between fragmentation and accuracy**: Reducing ID fragmentation (fewer unique IDs) can hurt overall tracking accuracy if done too aggressively.
2. **Track persistence requires tuning**: 30 frames was too long for this sequence (causing ghost tracks), while 10 frames provided a better balance.
3. **Kalman predictions diverge during occlusions**: Using raw detection coordinates for re-found tracks prevents bounding box drift.
4. **IDF1 as the key metric**: For identity preservation, IDF1 is the most relevant metric, and Test 3 achieved the best IDF1 score.

6 Visualization

The `visualize_tracking.py` script generates videos with:

- Bounding boxes colored by track ID
- ID labels displayed above each box
- Frame counter in the top-left corner
- Resizable display window (75% of original resolution)

Five output videos were generated:

1. `tracking_output.mp4` - IoU-based tracking
2. `filtered_tracking_output.mp4` - IoU + Kalman Filter
3. `appearance_tracking_output.mp4` - Appearance-aware (Yolov5l detections)
4. `appearance_yolo11x_output.mp4` - Appearance-aware (YOLO11x detections in batch mode)
5. `end_to_end_output.mp4` - End-to-end pipeline (tracking done streaming mode, picking up YOLO11x detections as they come)

7 Conclusion

This project successfully implemented both Single Object Tracking and Multi-Object Tracking systems:

7.1 Part 1: Single Object Tracking

- Implemented a complete Kalman Filter from scratch with predict/update cycles
- Integrated with edge-based object detection for centroid tracking
- Demonstrated the difference between raw detections, predictions, and filtered estimates
- Visualized the tracking trajectory over time

7.2 Part 2: Multi-Object Tracking

Developed progressively sophisticated tracking components:

1. **Baseline IoU Tracker**: Simple but effective for objects with high spatial overlap between frames
2. **Kalman Filter Enhancement**: Added motion prediction, though this paradoxically increased ID fragmentation
3. **Appearance Features**: Deep learning-based re-identification with OSNet for identity matching
4. **End-to-End Pipeline**: Complete system from raw images to tracked outputs using YOLO11x

7.3 Experimental Findings

Through iterative experimentation, we discovered:

- The baseline appearance-aware tracker produced 142 unique IDs for 24 ground truth pedestrians
- Aggressive track persistence (Test 2) reduced IDs to 32 but degraded overall accuracy
- Balanced configuration (Test 3) achieved the best IDF1 (48.43%) with 52 unique IDs
- The main failure modes are occlusions and pose changes
- Kalman Filter predictions can diverge during occlusions, requiring careful handling of refound tracks

7.4 Future Improvements

- Integrate Kalman Filters more intelligently. For example use a more complex model instead of the constant positive acceleration one (EKF).
- Enrich the dataset to improve detection and feature extraction (more poses of the same object instances, more frames of entering and exiting the scene...).