Multi-Object Tracking System

Machine Learning for Visual Object Tracking - Practical Work Report

Yahya Ahachim

November 26, 2025

Abstract

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) using a Kalman Filter for centroid-based tracking, 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 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.

Contents

1 Introduction

Object tracking is a fundamental computer vision task with applications in surveillance, autonomous driving, sports analysis, and human-computer interaction. This project implements both Single Object Tracking (SOT) and Multi-Object Tracking (MOT) systems.

Part 1 - Single Object Tracking: Implements a Kalman Filter-based centroid tracker for following a single object (ball) in a video sequence.

Part 2 - Multi-Object Tracking: Develops a tracking-by-detection system where object detections are associated across frames using various matching strategies:

- IoU-based matching with Hungarian algorithm
- Kalman Filtering for motion prediction and smoothing
- Appearance-based re-identification using deep features
- End-to-end detection and tracking with YOLO11x

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
`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:

2.3.1 Initialization (__init__)

The filter is initialized with six parameters:

- `dt`: Sampling time (time for one cycle)
- `u_x, u_y`: Accelerations in x and y directions
- `std_acc`: Process noise magnitude (standard deviation of acceleration)
- `x_std_meas, y_std_meas`: Measurement noise standard deviations

State Vector: $\mathbf{x}_k = [x, y, v_x, v_y]^T$ (position and velocity)
Control Input: $\mathbf{u} = [u_x, u_y]^T$
State Transition Matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Control Input Matrix:

$$\mathbf{B} = \begin{bmatrix} \frac{1}{2}\Delta t^2 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \tag{2}$$

Measurement Matrix:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{3}$$

Process Noise Covariance ($\mathbf{Q}$): Derived from acceleration noise σ_a^2
Measurement Noise Covariance:

$$\mathbf{R} = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix} \tag{4}$$

2.3.2 Predict Function

The prediction step projects the state forward in time:

$$\hat{\mathbf{x}}_k^- = \mathbf{A}\hat{\mathbf{x}}_{k-1} + \mathbf{B}\mathbf{u} \tag{5}$$
$$\mathbf{P}_k^- = \mathbf{A}\mathbf{P}_{k-1}\mathbf{A}^T + \mathbf{Q} \tag{6}$$

2.3.3 Update Function

The update step incorporates the measurement $\mathbf{z}_k$:

$$\mathbf{S}_k = \mathbf{H}\mathbf{P}_k^-\mathbf{H}^T + \mathbf{R} \tag{7}$$
$$\mathbf{K}_k = \mathbf{P}_k^-\mathbf{H}^T\mathbf{S}_k^{-1} \tag{8}$$
$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}\hat{\mathbf{x}}_k^-) \tag{9}$$
$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_k^- \tag{10}$$

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. **Detection**: Use `detect()` function to find object centroids using Canny edge detection
4. **Tracking Loop**: For each frame:
   - Call `predict()` to get predicted position
   - Call `update()` with detected centroid
   - Store estimated position in trajectory

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

Listing 1: Core tracking loop

```python
while True:
    ret, frame = cap.read()
    centers = detect(frame)
    if len(centers) != 0:
        k_filter.predict()
        predicted = k_filter.xk.copy()

        measurement = centers[0]
        k_filter.update(measurement)
        estimated = k_filter.xk.copy()

        # Visualization code...
        trajectory.append((estimated_x, estimated_y))
```

3 Part 2: Multi-Object Tracking

4 Part 2: Multi-Object Tracking - Project Structure

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

4.1 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

Table 2: Python scripts and their purposes

4.2 Detection Files

Pre-computed detections are stored in the `det/` directory:

- `det/Yolov5l/det.txt` - YOLOv5-large detections
- `det/Yolov5s/det.txt` - YOLOv5-small detections
- `det/yolo11x/det.txt` - YOLO11x detections (generated)

4.3 Output Files

Results File	Video Output
`tracking_results.txt`	`tracking_output.mp4`
`filtered_tracking_results.txt`	`filtered_tracking_output.mp4`
`appearance_tracking_results.txt`	`appearance_tracking_output.mp4`
`appearance_yolo11x_results.txt`	`appearance_yolo11x_output.mp4`
`end_to_end_results.txt`	`end_to_end_output.mp4`

Table 3: Tracking results and corresponding visualization videos

5 Part 2: Multi-Object Tracking - Methodology

5.1 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).

5.2 IoU-Based Tracker

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

1. **Detection Loading**: Parse detections from text files, organized by frame number
2. **Similarity Matrix**: Compute Intersection over Union (IoU) between all tracks and detections:
$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{11}$$
3. **Hungarian Algorithm**: Use `scipy.optimize.linear_sum_assignment` to find optimal matching that maximizes total IoU
4. **Track Management**:
    - Matched detections inherit the track ID
    - Unmatched detections create new tracks with incremented IDs
    - Unmatched tracks are terminated

5.3 Kalman Filter Integration

The enhanced tracker (`IoU_KF_tracker.py`) adds motion prediction using a 2D Kalman Filter:
State Vector: $\mathbf{x} = [x, y, v_x, v_y]^T$ (position and velocity)
State Transition Model:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{12}$$

Process:

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 helps handle:

- Noisy detections
- Brief occlusions
- Prediction during missed detections

5.4 Appearance-Aware Tracker

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

Similarity Score:

$$S = \alpha \cdot \text{IoU} + \beta \cdot \text{ReID\_similarity} \tag{13}$$

Where $\alpha = \beta = 0.5$ (equal weighting).

Re-Identification Features:

- Model: OSNet (x0.25) trained on Market1501 dataset
- Format: ONNX for efficient inference
- Process: Extract 512-dimensional feature vectors from cropped bounding boxes
- Similarity: Cosine similarity between feature vectors

Preprocessing Pipeline:

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

5.5 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

6 Challenges and Solutions

6.1 NumPy Array Shape Mismatch

Problem: The Kalman Filter state vector `xk` had shape issues when extracting scalar values.
Cause: `k_filter.xk[0]` returned a 1D array instead of a scalar due to matrix operations.
Solution: Created a helper function to safely extract scalar values:

```
def get_scalar(val):
    try:
        return float(val)
    except TypeError:
        return float(val[0])
```

6.2 Kalman Filter Broadcasting Error

Problem: The control input `u` was defined as a 1D array, causing shape mismatch during matrix multiplication.
Cause: `np.dot(B, u)` with shapes $(4, 2)$ and $(2,)$ produced unexpected broadcasting.
Solution: Define `u` as a column vector:

```
self.u = np.array([[u_x], [u_y]])  # Shape (2, 1) instead of (2,)
```

6.3 ONNX Model Input Name

Problem: ReID model inference failed with "Required inputs missing" error.
Cause: Hardcoded input name `'input'` didn't match model's expected `'modelInput'`.
Solution: Dynamically retrieve input name from session:

```
input_name = reid_session.get_inputs()[0].name
features = reid_session.run(None, {input_name: input_tensor})[0]
```

6.4 YOLO Duplicate Detections

Problem: YOLO11x produced many overlapping detections for the same object.
Cause: Missing Non-Maximum Suppression (NMS) in post-processing.
Solution: Implemented NMS using OpenCV:

```
indices = cv2.dnn.NMSBoxes(boxes, scores, conf_threshold, nms_threshold)
```

6.5 Negative Bounding Box Coordinates

Problem: Kalman Filter predictions sometimes produced negative coordinates, causing OpenCV resize errors.
Solution: Clamp coordinates to image boundaries in preprocessing:

```
x1 = max(0, x)
y1 = max(0, y)
x2 = min(img_w, x + w)
y2 = min(img_h, y + h)
```

6.6 TrackEval Compatibility

Problem: TrackEval failed with `np.float` deprecation errors.
Cause: NumPy 1.24+ removed deprecated aliases like `np.float`.
Solution: Replace deprecated types in TrackEval source:

```
find . -name "*.py" -exec sed -i 's/np\.float\b/np.float64/g' {} \;
find . -name "*.py" -exec sed -i 's/np\.int\b/np.int64/g' {} \;
```

7 Evaluation Results

The appearance-aware tracker with YOLO11x detections was evaluated using TrackEval on the ADL-Rundle-6 sequence.

7.1 HOTA Metrics

Metric	**Value**
HOTA	38.896
DetA (Detection Accuracy)	46.193
AssA (Association Accuracy)	33.815
LocA (Localization Accuracy)	79.971

Table 4: HOTA metrics breakdown

7.2 CLEAR Metrics

Metric	**Value**
MOTA	50.01
MOTP	76.876
ID Switches (IDSW)	88
Fragmentations	104
CLR_TP (True Positives)	3477
CLR_FP (False Positives)	884
CLR_FN (False Negatives)	1532

Table 5: CLEAR MOT metrics

7.3 Identity Metrics

Metric	**Value**
IDF1	46.574
IDR (ID Recall)	43.562
IDP (ID Precision)	50.034

Table 6: Identity metrics

7.4 Count Statistics

Metric	**Tracker**	**Ground Truth**
Total Detections	4361	5009
Unique IDs	142	24

Table 7: Detection and identity counts

7.5 Analysis

- **HOTA = 38.9%**: Balanced metric showing moderate overall performance
- **AssA = 33.8%**: Association accuracy is lower than detection accuracy, indicating room for improvement in identity maintenance
- **LocA = 79.97%**: Good localization accuracy shows bounding boxes are well-aligned
- **ID Switches = 88**: Moderate number of identity switches, could be reduced with better appearance matching or track management
- **142 vs 24 IDs**: The tracker creates many more identities than ground truth, suggesting fragmentation issues

8 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)
5. `end_to_end_output.mp4` - End-to-end pipeline

9 Conclusion

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

9.1 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

9.2 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**: Improved robustness to noise and brief occlusions through motion prediction
3. **Appearance Features**: Deep learning-based re-identification helps maintain identity during challenging scenarios
4. **End-to-End Pipeline**: Complete system from raw images to tracked outputs using YOLO11x

Key Learnings:

- Understanding the Kalman Filter equations and their role in state estimation
- The importance of proper matrix dimensions (column vectors vs row vectors)
- The importance of proper data format handling (MOTChallenge format)
- NMS is critical for modern object detectors to avoid duplicate detections
- Balancing IoU and appearance features (α, β weights) affects tracking quality
- Kalman Filter parameters significantly impact smoothness vs. responsiveness

Future Improvements:

- Implement track lifecycle management (tentative, confirmed, deleted states)
- Add motion-based gating before appearance matching
- Tune α/β weights based on sequence characteristics
- Implement cascade matching (high-confidence first)

Appendix: Running the Code

Part 1: Single Object Tracking

```
cd 2D_Kalman-Filter_TP1/
python objTracking.py
```

Part 2: Multi-Object Tracking

Dependencies:

```
pip install numpy scipy opencv-python onnxruntime
```

Run Trackers:

```
python IoU_based_tracker.py       # Basic IoU tracking
python IoU_KF_tracker.py          # IoU + Kalman Filter
python appearance_aware.py        # Appearance-aware tracking
python end_to_end.py              # Full pipeline with YOLO11x
```

Generate Visualizations:

```
python visualize_tracking.py
```

Run Evaluation (from TrackEval directory):

```
python scripts/run_mot_challenge.py \
    --BENCHMARK MOT17 \
    --SPLIT_TO_EVAL train \
    --TRACKERS_TO_EVAL yolo11x_w_appearance \
    --METRICS HOTA Identity CLEAR \
    --DO_PREPROC False
```