OPTION 2: Re-Identification in a Single Feed

Aryan Agarkar aryanagarkar09@gmail.com

09/07/2025

1. System Overview

This system is designed to identify each player in a 15-second soccer video, even if they leave and later reappear in the frame. The goal is to give the same identity to a player every time they appear, regardless of when or where they show up.

1. Frame and Object Detection

Each frame of the video is passed into a YOLOv11 object detection model. This model finds all the objects of interest players, goalkeepers, referees, and the ball and returns bounding boxes around them. Each bounding box is represented as:

- Upper-left corner: (x_1, y_1)
- Bottom-right corner: (x_2, y_2)
- Confidence score: how sure the model is
- Class ID: indicating whether it's a player, ball, etc.

We only keep the bounding boxes that are:

- Detected as players (Class ID = 2),
- Tall enough (height $\geq$ 80 pixels),
- Wide enough (width $\geq$ 40 pixels).

Each of these bounding boxes is used to crop out the player from the frame — these cropped images are saved for further processing.

2. Feature Extraction

Each cropped image of a player is passed through a special neural network called OSNet (specifically, `osnet_x1_0`). This model outputs a feature vector a list of numbers that uniquely represent the visual appearance of the player.

Let's say we detect 10 players in a frame. We now have 10 feature vectors, each one a numerical summary of how that player looks. These are stored in a matrix called $\mathbf{F}_t$, where:

- Each row is a feature vector for one player.
- The number of rows = number of players detected.
- The number of columns = size of the feature vector (typically 512).

3. Finding Similar Players in the Same Frame

We now compare every player with every other player in the same frame using cosine similarity. This tells us how visually similar two feature vectors are.

We create a similarity matrix $\mathbf{S}_t$, where:

- S_{ij} is the similarity score between player i and player j.
- Values range from -1 to 1; closer to 1 means they look very similar.

We then threshold this matrix keeping only those pairs that have similarity above 0.83 and form a binary matrix $\mathbf{A}_t$.This helps to group players in same team.In this matrix:

- 1 means "these two players look similar".
- 0 means "they don't look similar".

4. Grouping Similar Players

From the binary similarity matrix $\mathbf{A}_t$, we form sets for each player that contain the indices of players who look similar to them. For example:

- Player 0 might look similar to players 1 and 2, so the set is $\{0, 1, 2\}$.
- Player 3 might only look similar to player 4, so the set is $\{3, 4\}$.

We then merge overlapping sets to form final "identity groups". If two sets share members, we assume those players belong to the same real-world player. This gives us groups like:

- Group A: $\{0, 1, 2\}$
- Group B: $\{3, 4\}$
- Group C: $\{5\}$

Each group represents one unique player in the current frame.

5. Keeping Track of Players Across Frames

To recognize the same player later, we maintain a database of identities.
Each identity in the database stores:

- The average feature vector of all appearances of the player.
- The individual feature vectors seen so far.

When a new group of players is found:

- We calculate the group's average feature vector.
- We compare it with the feature vectors in our database.
- If it matches a known player (similarity ¿ 0.85), we update that entry.
- Otherwise, we create a new entry in the database.

6. Assigning IDs to Each Detected Player

Within each group, we assign an ID to each individual player detection:

- We compare each detection to the saved feature vectors of the matched identity.
- If it matches one of them (similarity ≥ 0.93), we give it the same ID.
- If it doesn't match and the identity has fewer than 12 known examples, we assign a new ID and update the database.

This ensures that even if a player disappears and comes back later, the system can recognize them and give them the same ID again.

7. Feature Extractor Details

The feature extractor used is:

- **Model:** OSNet_x1_0
- **Checkpoint file:** `osnet_x1_0.pth`
- **Hardware:** GPU (CUDA)

2. MATHS

2.1 Player Detection Module

We begin by processing each video frame $I_t \in \mathbb{R}^{H \times W \times 3}$ using a pre-trained YOLOv11 object detection model $\mathcal{M}_{\text{YOLO}}$, which outputs a set of detections:

$$\mathcal{B}_t = \left\{ (x_1^{(i)}, y_1^{(i)}, x_2^{(i)}, y_2^{(i)}, \text{conf}^{(i)}, c^{(i)}) \right\}_{i=1}^{N_t}$$

We filter the detections to retain only player instances:

$$\mathcal{P}_t = \{(x_1, y_1, x_2, y_2) \in \mathcal{B}_t \mid c = 2,\ (x_2 - x_1) \geq 40,\ (y_2 - y_1) \geq 80\}$$

Each valid bounding box is cropped from the frame:

$$C_t^{(j)} = I_t[y_1^{(j)} : y_2^{(j)},\ x_1^{(j)} : x_2^{(j)}]$$

Each crop is passed through a feature extractor $\mathcal{F}$, yielding:

$$\mathbf{f}_t^{(j)} = \mathcal{F}(C_t^{(j)}) \in \mathbb{R}^d$$

All feature vectors are stacked as a feature matrix:

$$\mathbf{F}_t \in \mathbb{R}^{M_t \times d}$$

2.2 Feature Extraction Module

We use the OSNet architecture to extract appearance features from each cropped image. Specifically:

- **Model:** `osnet_x1_0`
- **Checkpoint:** `osnet_x1_0.pth`
- **Device:** `cuda`

Each crop $C_t^{(j)} \in \mathbb{R}^{h \times w \times 3}$ is passed through the model to yield:

$$\mathbf{f}_t^{(j)} = \mathcal{F}(C_t^{(j)};\ \Theta)$$

where Θ are the learned model weights.

2.3 Cosine Similarity Matrix Computation

Given the feature matrix $\mathbf{F}_t$, we compute the pairwise cosine similarity matrix:

$$\mathbf{S}_t = \frac{\mathbf{F}_t \mathbf{F}_t^\top}{\|\mathbf{F}_t\|_{\text{row}} \cdot \|\mathbf{F}_t\|_{\text{row}}^\top}$$

Each entry in the similarity matrix is:

$$S_{ij} = \frac{\mathbf{f}_t^{(i)} \cdot \mathbf{f}_t^{(j)}}{\|\mathbf{f}_t^{(i)}\|_2 \cdot \|\mathbf{f}_t^{(j)}\|_2}$$

2.4 Similarity-Based Clustering

We threshold the cosine similarity matrix to generate a binary adjacency matrix:

$$\mathbf{A}_t(i,j) = \begin{cases} 1 & \text{if } \mathbf{S}_t(i,j) \geq \tau \\ 0 & \text{otherwise} \end{cases} \quad \text{with } \tau = 0.83$$

This defines local similarity neighborhoods:

$$\mathcal{G}_t^{(i)} = \{j \mid \mathbf{A}_t(i,j) = 1\}$$

2.5 Identity Group Formation

To form unique identity groups, we iteratively merge overlapping similarity neighborhoods. We initialize:

$$\mathcal{I}_t = \left[\mathcal{I}_t^{(1)},\ \mathcal{I}_t^{(2)},\ \ldots\right]$$

Each new set $\mathcal{G}_t^{(i)}$ is added to $\mathcal{I}_t$ if it is not already fully contained within an existing group:

$$\mathcal{I}_t^{(k)} \cap \mathcal{G}_t^{(i)} = \mathcal{I}_t^{(k)} \Rightarrow \text{do not add}$$

Otherwise, it becomes a new identity group.

2.6 Identity Assignment and Database Update

For each identity group $\mathcal{I}_t^{(i)}$, we compute the average feature vector:

$$\bar{\mathbf{f}}_i = \frac{1}{|\mathcal{I}_t^{(i)}|} \sum_{j \in \mathcal{I}_t^{(i)}} \mathbf{f}_t^{(j)}$$

2.7 Matching to Existing Identities

We compare $\bar{\mathbf{f}}_i$ to each identity in the database $\mathcal{D}$, using cosine similarity:

$$k^* = \arg\max_k \cos(\bar{\mathbf{f}}_i, \mathbf{r}^{(k)}) \quad \text{where} \quad \cos(\cdot) \geq \theta_1 = 0.85$$

If matched, the representative vector $\mathbf{r}^{(k^*)}$ is updated:

$$\mathbf{r}^{(k^*)} \leftarrow \mathbf{r}^{(k^*)} + \bar{\mathbf{f}}_i$$

2.8 Assigning IDs to Group Members

Each member $\mathbf{f}_j$ of the group is compared with historical features of identity k^*:

$$\cos(\mathbf{f}_j, \mathbf{d}_\ell) \geq \theta_2 = 0.93$$

If a match is found, assign ID $\ell + 1$. If not, and if $n < 12$:

- Add $\mathbf{f}_j$ to $\mathcal{D}^{(k^*)}$
- Recompute the average feature
- Assign ID $n + 1$

3. Improvements and Better Approach (NOT CURRENTLY IMPLEMENTED)

While the current system works well under simple conditions, it can struggle in situations with occlusion, long absence from the frame, or very similar appearances between players. To improve robustness and accuracy, a better approach would combine appearance-based re-identification with motion tracking using a Kalman Filter.

1. Using a Kalman Filter for Temporal Tracking

A Kalman Filter is a recursive algorithm that predicts the future position of an object based on its past movements. It can help us:

- Track players across frames, even when their detections are momentarily lost.
- Predict their location when they are partially occluded.
- Handle fast motion or temporary disappearance from the camera view.

Each detected player would be assigned a Kalman Filter tracker. At each frame:

1. The filter predicts the player's next position.
2. The YOLO detector gives actual detections.
3. The prediction and actual detection are matched using Intersection-over-Union (IoU).

Timeout logic: If a player is not detected for several frames (e.g., 10 frames), we mark their tracker as "inactive" or "lost". If a new player appears later and their location and appearance match a previously lost player, we can reassign the same ID.

2. Combining Motion and Appearance Features

Motion-based tracking alone is not sufficient, especially in sports where players often cross paths or overlap. Thus, we also incorporate appearance-based features using the same feature extractor (`osnet_x1_0`).

Each tracker's state includes:

- The Kalman Filter position.
- The last known feature vector.
- A short history of previous features (optional).

When a detection is matched to a track:

- We check if the spatial location matches using Kalman prediction.
- We compare the extracted appearance feature to the track's stored feature.
- If both are similar enough (e.g., cosine similarity > 0.85), we assign the detection to the track.

3. Benefits of This Combined Approach

- **Better Re-Identification:** Tracks that go missing for a few seconds can still be recovered.
- **Reduced Identity Switches:** Using both direction (motion) and appearance reduces false matches.
- **Efficient Tracking:** Fewer comparisons are needed as we match only nearby candidates.
- **Smoother IDs:** IDs are retained even across temporary occlusions or edge-of-frame exits.

4. Summary

In short, a better system would:

1. Use a Kalman Filter for tracking the motion of each detected player.
2. Use deep appearance features to distinguish between visually similar players.
3. Match new detections to existing tracks using both position and feature similarity.
4. Handle lost-and-reappeared players using timeout and similarity thresholds.

This combination of temporal tracking and visual identity embedding is common in state-of-the-art multi-object tracking systems and would greatly improve the robustness of the re-identification pipeline in a sports setting.

4. Challenges Faced

1. Time Constraints and Environment

One of the biggest challenges during this assignment was the time constraint. Although one week is typically sufficient for a problem of this scale, the assignment happened to coincide with my family trip to Goa. Despite being on vacation, I tried to extract 45 minutes to an hour each day to work on the assignment.

Additionally, this report itself is being written while returning to my hometown, with minimal internet access. As a result, verifying resources and testing some modules in real-time was difficult. Nevertheless, I ensured consistent progress was made every day, albeit slowly.

2. New Domain and Limited Prior Exposure

Another difficulty was the novelty of the problem. Re-identification and multi-object tracking in sports videos was an area I had not worked on before. To familiarize myself, I conducted as much research as possible within the limited timeframe.

Some of the helpful resources I came across include:

- **SoccerNet Challenge:** A benchmarking competition for soccer video understanding.
- **TorchREID:** A PyTorch-based library for person re-identification.
- **BoxMOT:** A minimal yet efficient framework for box-based multi-object tracking.

These references helped me understand how re-ID systems are evaluated and implemented, even though integrating them directly was beyond the scope of this assignment.

3. Efficiency and Real-Time Processing

One of the key technical challenges was to make the system efficient enough to be executed almost every frame. This meant minimizing the number of computations while keeping accuracy intact.

To start with, I prototyped the basic logic using simple `for`-loops for clarity. Once the logic was verified, I optimized performance by replacing iterative operations with linear algebra tricks, such as:

- Using `np.matmul` for bulk dot product calculations,
- Applying `np.outer` and broadcasting for matrix construction,
- Avoiding Python loops wherever vectorized NumPy operations were possible.

Both the YOLOv11 model and the OSNet feature extractor were loaded on the GPU, which significantly improved inference time and allowed me to experiment more quickly with frame-level processing.