VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Image Processing and Computer Vision - CO3057

Assignment Report

Using YOLO in Image Processing for Soccer Video Analysis

Supervising Lecturer: Dr. Nguyễn Đức Dũng

Student: Đinh Xuân Quyết 2212854

HO CHI MINH CITY, MARCH 2025

Mục lục

1 Introduction **5**

2 Theoretical Foundation **5**

2.1 YOLO: Object Detection 5
2.1.1 How YOLO Works 6
2.1.2 Advantages of YOLO 6
2.1.3 Changing coordinate xyxy to xywh in Yolo 6
2.1.4 Conversion Formula 7
2.2 Roboflow: Dataset Management 7
2.2.1 Features of Roboflow 7
2.2.2 Using Roboflow 7
2.3 OpenCV: `cv2` 8
2.3.1 Key Functions in `cv2` 8
2.4 Supervision 8
2.4.1 Types of Supervision 8
2.4.2 Supervision in Object Detection 9
2.5 Theoretical Basis of k-Means Clustering 9
2.6 Perspective Transformer 10
2.6.1 Homography Transformation 10
2.6.2 Transformation Process 10
2.6.3 Applications 11

3 Objectives and Desired Outcomes **11**

4 Analysis **13**

4.1 Object Detection (YOLO) and Tracking 13
4.1.1 Training the Model 13
4.2 Tracker Class Description 15
4.2.1 Features 15
4.2.2 Key Methods 16
4.2.3 Use Case 16
4.3 Player Color Assignment 16
4.3.1 Using k-Means for Team Classification by Jersey Color 16
4.4 Camera Movement Estimator 17
4.4.1 Initialization 17

4.4.2 Camera Movement Calculation 18
4.4.3 Position Adjustment 18
4.4.4 Visualization 18
4.5 Perspective Transformer 18
4.5.1 Initialization 19
4.5.2 Point Transformation 20
4.5.3 Adding Transformed Positions to Tracks 20
4.6 Speed and Distance Estimator 21
4.6.1 Distance Calculation 21
4.6.2 Speed Calculation 21
4.6.3 Tracking Speed and Distance in Video Frames 22

5 Future Improvements **22**

6 Source code in Github **22**

1 Introduction

In recent years, the application of computer vision techniques has become increasingly prominent in sports analytics, particularly in soccer. One of the most effective methods for analyzing soccer videos is object detection, which allows for the identification and tracking of players, the ball, and other key elements in a game. YOLO (You Only Look Once) is a state-of-the-art real-time object detection algorithm that has gained widespread adoption due to its speed and accuracy. YOLO's ability to detect multiple objects in an image or video frame in a single pass makes it an ideal choice for soccer video analysis, where multiple objects (players, ball, goalposts) need to be tracked simultaneously.

To enhance the efficiency of the analysis process, tools like Roboflow are often used to manage datasets, including uploading them in various formats, annotating, and preparing them for training machine learning models. Roboflow supports a variety of data formats, enabling seamless integration with YOLO for object detection tasks.

Furthermore, OpenCV (cv2) plays a crucial role in processing and manipulating video frames. OpenCV provides a robust set of tools for handling image and video data, making it an essential library for video analysis tasks such as frame extraction, object tracking, and visualization.

This work explores the application of YOLO for object detection in soccer video analysis, focusing on the detection and tracking of players and the ball, as well as the integration of tools like Roboflow for dataset management and OpenCV for video processing. By leveraging these technologies, it is possible to automate the extraction of meaningful insights from soccer games, such as player movements, ball possession, and game dynamics, which can be valuable for coaches, analysts, and fans alike.

In this project, I utilize essential tools to perform various image processing tasks, including object detection (YOLO) and tracking, player color assignment, ball position interpolation, camera movement estimation, perspective transformation, and speed and distance estimation.

2 Theoretical Foundation

2.1 YOLO: Object Detection

YOLO (You Only Look Once) is a state-of-the-art object detection algorithm that performs object localization and classification in a single forward pass of a neural network. It is known for its high speed and accuracy, making it suitable for real-time applications.

2.1.1 How YOLO Works

- **Grid Division:** The input image is divided into an $S \times S$ grid. Each grid cell predicts bounding boxes and their confidence scores.
- **Bounding Box Prediction:** Each grid cell predicts B bounding boxes, where each box includes:
  - Coordinates (x, y): Center of the box relative to the grid cell.
  - Dimensions (w, h): Width and height of the box, normalized by the image size.
  - Confidence Score: Indicates the likelihood of an object and the accuracy of the box.
- **Class Prediction:** Each grid cell predicts C class probabilities, indicating the likelihood of each class.
- **Non-Maximum Suppression (NMS):** Overlapping boxes are filtered using IoU (Intersection over Union) to retain the most confident predictions.

2.1.2 Advantages of YOLO

- Real-time performance with high FPS (frames per second).
- Unified architecture simplifies the detection pipeline.
- Generalizes well to new datasets.

2.1.3 Changing coordinate xyxy to xywh in Yolo

In object detection, bounding boxes are often represented in different formats. Two common formats are:

- `xyxy`: Specifies the top-left $(x_{\text{min}}, y_{\text{min}})$ and bottom-right $(x_{\text{max}}, y_{\text{max}})$ corners of the bounding box.
- `xywh`: Specifies the center coordinates $(x_{\text{center}}, y_{\text{center}})$ and the width and height (w, h) of the bounding box.

This document provides the formula and Python code to convert bounding boxes from `xyxy` to `xywh` format.

2.1.4 Conversion Formula

Given a bounding box in `xyxy` format:

$$x_{\text{min}}, y_{\text{min}}, x_{\text{max}}, y_{\text{max}}$$

The conversion to `xywh` format is done as follows:

$$x_{\text{center}} = \frac{x_{\text{min}} + x_{\text{max}}}{2}$$

$$y_{\text{center}} = \frac{y_{\text{min}} + y_{\text{max}}}{2}$$

$$w = x_{\text{max}} - x_{\text{min}}$$

$$h = y_{\text{max}} - y_{\text{min}}$$

2.2 Roboflow: Dataset Management

Roboflow is a platform that simplifies dataset management for computer vision tasks, including object detection, classification, and segmentation.

2.2.1 Features of Roboflow

- **Dataset Upload:** Supports uploading datasets in various formats (e.g., COCO, Pascal VOC, YOLO).
- **Dataset Preprocessing:** Offers tools for resizing, augmenting, and splitting datasets.
- **Dataset Management:** Enables annotation editing, dataset versioning, and format conversion.
- **Export Formats:** Supports exporting datasets in multiple formats compatible with popular frameworks (e.g., TensorFlow, PyTorch, YOLO).

2.2.2 Using Roboflow

1. Create a project on Roboflow and upload your dataset in its current format.
2. Use the preprocessing tools to augment and prepare your dataset.
3. Export the dataset in the desired format for training your model.

2.3 OpenCV: `cv2`

OpenCV (Open Source Computer Vision Library) is a popular library for image and video processing. The `cv2` module in Python provides a comprehensive set of tools for computer vision tasks.

2.3.1 Key Functions in `cv2`

- **Reading and Writing Images:**
  - `cv2.imread(path)`: Reads an image from the specified path.
  - `cv2.imwrite(path, image)`: Saves an image to the specified path.
- **Video Processing:**
  - `cv2.VideoCapture(path)`: Captures video from a file or camera.
  - `cv2.VideoWriter(path, codec, fps, size)`: Writes video frames to a file.
- **Image Transformations:**
  - `cv2.resize(image, size)`: Resizes an image.
  - `cv2.cvtColor(image, code)`: Converts an image to a different color space.

2.4 Supervision

Supervision refers to the process of monitoring and managing the performance of models or systems during training and evaluation. It ensures that the model learns effectively and generalizes well to unseen data.

2.4.1 Types of Supervision

- **Supervised Learning:** Uses labeled data for training. The model learns to map inputs to outputs based on the provided labels.
- **Unsupervised Learning:** Relies on unlabeled data to find patterns or structure in the data.
- **Semi-Supervised Learning:** Combines labeled and unlabeled data for training.

2.4.2 Supervision in Object Detection

In the context of object detection, supervision involves:

- Annotating datasets with bounding boxes and class labels.
- Monitoring model performance metrics (e.g., mAP, precision, recall).
- Fine-tuning hyperparameters to improve detection accuracy.

2.5 Theoretical Basis of k-Means Clustering

k-Means is an unsupervised learning algorithm that partitions a dataset $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ into k clusters $\{C_1, C_2, \ldots, C_k\}$. The algorithm iteratively minimizes the within-cluster sum of squares (WCSS), defined as:

$$J = \sum_{j=1}^{k} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2,$$

where μ_j is the centroid of cluster C_j.

The algorithm proceeds as follows:

1. **Initialization:** Randomly select k initial centroids $\{\mu_1, \mu_2, \ldots, \mu_k\}$.
2. **Assignment:** Assign each data point x_i to the nearest centroid μ_j based on the Euclidean distance:

$$C_j = \{x_i : \|x_i - \mu_j\|^2 \leq \|x_i - \mu_l\|^2, \forall l = 1, \ldots, k\}.$$

3. **Update:** Recompute centroids as the mean of points in each cluster:

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i.$$

4. **Convergence:** Repeat the assignment and update steps until centroids stabilize or a maximum number of iterations is reached.

k-Means is computationally efficient and widely used for clustering due to its simplicity. However, it requires specifying k in advance and may converge to local minima depending on the initialization of centroids.

2.6 Perspective Transformer

The Perspective Transformer is a technique used to correct or modify the perspective of an image or video frame. This method is particularly useful in computer vision applications where the goal is to map points from one perspective to another, such as in top-down views or correcting distorted images. The key idea is to apply a homography transformation, which maps the points in the original perspective to a new perspective, typically involving a projective transformation.

2.6.1 Homography Transformation

A homography is a transformation that relates the coordinates of points in one plane to another plane. It is represented by a 3×3 matrix H, which is used to perform a projective transformation. Given a point (x, y) in the original image, the transformed point (x', y') is calculated as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where: - (x, y) are the coordinates of a point in the original image. - (x', y') are the coordinates of the transformed point in the new perspective. - H is the homography matrix.

2.6.2 Transformation Process

The process of perspective transformation involves the following steps:

- **Define Source and Destination Points:** First, a set of corresponding points from the original image (source points) and the target perspective (destination points) are selected. These points should ideally be chosen from the corners or distinct features in the image.
- **Compute the Homography Matrix:** Using the selected points, a homography matrix H is computed. This matrix can be calculated using methods such as Direct Linear Transformation (DLT) or other numerical techniques.
- **Apply the Transformation:** Once the homography matrix is computed, it is used to transform every point in the original image to the new perspective. This is done by multiplying the point coordinates with the homography matrix.

- **Warp the Image:** The entire image is transformed by applying the homography to each pixel. This results in an image that has been warped to the new perspective.

2.6.3 Applications

Perspective transformation is widely used in various computer vision tasks, including:

- **Bird's Eye View Generation:** Converting a perspective view into a top-down view, often used in autonomous driving and robotics.
- **Rectifying Distorted Images:** Correcting the distortion caused by camera angle or lens effects.
- **Image Stitching:** Aligning multiple images to create panoramas by transforming the perspectives of individual images to match.
- **Augmented Reality:** Mapping virtual objects onto real-world images, where perspective correction is crucial for realistic rendering.

3 Objectives and Desired Outcomes

In this project, I aim to apply image processing techniques and utilize the necessary tools to analyze machine learning models for short video clips of soccer matches. This analysis can provide commentators and analysts with deeper insights into the game, helping them gain a more comprehensive understanding of the match

I am using the dataset with link https://universe.roboflow.com/roboflow-jvuqo/football-players-detection-3zvbc/dataset/1 on Roboflow with titled **Football-Players-Detection**, is designed for the detection of football players in images. The dataset is available under the `CC BY 4.0` license and can be accessed through the following Roboflow project:

- **Project Name**: football-players-detection-0zybd
- **Roboflow Workspace**: roboflowjvuqo-qjctr
- **Version**: 1

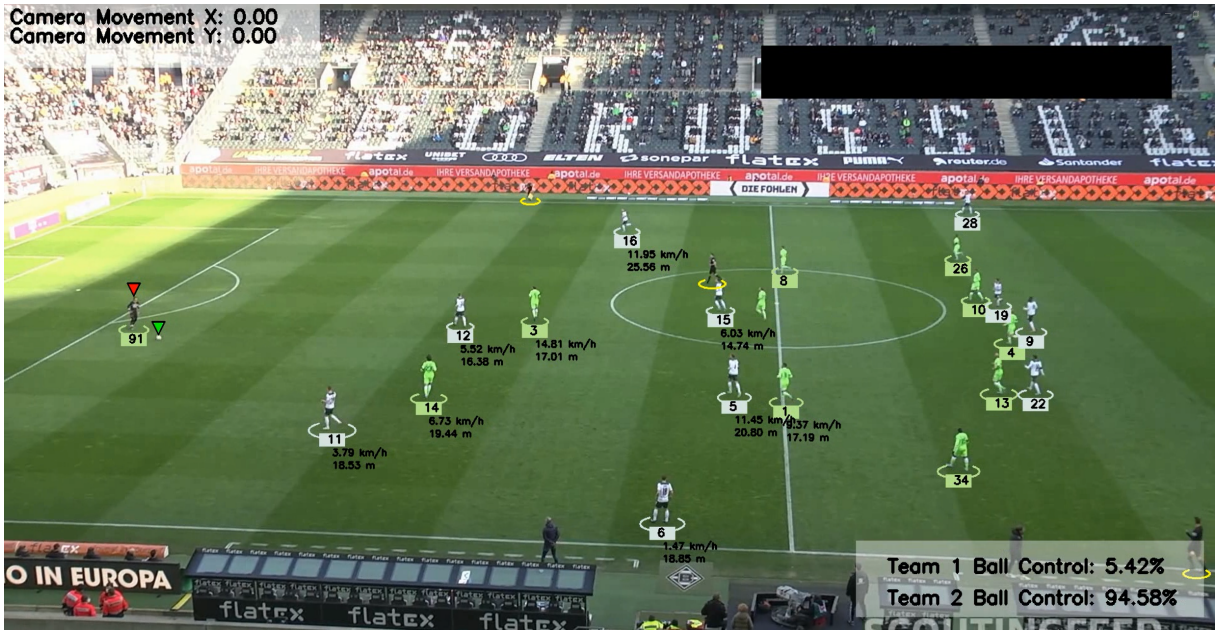
Dataset Structure:

- **Training Data**: football-players-detection-1/train/images

- **Validation Data**: football-players-detection-1/valid/images
- **Test Data**: ../test/images

The image processing techniques applied in this project include:

- **Object Detection (YOLO) and Tracking**: Detecting and tracking objects such as players and the ball in soccer videos.
- **Player Color Assignment**: Identifying and assigning colors to players to differentiate between teams.
- **Ball Interpolation**: Estimating the ball's position in frames where it may be missing or unclear.
- **Camera Movement Estimator**: Analyzing and adjusting for camera movement to stabilize tracking data.
- **Perspective Transformer**: Applying perspective transformations to correct for changes in view and angle in the video.
- **Speed and Distance Estimator**: Calculating the speed and distance traveled by players and the ball during the game.



Hình 3.1: Desired Outcomes

4 Analysis

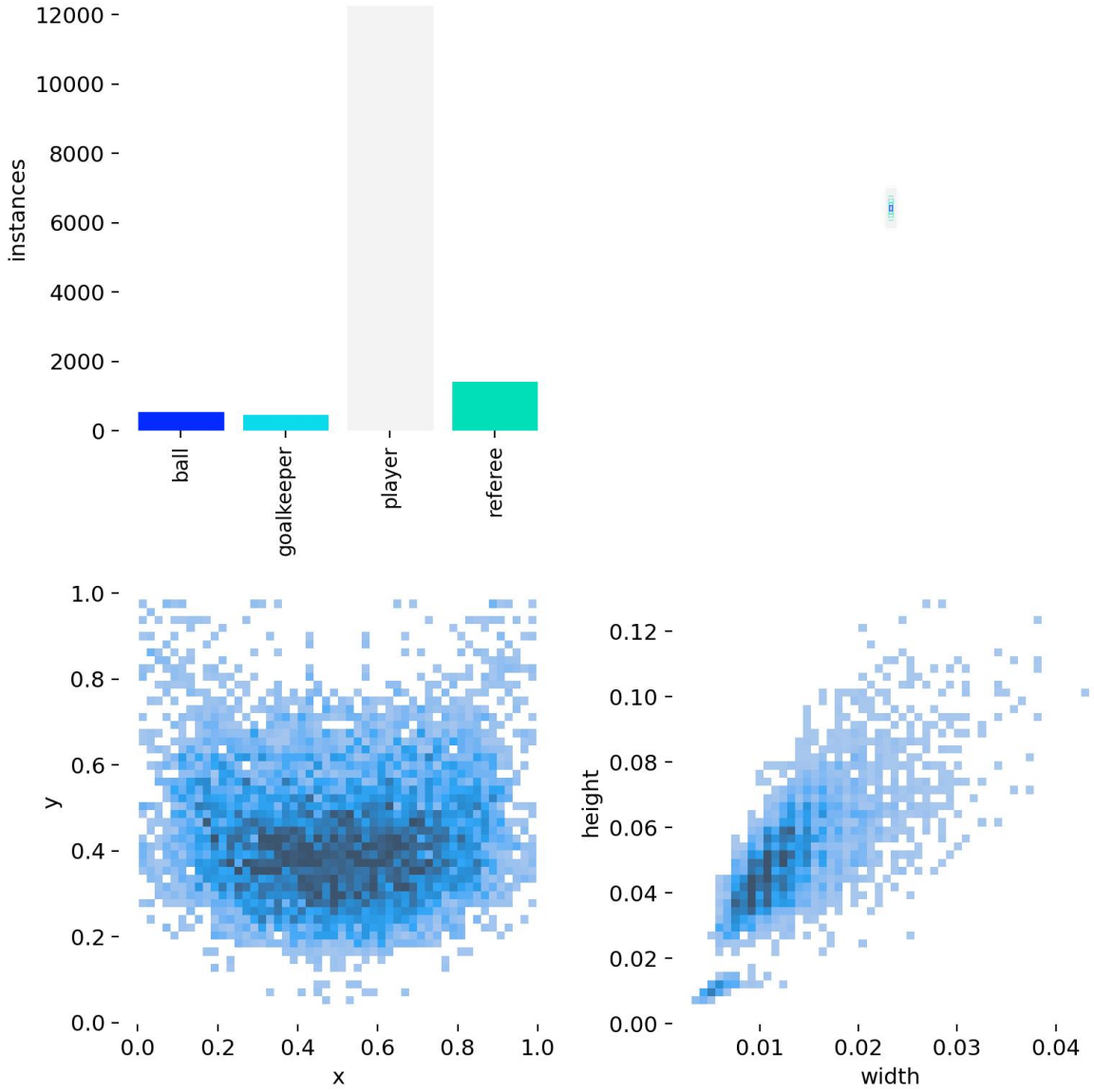
4.1 Object Detection (YOLO) and Tracking

4.1.1 Training the Model

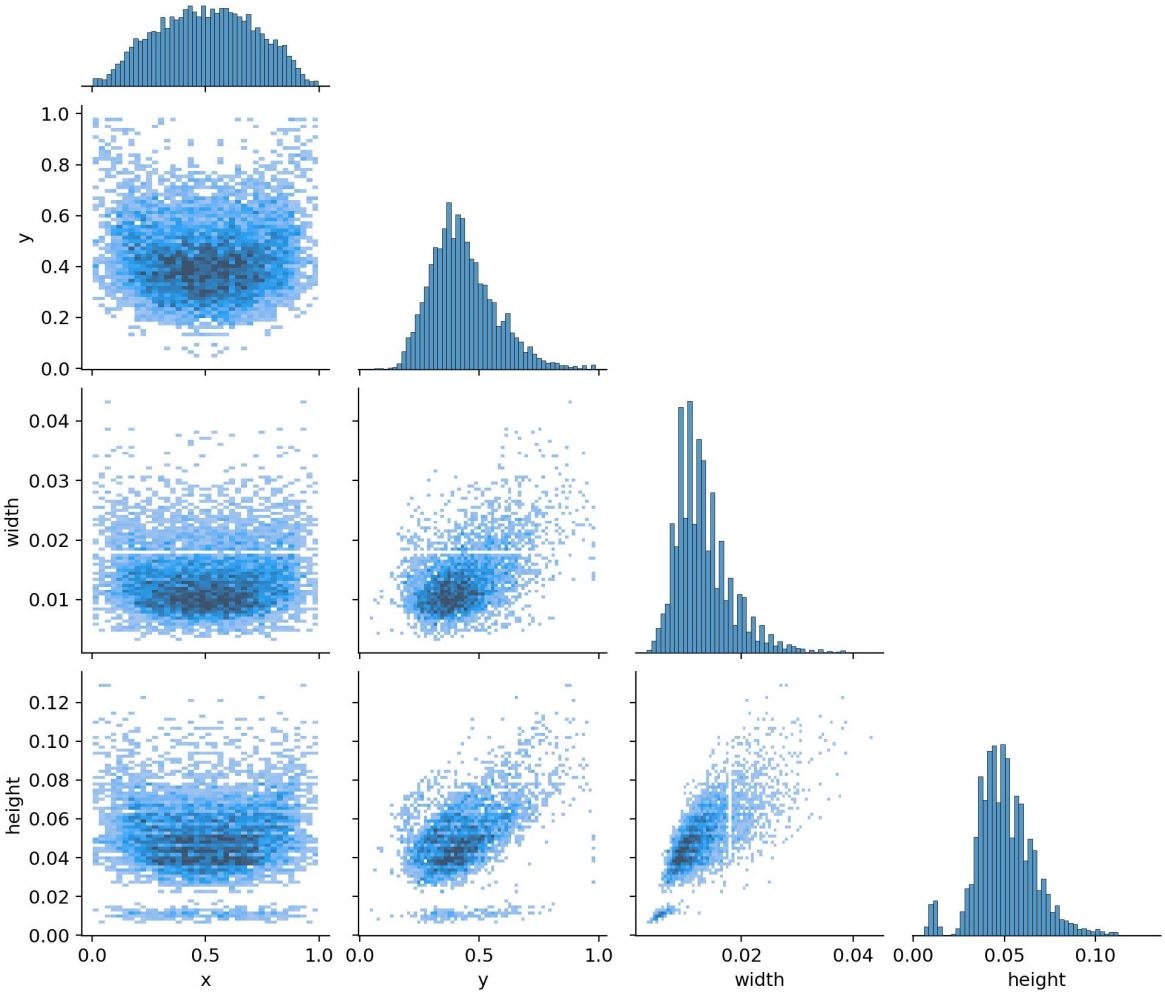
The YOLOv5 model was trained using the pre-trained `yolov5x.pt` model. The training process was conducted for 100 epochs with an image size of 640x640. The model was trained to detect football players in images, using the dataset provided by Roboflow.

The training resulted in two key models:

- **best.pt**: The model that achieved the best performance on the validation dataset.
- **last.pt**: The model from the final epoch of training.



Hình 4.1: Labels when analysis



Hình 4.2: Labels correlogram when analysis

4.2 Tracker Class Description

The `Tracker` class facilitates object detection, tracking, and annotation for football analysis using the YOLO model and ByteTrack algorithm.

4.2.1 Features

- **Object Categorization:** Detects and tracks players, referees, and the ball. Converts "goalkeeper"detections into the "player"category for consistency.

- **Position Tracking:** Adds positions (center or foot) to tracked objects. Interpolates missing ball positions for smooth trajectories.
- **Visualization:**
  - Draws ellipses for players and referees, and triangles for the ball or players holding the ball.
  - Displays team ball control percentages with overlays.
- **Stub Support:** Saves and loads precomputed tracking data for faster testing and debugging.

4.2.2 Key Methods

- `add_position_to_tracks`: Computes and adds positional data to tracks.
- `interpolate_ball_positions`: Interpolates missing ball position data.
- `detect_frames`: Detects objects in video frames using YOLO.
- `get_object_tracks`: Tracks players, referees, and the ball across frames.
- `draw_annotations`: Annotates video frames with tracks, team colors, and ball control stats.

4.2.3 Use Case

The `Tracker` class is suitable for analyzing football games, including player movement, ball tracking, and team performance evaluation.

4.3 Player Color Assignment

4.3.1 Using k-Means for Team Classification by Jersey Color

The k-Means clustering algorithm is applied to distinguish players' teams based on their jersey colors in a soccer game image. The process is outlined as follows:

Image Preprocessing

- The input image is read and converted to the RGB color space.
- Only the top half of the image, which typically contains most players, is extracted for analysis.

Clustering

- The image is reshaped into a 2D array of pixel RGB values.
- k-Means clustering is performed with two clusters ($k = 2$) to classify pixels based on color similarity.

Cluster Identification

- The clustered image is analyzed by checking the cluster labels of the four corners, assuming non-player regions (e.g., field or background) dominate these areas.
- The cluster with the majority presence in the corners is identified as the non-player cluster, while the other represents players.

Output

- The RGB values of the cluster center corresponding to the players are extracted, representing the dominant jersey color of the team.

4.4 Camera Movement Estimator

The CameraMovementEstimator class is designed to estimate and adjust the movement of the camera across a sequence of frames in a video. The process involves detecting the movement of features between consecutive frames and adjusting object positions accordingly. The steps are as follows:

4.4.1 Initialization

- The class is initialized with a frame, and the minimum movement distance is set to 5 pixels.
- The Lucas-Kanade optical flow parameters (`lk_params`) are set, including the window size, maximum pyramid levels, and criteria for termination.
- A mask is applied to the first frame to limit feature detection to specific regions, excluding the edges of the frame.
- The feature detection parameters are defined, including the maximum number of corners to detect, quality level, minimum distance, and block size.

4.4.2 Camera Movement Calculation

- The method `get_camera_movement` computes the camera movement by calculating the optical flow between consecutive frames.
- The first frame is converted to grayscale, and the good features to track are identified using the `cv2.goodFeaturesToTrack` function.
- For each subsequent frame, optical flow is calculated using `cv2.calcOpticalFlowPyrLK`, and the movement is estimated by comparing the displacement of features between frames.
- The maximum movement distance is used to detect significant camera movement, and the movement is stored for each frame.
- The calculated camera movement is saved to a stub file for future use.

4.4.3 Position Adjustment

- The method `add_adjust_positions_to_tracks` adjusts the positions of objects in the video frames based on the estimated camera movement.
- For each object and frame, the object's position is adjusted by subtracting the corresponding camera movement vector.

4.4.4 Visualization

- The method `draw_camera_movement` visualizes the camera movement on the frames.
- A semi-transparent overlay is drawn on the top-left corner of each frame, displaying the X and Y components of the camera movement.
- The adjusted frames are returned as output.

4.5 Perspective Transformer

The View Transformer algorithm is designed to transform the coordinates of points from a perspective view to a top-down view using a perspective transformation matrix. This is particularly useful in applications like sports analytics, where the objective is to map points from a camera's perspective to a standardized court layout. The steps of the algorithm are as follows:

4.5.1 Initialization

The algorithm initializes two sets of points:

- **Pixel Vertices:** These are the coordinates of four points in the image that correspond to the corners of the court in the camera's perspective. These points are manually selected from the image.
- **Target Vertices:** These are the coordinates of the corresponding points in the top-down view, which represent the corners of the court in a predefined coordinate system (in this case, the court's dimensions in meters).

The coordinates are stored as NumPy arrays, and the data types are converted to `float32` to ensure compatibility with OpenCV functions. The transformation matrix is then computed using the `cv2.getPerspectiveTransform` function, which generates the homography matrix that will be used for transforming points from the original perspective to the target view.

```
class ViewTransformer():
    def __init__(self):
        court_width = 68
        court_length = 23.32

        self.pixel_vertices = np.array([[110, 1035],
                                        [265, 275],
                                        [910, 260],
                                        [1640, 915]])

        self.target_vertices = np.array([
            [0, court_width],
            [0, 0],
            [court_length, 0],
            [court_length, court_width]
        ])

        self.pixel_vertices = self.pixel_vertices.astype(np.float32)
        self.target_vertices = self.target_vertices.astype(np.float32)

        self.persepctive_trasnformer = cv2.getPerspectiveTransform(self.pixel_verti
```

4.5.2 Point Transformation

The `transform_point` method is used to apply the perspective transformation to individual points. The method checks if the point lies inside the defined polygon (the court's boundary in the image) using `cv2.pointPolygonTest`. If the point is inside, it reshapes the point to the required format and applies the transformation using `cv2.perspectiveTransform`. The transformed point is then returned.

```
def transform_point(self, point):
    p = (int(point[0]), int(point[1]))
    is_inside = cv2.pointPolygonTest(self.pixel_vertices, p, False) >= 0
    if not is_inside:
        return None

    reshaped_point = point.reshape(-1, 1, 2).astype(np.float32)
    tranform_point = cv2.perspectiveTransform(reshaped_point, self.persepctive_trasn
    return tranform_point.reshape(-1, 2)
```

4.5.3 Adding Transformed Positions to Tracks

The `add_transformed_position_to_tracks` method iterates through a dictionary of tracked objects and their positions across frames. For each position, it calls `transform_point` to get the transformed coordinates. If the point is successfully transformed, the new position is added to the track data as `position_transformed`. This allows for the visualization and analysis of object movements in the top-down view, which is useful for tasks such as player tracking in sports games.

```
def add_transformed_position_to_tracks(self, tracks):
    for object, object_tracks in tracks.items():
        for frame_num, track in enumerate(object_tracks):
            for track_id, track_info in track.items():
                position = track_info['position_adjusted']
                position = np.array(position)
                position_trasnformed = self.transform_point(position)
                if position_trasnformed is not None:
                    position_trasnformed = position_trasnformed.squeeze().tolist()
                tracks[object][frame_num][track_id]['position_transformed'] = positi
```

4.6 Speed and Distance Estimator

The algorithm for calculating speed and distance is designed to track the movement of objects (such as players or the ball) across frames in a video. By using the positions of the objects in consecutive frames, the algorithm computes the distance traveled and the speed of the object. The steps of the algorithm are as follows:

4.6.1 Distance Calculation

The distance traveled by an object between two consecutive frames is calculated using the Euclidean distance formula. Given two points, (x_1, y_1) and (x_2, y_2), the distance d is computed as:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

In the code, this calculation is done using the function `measure_distance`, which takes the coordinates of two points and returns the Euclidean distance between them.

```
def measure_distance(point1, point2):
    return np.sqrt((point2[0] - point1[0])**2 + (point2[1] - point1[1])**2)
```

4.6.2 Speed Calculation

The speed of an object is defined as the distance traveled over time. In the context of video frames, time is represented by the frame rate (frames per second, FPS). The speed v is computed as:

$$v = \frac{d}{\Delta t}$$

where d is the distance traveled between two frames, and Δt is the time between those frames, which is the inverse of the frame rate (i.e., $\Delta t = \frac{1}{\text{FPS}}$).

In the code, the speed is calculated by dividing the distance by the time between frames. The `measure_xy_distance` function computes the difference in coordinates between two consecutive points, and the speed is then derived by dividing the distance by the frame time.

```
def measure_xy_distance(point1, point2):
    return point2[0] - point1[0], point2[1] - point1[1]
```

4.6.3 Tracking Speed and Distance in Video Frames

The algorithm iterates through the video frames, calculates the distance between the current and previous frame positions, and computes the speed by dividing the distance by the time difference (frame rate). These values are stored for each object in the video, and the speed and distance traveled can be visualized or used for further analysis.

```
def calculate_speed_and_distance(frames, tracks, fps):
    for object, object_tracks in tracks.items():
        for frame_num, track in enumerate(object_tracks):
            for track_id, track_info in track.items():
                position = track_info['position']
                if frame_num > 0:
                    prev_position = object_tracks[frame_num - 1][track_id]['position']
                    distance = measure_distance(position, prev_position)
                    time = 1 / fps
                    speed = distance / time
                    track_info['distance'] = distance
                    track_info['speed'] = speed
```

5 Future Improvements

- Improve performance for video analysis as video processing takes a significant amount of time for detection.
- Use a variety of datasets to enhance the model's robustness.
- Apply image processing techniques to add more features, making it more convenient for users such as football commentators and post-match analysts.
- Enhance efficiency by utilizing algorithms in image processing for detection, training, and perspective transformation.

6 Source code in Github

https://github.com/Qdaika22/ipcv_ass.git

Tài liệu

[1] Lecture on Digital Image Processing and Computer Vision

[2] Szeliski, R. (2020). Computer vision: Algorithms and applications (2nd ed.). Springer.

[3] Tutorial: https://www.youtube.com/watch?v=neBZ6huolkg&t=2076s