École Polytechnique Fédérale de Lausanne

Master Semester Project

Report

Automatic tracking, event detection and visualization in sports video for summarization and statistical analysis

Student : Mr. Vincent Rinaldi

Supervisor : Mr. Adam Scholefield

January 15, 2021

Table of contents

Introduction……..……..……..3
- Goal of the Project……..……4
- Yolov4……..……4
- DeepSort..……...……..……..……7
- DeepBall……..……...……..8
- Related Work..……..……..……....12

Methodology..……..……..……..12
- Preliminary steps…….……..12
  - Input & output video settings…….……...13
  - Landmark areas, polygons and matrices definition……...14
  - Computation lists and frame folders creation……...……17
- Input videos processing…….……...18
  - Background subtraction……..…..……19
  - Ball detection process……..……..…20
  - Jersey color tracking and team association……..…..……..23
  - Bboxes handling and coordinates conversion……..…..25
- Output video writing……..……27
  - Frame selection……..…..…..27
  - Statistics computation……..……..29
  - Mini-map plot and heatmap values update……..……...31
- Heatmap build……32

Experimental Results…….....……..…..……..…...35

Conclusion……..……..……43

Acknowledgments..……..……..……45

Citations and References.……....……....……..……45

1. Introduction

The influence of data analytics has grown in many domains like healthcare, media and sports. In the latter, the use of analytics software has strongly evolved, especially during the last two decades. Until a few years ago, it was thought that sports like football were immune to this trend. Today, the early adopters of this technology are greatly taking advantage of this huge competitive edge given by their investment in data analytics, to the point where sport clubs that are still not planning to jump on the analytics bandwagon risk being left far behind in terms of sporting result and business development.

Managers and coaches are now able to watch and review multiple games of any sport team and, at the same time, gather in-game statistics to quickly identify their different strengths and weaknesses. This facilitates decision making both during and prior to sporting events, like when planning an appropriate training program after pointing out the flaws of each player of the team in order to improve their performance in a very short time, or establishing a specific game tactic in order to take advantage of the vulnerabilities of the opposing team.

Concerning those different existing types of softwares, automated video analysis, as an example, are widely used by football teams to track the individual movement patterns of each player[1-2]. In basketball, *Intel RealSense 3D* depth cameras are used to track and analyze every shot, including trajectory and location[3], whereas statistics like player position, performance and wellness on the court are recorded thanks to wearable wristband technologies like the popular *Kinexon* tracking system[4]. Sophisticated *FIELDf/x* camera tracking systems, in Major League Baseball, transform every pitch, hit, and out into raw data in every ballpark[5]. The list goes on in hockey, tennis, and more.

Sports video analysis became even more popular with the help of modern object detection and tracking techniques. Today, many academic researches and commercial applications are focusing on player tracking and identification[6-9]. The problem of labeling and tracking of multiple targets in video, like players during a sports game, is constantly being investigated and many implementations dealing with this challenge have been presented[10-17] during the last 40 years.

1.1. Goal of the project

The goal of this project was to develop Computer Vision methods in order to be able to establish statistical analysis and summarization of a sport game through one or several video filming the given opposition with fixed camera angles. Techniques like automatic ball detection as well as labelling and tracking of players on the field were used to detect events related to specific playing movements in order to update, at each frame, the statistics we aim to compute. Some statistics will be immediately displayed on the screen while the video of the game is playing for better visualization and extended game analysis, while others will be stored on the filesystem after their full computation for later examination.

The main elements we use for the realization of this work are the modern, recent and very popular object detector and tracker, respectively called *YOLOv4*[18] and *DeepSORT*[19], as well as a sports ball detector, mainly designed for football games, that produced very interesting results on the video we were working on, named *DeepBall*[20].

1.2. YOLOv4

YOLOv4 (*YOLO* is short for *You Only Look Once*) is a state of the art one-stage object detector using deep convolutional neural networks in order to detect objects in a given image or frame. It is actually one of the fastest and most accurate existing object detection systems in the scientific field.

This real-time recognition system is able to recognize multiple objects from an image at the same time and draw a boundary box, also known as *“bounding box”*, around the detected object. This boundary box represents the portion of the image where the object is located. The object detector also predicts probabilities for the different possible classes an object can belong to. Finally, *YOLOv4* can be easily trained and deployed in a production system.

There are two types of object detection models: one-stage or two-stage. A one-stage model can detect objects without the need for a preliminary step, while a two stage detector performs, as a preliminary step, the detection and classification of regions of importance in order to see if, in these regions, an object has been detected. The advantage of a one stage detector, as *YOLOv4*, is its speed: it is able to make predictions faster which makes it more suitable for real time use. Figure 1 describes schematically the architecture of one-stage and two-stage object detectors.

YOLOv4’s architecture is composed of:

- *CSPDarknet53* as a backbone, which is a deep neural network mainly composed of convolution layers and whose main objective is to extract the essential features
- *Spatial Pyramid Pooling (SPP)* as additional module to increase the reception field and separate out the most significant context features
- *PANet* path-aggregation neck *(replacing Feature pyramid networks (FPN) used in YOLOv3*) which are extra layers, going in between the backbone and head, that are used to extract different feature maps of different stages of the backbone
- YOLOv3 head (anchor based), which is a network in charge of actually doing the detection part (classification and regression) of bounding boxes

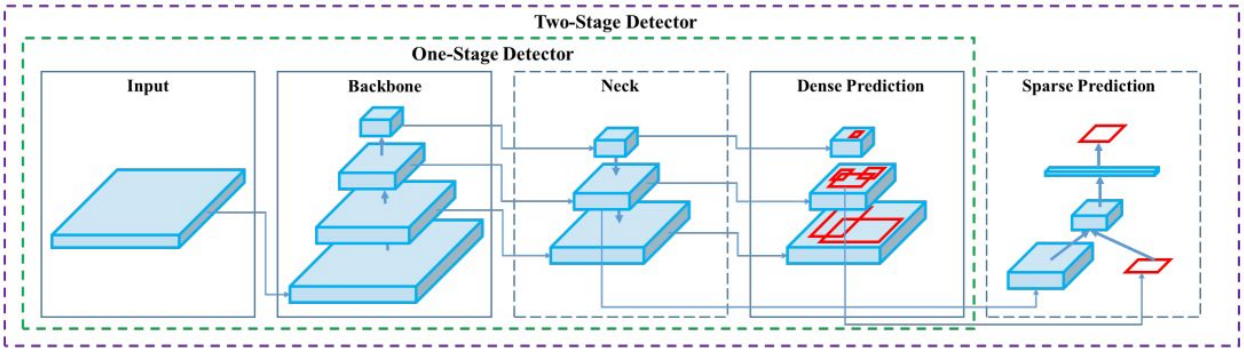

Figure 1: Comparison of one-stage and two-stage detector architectures

According to Figure 2, *YOLOv4* achieves state-of-the-art results at a real time speed on the *MS COCO* dataset with 43.5 % *AP* (Average Precision) running at 65 *FPS* (Frames Per Second) on a Tesla V100 GPU. In addition, *AP* and *FPS* increased by 10% and 12% compared to *YOLOv3.*

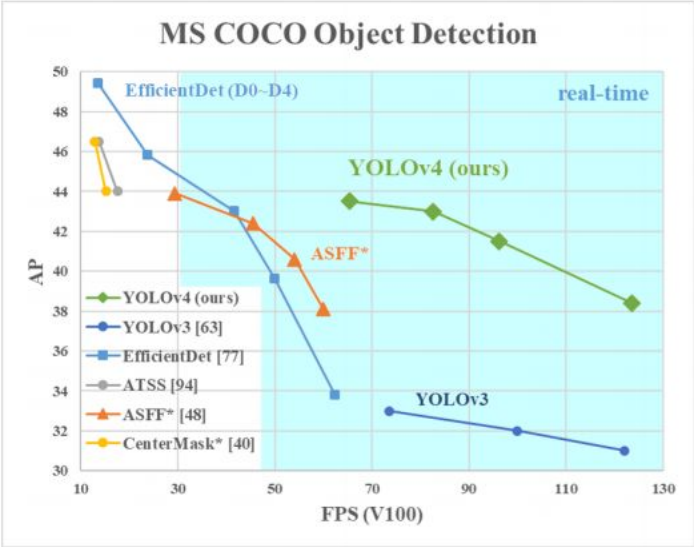

Figure 2: Comparison of *YOLOv4* and other state-of-the-art object detectors

YOLOv4, is then faster and more accurate than most of the available alternative detectors, places itself among the best object detectors of the moment, and can be considered as a futuristic recognizer.

The object detections that YOLOv4 outputs can actually be fed into *DeepSORT* in order to create a highly accurate object tracker.

1.3. DeepSORT

Popular traditional methods for objects tracking like *Joint Probabilistic Data Association Filter* (*JPDAF*), *Multiple Hypothesis Tracking* (*MHT*), or even their recent revisited version[21-22] showed promising results. However, the performance of these methods comes at increased computational and implementation complexity.

A more recent method called *Simple Online and Realtime Tracking* (*SORT*)[23], a much simpler framework, performs *Kalman filtering* in image space and frame-by-frame, and use the *Hungarian algorithm* with an association metric measuring bounding box overlap in order to compute data association. While this method produces overall good performance in terms of tracking precision and accuracy, it returns a relatively high number of identity switches (also called “id switches”), due to the fact that the association metric that is employed is only accurate when the state estimation uncertainty is low. This makes *SORT* pretty weak in situations when many occlusions occur, as they typically appear in frontal-view camera scenes.

This issue has been overcome by replacing the association metric with a more informed metric combining motion and appearance information, giving birth to *DeepSORT*, an object tracker way more robust against misses and occlusions.

Here is a summarization of the main special features of *DeepSORT*:

- It is considered as being an extension to the *SORT* object tracker
- As *SORT*, it uses *Kalman filter* for unbiased estimates of system states, and *Hungarian algorithm* for data association
- Unlike *SORT,* it uses, as distance metrics, the (squared) *Mahalanobis distance* to incorporate motion information and the brand new *Appearance descriptor* to take into account the “appearance” of the object

A pre-trained CNN is applied to compute bounding box appearance descriptors. The architecture of this network is described in Table 1.

Name	Patch Size/Stride	Output Size
Conv 1	$3 \times 3/1$	$32 \times 128 \times 64$
Conv 2	$3 \times 3/1$	$32 \times 128 \times 64$
Max Pool 3	$3 \times 3/2$	$32 \times 64 \times 32$
Residual 4	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 5	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 6	$3 \times 3/2$	$64 \times 32 \times 16$
Residual 7	$3 \times 3/1$	$64 \times 32 \times 16$
Residual 8	$3 \times 3/2$	$128 \times 16 \times 8$
Residual 9	$3 \times 3/1$	$128 \times 16 \times 8$
Dense 10		128
Batch and ℓ_2 normalization		128

Table 1: Overview of the *DeepSORT* CNN architecture

Thanks to this extension, objects can be tracked through longer periods of occlusions, which by consequences reduces the number of potential identity switches: experimental evaluation shows that identity switches have been reduced by 45% with *DeepSort*, making this object tracker a strong competitor to state-of-the-art online tracking algorithms.

1.4. DeepBall

While we could use *YOLOv4* to track the sports ball, it seems, during our experiments, that the object detector had troubles to detect the ball efficiently, certainly due to the fact that we didn't experiment with high quality videos, but also because the size of the ball was very small, and could sometimes be confused with the penalty spot.

It is already known that detecting the ball from long-shot video footage of a football game is not trivial to automate. In addition to the very small size of the ball compared to the other visible object, the ball display on the screen can become blurry and elliptical, with a shape not always circular, when it moves at high velocity.

However, it is mandatory to work with an efficient ball detector in order to compute and establish game statistics, as the ball is the main element triggering any game situations. After some research and documentation, we decided to reuse a method that has been recently developed, named *DeepBall.* Figure 3 highlights the differences in terms of performances on the detection of the ball between our own *YOLOv4* detector and *DeepBall* for 3 different frames picked in a very short time interval.

While traditional ball detection methods show their limits when it is possessed or partially occluded by a player, *DeepBall* can deal with situations where the perceived ball shape is not circular (due, for example, to the motion blur). The ball detector is also able to detect the ball when it is very close to a player's body, or even when the ball is partially occluded by it.

DeepBall, in summary, brings modifications to a one-stage object detector typical architecture and specializes it in ball detection tasks by increasing its accuracy when trying to locate small objects on the frame. The method uses *hypercolumn concept*[24], where feature maps from different hierarchy levels of the CNN are combined and jointly fed to the convolutional classification layer. This boosts detection accuracy, because larger visual context around the object of interest is taken into account and other objects on the frame that highly resemble the ball can be correctly classified.

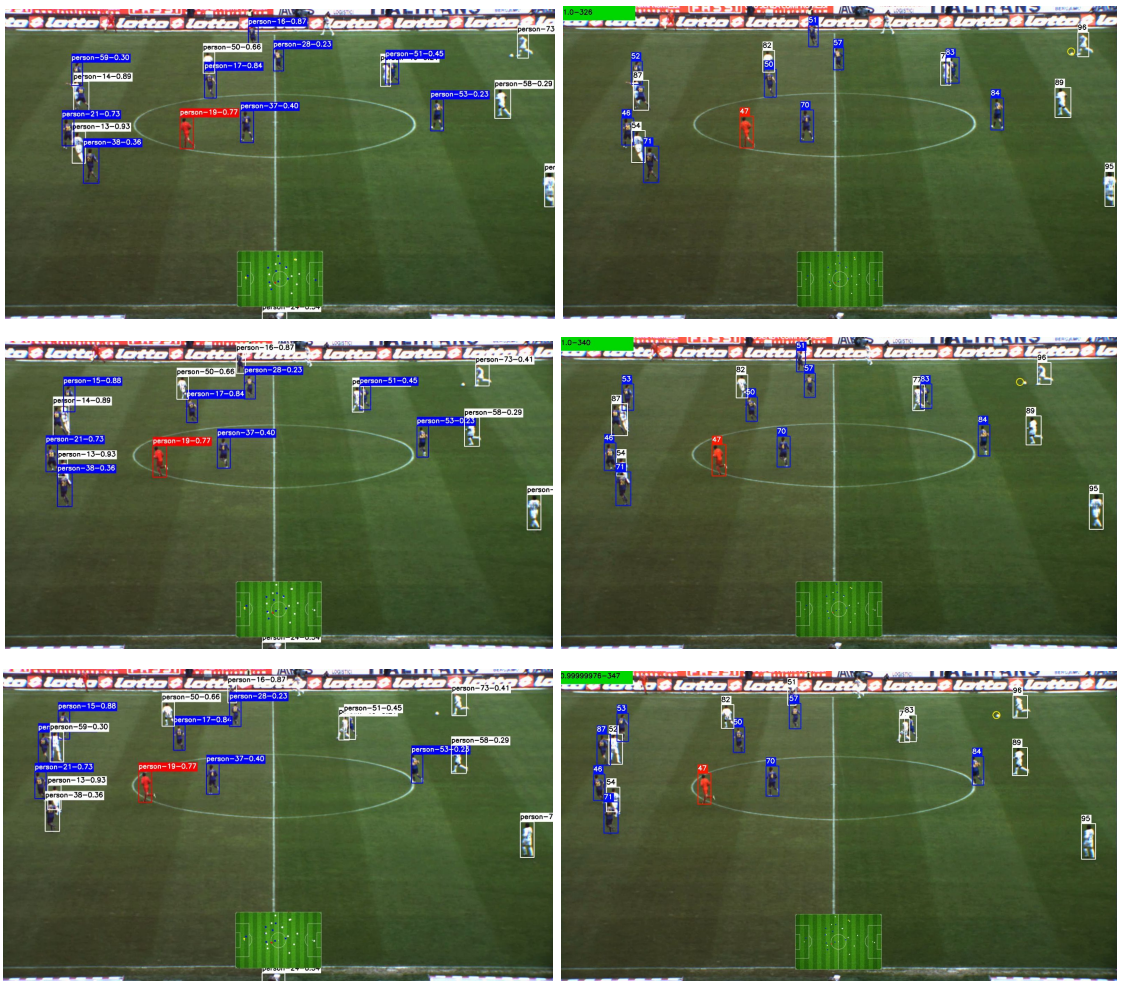

Figure 3: Example where *YOLOv4* (left) doesn't detect the ball
for frames 326, 340 and 347 while *DeepBall (right)* detects it accurately

The method achieves state-of-the-art results when tested on publicly available *ISSIA-CNR Soccer Dataset*[25]. This dataset is quite demanding in terms of performances: the videos of this dataset have a moderate quality and are a bit blurry. Also, there is a team wearing white jerseys, making it even harder to not confuse some parts of the white team players body with the ball when the latter is close (or not) to one of those players.

The model takes a video frame as input, and outputs a scaled down *ball confidence map* encoding probability of ball presence at each location. Concerning its architecture, summarized in Table 2, the input frame is processed by three convolutional blocks (named *Conv1*, *Conv2* and *Conv3* on the provided Table 2) producing convolutional feature maps with decreasing spatial resolution and increasing number of channels. The outputs, from each of these convolutional blocks, are all concatenated together, just after feature maps *Conv2* and *Conv3* get upsampled. This concatenation forms what we call a *hypercolumn*. The resulting concatenated feature map is then fed to the final fully convolutional classification layer (which is *Conv4* on Table 2). This classification block consists of two convolutional layers followed by the softmax layer.

Block	Layers	Output size
Conv1	Conv: 8 7x7 filters	
	stride 2	
	Conv: 8 3x3 filters	
	Max pool: 2x2 filter	(8, 268, 480)
Conv2	Conv: 16 3x3 filters	
	Conv: 16 3x3 filters	
	Max pool: 2x2 filter	(16, 134, 240)
Conv3	Conv: 32 3x3 filters	
	Conv: 32 3x3 filters	
	Max pool: 2x2 filter	(32, 67, 120)
Conv4	Conv: 56 3x3 filters	
	Conv: 2 3x3 Filters	(2, 268, 480)
Softmax	Softmax	(2, 268, 480)

Table 2: Overview of *DeepBall* CNN architecture
Output size has the format (number of channels, height, width) and each convolutional layer is followed by a *BatchNorm* layer and *ReLU* non-linearity

The proposed network has been able to record an average precision of 0.877 and an accuracy of 0.951 on the *ISSIA-CNR Soccer Dataset* and outperforms some other recent neural network-based ball detections methods[26-27], while having lower number of trainable parameters and significantly higher frame rate.

1.5. Related Work

The starting point of this work was the object tracking application implemented with TensorFlow and combining both *YOLOv4* object detection and *DeepSORT* object tracker, provided by Jack Wotherspoon, Microsoft Student Ambassador from Queen's University, Kingston, Canada. Jack Wotherspoon also owns a YouTube channel named *The AI Guy* dedicated to machine learning and artificial intelligence content where he explains how to use the tracking application our project relies on. This application is available on his GitHub under the repository name *theAIGuysCode/yolov4-deepsort*. The provided YOLOv4 object detector has been pre-trained with the COCO[28] (Common Objects In Context) large-scale object detection, segmentation, and captioning dataset. The input videos we used to develop our algorithm and different features are 3 sequences of 2 minutes length from the same football match filming a different part of the field at the same time. Those videos are taken from the ISSIA-CNR Soccer Dataset, the same dataset used to train the *DeepBall* object detector.

2. Methodology

In this section we will describe in detail the different methods we used to implement the core functionalities of the project. We will not go in details on the parts of the algorithm that were already implemented and provided by Jack Wotherspoon's tracking application, on which we built our algorithm.

2.1. Preliminary steps

Before we start iterating over the frames of each input video, we need to provide a way to load the frames of our input videos, create a *VideoWriter* for the build of the output video, and initialize, or set, the different objects storing the important values that will be used during the whole process.

2.1.1. Input & output video settings

After initializing and configuring the provided *DeepSORT* tracker and *YOLOv4* object detector, we load the pre-trained *DeepBall* keras model we decided to use for a more accurate ball detection.

We then start the video capture and load the input videos using the *OpenCV* function *VideoCapture*. The paths to the several different input videos filming the football game can be specified through the *VIDEO* flag. The argument must be written under the form *./path/to/video1+./path/to/video2* where *./path/to/video1* is the location of *video1* in the filesystem. Each provided video path has to be separated from others by a **'+'**. Any number of video paths can be provided to the algorithm. The user is still required to load videos that are all recording the same football match at the same moment with fixed camera angle, and must specify them through the *VIDEO* flag argument in a specific order: the first specified video must be filming the left end side of the field, the second one must be filming the portion of the field immediately on the right of the portion filmed by the first camera angle, and so on until we reach the right end side of the field. Figure 4 gives more precision by showing a valid ordered set of videos.

Figure 4: Example of video camera angles where the angle on the left must be specified as the first video to be processed, followed by the angle on the center, and then finally by the angle on the right

At the moment, it is required that the cameras are all placed on the same side of the field. As further work, we may consider dealing with the case where cameras on the opposite side of the field are also recording at the same time.

We then have to define the attributes of our output video: its width, height, frame rate (*FPS* - Frames Per Second), codec and length. The width and height are fixed and respectively set to 1920 and 1088 pixels, since the *DeepBall* object detector has been trained to optimally work with these frame dimensions. We then highly recommend to use input videos with frame dimensions close to *1920x1088* to not lose too much information after the resizing process (mandatory process in order to feed each frame to the input layer of the sports ball detector). The frame rate of the output video is set to the highest existing *FPS* attribute among every specified input video. The video codec used is always set as *XVID*. The length of the output video is set to the shorter existing video length among the chosen input videos. The *VID_LEN* flag lets the user specify an even shorter length for the output video in order to process only the first *VID_LEN* frames of each input video. The argument must be an integer lower than the number of frames of the shortest input video, otherwise it has no effect. The user is also invited to specify a filesystem path to save the output video and the end of the computation thanks to the *OUTPUT* flag with an argument of the form *./path/to/output*, where *output* is the name given to the output video. If no path is specified, no output video will be computed. A *VideoWriter* object for the output video computation will then be created using the defined width, height, codec and fps attributes as well as the specified filesystem path.

2.1.2. Landmark areas, polygons and matrices definition

We will display a mini-map, representing a football field by bird-eye view, at the bottom of the output video. This mini-map will record the position of the tracked players at every frame. To do it, we first need to define a landmark area for each video. Each one of these areas will be linked to a different landmark area on the mini-map that we also need to define. Then, we will create our perspective transformation matrices in order to convert every 2D coordinate on a given video into its corresponding 2D coordinate on the mini-map.

To specify the different landmark areas for the videos and the mini-map, the user has to respectively use the *LANDMARK_COORDS* and *MINI_LANDMARK_COORDS* flags. The argument for each of those two flags has to be written under the form x_{11},y_{11}-...-x_{14},y_{14}+x_{21},y_{21}-...-x_{24},y_{24} where x_{AB},y_{AB} represents the 2D coordinate of point *B* among the landmark area *A*. Each 2D point among the same landmark area is separated by a **'-'** and each list of points, representing a different landmark area, is separated by a **'+'**.

For video landmark areas, the user is expected to provide 5 points for the landmark area of the input videos that are filming either end of the field in the following order: {top-left → fifth point → bottom-left → bottom-right → top-right} for the left end of the field and {top-left → bottom-left → bottom-right → fifth point → top-right} for the right end. For the landmark area of the other parts of the field, the user is expected to provide 4 points in the following order: {top-left → bottom-left → bottom-right → top-right}. In the case we only have one input video, which is then filming the whole field, the user has to provide only one 4-points, or 6-points, landmark area, respectively under the form {top-left → bottom-left → bottom-right → top-right} or {top-left → fifth point → bottom-left → bottom-right → sixth point → top-right}. Here is the technical meaning of each word written in the landmark areas points order: *top-left*, *bottom-left*, *bottom-right* and *top-right* respectively correspond to the top-left, bottom-left, bottom-right and top-left corner of the landmark area, while *fifth point* corresponds to the base of goal post that is the closest to the bottom of the screen. We give the possibility to define a 6-points landmark area in the situation we only have one video as input that is filming the whole field, because the bottom goal post base from both sides will be visible.

For each existing mini-map landmark area, the user is expected to always provide 4 points in the following order: {top-left → bottom-left → bottom-right → top-right}. The user has also to be careful on the order of specification of the different landmark areas when providing them through both *LANDMARK_COORDS* and *MINI_LANDMARK_COORDS* flags: the first one must be related to the first video specified through the *VIDEO* flag, the second one to the second video, and so on. For example, if the *VIDEO* flag is set to *./path/to/video1+./path/to/video2* and the *MINI_LANDMARK_COORDS* flag is set to x_{11},y_{11}-...-x_{14},y_{14}+x_{21},y_{21}-...-x_{24},y_{24} then x_{11},y_{11}-...-x_{14},y_{14} must be the mini-map landmark area related to the video *./path/to/video1*. The same goes for the *LANDMARK_COORDS* flag with 4, 5 or 6-points area landmarks.

Each video landmark area defined will then be turned into a *Polygon* thanks to the *Shapely* library. Each *Polygon* will bound the limits of a different filmed portion of the field. This process is made to remove detections that are not inside one of those *Polygons*, meaning they are outside of the field limits, like people in the crowd or security agents walking around the field. In the specific case we have 3 videos as input, we create an additional *Polygon* with vertices being the mini-map central landmark area points (representing the central portion of the field filmed by the “second” camera).

This *Polygon* is created to remove potential duplications of players on the mini-map, since one or several small portions of the field may be filmed by two (or more) different cameras at the same time. This duplication may occur if the two cameras in question are, for example, not placed exactly at the same distance from the field or not oriented the same way. When this happens, it is difficult to find precise landmark 2D points and make sure that every landmark area on the mini-map represents precisely their corresponding video landmark area. At this moment, when transforming the coordinates from the video into the mini-map via the corresponding transformation matrices, we may observe two overlapped dots on the portions of the mini-map where the two camera angles intersect: this corresponds to a duplicated detection of a given player, showing us that there is some precision error with our landmark areas definition.

By creating this additional *Polygon*, we make sure that, when this situation occurs, only the detection recorded on the central camera will be displayed on the mini-map. As further work, this mechanism has to be extended to deal with the cases where there are more (or less) than 3 input videos.

Finally, we can define each of our 2D coordinate transformation matrices. Thanks to the function *getPerspectiveTransform*, from the *OpenCV* library, we create each matrix that will be used to transform the coordinate of each detection from the associated input video to their corresponding one on the mini-map. We have to pass, as arguments to the function, two *float32 numpy* arrays of 4 elements. Those elements are, in order, the *top-left, bottom-left, bottom-right* and *top-right* landmark area 2D coordinates. The first array has to be a video landmark area previously defined, while the second array must be its corresponding mini-map landmark area.

2.1.3. Computation lists and frame folders creation

Before starting the video reading process, we define a few lists to store some important values that will be used later in the algorithm to compute the output video.

→ The lists *ball_detection_scores, recorded_ball_positions* and *nearest_bbox_from_ball* are two-dimensional lists where each row represents a frame number and a column represents the index of an input video (e.g. the first row corresponds to the first frame and the first column corresponds to the first input video the user specified through the *VIDEO* flag):

- *ball_detection_scores* stores the ball detection confidence score if it is higher than a set threshold
- *recorded_ball_positions* stores the position of the ball
- *nearest_bbox_from_ball* stores information about the *bounding box* (player detection area) that is the nearest from the ball if the latter has been detected in the given frame

→ We define, in addition, two other two-dimensional lists that we named *count_best_frame_ball_detection* and *total_recorded_points*:

- *count_best_frame_ball_detection* stores the number of times each input video proposed the most interesting frame during one given second (what we can call a *'focus'* score), in order to choose which camera angle to display at every second (one second corresponding to *fps* frames, defined earlier) on the output video (the n^{th} row corresponds to the n^{th} second of the output video while the i^{th} column will record the *focus* score of input video *i* for each second).
- *total_recorded_points* stores the mini-map coordinates of every detection at the same time step for every video (e.g. the first row corresponds to the first frame of every input video and contains the mini-map coordinates of every detections recorded at the first frame of each input video, while the columns have no real meaning for this list).

We also create a *processing* folder, with, inside it, two subfolders for each dedicated input video, to store the original processed frames and their background subtraction version. We store the frames to precisely analyse how the computation went but also to later build our resulting output video.

2.2. *Input videos processing*

We now can start our iteration over each frame of every input video. As a preliminary step, just before reading the first frame of the next video, we initialize our background subtractor that will be used to enhance ball detection performance. Also, the tracks of the *DeepSORT* tracker, from the previous processed input video, that are still active must be deleted, since we are analysing a brand new video sequence.

Each input video will be processed in the same order they have been specified through the *VIDEO* flag. When trying to read the next frame of the currently processed video, if the *read* function returns *False*, then it indicates that all the frames of the current video have been read and we can start reading the next video. Otherwise, we process the current frame by first resizing it to the dimensions of the output video we defined earlier, and then pass the frame to the *YOLOv4* detector and the *DeepSORT* tracker that are already at our disposal. We simply make sure that the *YOLOv4* detector only focuses on the *Person* object class, since we only want to detect and track players with *YOLOv4* and *DeepSORT*.

2.2.1. Background subtraction

The next step is to apply background subtraction on the current processed frame with the *KNN* algorithm. The function *apply* lets us create a foreground mask using our background subtractor that we then simply have to apply on our frame with the *OpenCV* function *bitwise_and* in order to subtract the background and only keep moving objects. The *apply* function also updates the background subtractor model to process optimally the next frame of the video.

We found that the *DeepBall* model, according to Figure 5, is able to give a better confidence score and has more facility to spot the correct location of the ball on a frame with its background subtracted. We experimented with 2 different background subtraction algorithms, which are *KNN*[29] and *MOG2*[30]. We decided to retain the *KNN* algorithm since, according to Figure 6, he is the one that produces the less noise in the resultant image, and generally tends to be more precise in terms of correct ball detection according to our experiments.

In addition, the *KNN* background subtractor is known to be an excellent choice in terms of performance on some implementations[31].

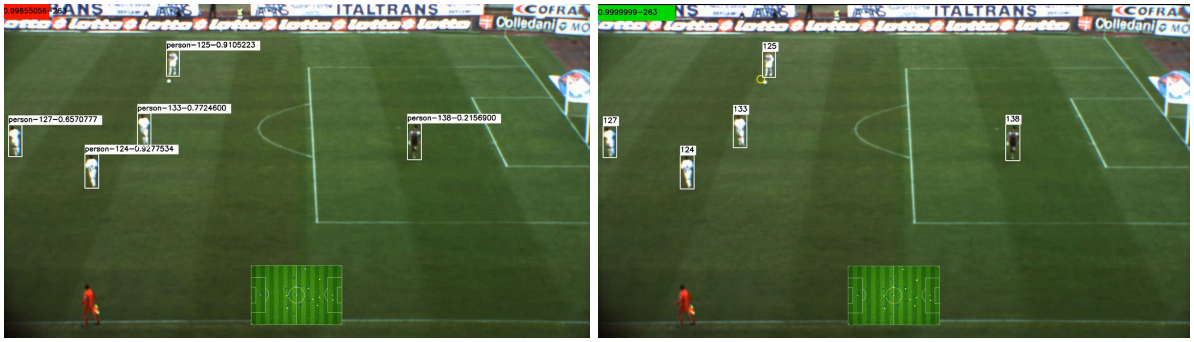

Figure 5: Example of a case showing the differences in terms of ball detection efficiency between two identical frames, but one without applying background subtraction (left, highest ball confidence score equal to 0.99655056, ball not detected) and with background subtraction applied (right, highest ball confidence score equal to 0.9999999, ball detected)

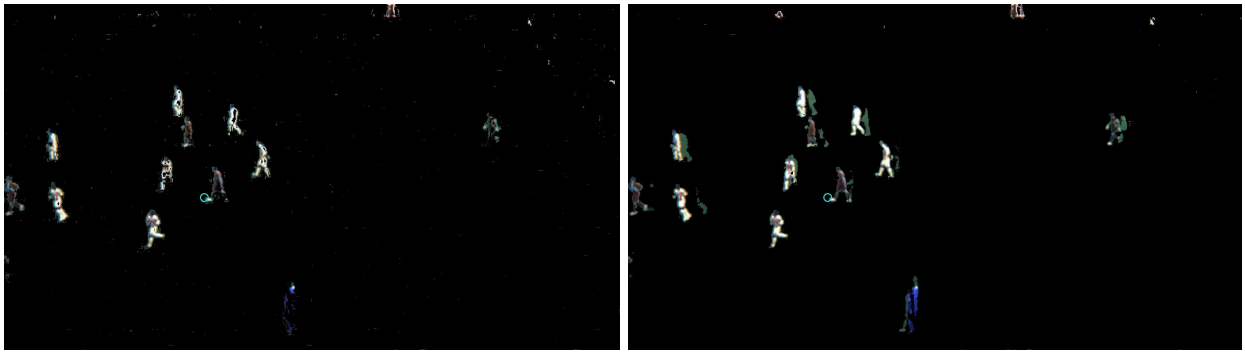

Figure 6: Comparison between a frame with background subtracted using the MOG2 algorithm (left) and the KNN algorithm (right)

2.2.2. Ball detection process

We then run our *DeepBall* model in order to detect the ball. After resizing our frame to match the required dimensions for the input layer, which are *480x272*, the model will output a confidence map containing the confidence score of each pixel of the frame given by *DeepBall*: the higher is the confidence score for a given pixel, the higher the chances there is a ball at this location of the frame. We only focus on the pixel in the confidence map having the highest returned confidence score.

If this confidence score is higher than the set threshold (0.999999 by default which can be changed through the *BALL_THRESHOLD* flag), we consider that the detection correctly corresponds to the ball and we retrieve the coordinates of the corresponding pixel on the full size frame by multiplying the pixel coordinates from the confidence map by a scaling factor equal to *1920/480 = 1088/272 = 16* (we recall that our frame has dimensions *1920x1088* while the input layer of the *DeepBall* model accepts frames of dimensions *480x272*).

In the case the highest confidence score is lower than the set threshold, we consider that the ball has been missed, or is simply not visible on the frame, and we update the "*ball position range*" in both x (width axis of frame) and y (height axis of frame) coordinates. This range corresponds to the realistic distance in pixels, on both axes, between the location of the last correct ball detection, and the current position of the ball. Thanks to this range limit, in the case a pixel of an other object than the real ball has the highest confidence score at the next frame (because the ball is hidden or because the object highly resembles a ball), we can invalidate the detection if this object is realistically too far from where the real ball can be.

When a detection gets invalidated because the threshold has not been reached, or the location of the current detection is too far from the previous valid detection, then we increase the possible range by the value the user set through the *BALL_RANGE_UPDATE* flag (100 by default for both x and y axes). If, later, a valid detection occurs, then the range, for both axes, is set back to the *BALL_RANGE_UPDATE* value. In the case the user wants to choose his own values for the initialization and update of the realistic ball range position, for both axes, the argument, for this flag, has to take the form *x,y* with *x* and *y* being integers, and respectively corresponding to the range value of the x axis and y axis (note that a comma has to separate both values).

In the case we consider that the ball has been correctly detected, after retrieving the pixel coordinates on the original processed frame, we save the confidence score in the *ball_detection_scores* list described earlier.

We also save the ball position in a variable to use it later in the current frame process. Then, we draw a circle on the frame at the location of the detection to let the user visualize the success of the detection, and finally we transform the coordinates of the ball into their corresponding mini-map coordinates using the coordinates transformation matrix previously defined for the currently processed input video. To make the transformation, we use the formula on Figure 7. The mini-map point is then saved in a temporary list called *points_info* that gathers, for the current frame, every detection track id, associated color and mini-map coordinates. A track id (positive number) is associated by *DeepSORT* to an object currently being tracked. Since we do not use *DeepSORT* to track the ball, we always associate the track id number 0 for the ball.

$$\mathtt{dst}(x,y) = \mathtt{src}\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right)$$

Figure 7: Formula of coordinates conversion, with $M_{a,b}$ being the value at row *a* and column *b* of the perspective transform matrix

After having run the *DeepBall* detector, if we are in the case that we are processing the last input video at a given frame *n*, we are able to tell which video proposed the frame *n* with the highest ball detection confidence score. We note, for a given detection, that the ball confidence score, in the *ball_detection_scores* list, will be saved as being equal to 0 if the detection threshold is not reached, since we consider, under this threshold, that the real ball has not been correctly spotted by the ball detector. We then increase by 1 the *focus* score of the input video that outputs the highest confidence score for the current frame *n*. In the case of equal ball detection scores between two or more input videos for frame *n*, we increase the *focus* scores of every of these videos by 1. This *focus* score is related to the $(n/fps)^{th}$ second (truncated value) of the output video. To recall, every *focus* score is stored in the *count_best_frame_ball_detection* list.

2.2.3. Jersey color tracking and team association

We now have to detect the jersey color of every detected entity on the field (referees, home team and away team players and goalkeepers). Again, we will work with the resulting frame after applying background subtraction to it, so that the field does not interfere in the process. Indeed, since our jersey color detection process will be based on the pixel values that are into the bounding box of each detection, it is better to have every pixel belonging to the field being constantly nullified so that we don't risk taking into account any of them in the case some fall into one of the color ranges we define for each jersey.

At the moment, due to the complexity of the jersey color detection feature of this project, the range of jersey color values for each entity is fixed and are made to match the jersey colors of the players appearing on the default input videos: red for referees, blue for home team, yellow for home team goalkeeper, white for away team and black for away team goalkeeper. We also note that, concerning the ball, we always highlight it on the frames with a yellow circle when being detected. As further work, the program should be adapted to deal with any kind of jersey colors, where the user may choose his own color space value ranges (e.g. through flags) in order to evaluate any kind of opposition. The user should also be able to choose in which color the circle that highlights the ball should be.

As a first step, we switch the color space of the frame to *HSV*. Figure 8 and Figure 9 show the *HSV* diagram for any existing colors. In Python, *Hue* range is [0,179], *Saturation* range is [0,255] and *Value* range is [0,255].

The *HSV* color ranges we defined are the following:

- Referees (red) : (0,100,100) to (9,255,255) & (170,100,100) to (179,255,255)
- Home team (blue) : (110,75,75) to (130,255,255)
- Home team goalkeeper (yellow) : (25,100,100) to (35,255,255)
- Away team (white) : (0,0,180) to (179,75,255)
- Away team goalkeeper (black) : (40,20,120) to (90,80,230)

The main difficulty was to find a color range that highlights the black pixels of the home team goalkeeper jersey without highlighting at the same time the pixels of the home team players dark blue jersey.

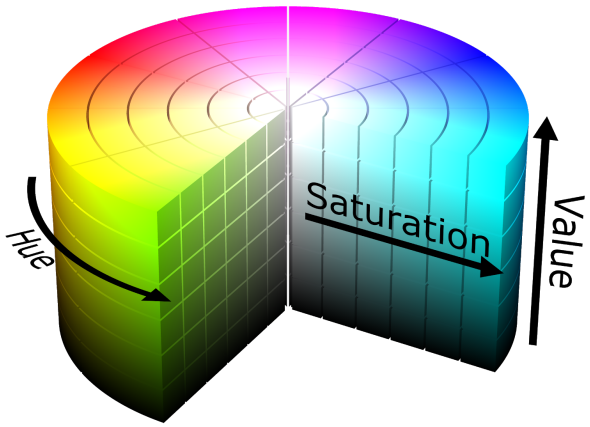

Figure 8: HSV diagram showing the resulting effects of the variations of Hue, Saturation and Value

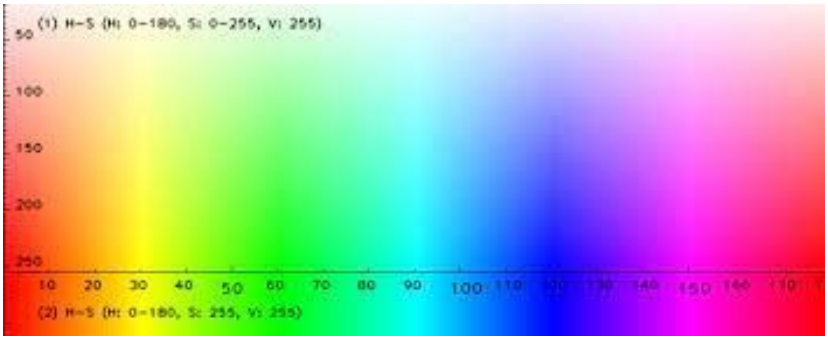

Figure 9: HSV diagram when working with *OpenCV Python* color ranges

To create our masks that will filter the frame to only keep the desired color, we use the *inRange* function from *OpenCV*. Note that we defined two color ranges for the red color since that, according to Figure 9, the red color is situated on both ends of the color diagram, resulting in the creation of two different masks for the red color. We combined these two masks with the *OpenCV* function *bitwise_or*. We finally create one filtered frame for each mask by applying them on the current processed frame, having its background subtracted, using the *OpenCV* function *bitwise_and*.

2.2.4. Bboxes handling and coordinates conversion

We will now plot the bounding box (bbox) of every object, still being tracked by *DeepSORT*, on the currently processed frame. For each of these objects, using the *x* and *y* coordinates, provided by the object tracker, of the top-left and bottom-right corner of their bounding box, we first retrieve their bounding box dimensions and the coordinates of their position, the latter being the 2D coordinates of the pixel that is at the center of the bottom edge of their bounding box. Then, we eliminate the bounding boxes that have unrealistic dimensions for a *Person* object class (e.g. a computed bounding box that has dimensions close to the width and height of the actual frame). The default value for the maximum width and height of a *Person* bounding box is set to respectively 110 and 170 pixels. Those values can be changed by the user through the flag *MAX_BBOX_DIM* by providing an argument under the form *max_width,max_height* (a comma has to separate both values, and both *max_width* and *max_height* are positive integers). We also eliminate the bounding boxes that are outside the field. To do that, we retrieve their coordinates position we just computed and check if this position is within the created *Polygon* associated to the currently processed input video. If it is not the case, then the tracked object is outside of the limits of the field and we don't take it into account among the detected entities for the current frame.

As a next step, by still using the *x* and *y* coordinates of the top-left and bottom-right corner of each bounding box, for each previously computed filtered (or masked) frame, we only keep the frame portion which corresponds to the area of the currently processed bounding box. We then calculate the "*black pixels ratio*" inside this bounding box, which is the *number of pixels that have been nullified after applying a given color mask divided by the total number of pixels inside the bounding box*. The bounding box will take the color associated with the masked frame that returns the lowest *black pixels ratio*. The bounding box can now be drawn on the frame around the concerned tracked entity with its associated color. The id of the entity, attributed by the *DeepSORT* tracker, is also specified above the top-left corner of the bounding box for visualization purposes.

The coordinates of the bounding box position on the frame is then converted into mini-map coordinates using the transformation matrix associated to the currently processed input video. The converted coordinates are saved in the temporary *points_info* list, introduced earlier, with the bounding box detection id and its attributed color. Another temporary list, named *bboxs_info*, will then come into play and will store the current bounding box top-left and bottom-right *x* and *y* coordinates in addition to its id and attributed color.

After processing every tracked object on the current frame, we add every mini-map point from *points_info* to the list *total_recorded_points*, which gathers every mini-map point to be plotted later on each frame of the output video. We recall that, in the case 3 input videos in total have to be processed, if we are currently processing a video filming the left or right end of the field, we do not add the mini-map points that are included in the previously defined mini-map *Polygon* (meaning they are also detected by the central camera) in order to avoid detection duplications on the mini-map of the output video.

Finally, we end a frame processing iteration by storing, in the *nearest_bbox_from_ball* list, the information of the bounding box of which center (the pixel at position [*(x_min+x_max)/2 , (y_min+y_max)/2*] where [*x_min,y_min*] is the position of the top-left corner of the bounding box and [*x_max,y_max*] is the position of its bottom-right corner) is the closest, in terms of euclidean distance, to the ball, if the latter has been detected in the current frame. We don't take into account the bounding boxes of the referees, since the list *nearest_bbox_from_ball* will be used to compute statistics on teams and players.

To compute every Euclidean distance, we first retrieve the position of each bounding box center, as well as the location of the ball. We then use the function *distance* from the *Shapely* library between the center of each bounding box and the position of the ball, and store in *nearest_bbox_from_ball* the bounding box that returns the shortest value.

2.3. Output video writing

2.3.1. Frame selection

After processing every input video, we can now start writing each frame of our output video. We initialize the variables *nb_possession_frame, possession_home* and *possession_away*, that will be used for the possession statistics, to 0. The variables *prev_possession* (set to None)*, currently_passing* (set to False)*, total_passes_home, correct_passes_home, total_passes_away* and *correct_passes_away* (all set to 0) will be used for the pass accuracy statistics. Finally, *heatmap_values*, a two-dimensional list with every entry initially set to 0, will be used for heatmap computation. We also load our mini-map image (the image can be chosen by specifying its path in the filesystem through the *IMG_INPUT* flag). The list *heatmap_values* has the same dimensions as the image (number of rows equal to the height of the image, and number of columns equal to the width).

The n^{th} frame of the output video will be the n^{th} frame of only one of the input videos. At each timestep, we then have to pick a processed frame from one video input. The list *count_best_frame_ball_detection* we described earlier comes into play. The video from which to pick the frame will be the one having the highest *focus* score in the corresponding row. We recall that each row corresponds to a given output video second. A second corresponds to *fps* frames. The value *fps* corresponds to the frame rate of the output video, which has been defined before the processing of each input video. The *focus* score of a video for a given second can then have a maximum value equal to *fps*.

In our experiments, the output video has a *fps* of 25. This means one second corresponds to 25 frames. Row *n* of *count_best_frame_ball_detection* will tell us which video has the highest *focus* score (which can't exceed 25 in our experiments) for the n^{th} second, or, in other words, which input video will be displayed during the n^{th} second of the output video (such that frames number *(n-1)*fps+1* to *(n-1)*fps+25* from the selected input video will be displayed in a row, if we consider that the first second corresponds to *n=1* and the first frame is noted *frame 1*). In case of equality, we keep the same camera angle we used for the previous second. If there is already an equality for the very first second, the input video with index equal to the truncated value of the *total number of input videos divided by 2* will be displayed. This mechanism makes sure that we don't oscillate between two or more different camera angles every few frames, in the case the *DeepBall* detector would incorrectly give a confidence score greater than the set threshold to an object that is not the ball, but highly resembles it, at the other end of the field.

By making sure the most interesting camera angle (the one where the ball is visible with highest probability) is displayed for one straight second, the viewer is able to clearly visualize the portion of the field requiring the greatest attention at any given time. After selecting the input video to be displayed for the currently processed second, we simply load the corresponding frame that has been processed and saved in our filesystem earlier during the processing of the input videos.

2.3.2. Statistics computation

We then update the possession and passing statistics. We retrieve the bounding box that was the closest to the detected ball at the current time step and we check its color to determine the team it belongs to. If the center of this bounding box is at a distance, in pixels, lower than a certain threshold defined through the *MAX_DIST_POSSESSION* flag (70 by default) then we consider being in a "*Possession*" phase. Otherwise we are in a "*Passing*" phase.

When being in a "*Passing*" phase, the boolean *currently_passing* is simply set to True. When being in a "*Possession*" phase, we check if *currently_passing* is set to True. If it is not the case, we simply update the *prev_possession* variable that stores the last player that was the closest to the ball during a *Possession* phase (that we can consider as being the last player that possessed the ball) before updating the possession statistics. To update the possession statistics, first, we increase by 1 the variable *nb_possession_frames* storing the number of frames of the output video in which we were in *Possession* phase. Then, according to the associated team of the bounding box, we increase by 1 *possession_home* if the bounding box was associated to the home team, or *possession_away* if it was associated to the away team. Finally, we compute the possession percentage for each team with the following formulas:

→ For the home team:
- *(possession_home/nb_possession_frames)*100*

→ For the away team:
- *(possession_away/nb_possession_frames)*100*

The passing accuracy statistics are updated in the case *currently_passing* is seen as True when we are in a *Possession* phase. Also, *prev_possession* must not be *None,* meaning that a player should have previously possessed the ball before passing it (we can say that we were previously in a *Passing* phase because *currently_passing* has been previously set to True). If those conditions are met, we distinguish two different cases. The first case is when the player receiving the ball is on a frame belonging to the same input video as the frame where the last player that passed the ball actually executes the pass. We then check if those two players are the same (if they have the same track id). If it is not the case, we can update the passing accuracy statistics. To update the passing accuracy statistics, first, we check the teams these two concerned players belong to:

→ If they are both from the same team:

- They are from the home team: the variables *correct_passes_home* and *total_passes_home* are increased by 1
- They are from the away team: the variables *correct_passes_away* and *total_passes_away* are increased by 1

→ If they are not from the same team:

- The player that received the pass is from the home team: the variable *total_passes_away* is increased by 1
- The player that received the pass is from the away team: the variable *total_passes_home* is increased by 1

The second case is when the player receiving the ball is on a frame belonging to a different input video than the frame where the last player that passed the ball actually executes the pass. In this case, we consider that the new player in possession of the ball is different from the previous one, since it is difficult to know if this is the same player or not because of the fact that the tracked player ids are never the same from an input video to another, and we update the passing accuracy statistics the same way as described previously.

We compute the passing accuracy percentage for each team with the following formulas:

→ For the home team:
- *(correct_passes_home/total_passes_home)*100*

→ For the away team:
- *(correct_passes_away/total_passes_away)*100*

We also decided to add an indication on the *'state'* of the ball to tell if the ball is on the field or outside the limits of the field. To do it, we retrieve the ball position that has been recorded on the frame we are currently writing to the output video, and check if this position is within the *Polygon* associated with the input video of the frame we are writing. If it is the case, the ball is "On Field", and if not, it is "Out of Bounds". This feature can be extended as further work to compute events like throw-ins, goal kicks, corner kicks or when a goal is scored.

2.3.3. Mini-map plot and heatmap values update

The last step we need to perform in order to finish writing the frame to the output video is the plot of the mini-map with the current position of every entity on the field (players, referees and the ball).

After getting the mini-map image from our filesystem, we simply plot every point that is present in the corresponding row of the *total_recorded_points* list using their converted mini-map coordinates we previously computed. We just have to make sure that only one ball is drawn. In the case a ball has been detected in two or more different input videos at frame *n*, we will have several points having their track id equal to 0 at row index *n-1* of the list (if n takes integer values equal or greater than 1). The only point with track id 0 we will display on the mini-map is the one that has been detected by the camera of the input video of which we are currently writing the frame to the output video.

In addition, if a point fulfills the conditions for the heatmap computation (e.g. the point belongs to the home team and the user requested to compute the heatmap of the home team), we increase by 1 the corresponding entry of the *heatmap_values* list (if a point has mini-map coordinates *(x,y)* then the value at row *y* and column *x* of the list is increased by 1). The user can specify the entity of which he wants to compute the heatmap through the flag *HEATMAP* by giving, as argument, a positive integer, or the words “ref”, “home” or “away” (for respectively computing the heatmap of every referees, home or away players). If the HEATMAP flag is not used, no heatmap will be computed.

Finally, we plot the mini-map at the bottom center of the frame after shrinking it. The scale of the shrink can be controlled through the *IMG_SCALE_SHRINK* flag. The provided argument must be a positive integer and represents a shrink percentage (e.g. if the user enters 35 then the image new dimensions will be equal to the original dimensions multiplied by 0.35).

We can then call the *OpenCV* function *write* on the VideoWriter created at the very beginning to write our frame to the output video and start processing the next frame until no more frames have to be written, ending the whole output video writing process.

2.4. Heatmap build

The whole program ends with the heatmap computation process. We first switch the original image used for the mini-map to the *BGRA* color space in order to play with pixel transparency. This image will act as the background of our mini-map.

We will apply a colormap on this background image. The colors will vary according to the values in each entry of our *heatmap_values* list. We then have to convert the range of existing values in *heatmap_values* to the range of values of the colormap (0 to 255).

To do this, we retrieve the maximum and minimum values in *heatmap_values* and apply the following linear conversion formula to each value of *heatmap_values*:

*newValue = (((oldValue - oldMin) * (newMax - newMin)) / (oldMax - oldMin)) + newMin*

Here, *newValue* is the result of the conversion of the original value, *oldValue* is the original value we want to convert to the new range, *oldMin* and *oldMax* are respectively the minimum and maximum values of the *heatmap_values* list before conversion, and *newMin* and *newMax* are respectively the minimum and maximum values of the colormap range (which are then 0 and 255 respectively).

We then transform *heatmap_values* into a “frame” to form the desired color map to be overlayed on the background image. We first turn *heatmap_values* into a *numpy* array. Then, we set its type as *uint8*. Finally, we apply the *OpenCV* function *applyColorMap* using the colormap type called *COLORMAP_JET*. Figure 10 shows the color scale of the *JET* colormap that we will apply to our foreground frame.

Figure 10: Color scale of the *OpenCV* JET colormap

We then switch our computed colormap frame to *BGRA* color space to make the zero-valued pixels transparent. A zero-valued pixel corresponds to a position that the tracked entity never visited during the entire video. If we refer ourselves to the scale of the colormap *COLORMAP_JET*, the *BGRA* color code for the lowest value is (128,0,0,255) which corresponds to dark blue with full opacity. We turn the *alpha* channel parameter of every of those pixels to 0 in order to make them completely transparent so they don’t appear on the heatmap.

The very last step is to perform “over” compositing. Thanks to the *alpha* channel, we will combine our two frames, such that our colormap appears in the foreground and our original field mini-map image appears in the background, using the compositing operation commonly called "over". The corresponding formula is on Figure 11.

$$C_o = \frac{C_a \alpha_a + C_b \alpha_b (1 - \alpha_a)}{\alpha_a + \alpha_b (1 - \alpha_a)}$$

Figure 11: Formula for the “over” operator (C_o is the result of the operation, C_a is the color of the pixel in element A, C_b is the color of the pixel in element B, and α_a and α_b are the alpha of the pixels in elements A and B respectively)

Before applying this formula, we normalize the alpha channel of both frames from the range 0-255 to the range 0-1:

alpha_background = heatmap_background[:,:,3] / 255.0
alpha_foreground = heatmap_values[:,:,3] / 255.0

Here, *heatmap_background* is the frame of our background which is the mini-map background, and *heatmap_values* is our list that has been turned into a frame that will represent the foreground, or in other words, the colormap. We recall that both *heatmap_background* and *heatmap_values* are 2D images with a third dimension containing four channels *B,G,R* and *A* where *B,G,R* are the color channels and *A* is the alpha channel for the transparency of the pixel, and are respectively indexed by 0,1,2 and 3.

Then we apply the formula on Figure 11 to adjust the frames BGR colors. In the following formula, *hb* stands for *heatmap_background*, *hv* for *heatmap_values*, *ab* for *alpha_background* and *af* for *alpha foreground*:

for i taking values in {0,1,2}:
    *hb[:,:,i] = af * hv[:,:,i] + ab * hb[:,:,i] * (1 - af)*

We now have to adjust the alpha channel. The corresponding formula is on Figure 12.

$$\alpha_o = 1 - (1 - \alpha_a)(1 - \alpha_b)$$

Figure 12: Formula to adjust the alpha channel, being the last step of the whole “over” operation (α_a and α_b are the alpha of the pixels in elements A and B respectively)

We then use this formula and denormalize the alpha channel back to the 0-255 range:

*hb[:,:,3] = (1 - (1 - af) * (1 - ab)) * 255*

More explanation about those formulas can be found on *Wikipedia* in the article *Alpha composing*.

Our heatmap has finally been computed and we can save it in our filesystem. The save path can be chosen manually through the flag HEATMAP_PATH with an argument of the form *./path/to/heatmap.png* where *heatmap* is the name of our heatmap image. The *PNG* extension is mandatory since it supports pixel transparency.

This ends our whole implementation description.

The source code is publicly accessible under the GitHub repo *vincentrinaldi/MasterSemesterProject*.

3. Experimental Results

We ran our implementation using *Google Colaboratory* service for better speed performance, which allowed us to process our input video on a larger amount of frames.

The minimum accepted confidence score for the detection of a *Person* object to be outputted by *YOLOv4* was set to 0.05 during our experiments in order to make sure all the players are constantly being detected at every frame of any input videos.

The obtained results are pretty interesting and positive. Most of the objectives set at the beginning of the semester have been reached, in particular concerning the computation of statistics.

We can see on Figure 13 that the jerseys are, for the most cases, correctly tracked, since that each visible bounding box representing a *Person* has their correct associated color.

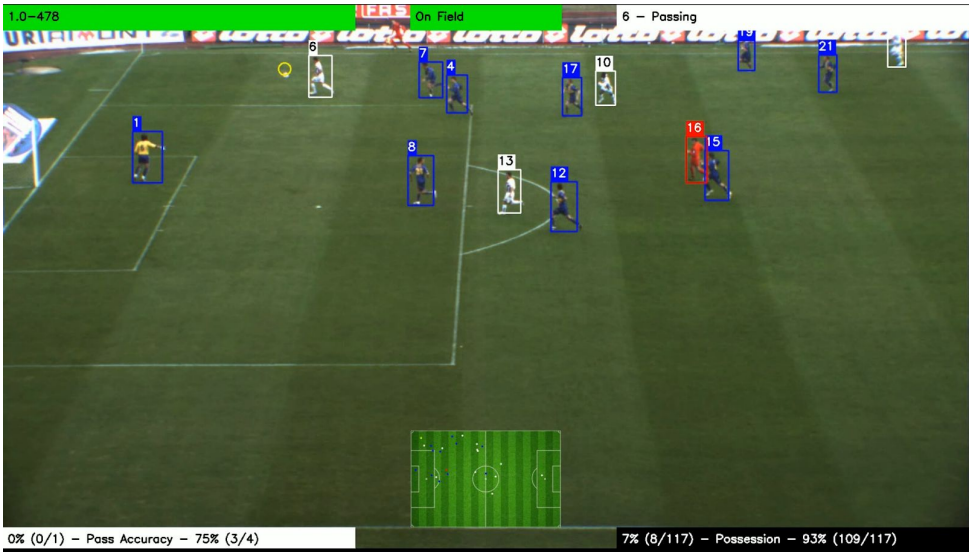

Figure 13: Processed frame where all the players and referees have been correctly associated to their corresponding team

Our custom technique to moderate incorrect ball detections that are too far from the previous correct one showed interesting results, and Figure 14 shows a situation with an incorrect ball detection (ball confidence score greater than the ball detection threshold set to 0.999999 for an object that is not the ball but resembles it) that is ignored because of this detected position is way too far from the actual position of the ball, which has been correctly detected at the previous frame.

Figure 14: Sequence of frames in the following order: 1.top-left: the real ball is correctly detected (top-left); 2.top-right: an object as been designated by *DeepBall* as being the ball, while it is not, but the detection is invalidated (indicated by the blue rectangle at the top-left corner of the frame) because it is too far from the previous correct location of the ball; 3.bottom-left: No ball has been detected; 4.bottom-right: the real ball is again correctly detected and no incorrect detection has been displayed even if the real ball was not detected during two consecutive frame

Figure 15 shows two heatmaps that have been computed for the full 2-minutes sequence. One is the combined heatmap of every player belonging to the home team, and the other is the heatmap of the player with track id 1, which is the goalkeeper of the home team. The dark blue color is the most predominant, meaning that the player visited those position very rarely (only one or two times), while only very few pixels are bright blue, and only one or two are dark red (close to highest value on the colormap scale) because a duration of two minutes is realistically too short to visit several time a many different position with high probability. It would be more relevant to compute a heatmap on a full 90-minutes football match.

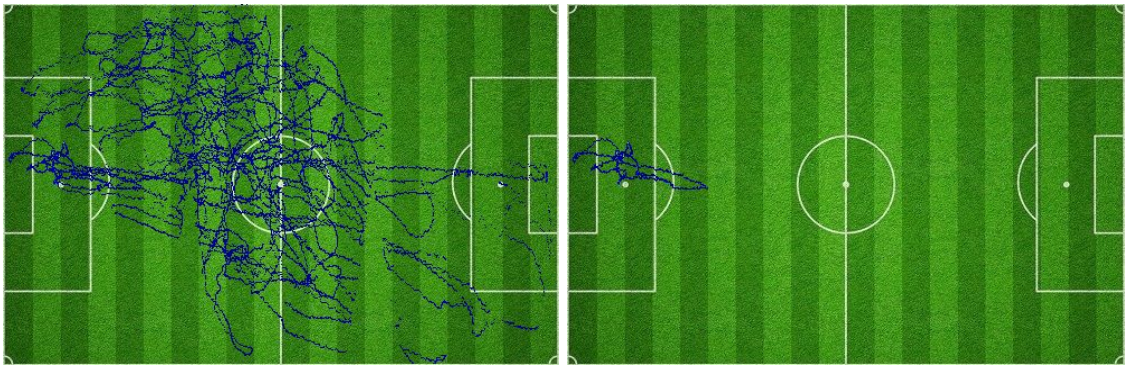

Figure 15: Heatmap of the home team (left) and the home goalkeeper with track id 1 (right) after the processing of the full 2-minutes video sequence

We could also see, and admit, with the different figures showing a processed frame, that the mini-map correctly updates frame-by-frame the associated bird-eye position of each detected player on the whole field according to their position on every input video, even if we are only displaying a frame of only one of the video inputs we processed. This shows that the area landmarks we define for the videos and the mini-map are consistent, and that our perspective transform matrices have been correctly computed.

Finally, Figure 16 shows situations where possession and passing accuracy statistics are correctly updated, as well as the state of the ball according to its position on the field.

To watch the fully processed 2-minutes video sequence, you can download the video named *football_match_tracking.avi* from the GitHub repo *vincentrinaldi/MasterSemesterProject* located in the folder *output*.

However, while some issues that were already present at the beginning, which could be mostly seen as challenges to deal with, have not been completely solved, our own implementation also suffers from some slight problems.

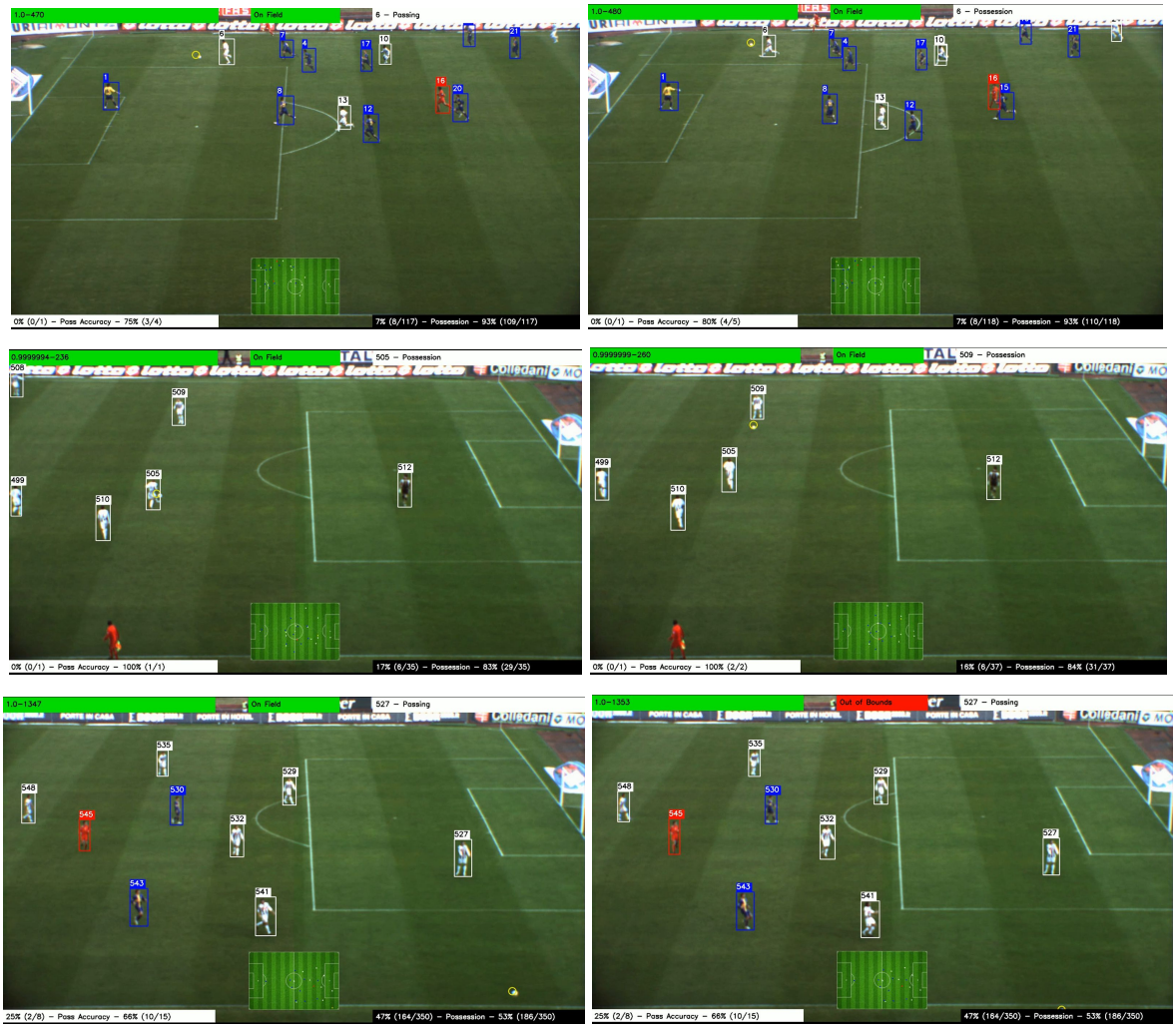

Figure 16: Correct update of possession statistics (top-left → top-right, check the value on the right in the black rectangle where *Possession* is written), correct update of passing accuracy statistics (center-left → center-right, check the value on the right in the white rectangle where *Passing accuracy* is written), and correct “ball state” detection (bottom-left → bottom-right, check the value and background color of the top-center rectangle where *On Field* then *Out of Bounds* are written)

The main challenges were to deal with some id switches that the *DeepSORT* tracker sometimes mistakenly induces because of some situations of player occlusion that *DeepSORT* couldn’t properly handle despite being very efficient to deal with objects occlusion compared to his predecessors.

This prevented us to use deterministic method to associate the same jersey color to a player at every frame (e.g. store the bounding box color associated to a player id at each frame during the processing of the input videos and, at the end, definitely attribute to this player id the color that has been associated the most time during the output video writing process in order for each player id to be always displayed with the same color instead of being displayed with the wrong color for a small bunch of frames on the output video). This also makes the resulting heatmap a bit skewed when tracking a specific player id, since we are not tracking the same player after the id switch occurs. An id switch situation is shown on Figure 17. A clear solution to this problem is to continue researching on the improvement of object trackers.

Figure 17: Situation where two player ids are switched : blue player 38 is switching id with white player 6 (the order of the frames is left → center → right)

Another challenge was to deal with the perspective. Since we are working on 2D frames, we don't have access to the third dimension, which is the field depth. This is a problem when computing the distance between the position of the ball and the nearest bounding box center. Sometimes, while a player can be very far from the ball, if the ball gets in front of the player on the frame (e.g. when the ball is kicked in the air and the player is at the top edge of the fields limits as shown in Figure 18), our algorithm will think that a pass has been made to this player. Also, when the ball is kicked very high, since that the crowd will be behind the ball at this moment on the frame, our algorithm will output that the ball is out of the limits of the field, which is not correct. This perspective issue can be solved with different possible techniques like checking when the ball trajectory drastically changes, which can be interpreted as a player receiving and controlling the ball when a pass has been completed.

Figure 18: Examples of situations with skewed computation results due to the fact that perspective and field depth are not taken into account: in the top-left → top-right situation, the ball is kicked in the air by the blue player, but the program outputs that a pass has been successfully completed by the blue team at the moment the ball becomes close to a blue player on the frame, while still being in the air; in the bottom-left → bottom-right situation, the ball is kicked in the air by the blue player, but the program outputs that the ball is out of the field limits at the moment the ball is in front of the crowd on the frame, while still being inside the limits of the field despite being in the air

Concerning our own methodology, the associated team color sometimes oscillates during a few frames between two different colors.

As said earlier in the *Methodology* part when we talked about our method to detect and track the jersey color of each players, we had difficulties to track the black pixels of the away goalkeeper jersey without taking into account, at the same time, the dark blue pixels of the home team jersey, due to their closeness in terms of brightness and the difficulty to spot the best HSV color ranges to track them.

Figure 19 shows an example where a blue jersey player sees himself being associated with both teams alternatively for a couple of frames due to the difficulty to distinguish his blue jersey from the black one of the goalkeeper playing in the opposing team that we can also see on the same frame. Sometimes, this can skew the statistics on passing accuracy if a blue player is considered being among the white team at the moment he is making a pass to his blue teammate, resulting in a missed pass, while the pass is unquestionably successful.

Figure 19: Example of a situation where a player with a blue jersey is being associated to another team (the bbox color is changing) at the next frame for several frames in a row

In addition, although the *DeepBall* detector is very efficient, it still makes a few detection errors at certain frames. It happens that, because of the mid-quality of the video, the white boots, white socks and some body parts covered by a white clothing on some players may sometimes be detected as being the ball with an almost perfect confidence score, which skews the statistics related to possession, and consequently, passing accuracy as well. Figure 20 describes this issue with more precisions.

For testing purposes, you can set your *Google Colaboratory* environment to run this project by following the steps described in the video *YOLOv4 in the CLOUD: Build Object Tracking Using DeepSORT in Google Colab (FREE GPU)* on the YouTube channel of Jack Wotherspoon named *The AI Guy.* The main file to be edited to add, remove or modify some features is *object_tracker.py*.

Figure 20: The white player at the bottom-center of the frame is receiving the ball and tries to pass it to the white player at the bottom-right of the frame. After the ball leaves the first white player, an incorrect ball detection occurs during a few frames, indicating that the ball is possessed by the blue player close to them, which leads to a *Possession* phase for the blue team. Then the ball is being correctly detected again before it reaches the second white player, leading to a *Passing phase*. The second white player then receives the ball, leading to a *Possession* phase for the white team, and the algorithm will then output that the blue team just missed a pass while it is actually the white team that made a successful pass

4. Conclusion

Automatic player localization, labeling and tracking is a very important feature to establish accurate team tactics and playing strategies, but also to analyse the activity of the players in order to spot the deficient areas that have to be improved by planning specific and optimal training programs, and to maximize the attractiveness of broadcast sports videos.

However, these techniques are one of the biggest challenges in the Computer Vision field due to well-known problematic situations[32], such as player-to-player occlusion, the fact that two players can have a similar appearance or clothes of almost same color, the camera motion that may be too brutal and difficult to handle, the quality of the video that may cause some computation errors because of various noises, blurred areas, when complicated motion patterns are performed by some players during the match, or the situation when some players disappear from the view of the camera.

At the end of our project, we were able to find, develop and provide computation methods that can actually output very performant and accurate results when performing the computation in very favorable conditions (high video quality, perfect tracking of players, ease of finding HSV jersey color ranges for any entity). However, it is unlikely to process sports video in those conditions. Our algorithm, at times, shows some limitations that need to be fixed in the future in order to be able to use it at its full potential. It would also be very interesting to continue its development in order to extend its set of features and use it at real-life sports events.

As further work, we then must focus on extending and enhancing the algorithm in order to:

- deal with cameras that are also filming at the opposite side of the field at the same time and develop a method that correctly retrieves the mini-map coordinates of the objects tracked by the cameras on the opposite side of the field
- being able to remove player duplications on a mini-map, that may occur when two cameras are detecting the same player at the same time, in the case more than 3 (or just 2) cameras are each filming a different part of the field
- let the user choose the HSV jersey color ranges of any entity on the field (including choosing the color of the ball detection indicator) through flags in order to match any jersey color that can appear on an input video, making the algorithm more flexible and able to evaluate any kind of opposition
- expand the “ball state” feature to compute different kind of events like throw-ins, goal kicks, corner kicks or even when goals are scored
- take into account the perspective of the field and the field depth of the frames
- improve the object detectors and trackers performances, either for the players or the ball, in order to reduce the number of ball detection false positives as well as the frequency of id switches

People are more and more feeling the need to quantify sports performance, encouraging the scientific community to start new ambitious projects and develop new softwares that are getting more and more performant, resulting in very important game-changing results. The more sports domain enthusiasts will be able to access large amounts of different types of data, the more the use of analytics in the sports industry will increase.

5. Acknowledgments

I would first like to kindly thank my supervisor Mr. Adam Scholefield who advised and guided me during the whole semester. I also thank the *LCAV* staff for the organization of the semester projects as well as the oral presentations during which I greatly enjoyed attending to discover the excellent work of the other students and to present mine.

This has been an amazing experience and one of the most exciting projects I have done during my academic career. Thanks to this opportunity, I have been able to extend my academic background and learn from a domain I was less familiar with before starting this work. I would definitely take advantage of the next coming opportunities to extend my expertise in the Computer Vision field, and it would also be a great pleasure to work again for the *EPFL AudioVisual Communications Laboratory* in the future.

6. Citations and References

[1] U. Feuerhake, “Recognition of Repetitive Movement Patterns—The Case of Football Analysis,” International Journal of Geo-Information, 2016

[2] S. Barris, and C. Button, “A Review of Vision-Based Motion Analysis in Sport,” Sports Medicine, 2008

[3] S. Leibrick, “Tracking NBA Shooting Stats with RSPCT Using Intel® RealSense™ Technology,” Intel® RealSense™ blog, 2018

[4] J. Portch, “The Growth of Athlete Monitoring in the NBA,” Learders™ blog, 2019

[5] M. Hickins, “Baseball All-Stars' Data Gets More Sophisticated With Field F/X,” The Wall Street Journal, 2013

[6] Q. Ye, Q. Huang, S. Jiang, Y. Liu, and W. Gao, “Jersey number detection in sports video for athlete identification,” Proceedings of the SPIE Visual Communications and Image Processing, pp. 1599-1606, 2005

[7] G. Zhu, C. Xu, Q. Huang, W. Gao, and L. Xing, “Player action recognition in broadcast tennis video with applications to semantic analysis of sports game,” Proceedings of the ACM International Conference on Multimedia, Santa Barbara, CA, USA, pp. 431-440, 2006

[8] M. C. Hu, M. H. Chang, J. L. Wu, and C. Lin, “Robust camera calibration and player tracking in broadcast basketball video,” IEEE Transactions on Multimedia, vol. 13, no. 2, pp. 266-279, 2011

[9] C. W. Lu, C. Y. Lin, C. Y. Hsu, M. F. Weng, L. W. Kang, and H. Y. M. Liao, “Identification and Tracking of Players in Sport Videos,” Proceedings of the Fifth International Conference on Internet Multimedia Computing and Service, 2013

[10] K. Okuma, A. Taleghani, N. de Freitas, J. J. Little, and D. G. Lowe, “A Boosted Particle Filter: Multitarget Detection and Tracking,” Proceedings of the European Conference on Computer Vision (ECCV), 2004

[11] Y. Bar-Shalom, and T. Fortmann, “Tracking and Data Association,“ Academic Press, San Diego, CA, USA, 1988

[12] D. Reid, “An Algorithm For Tracking Multiple Targets,” IEEE Transaction on Automatic Control, vol. 24, no. 6, pp. 843–854, 1979

[13] S. Oh, S. Russell, and S. Sastry, “Markov Chain Monte Carlo Data Association for Multiple-Target Tracking,” University of California, Berkeley, Technical Report UCB//ERL M05/19, 2005

[14] A.G.A. Perera, C. Srinivas, A. Hoogs, G. Brooksby, and W. Hu, “Multi-Object Tracking Through Simultaneous Long Occlusions and Split-Merge Conditions,” Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR), 2006

[15] J. Sullivan and S. Carlsson, “Tracking and Labeling of Interacting Multiple Targets,” Proceedings of the European Conference on Computer Vision (ECCV), 2006

[16] P. Nillius, J. Sullivan and S. Carlsson, “Multi-Target Tracking – Linking Identities using Bayesian Network Inference,” Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR), 2006

[17] J. Liu, X. Tong, W. Li, T. Wang, Y. Zhang, and H. Wang, “Automatic player detection, labeling and tracking in broadcast soccer video,” Pattern Recognition Letters, vol. 30, no. 2, pp. 103-113, 2009

[18] A. Bochkovskiy, C. Y. Wang, and H. Y. M. Liao, “YOLOv4: Optimal Speed and Accuracy of Object Detection,” arXiv:2004.10934 [cs.CV], 2020

[19] N. Wojke, A. Bewley, and D. Paulus, “Simple Online and Realtime Tracking with a Deep Association Metric,” arXiv:1703.07402v1 [cs.CV], 2017

[20] J. Komorowski, G. Kurzejamski, and G. Sarwas, “DeepBall: Deep Neural-Network Ball Detector,” arXiv:1902.07304v1 [cs.CV], 2019

[21] C. Kim, F. Li, A. Ciptadi, and J. M. Rehg, “Multiple hypothesis tracking revisited,” Proceedings of the International Conference on Computer Vision (ICCV), pp. 4696–4704, 2015

[22] S. H. Rezatofighi, A. Milan, Z. Zhang, Qi. Shi, An. Dick, and I. Reid, “Joint probabilistic data association revisited,” Proceedings of the International Conference on Computer Vision (ICCV), pp. 3047–3055, 2015

[23] A. Bewley, G. Zongyuan, F. Ramos, and B. Upcroft, “Simple online and realtime tracking,” Proceedings of the IEEE International Conference on Image Processing (ICIP), pp. 3464–3468, 2016

[24] B. Hariharan, P. Arbelaez, R. Girshick, and J. Malik, “Hypercolumns for object segmentation and fine-grained localization,” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015

[25] T. D’Orazio, M. Leo, N. Mosca, P. Spagnolo, and P. L. Mazzeo, “A Semi-Automatic System for Ground Truth Generation of Soccer Video Sequences,” Proceeding of the 6th IEEE International Conference on Advanced Video and Signal Surveillance, Genoa, Italy, 2009

[26] D. Speck, P. Barros, C. Weber, and S. Wermter, “Ball localization for robocup soccer using convolutional neural networks,” RoboCup 2016: Robot World Cup XX, 2017

[27] V. Reno, N. Mosca, R. Marani, M. Nitti, T. D’Orazio, and E. Stella, “Convolutional neural networks based ball detection in tennis games,” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2018

[28] T. Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. Lawrence Zitnick, and P. Dollár, “Microsoft COCO: Common Objects in Context”, arXiv:1405.0312 [cs.CV], 2014

[29] Z. Zivkovic, and F. Van der Heijden, “Efficient adaptive density estimation per image pixel for the task of background subtraction”, Pattern Recognition Letters, 2006

[30] Zivkovic Z., “Improved adaptive gaussian mixture model for background subtraction”. Proceedings of the 17th International Conference on Pattern Recognition, 2004

[31] T. Trnovszky, P. Sykora, and R. Hudec, “Comparison of Background Subtraction Methods on Near Infra-Red Spectrum Video Sequences,” Proceedings of the International scientific conference on sustainable, moder and safe transport, 2017

[32] W. L. Lu, J. A. Ting, K. P. Murphy, and J. J. Little, “Identifying players in broadcast sports videos using conditional random fields,” Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Colorado Springs, USA, pp. 3249-3256, 2011