Visualization of large data set on Small screens

HAMMAD ALI

MS(Computer)

Supervisor: Veronica Estrada Galiñanes

A thesis submitted in fulfilment of the requirements
for the degree of
Master of Computer Science (Data Science)

Faculty of Science and Technology
The University of Stavanger
Norway

14 June 2020

Acknowledgment

I would like to thank my Mother, who love me more than anyone in this world can ever do. I would like to thank my Father, who always provided me with an immense confidence, to tackle every problem in life. Thank you to my supervisor Veronica Estrada Galiñanes, for giving me guidance and appreciation through out my work. Thank you to my brothers for not believing in me, which gave me an extra push. I would like to thank my loving wife, whose love for me has never lessened. And last but not least, I would like to thank all the cups of coffee which made me stay up, through out the night.

Abstract

Representing large, valuable information in a comprehensible manner has been a key challenge of 21st century. Human cognition relies mainly on their visual system. However, visual analysis includes both, visual complexities, as well as computational challenges. Visual complexities include the limitations of the display devices; whereas computational challenges involve the complex algorithm to extract and process the information from large data sets. This thesis aims to devise strategies in advancing the techniques used for visual analysis in the sports industry. Different algorithms are used to extract the location of the player from a video stream of football game. The obtained information is processed in a way, to be presented on a smaller screen like cell phones. The integration between the two devices is managed by a cloud, which works in real time. Furthermore, conclusions are made based on the average precision and frame throughput. Moreover, most processes are automated to minimize the human input and be adopted by multiple application domains.

List of Acronyms

AI Artificial Intelligence

API Application Programming Interface

AUC Area Under the Curve

BS Background Subtraction

BIC Bayesian information criterion

CNN Convolutional Neural Network

csv comma-separated values

DOM Document Object Model

DNN Deep Neural Network

FCL Fully Connected Layer

FCNN Fully Connected Neural Net

FN False Negative

FP False Positive

FPS Frames per second

GMM Gaussian Mixture Model

GPS Global Positioning System

HSV Hue Saturation Value

IOU Intersection over Union

JSON JavaScript Object Notation

KDE Kernel Density Estimation

MOT Multiple Object Tracking

MLP Multilayer perceptrons

NMS Non Maximum Suppression

PDF Probability Density Function

R-CNN Region-based Convolutional Neural Network

ReLU Rectified linear unit

ResNet Residual Neural Network

RGB Red, Green, Blue

RNN Recurrent Neural Network

ROI Region of Interest

RPN Region Proposal Network

SE Structuring Elements

SORT Simple Real time Tracking

SSD Single Shot Detection

TAM Temporal Averaging Method

TN True Negative

TP True Positive

VGG Visual Geometry Group

VOC Visual Object Classes

XML eXtensible Markup Language

YOLO You Only Look Once

Contents

List of Figures

CHAPTER 1

Introduction

Sports analysis has always been of major interest by elite clubs, it has proven to be correlated with the development of an athlete's performance as well as team capabilities. Analysis in the past, was mainly done by viewing the footage, captured during the competition or training environment. Performing analysis using this method was very labour intensive and could only be performed after the match has been concluded.

Post advancement in technology, the focus was shifted to using the sensor technology, where each player would be needing to wear a sensor, which in turn would provide useful information, such as position of the player, speed and acceleration . An example of this sensory technology is used at Alfheim Stadium - the home arena forTromsø IL (Norway), which employed radio-based system, called ZXY Sports Tracking [2]. This sensor based technology collects the data in real-time with high accuracy and platforms like "Bagadus" [3] provided an interface for data collected, which allow coaches and sports scientists to make more informed decisions.

Soccer being game of two teams competing, the data of the opposing team is considered to be of high importance. Since the data collected using sensor technology is made confidential by every team, only the video footage can be used to analyze the performance of other team and its players. Elite clubs with high budget employ number of analyst, to extract information regarding players of the opposition, from the video footage and provide valuable feedback. This can not be the case for small clubs with low budget.

Recent advancement in computer vision, allow scientists to extract useful information from an image or a video stream. These techniques, if applied to a video stream of a football game, can extract information like position of the player and other matrices can be derived from

this positional data. This thesis presents a system which would extract positional data of the players from a video footage and display it on to a small screen.

1.1 Problem Identification

The current systems used in sports analysis require expensive equipment and team of analysts, which makes it difficult to be employed by small clubs. In this thesis I would be exploring the latest techniques used for Object detection and Multiple Object Tracking (MOT), to extract positional data of the players from both teams, using a video footage and provide an animated representation of the collected data on to a mobile phone in real-time. This would allow small clubs with low budget, to employ this system and use the data collected in enhancing their players and teams performance.

1.2 Scope and Limitations

This thesis will specify design and implementation of a prototype to perform a real-time data analysis of a football game. The system shall be able to capture and analyze the positional data of a player and provide a user interface to be viewed on a mobile phone.

- Various object identification techniques will be explored for the purpose of identifying the players on the football pitch. These techniques will be evaluated on the basis of their accuracy and processing time, to build a robust system.
- The system identifying and processing the positional data of the players, will store the detections in a real-time data base using cloud solution.
- For the purpose of visualization a mobile phone application will be developed, which would be integrated with the cloud data base to perform the analysis in real-time
- Necessary tasks needed to be performed by the system will be automated, so the users with low level of technical background would be able to use the system without any hindrance.

1.3 Outline

The thesis has been divided into 8 chapters, with chapter 2 providing only the details on the data used and created for the development of the system. Chapter 3 outlines a brief introduction of detection algorithms currently researched and introduced by scientific community.

Chapter 4 explores different algorithms used for the purpose of player identification on the football pitch. Whereas, The quantification techniques used for the evaluation of these algorithms are provided later in chapter 7. Chapter 5 and Chapter 6 explains the pre-processing and technique used for visualization, respectively.

CHAPTER 2

Data set

2.1 Video Data

The data set used for this project was captured at Alfheim Stadium - the home arena for Tromsø IL (Norway) [4]. This comprised of video footage depicting football games played between Tromsø IL and three other teams Strømsgodset, Anzhi and Tottenham Hotspurs. The videos of played games are generated into 3-second individual video clips encoded with H. 264 compression; which are shot by array of camera covering each part of the football field. To have a complete angle in view, footage from different cameras are stitched together to produce a panoramic view, by the data set provider. Since videos are in short 3-second intervals, all the clips were concatenated by the use of ffmeg [5].

FIGURE 2.1. 3 different Camera Angles from match against Strømsgodset

The stitching implementation was found best in the match against Tottenham Hotspurs, as other panoramic views were shot behind the public stands. This resulted in, the footage having a lot of noise, of spectators walking in front of camera. To create the panoramic view of figure [2.1], a stitching algorithm, developed by an open source library OpenCV, which relies on key point feature matching, was used. The stitched video revealed the angle similar

to figure [2.3], which is a distortion in angle and creating a fish eye view, shown in figure [2.2]. For this reason, stitching was taken off of the pipline and match played on 2013-11-28 between Tromsø IL - Tottenham Hotspurs was taken under consideration

FIGURE 2.2. Stitched Panoramic View of 3 cameras shown in 2.1, using OpenCV stitcher

FIGURE 2.3. Panoramic view from match against Tottenham

The dataset also included the ground truth of players position. These positions were collected by ZXY SPORTS TRACKING SYSTEM which relies on radio-based signaling. Data collected from this technology was found to be more accurate than GPS [6]. Furthermore, data contained values like 'timestamp', 'direction', 'speed' etc. as shown in table 2.1.

timestamp	tag-id	xpos	ypos	heading	direction	energy	speed
2013-11-03 18:30:00.000612	1	31278	31278	49.366	2.2578	3672.22	3672.22
2013-11-03 18:30:00.013407	3	74.5904	71.048	-0.961152	0	2.37406	0
2013-11-03 18:30:00.004524	11	45.386	49.8209	0.980335	1.26641	5614.29	3672.22

TABLE 2.1. First 3 entries of the comma-separated values (csv) file containing data related to video

2.2 Manual Annotations

For the purpose of training deep learning models discussed in section 4.2, a custom data set was created. This data set included the manual annotations of the players present on the football field. $\sim$ 150 images were extracted from the video stream, every 15 seconds and were manually annotated. The annotations included the presence of the player, defined by a bounding box around it. This task was performed using labelImg [7], an open source software, which helps to draw bounding box around an object and stores the coordinates of the bounding boxes (see figure [2.4] for reference), in eXtensible Markup Language (XML) format. The yielded XML files for each image contained the name and dimensions of the image along with the objects and their bounding box coordinates.

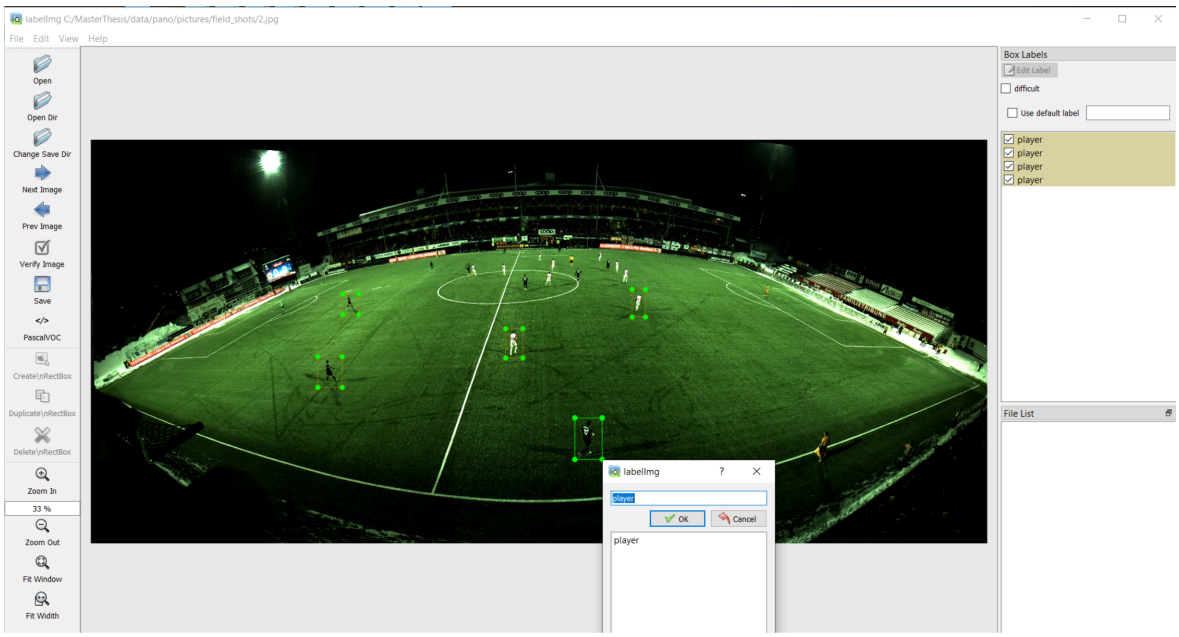


FIGURE 2.4. Screen Shot of LabelImg, to annotate the objects in the image

For the purpose of training deep learning models, these XML files along with the image it represents, were split into 2 folders, train and test. These XML files were concatenated and stored into a csv format, where each csv file contained the information of all the files present in each folder. The columns included in these csv files comprised of, filename, width, height, class, xmin, ymin, xmax, ymax (as shown in table (2.2)). Here **filename** and **class** is the name of the image file on which annotations were made and the class of the object annotated, respectively. **Width** and **height** are the width and height of the image in pixels, and rest of the entries define the bounding box around the object, in pixels.

filename	width	height	class	xmin	ymin	xmax	ymax
100.jpg	4450	2000	player	3276	684	3307	765
100.jpg	4450	2000	player	2456	788	2499	899
100.jpg	4450	2000	player	2435	623	2480	702
100.jpg	4450	2000	player	2420	578	2448	647

TABLE 2.2. First 4 entries of the csv file generated form the XML files produced by labelImg

CHAPTER 3

Literature Review

3.1 Sports Analysis

Sports Analysis is the technique of extracting information during a competition or training environment, which can assist in improving the performance of an individual or the team. Almost all the major clubs or teams, competing in a competition have their own analysts. These analysts are expert at recognizing patterns by viewing the video footage or the live game being played. The elite clubs, hire team of analysts to collect data for each individual player, which then can be analyzed to optimize the result. The data collected includes the position of the player, distance travelled by the player, goals scored, passes completed etc.. This is how statistics of the teams are created and presented in a way which can easily be understood by the coaches and players themselves.

Early days the data was collected by manually viewing the video recording of the event, which was a very tedious task. Later after the evolution of sensory technology, the players had to wear sensors, which would give their positional data and other statistics such as, distance covered, acceleration and speed were computed from this positional data. Collecting data using sensory technology, is limited to acquiring information of ones own team and their players. Data for the other team is collected using the mouse on each individual frame of a video footage. Due to tremendous amount of effort involved in collecting the data, small clubs have not been able to perform the sports analysis on their players, to the fullest.

3.2 Background Subtraction

Background Subtraction (BS) models were one of the earliest models in the field of computer vision. The main idea behind BS was to differentiate between the background and the foreground pixels in a frame of a video stream. This was used widely in computer vision tasks, such as, video surveillance, tracking and even human pose estimation. The main idea behind all the models introduced, was to achieve high accuracy at differentiating, background from the foreground. It was first achieved by taking an absolute difference between a static background and the moving pixels on top in a video stream. Using this simple technique, masks of the moving pixels could be obtained and hence location of the moving objects could be extracted.

(a) (b) (c)

FIGURE 3.1. Absolute difference between the image (a) and image (b) shown in (c)

This technique worked the best, with a stream of video coming from a static background. If the source of the video capturing device was found to be moving, the background subtraction method, using absolute difference would not work. This technique also widely depends on the light conditions, since the algorithm is subtracting each pixel of two images, the variance luminosity, would also effect the result and would not be able to yield good results.

3.2.1 GMM

This model is the modification of One Gaussian (1-G), which models background pixel with a probability density function (PDF) learned using series of frames. Thresholding pixel values from PDF can provide with background and foreground. Pixels with low probability are

considered to be moving objects and corresponds to foreground segmentation. Gaussian distribution $\mathcal{N}(\mu_{s,t},\ \Sigma_{s,t})$ of every background pixel can be used to account for noise. Here $\mu_{s,t}$ and $\Sigma_{s,t}$ corresponds to average background color and covariance matrix at pixel s and time t respectively.

In GMM multimodal PDFs were used, which proposes the use of K Gaussians, to model every pixel. PDF of each pixel in this method can be found by using equation (3.1).

$$P(I_{s,t}) = \sum_{i=1}^{K} \omega_{i.s.t} \times \mathcal{N}(\mu_{i,s,t}, \Sigma i, s, t) \tag{3.1}$$

In (3.1) $\mathcal{N}(\mu_{i,s,t}, \Sigma i, s, t)$ is the i^{th} Gaussian model and $\omega_{i,s,t}$ is its weight. The weights are updated in series of frames using the formula (3.2) where α is the learning rate predefined. This way we can keep track of history in the frames and define the moving pixels and background pixels on the basis of their values, with lower ones corresponding to the background. Using the equation (3.3) we can calculate the distance matrix, hence acquiring the mask of moving pixels for color image or (4.2) for gray scaled.

$$\omega_{i,s,t} = (1 - \alpha)\omega_{i,s,t-1} + \alpha \tag{3.2}$$

$$d = (I^R_{s,t} - B^R_{s,t})^2 + (I^B_{s,t} - B^B_{s,t})^2 + (I^G_{s,t} - B^G_{s,t})^2 \tag{3.3}$$

Here R, B, G denotes Red, Blue and Green color channels respectively.

3.2.2 Kernel Density Estimation KDE

This model uses a unstructured approach to model the multimodal Probability Density Function (PDF). The preposition was to use Parzen-window to estimate the background pixels [8]

$$P(I_{s,t}) = \frac{1}{N} \sum_{i=t-N}^{t-N} K(I_{s,t} - I_{s,j}) \tag{3.4}$$

A global threshold θ_{fg} for all the images is used, to define the foreground and background, i.e, if the $P(I_{s,t}$ is smaller than θ_{fg} the pixel belongs to the foreground and other wise it is a background pixel. In (3.4) $I_{s,j}$ is pre-estimated [9].

3.2.3 Morphological Transformation

Morphological Transformations also known as mathematical morphology, are the range of Non-linear image processing techniques. Morphological transformations are mostly applied on a binary image, to reduce the noise as much as possible without losing essential features. These transformation techniques use a small shape matrix known as Structuring Elements (SE), analogous to the kernel in convolution, to check the neighbouring pixels in an image. The SE defines the nature of morphology being applied to the image. Two basic, commonly used transformations are Erosion and Dilation [10].

3.2.3.1 Erosion

In this transformation, SE slides on the 2D binary image, since the binary images are represented as 1 or 0, if all the pixels under the SE are not 1, all the pixels are replaced by 0. Using this technique the border of an object present in a binary image are reduced as well as any pixel which is not part of the object is also counted as noise and gets replace by 0.

3.2.3.2 Dilation

Acting on the same technique of sliding window, dilation enlarges the width of maximum regions. If any value under the SE is 1, all the 0 pixels are also replaced by 1. This makes the boundaries of the object in a binary image to inflate, hence making small object in an image bigger.

3.2.4 Contours

Contours are the representation of joining curves in a binary image. For topological structural analysis on binary image, contours are used to retrieve the shape of the object. The idea behind finding contours is that all the connected pixels in a binary image belong to the same object. This is done using "Green Theorem" [11], which takes the line integral of a closed curve and double integral over the bounding plane. Using contours the inner and outer boundary on an object, with similar intensity, can be located.

3.3 Neural Networks

Vaguely inspired by human brain, Neural Networks are set of algorithms which perform tasks without being programmed specific rules for execution. Since every data, be it images, sound, text or timestamps can be represented in a numerical form, these algorithms can be used to recognize patterns in numerical vectors. Neural networks can be considered as algorithms which can learn to classify and cluster the labeled or unlabeled data, based on their similarities.

A neural network can also be understood as Multilayer perceptrons (MLP), where, a **perceptron** is a unit of a neural network, which performs certain computations and out puts a numerical result. A perceptron introduced in 1957 by Frank Rosenblatt, receives an input as a numerical vector($X_1, X_2, ..., X_n$) and multiply each entry in the vector with their corresponding weight ($w_1, w_2, ..., w_n$) and add them together. The sum of all these values are passed through an activation function (f) , which yields an output within a range, depending on the activation function used. Based on this output, features or business intelligence can be gained from the perceptron, figure 3.2.

Multiple perceptrons can be stacked together in layers to from a neural network (figure 3.3) . These networks can solve much complex classification problem, as more weights are introduced to achieve better results. Following the same principle, much larger models can be created, whose size depends on the complexity of the problem.

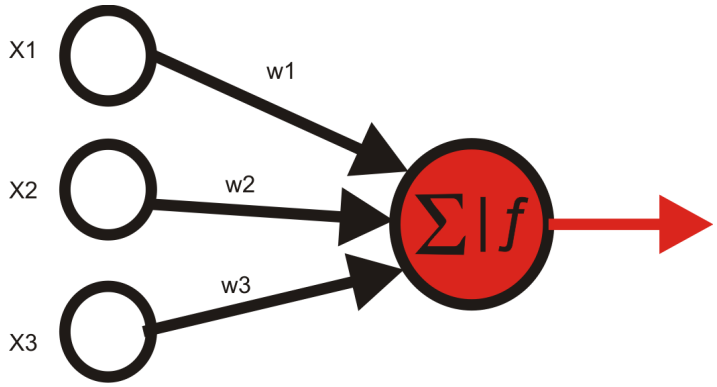


FIGURE 3.2. Perceptron, Single unit of a neural network

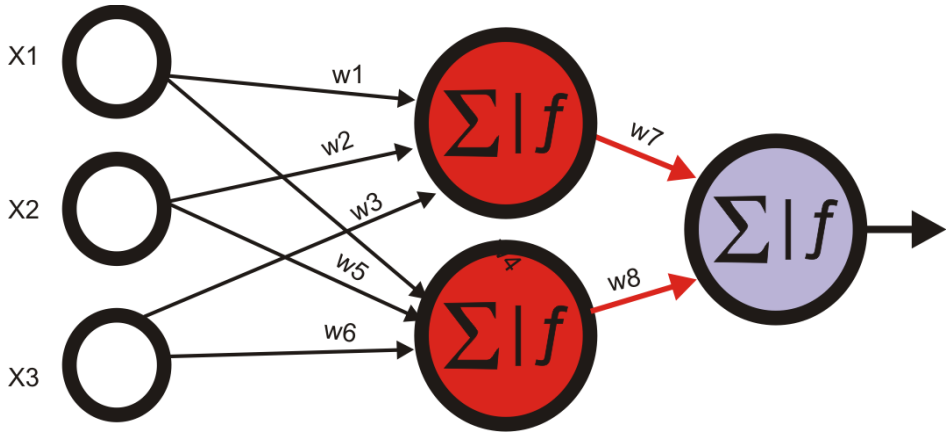


FIGURE 3.3. Neural Network with 1 hidden layer

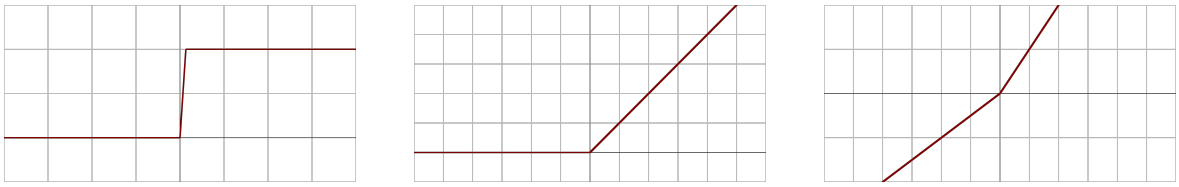
3.3.1 Activation Functions

Several activation functions are introduced hitherto, by researchers. These activation functions are used to bound the result produced by last node of each perceptron, with in a range of values. Based on the values produced by these activation function, a classification problem can be solved. One of the most basic activation function is binary step function (figure 3.4 (a)), which yields either 0 or 1 representing "ON" or "OFF" of the unit, using (3.5). More advanced functions include Rectified linear unit (ReLU) (figure 3.4(b)) or Leaky ReLU (figure 3.4(c)), which yields the value between $(0, \infty)$ and $(-\infty, \infty)$ respectively, using equations (3.6) , (3.7). These advanced activation functions allow the network to predict multiple classes.

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases} \tag{3.5}$$

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} \tag{3.6}$$

$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{3.7}$$



(a) (b) (c)

FIGURE 3.4. Activation functions on a plot

3.3.2 Optimization

It is hard to predict the correct value of weights to achieve the desired classification at very beginning. For this purpose, optimization algorithms are used to update the values of weights and reduce the error in the output, during the training of a network. When network is shown the labeled outputs which are required, error is calculated against these true labels and using the derivative techniques, the error produced by the network is minimized, by updating the weights (figure 3.5). There are several algorithms introduced by researchers, which uses different techniques, to find the global optima of the network. The global optima is considered as the point where the error produced by the network is minimum and network can perform the classification with more accuracy. Upon achieving the certain threshold of accuracy, the network is considered suitable to perform classification on general data.

Some of the best algorithms used for optimization of a neural network includes adaptive moment estimation (Adam) [12] and Gradient descent with momentum [13].

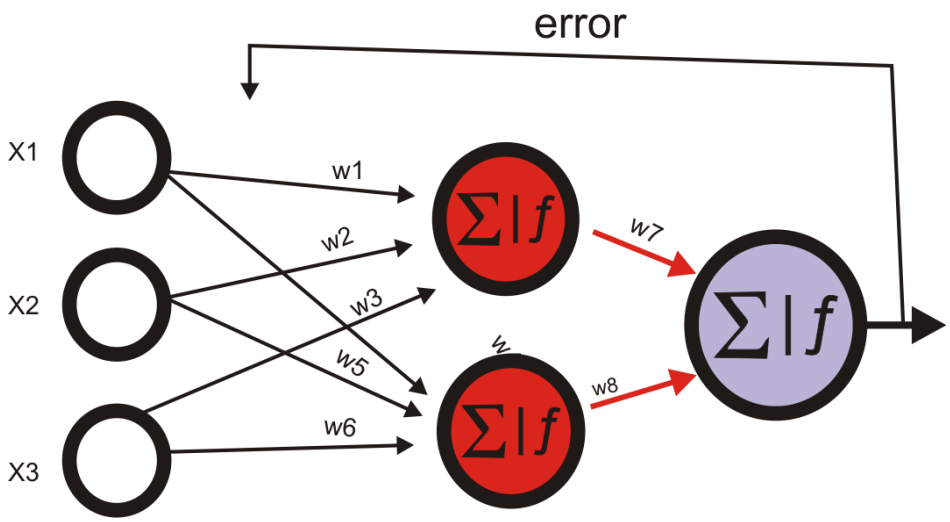


FIGURE 3.5. Optimization of a neural network

3.4 Convolutional Neural Network CNN

An image in computers is nothing but numerical vectors stacked together, where each numerical value represents the pixel value of corresponding channel. We see colors by observing the variance in wavelength produced by the object, light is bouncing off of. This allows us to see colors in a certain range of wavelength and large variety of it produced by the mixture of these waves in photon particle. Computer on the other hand looks at color using the mixture of 3 primary colors, Red, Green, Blue (RGB), which can produce a wide range of colors.

For instance if a colored image has a size of 1024×764 the to number of numerical values used to create this image would sum up to $((1024 \times 764 \times 3)$ 2347008. Using these numbers of inputs in a complex neural network, would increase the number of parameters being used to train the network drastically, as the network grows. To overcome this problem an approach was devised, first introduced in 1980 by Dr. Kunihiko Fukushima, by reducing the size of the image in such a manner that it should not loose the features, important to achieve higher accuracy.

The size of the image is reduced by introducing two layers, i.e. Conv layer and pooling layer. A Conv layer or convolutional layer uses a kernel of pre-defined size which slides on top of the image, on step at a time, covering all parts of the images. The main idea is to reduce

the size of the image by collecting as much information as possible, by covering part of the image which is equal to the size of the Kernel and then move to the next part in the image. The computations happening inside the kernel sliding technique is, all the values in the image gets multiplied by the values in the corresponding kernel cell and then summed up. This way a image of size $(5 \times 5 \times 1)$, using a kernel of size (3×3) will reduce down to (3×3).

Pooling layer on the other hand uses a kernel as well, but in this layer we take the average, maximum or minimum value among the numerical values of the image covered by the kernel. Type of the pooling layer, depends on the functionality required by the network. However, it is used to further decrease the size of the image, so computation for the purpose of classification can be applied. Some of the popular CNN architectures are VGG and ResNet.

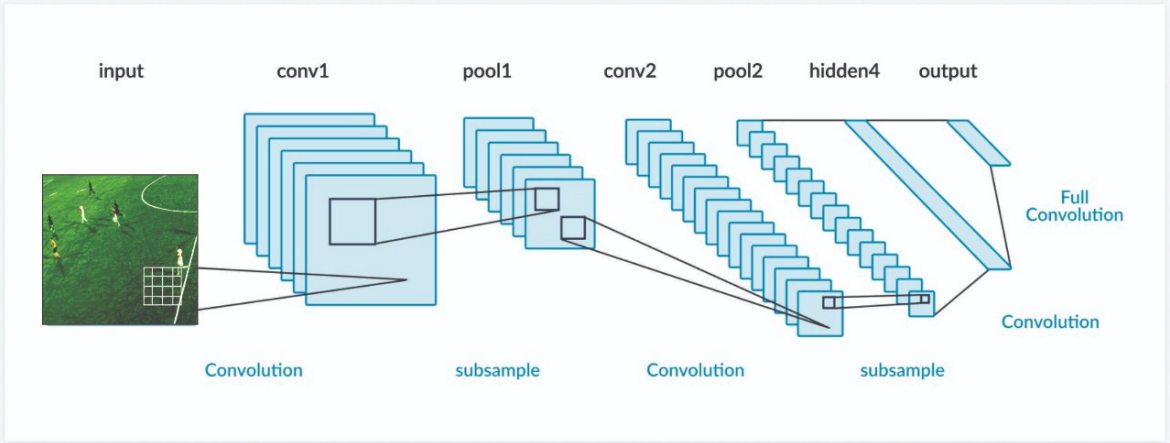


FIGURE 3.6. Intuition behind CNN

3.4.1 Visual Geometry Group (VGG)

This architecture includes 16 convolutional layers, with approximately 138 million parameters [14]. This architecture was a runner up at ILSVRC in 2014, a image recognition competition, hosted by ImageNet. This network is famous for its uniform architecture, which uses (3×3) convolution kernels through out with multiple filters.

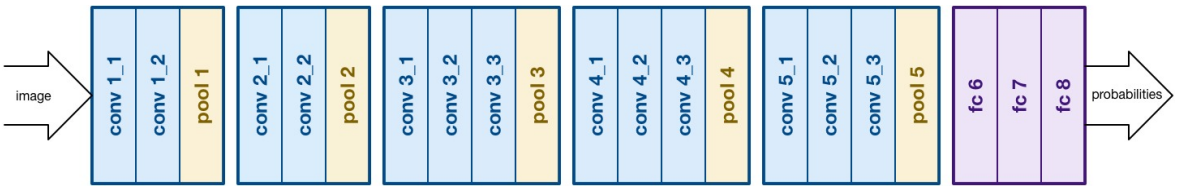


FIGURE 3.7. Layers of VGG

3.4.2 Residual Neural Network (ResNet)

Working on the basis of Recurrent Neural Network (RNN), this network architecture keeps track of the history by skipping to the next layer with the result from the previous layer. This skipping connection method allows ResNet to compute heavy batch normalization and act as a gated units showing similarity with RNN (figure 3.8)

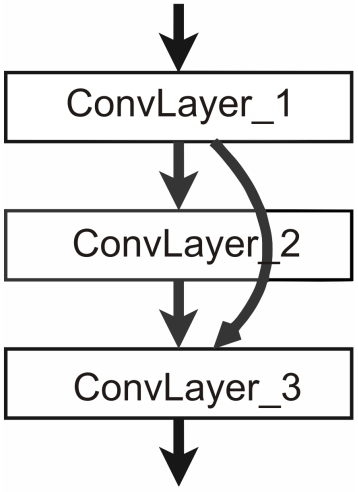


FIGURE 3.8. Intuition behind ResNet

CHAPTER 4

Experimentation

Several methods are proposed in this thesis for the purpose of detecting a football player on the field and comparing the acquired result. They experience different trade-offs, described later in the document. Data collected from the algorithms discussed in this section are fed to procedures mentioned in chapter 5. This apply further preprocessing on the data and refine it before being displayed.

4.1 Background Subtraction

Traditional approach of BS, to detect players on the pitch was applied in this implementation. Background subtraction has proven to be successful, if the video stream is being fetched from the static camera. BS can be implemented using different approaches. Keeping accuracy and throughput speed under consideration two different methods for background subtraction were considered in this thesis. A brief overview of these algorithms is provided in this section.

4.1.1 Absolute Difference

Considered as "quick and dirty" way to localize the moving objects and taking under consideration that lightning of provided footage was constant. Temporal Averaging Method (TAM) [15] was applied on the video footage. The first step of TAM is to extract the background model, which comprises of motionless pixels in a scene. B_t (i.e. background) and can be estimated using (4.1)

$$B_{t+1_{(x,y)}} = \frac{t \times B_{t_{(x,y)}} + I_{t_{(x,y)}}}{(t+1)} \tag{4.1}$$

Where I_t is the current frame, t is the frame count and (x, y) is the pixel coordinate. Using (4.1) we can loop through all the frames in a video stream and calculate a background model shown in figure 4.1. see appendix A, section A1.1, for script.

FIGURE 4.1. Background Model

Each frame from the stream is subtracted pixel by pixel from B_t to extract foreground. Since extracted images contain 3 channels, both I_t and B_t are converted to gray scale first.

$$D_t = |I_t - B_t| \tag{4.2}$$

Using the equation (4.2) mask of a foreground segmentation was obtained and was applied a binary threshold θ_b, to only include values (30, 255), such that any pixel below θ_b will be zero and maximum other wise. The mask obtained doing so, still contained a lot of noise in it. The reason behind was players casting shadows on the field. Since the shadows are moving along side the player they were not considered in motion less pixels and accounted for pixels of foreground. The digital advertisement boards located on the side and back of the pitch change every couple of seconds, thus, were too not considered in motionless pixels (figure 4.2). Morphological transformations were applied to suppress noise, this included multiple

layers of erosion with the SE of 13×13 and a dilation of SE 10×10, having single iteration. Later, SE size of morph transformation were reduced by the factor of 2 keeping the same iteration. These transformations were able to mitigate the shadows of the player but were not able to remove side banners from the mask as can be viewed in figure 4.3. see appendix A, section A1.2 for code snipet.

FIGURE 4.2. Foreground segmentation

FIGURE 4.3. Foreground segmentation after noise cancellation

The presence of noise in the background resulted in False Positive (FP) and when occluded with the players, present on the pitch, made it difficult to separate players from the noise. I

tried to come up with different methods to overcome this problem, which are discussed in section 4.1.3.

4.1.2 Gaussian Mixture Model (GMM)

This technique, models each pixel as a distribution over number of Gaussians (n_c), instead of modeling each pixel as one value. To find the optimal number of n_c Bayesian information criterion (BIC) [16] method was used, which selects the model from finite set of models, components producing low BIC was used as n_c. See appendix A, section A1.3 for code snippet.

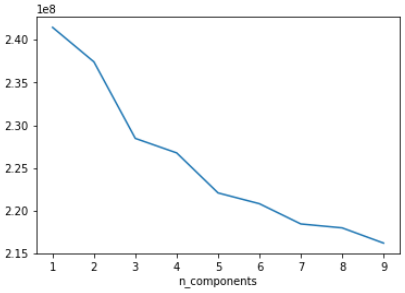


FIGURE 4.4. Number of Gaussians

This background subtraction method, also keeps track of the history of frames, i.e. it uses pre-defined n_h frames to detect the foreground. If the movement is not observed from the previous n_h frames, the object is discarded and considered as background. A value of 5 was for n_h in my case, was found to yield best results, which was found using trial and execution method.

After finding the suitable hyper-parameters, the model was applied on each frame from the video stream. This produced a binary image on which topological structural analysis could be performed. An example of binary image produced by the model, is shown in figure **??**.

Viewing the segmentation, it can be observed that the shadows were not completely removed and would accumulate towards FP. To mitigate this problem Morphological transformation was applied. Using layers of erosion and dilation noise and shadows were removed (figure 4.6). This did not eliminate the problem as a whole but provided decent results, and remaining noise was later handled in section 4.1.3. see appendix A section A1.4, for code snippet.

FIGURE 4.5. Gaussian Mixture Segmentation

FIGURE 4.6. After Morphological Transformation

4.1.3 Finding Contours

For topological structural analysis on retrieved binary image from section 4.1, [17] was used. Using this algorithm we can find the borders of connected components of 1-pixels. This way the 4 border points of the connected pixels (i.e., xMax, yMax, xMin, yMin) yielding the bonding box around an object are obtained. These contour points are used to create a rectangular space indicating the presence of an object inside. The yielded contours also included the points having FP, which in our case are the shadows or advertisement screens.

To manage this problem different series of checks were performed, these mainly included:

- Image was divided in grid to check the area and height as it varied with the depth.
- Area of the contour. (It was observed that area of advertising screens was significantly large and shadows to be really small w.r.t to the player contour)
- Height was always greater than the width of player's contour.

Performing these checks on each calculated contours, players were separated from other objects identified in the binary image. Since these were made to identify contours which represents humans, other humans present in the image were also identified. These included the players on the bench, line-men (referee), security personals, ball boys etc. To fit the criteria of detecting only players on the field currently playing, pixel location approach was devised, explained in chapter 5. See appendix A, section A1.5 for code of this section.

4.2 Deep Learning

Inspired by the recent developments in the area of computer vision, the following method uses the layers of CNN to detect the players on the field. Unlike Fully Connected Neural Net (FCNN), CNN's are used to preserve spatial dimensionality of the image and reduce their size. CNN's being comparatively less computationally expensive works better with image classification and object detection. Different CNN architectures are used in this section to achieve higher accuracy and frame throughput .

4.2.1 Faster R-CNN

Faster R-CNN runs CNN on top of the $\sim$2000 proposed regions produced by Selective Search. Being descendent of R-CNN, Faster R-CNN produces Region of Interest (ROI) using RPN. RPN produces anchor boxes with "objectness" score and 4 coordinates representing the bounding box of the region, using sliding window method at the last layer of an initial CNN. At last FCNN takes the input of proposed regions by RPN to predict object class

(classification) and Bounding box (Regression). The output of FCNN yields a confidence map with class, confidence score (of class detected) and the coordinates of the bounding box.

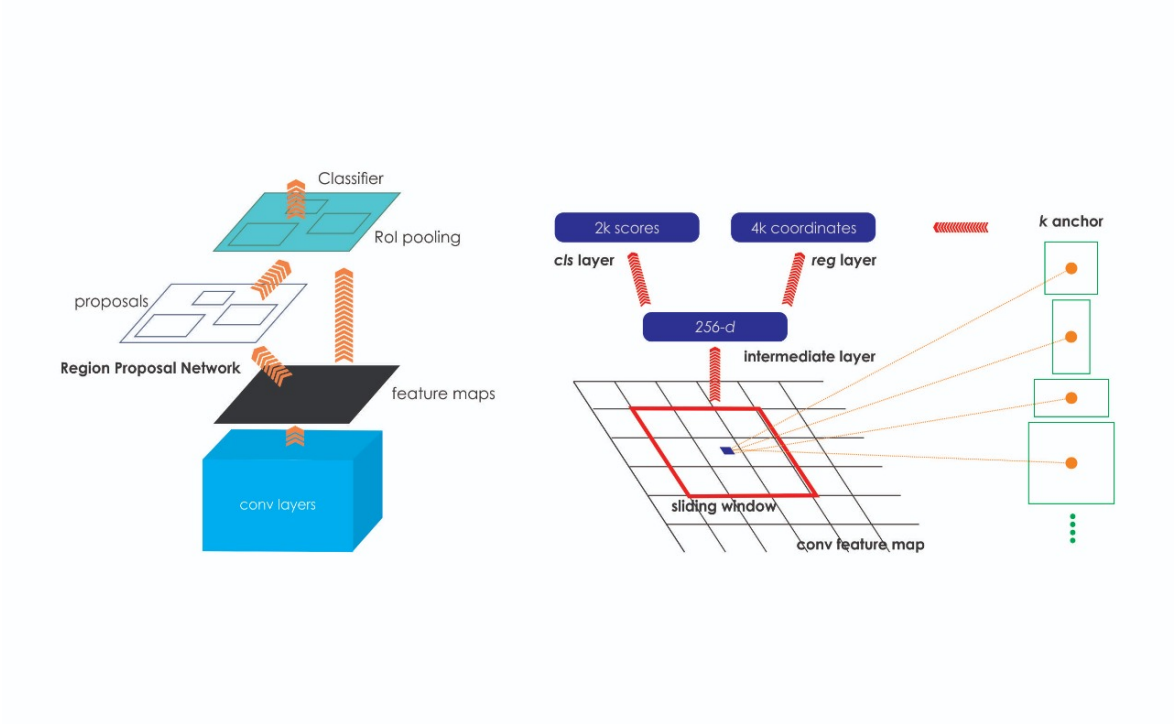


FIGURE 4.7. **Left**: Faster R-CNN, **Right**: RPN

To implement Faster R-CNN, a pre-trained model on COCO data set was downloaded and loaded using Tensorflow API [18]. This model contained a pre-trained inference graph which can detect numerous classes, but for the purpose of this project only class which can detected humans was used. For this purpose a a new .pbtxt format file was created, having only 1 entry, as player class.

The frames from video stream were collected synchronously and converted into a numpy array and then to a tensor, which is a multidimensional array. This tensor helps in performing faster computation and will be containing pixel values of the frame with 3 channels i.e. RGB. Feeding this tensor into the model loaded, yields a dictionary containing all the information regarding the frame. This information includes, number of detections, names of the classes detected, bounding box coordinates of the object detected and confidence of the object detected in the frame. Since the model is prone to have false detection, i.e. predicting a

detection when no object is present, only the detection with confidence score above the threshold $\theta_c \approx 0.3$ are considered.

The bounding box coordinates detected from this model are in normalized form, so these coordinates are multiplied with the height and width of the image to calculate actual coordinates with regards to the image. Furthermore, some bounding box overlap each other, this usually happens due to the multiple detection of the same object or due to the occlusion of the players on the pitch. These overlapping coordinates are suppressed using Non Maximum Suppression (NMS), which works on the principles of clustering proposals by spatial closeness measured by Intersection over Union (IOU) (Jaccard similarity) [19]. Since there could be many frames where players are very close to each other, a large threshold $\theta_t \approx 0.6 - 0.7$ was used to discard a bounding box. Filtered bounding box coordinates for each frame are stored in an array to further reduce the false detections, using methods discussed in chapter 5. See appendix A, section **??**, for complete implementation of Faster R-CNN.

4.2.2 SSD

SSD [20] (by C. Szegedy et al.) released by the end of November 2016 worked on single forward pass for object detection and localization. SSD's was built on venerable VGG-16 architecture. Modification in SSD was to discard the Fully Connected Layer (FCL) at the end of the network and use a set of auxiliary convolutional layers. This enabled the extraction of features at multiple scales and also decrease the size of input in each subsequent layer. Instead of RPN, SSD uses bounding box regression technique inspired by Szegedy's work on multibox [21].

The implementation of the SSD is identical to the implementation of Faster R-CNN provided in the section 4.2.1. For SSD, a model trained on COCO data set was downloaded from Model Zoo. Using similar manner described in section 4.2.1, the model was loaded and each frame was passed through the model. Predictions were collected, filtered and stored in an array for further processing.

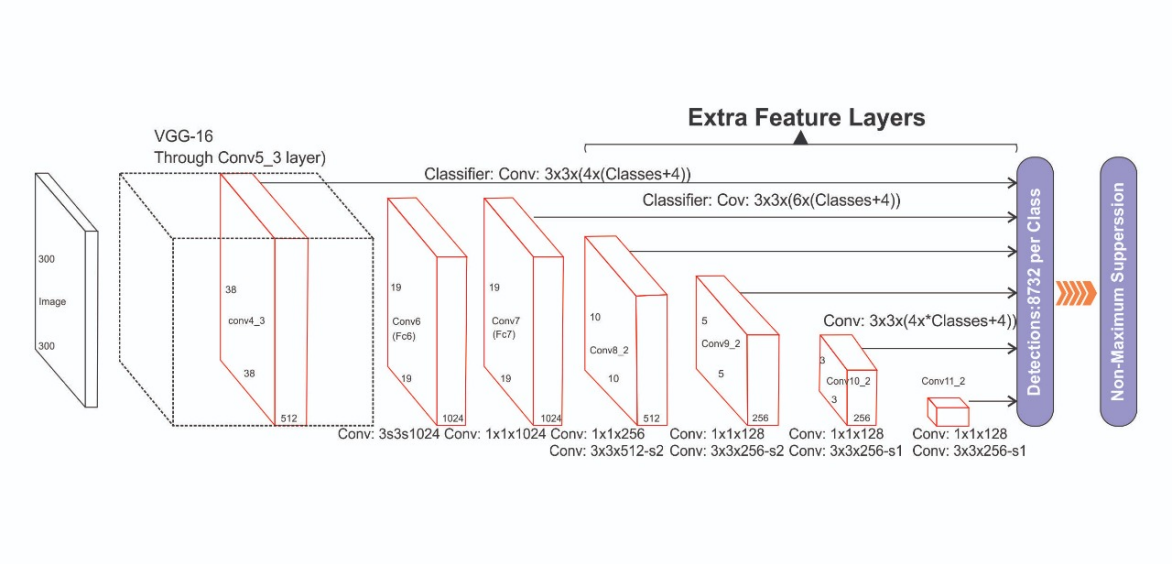


FIGURE 4.8. SSD architecture

4.2.3 YOLO

Comprising of 24 convolutional layers followed by 2 FCLs (figure 4.9), YOLO is able to achieve higher throughput with a less trade-off of accuracy [22] compared to SSD and R-CNN. YOLO uses (1×1) layer instead of inception layer to reduce the input size. It also treats object detection as a regression problem by dividing the image into $(S \times S)$ grid and assigning bounding boxes and class probabilities to the grid cells. These class probabilities reflect how confident the model is that the box contains the predicted class. This results in total of 5 predictions by each $(S \times S)$ gird cell i.e., x, y, w, h, c. The x, y represents the center of the bounding box relative to the grid and w, h are for width and height relative to the image. Here, c is the predicted class confidence. Since it sees the image as a whole pixel by pixel it is less prone to make error in detecting background patches as objects, in training and testing.

YOLO (version 3) was implemented in this project, using Deep Neural Network (DNN) module created by OpenCV [23]. Using DNN a faster through put can be achieved than tensorflow . Firstly, the configuration file and pre-trained weights were downloaded from the official website of YOLO and loaded to the network. Later, each frame from the video stream is preprocessed, where every frame is resized to 416×416, to aid the faster computation and maintain the aspect ratio. Mean subtraction, scaling and swaping of color channels was

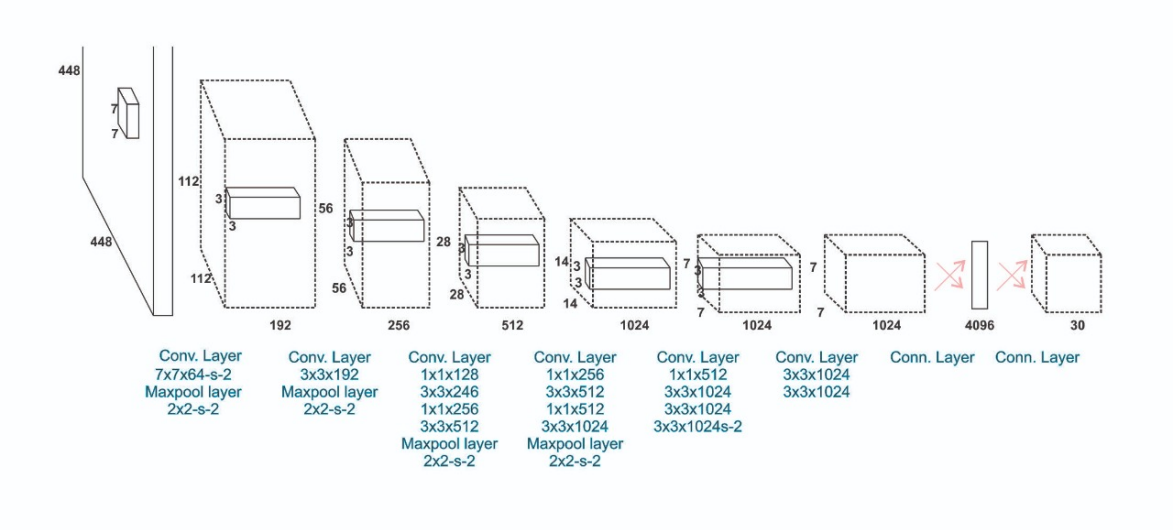


FIGURE 4.9. YOLO architecture

applied to each resized frame, because YOLO uses BGR instead of RGB order. Each of the processed frames are passed through the network for predictions, which yields a bounding box with normalized coordinates and confidence of the detection. Only the predictions with a confidence score, higher than 0.4 were considered valid and rest of the predictions were discarded. The predicted coordinates are normalized and hence were multiplied with frame's height and width to garner actual coordinates. The predicted bounding boxes also included duplicate detections of same object, these duplicate detections were suppressed using NMS method with a threshold value of 0.5. The filtered predictions of each frame were concatenated in an array for further processing. See Appendix A section A2.2 for complete detection implementation.

4.2.4 Observations

All the models explained in this section (i.e 4.2) are using pretrained weights. These weights are trained to classify general classes like (person, car, dog etc.). Although these weights are trained on large datasets like ImageNet [[24]] or coco data set [25]. It was observed that they are not able to perform up to the mark, as the result can be viewed in the figure 4.10 to 4.12.

FIGURE 4.10. Faster R-CNN ResNet50 pretrained on coco data set 2018

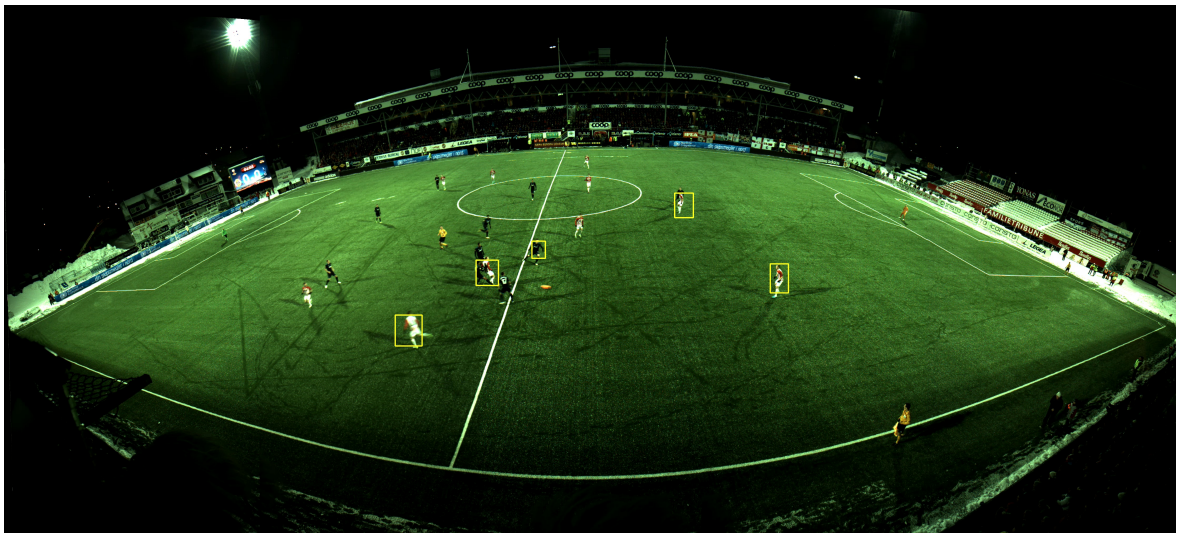


FIGURE 4.11. SSD inception V2 trained on coco data set 2018

4.3 Custom Trained Model

It was hypothesised that since the camera angle is distorted the pre-trained models are unable to work at their best. To tackle the problem a custom data set was created and discussed in section 2.2, was used to re-train the weights of the model. This data set contained 2 folders, train and test, containing images and their annotations, with a ratio of 80% and 20% respectively.

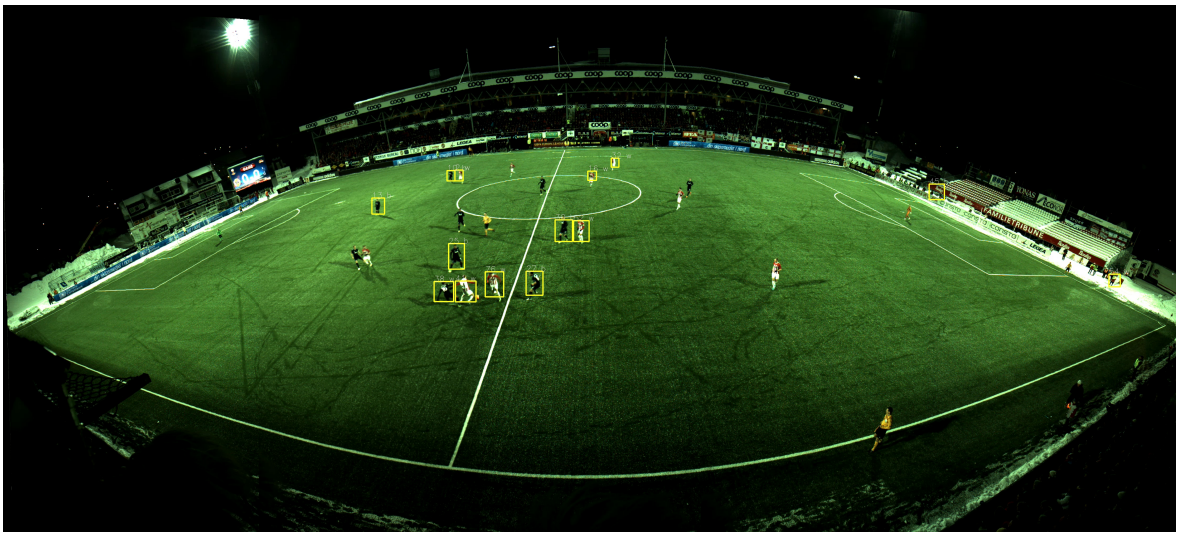


FIGURE 4.12. YOLOv3 with YOLO pre trained weights from yolo

The csv files were converted to Tensorflow's binary storage format, TFRecord, since it is a prerequisite to train the model using Tensorflow API. A latest pre-trained model of choice from Model Zoo was downloaded to acquire model architecture. For this project faster-rcnn-resnet101-coco was downloaded and the configuration file was edited to detect single class, using the images provided in train and test folder. Since training the model requires a lot of computations the model was trained using Tesla P100-PCIE-12GB GPU, for $\sim$ 2 hours or when the model reached the loss between 2 - 1, to avoid over fitting. It took almost 1800 epochs, for model to reach required loss, and snap shot at this checkpoint was saved. The script used for the purpose of training the model is provided by Tensorflow, at this URL.

A frozen inference graph was created from this checkpoint and was stored in a new folder along with data to reach the checkpoint and the edited configuration file of the model used, using the script provided by Tensorflow, at URL.

After testing the model using the code snippet, mention in appendix A, section A2.1, it was observed that the model is able to locate the players on the field with much more accuracy. A pictorial result of custom trained model, is shown in fig 4.13

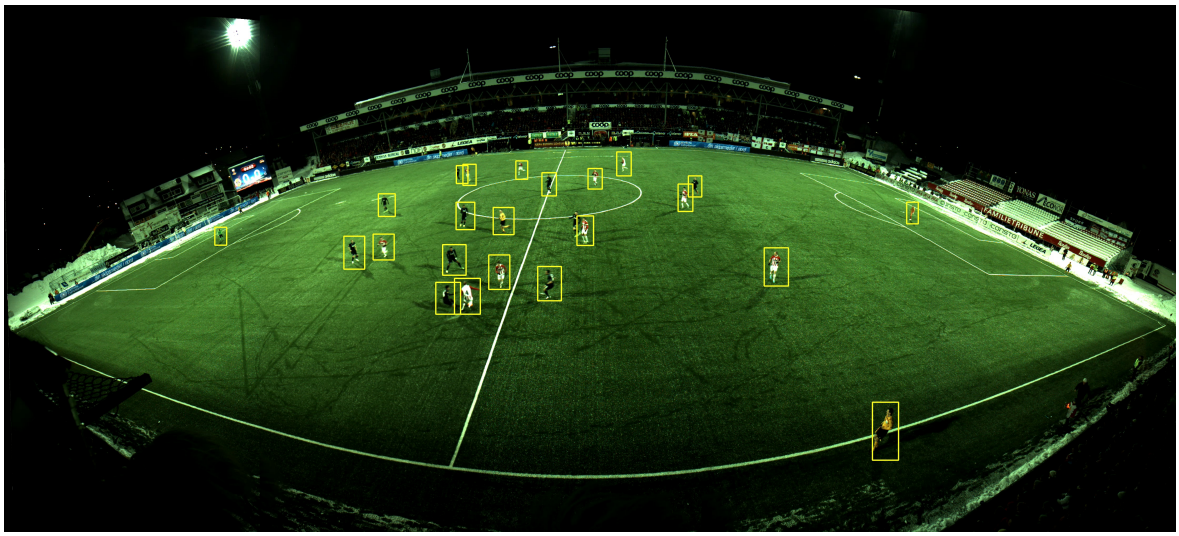


FIGURE 4.13. Custom trained model on faster-rcnn-resnet101-coco with

4.4 Deep Simple Real time Tracking (SORT)

Models to perform detection are computationally very expensive and take large processing time. To increase the throughput rate of the model, object tracking was introduced. After detecting the players in a frame, the coordinates of the predicted bounding boxes were stored in an array and predictions for the next frame were generated using the centroid tracking. This predicts the state of moving player in next frame, using its centroid and velocity. Once the number of detected objects are 23, i.e. 22 players and one referee, the detection are fed to centroid tracking, which keeps track of the moving player.

Centroid tracking can some time loose the track of a moving player or stop working when two bounding box are occluded. To cater this problem, detection model was asked to make new predictions every 30 frames and new tracking coordinates were generated. Once the model was able to make the required amount of predictions it was observed that the throughput rate of the model was slightly increased, as shown in table 7.1. See appendix A section A3 for code snippet.

CHAPTER 5

Player Identification

The task for the Object detection (i.e. players) was performed in chapter 4. The methods explained in the chapter 4 were able to perform the detection but were unable to differentiate between the players on the football field and the referees, spectators, security guards etc.. To make the computer understand the difference, an algorithm is devised which is explained in the section 5.1. The computer vision also could not identify which team the player belongs to. Based on the color of the jersey being worn, method explained in section 5.2, was used to perform the task of team identification.

5.1 Field Polygon

The player detecting algorithms preformed in chapter 4 are not able to distinguish between players inside the field or on the bench located in the background. The object detection algorithms, also detected the security personals and the ball boys present in the frame since they share the same anatomy or produce the contour with similar characteristics of a human. It was observed the difference between players on the pitch and off the field, is the position it self. This means that the detection observed inside the field can me marked as playing players and other observations can be discarded.

Since the field is forming a convex shape due to the distortion in the camera angle, hard coding the coordinates of the field would not perform best results. Therefore, an algorithm was devised which would require the user to identify the boundary points of the field. The user can pick **x** number of boundary points on the edge of the field and create a polygon with **x** points. This polygon will be considered as the football field and only the detection inside

this polygon will be considered valid. For the purpose of understanding, a polygon with 9 points is displayed in the figure 5.1.

FIGURE 5.1. Polygon of 9 selected points on the football field

The detected players from the chapter 4 are provided in the form of bounding boxes, i.e. 4 coordinates with top-left and bottom-right coordinates around the player. Since the players at the back can have their heads outside the field and still be playing due to camera angle, the feet of the players were put in the context. The coordinates of the feet were calculated by using the equation (5.1), where f_{cord} are the x and y coordinates of the feet.

$$f_{cord} = \frac{x_{min} + x_{max}}{2}, y_{max} \tag{5.1}$$

The f_{cord} obtained were checked if they fall inside the coordinates of the polygon drawn on the borders of the field. If the f_{cord} were observed to be inside the polygon, the detection related to it was considered to be a playing player. Doing so, the computer was able to identify all the players playing and false detections outside the field were discarded. The complete script of this section is provided in Appendix [A] section A3.1 and A3.2.

5.2 Team Detection

After filtering the detected players on the field we need to identify the team they belong to. The football match used for this project was played against teams wearing white and black kits. Just like human vision, this algorithm also relies on the color of jersey to identify the team. Since we humans distinguish the colors in terms of their wave length, the computer reads the images with RGB values of each pixel and differentiates them on the bases of these values.

To identify the team of the player we first need to calculate the range of pixel values, white and black colored jerseys fall under. For this purpose a script was used, mentioned in Appendix A section A3.3. This scripts reads an image with RGB values and converts it to HSV, as HSV is better to separate color information from luminance. This script provides with 6 toolbars to find the range of HSV values i.e. upper and lower range of each color. This can be viewed in the figure 5.2, where upper and lower range of white color is detected manually. The same way the color ranges for black team and referee jersey's were obtained.

The obtained ranges of HSV values were used to identify the player team. To perform this task, each bounding box detection was cropped out (as shown in figure 5.3) of the frame, to perform the masking method. As the images are being cropped from the frame of video stream they are in RGB format, these images were converted into HSV format. All the pixel values which fall inside upper and lower range of corresponding colors were stored in a temporary variable called mask and rest of the values not in range were discarded. A pixel by pixel **and** operation was performed on these images with the mask calculated. The resulting multi-dimensional array contained the active pixels produced after masking. This array was converted to gray scale to reduce the dimensionality and calculate the number of active pixels. The active pixels created a binary image and represented the strength of color present in the image.

The masking method was performed for each color to be detected, which in our case is white, black and yellow for Tromsø IL, Tottenham Hotspurs and referee respectively. Based on the strength of the color present in the cropped image, the team of the player was identified. The

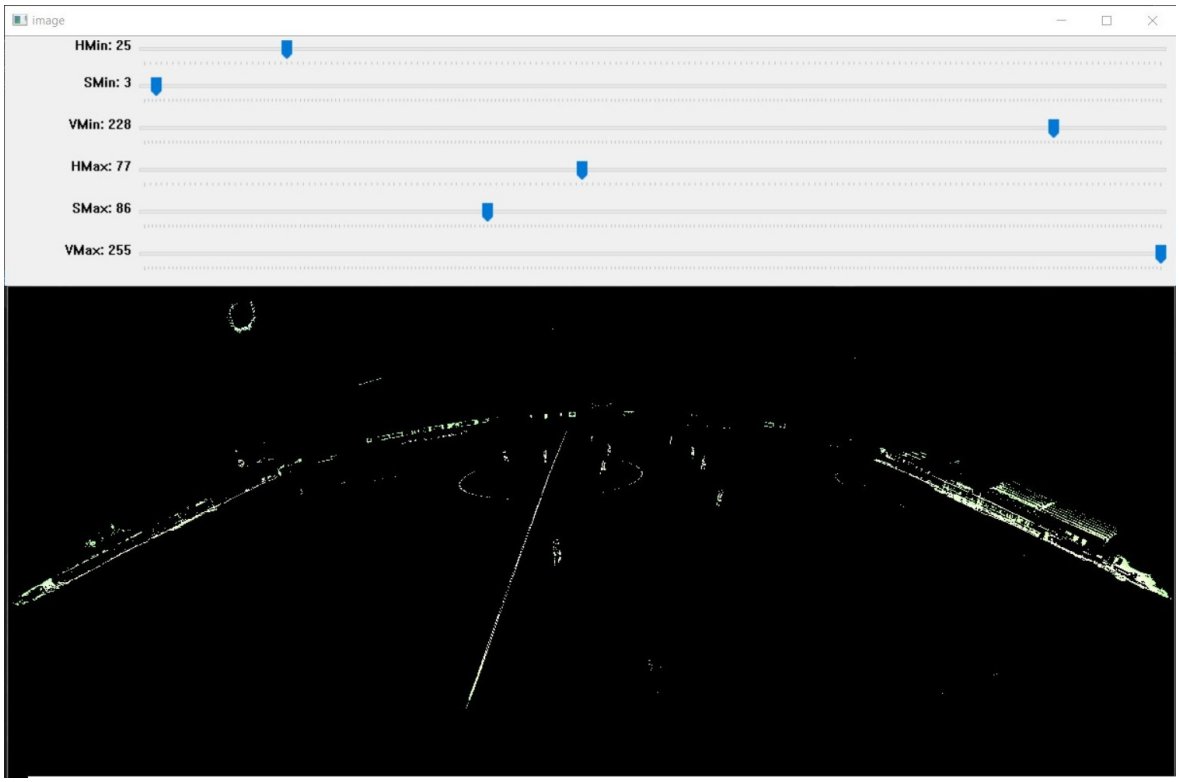


FIGURE 5.2. HSV upper and lower range of white team with toolbars

figure 5.4 shows the players with number of white pixels present in the bounding box. For the code of this method refer to appendix A section A3.4.

FIGURE 5.3. Players cropped with bounding box coordinates

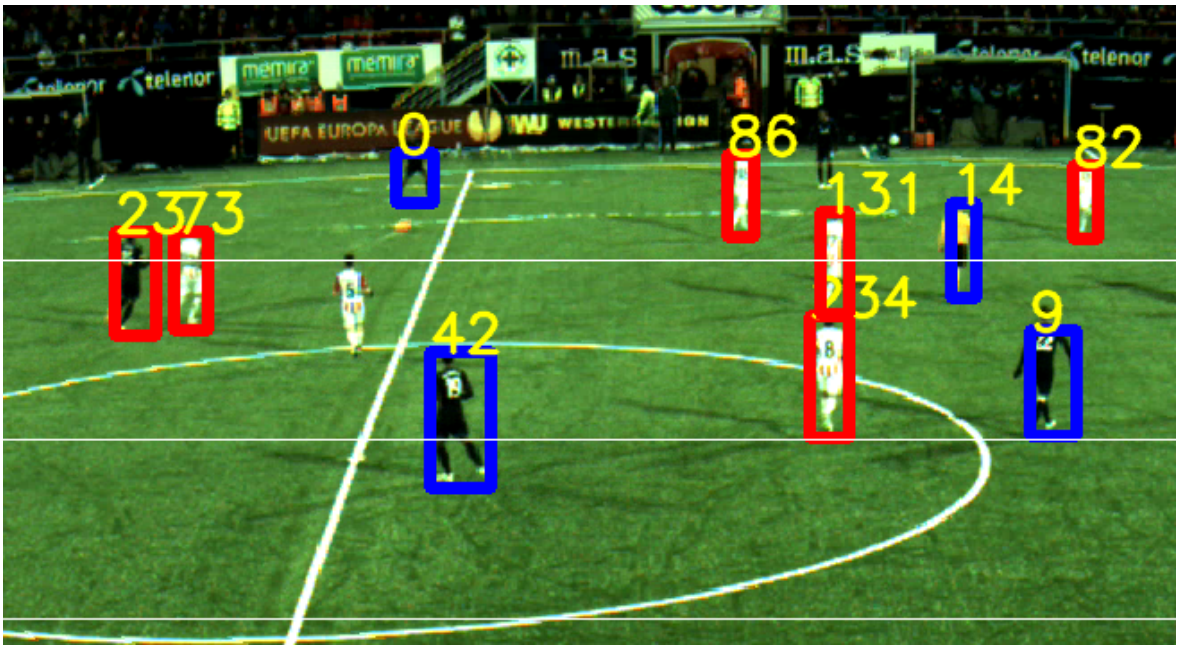


FIGURE 5.4. Players with number of white pixels present

5.3 Data Structure

From section 5.2 we were able to calculate the team of the player. The script used to perform player identification task, returned the initial of the team the player belongs to i.e. **"w"** for white (Tromsø IL), **"b"** for black (Tottenham Hotspurs) and **"r"** for referee. These initials were appended on to the filtered objects (players) identified in section 5.1. The resulting array contained the bounding box coordinates and the initials of the team corresponding to those bounding boxes, as seen below

```
x_min, y_min, x_max, y_max, team
 [[2299, 565, 2345, 645], 'w'],
 [[2023, 637, 2077, 725], 'w'],
 [[1685, 902, 1766, 1008], 'b'],
 [[1711, 614, 1751, 680], 'w'],
 [[799, 841, 843, 908], 'b'],
 [[2575, 651, 2631, 734], 'r']
 .........
```

CHAPTER 6

Visualization

From chapter 5 we were able to create an array with the pixel coordinates, of the players bounding boxes and the initials of the team they belong to. The resulting array from section 5.3 contained the coordinates based on the frame size from video stream. These coordinates needed to be converted with a scale, to be projected on the smaller screen, the method for this is explained in section 6.1.

Since the whole system is running in real time and we receive an array of detections every next frame, a cloud integration was made between the computer executing scripts and the mobile phone, where detections are being displayed. The method devised to make the display on mobile phone in real-time is explained in section 6.2.

6.1 Pixel Scaling

The yielded coordinates from the section 5.3 are in relation to the frame size of video (i.e. 2000×4450). Since the aim of this project is to visualize the positional data of players on a smaller screen, a scale was needed to translate the coordinates from video stream to a smaller image, shown in figure 6.1. This projection can be preformed by using the equation (6.1), where (x, y) are the coordinates of the small screen and (x', y') are the coordinates detected on a frame of video stream.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{6.1}$$

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

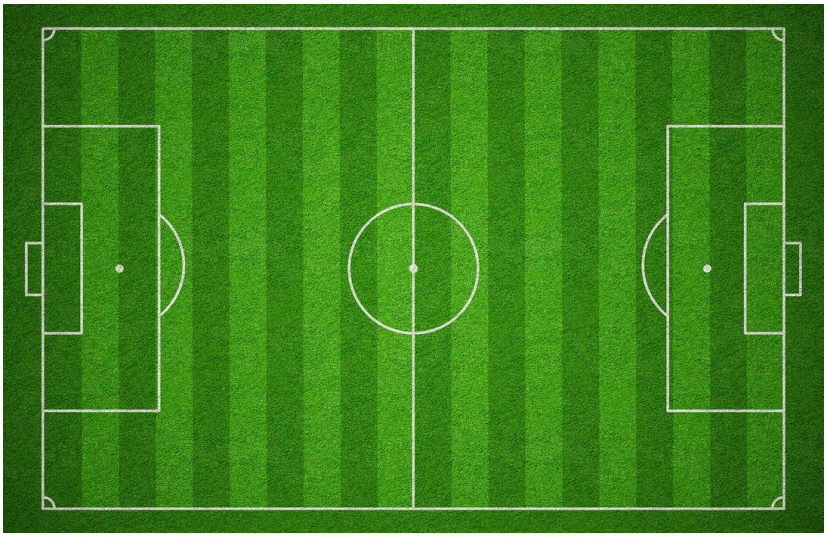


FIGURE 6.1. Image used on small screens with aspect ratio (19:9)

The transformation matrix H used in equation (6.1), is calculated by using the points to create polygon around the football field (shown in figure 5.1) and choosing similar points on smaller animated football field. A pictorial representation of the process is shown in figure 6.2. Having no similar features between the two, the task is performed manually using the script described in Appendix A section A3.1 with the source file changed to small image. The points of the polygon and the one collected from the small image should be in the same order. The two vectors, one representing the polygon and the other representing the points on the small image are used to calculate the H matrix. This can be done by creating a 2×9 matrix, for each corresponding point between the frame and the small image as shown in equation (6.2). Stacking the values for number of points in context, which in our case results in the

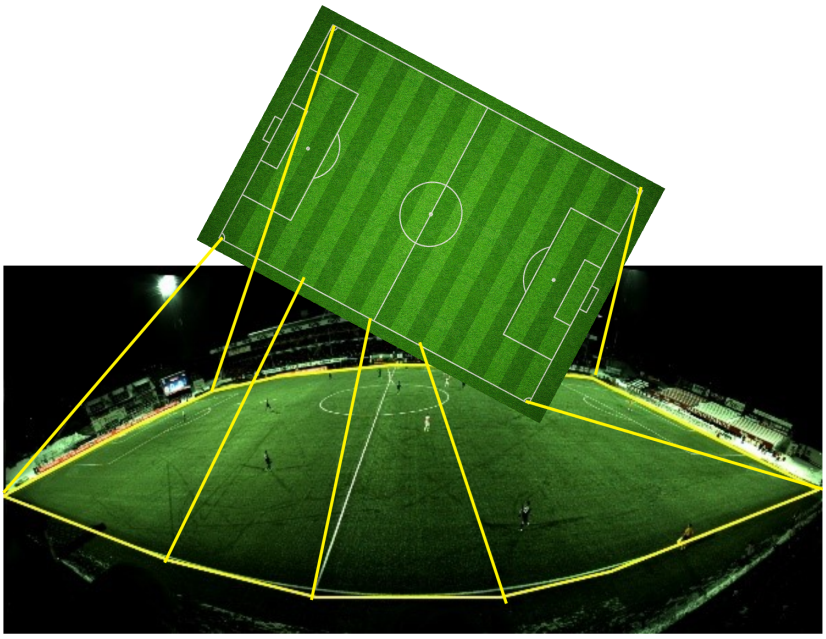


FIGURE 6.2. Choosing similar points on small screen image

matrix $P = (18 \times 9)$ we can compute the required homography matrix, from equation (6.3), from [26].

$$p_i = \begin{bmatrix} -x_i & -y_i & -1 & 0 & 0 & 0 & x_i x'_i & y_i x'_i & x'_i \\ 0 & 0 & 0 & -x_i & -y_i & -1 & x_i y'_i & y_i y'_i & y'_i \end{bmatrix} \tag{6.2}$$

$$PH = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1 x'_1 & y_1 x'_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1 y'_1 & y_1 y'_1 & y'_1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2 x'_2 & y_2 x'_2 & x'_2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2 y'_2 & y_2 y'_2 & y'_2 \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3 x'_3 & y_3 x'_3 & x'_3 \\ & & & & \dots & & & & \\ & & & & \dots & & & & \end{bmatrix} \begin{bmatrix} h1 \\ h2 \\ h3 \\ h4 \\ h5 \\ h6 \\ h7 \\ h8 \\ h9 \end{bmatrix} = 0 \tag{6.3}$$

In equation (6.1) a single coordinate is used, and currently the data structure from section 5.3 provides with 2 coordinates (i.e. top-left and bottom-right). Since in the section 5.1, we are using the feet position of the player for valid detection, the same coordinates are computed using equation (5.1) and used as the coordinates describing the position of the player. This was done to keep the representation of the player on small screen with high relevance.

The feet coordinates are multiplied with the H matrix to compute the position on the smaller screen. The resulting points are stored in a vector, which are then sent to the cloud so the mobile phone which is visualizing the data can fetch it. The computation of the pixel values for the smaller screen is shown in Appendix A section A3.5.

6.2 Cloud Integration

From previous section 6.1, we were able to create a vector containing the coordinates in terms of pixels, describing the player position. The yielded vector is changing with every frame, making the detections real time. To display these positions a database was created which would store the positions of the player and update it self as the new positions come in. The database should also send out a trigger response whenever it is updated so the mobile phone which is being used to display the positions would know if the values have been updated.

To preform the necessary, a cloud solution provided by Google called Firebase [27] was used. An integration was performed between the computer running the script and the cloud storage. This integration would update the database with every new frame and the detections predicted on those frames. A real time database integration was incorporated at the end of the pipeline, which takes the vector yielded from section 6.1 and pushes it on to the cloud. Since the vector is in numpy array format and firebase only accepts JavaScript Object Notation (JSON) format, the vector was first converted before being pushed. This was done by iterating over the array of detected positions and storing them in a python dictionary. The resulting dictionary contained team initials as the **key** and the relevant detections as the **values**, as shown below

'w': [[547, 137], [583, 177], [579, 188], [562, 212], [575, 226], [553, 231], [486, 289]],
'b': [[549, 173], [591, 207], [510, 249], [533, 244], [466, 239], [559, 287]]

The yielded dictionary was passed to the database, whose schema is pre-defined. Each iteration on the video frame results in a new dictionary which in turn updates the values in the cloud database. The figure **??** shows the values in the firebase database and the figure **??** shows the values being updated in real time, with highlighted boxes. As the values of the firebase real-time database updates, it generates a trigger to let all the connected devices know about the change. The generated trigger is used to make the mobile devices, where data is being viewed, to update its display and show the updated positions of the player. The integration of the mobile with the database is explained in section 6.3. The snippet code of this section is provided in Appendix A section A3.6

6.3 Mobile Application

To display the detections done in previous sections, a mobile application was developed. The development was carried out using React-native 16.9.0 developed by Facebook [28]. The purpose of this application was to visualize the detection that has been updated in the firebase database. Since the database is being updated in the real time, the application would be relying on the trigger from the firebase to update the detections on the screen.

Based on the internet connection, application is connected to the database in the cloud. The application keeps the listening port open for the firebase database, which allows it to act in real time with the updates detected in the database. This application takes the coordinates being stored in the database and use this data to represent players. The players in the application are represented as circles with two different colors (white and black) for each team as shown in figure 6.5.

The development of this native application was carried out on mobile phone Google pixel 2 and has been tested only on this device. The pixel2 has a 130mm screen size with the

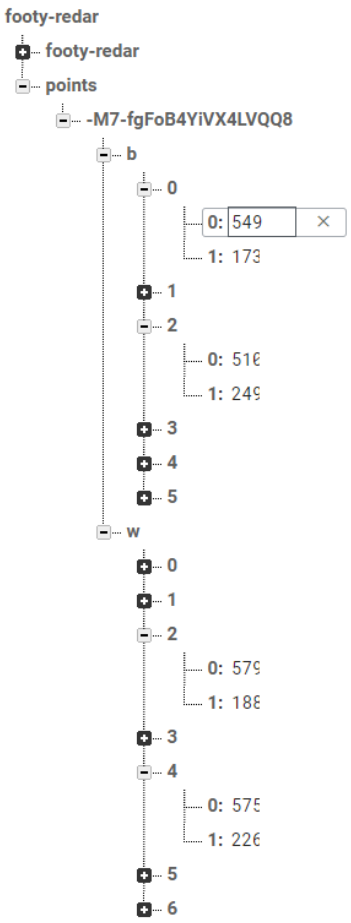


FIGURE 6.3. Values of pixel coordinates stored in firebase

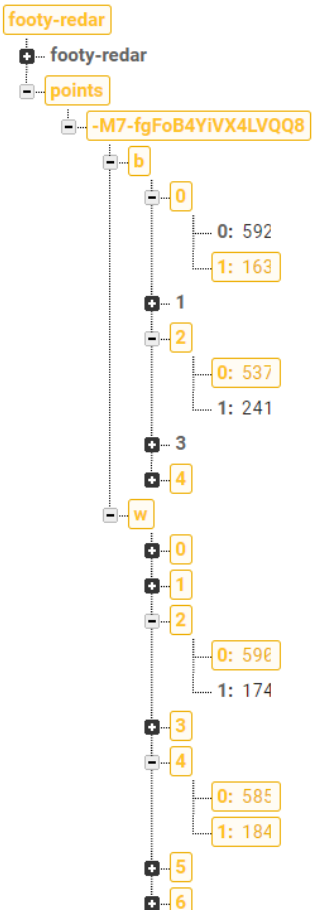


FIGURE 6.4. Highlighted boxes in firebase being updated in real time

resolution of 1080 (1920×1080), any devices having the screen aspects mentioned will yield the similar result.

Every time the data gets updated in the database, a snapshot of the data is taken. This snapshot is validated for having the same keys, to double check the data continuity. The data is stored in the states of the app (i.e white and black) with their corresponding keys. This makes the Virtual Document Object Model (DOM) update it self and the application knows that the new values have been collected. Using this technique, new circles are created on the image (background of football field), which makes the circles disappear and reappear on the next position. For the complete code of the app please refer to git repo [29]

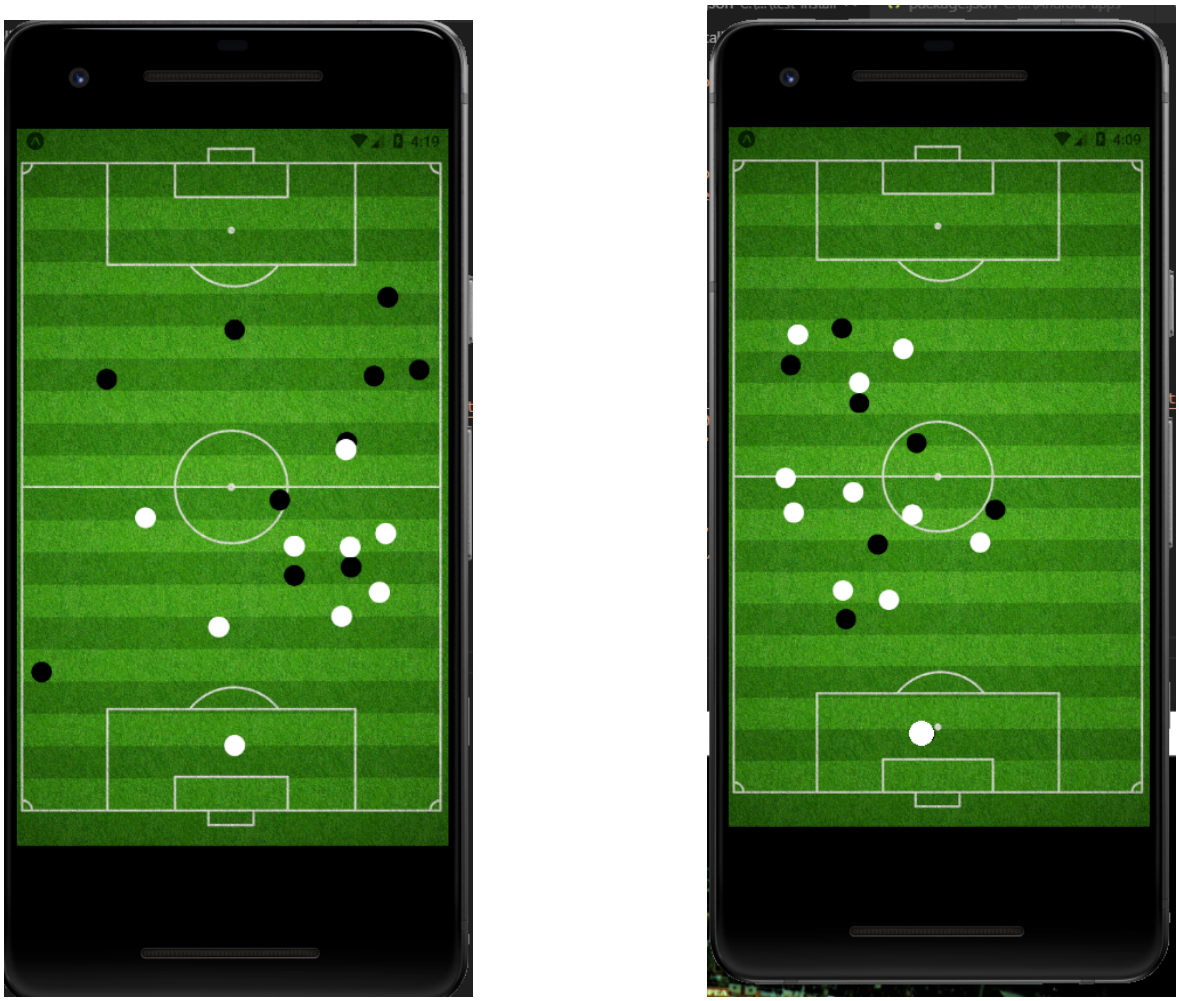


FIGURE 6.5. Screen shot of players being shown on mobile phone (Google Pixel 2 (emulator)

CHAPTER 7

Results

The implemented models are quantified on the bases of their speed through put and the precision of detecting the players accurately. All the models discussed in this project are tested on Intel(R) Core(TM) i7-8550U @ 1.80GHz. This CPU was used for both Background subtraction (section 4.1) as well as Deep Learning models (section 4.2). However, the training for deep learning models on custom data set were performed on Tesla P100-PCIE-12GB GPU.

7.1 Precision $\times$ Recall curve

A metric was required to evaluate the performance of the models, used in this project. For this purpose, a metric system used by PASCAL Visual Object Classes (VOC) [30] was used. This metric system relies on calculating Average Precision using Precision x Recall curve. The Precision and the Recall of a model can be calculated using the equations (7.1) and (7.2).

$$Precision = \frac{TP}{TP + FP} \tag{7.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{7.2}$$

Since the object detection yields the detections as bounding box, which are considered as our predicted results, the measurement of the True Positive (TP), FP and False Negative (FN) is

not straight forward. This was done using IOU, which takes into account the bounding boxes predicted by the model and calculates the score of detection using the equation (**??**). This method calculates the area which is overlapped by the predicted bounding box B_p with the ground truth bounding box B_{gt} for a particular object detected. The ground truth coordinates of the object are collected manually by annotating the image.

$$IOU = \frac{area(B_p \cap B_{gt}}{area(B_p \cup B_{gt})} \tag{7.3}$$

Same ground truth files used to train and test the model, containing the coordinates of the bounding box and file name of the image, were used. Since there was no way to maintain the order between the ground truth and the predicted bounding boxes, Euclidean distance [31] between the centroid of predicted and ground truth bounding boxes was computed. The minimum distance between the two boxes was considered as the detection of the same object in the frame. Calculating IOU of the corresponding detection we can set a precedent for the following:

- **True Positive**: Correct detection, where IOU $\geq \theta$
- **False Positive**: Invalid Detection, where IOU $\leq \theta$
- **True Negative**: Can not be computed, as there can be many bounding boxes where there is not any object.
- **False Negative**: No bounding box overlapping the ground truth

Using the equations above we can calculate the Precision and Recall of images whose ground truth is available. Furthermore, calculating the Precision and Recall of all the images we can plot the graph, where precision is represented on y-axis and recall on x-axis as shown in figure [7.1].

From the Precision x Recall curve we can calculate the Average Precision by estimating the AUC. This can be computed following two methods **11 point interpolation** or **All data points interpolation**. As PASCAL VOC challenge uses all data points method, the same

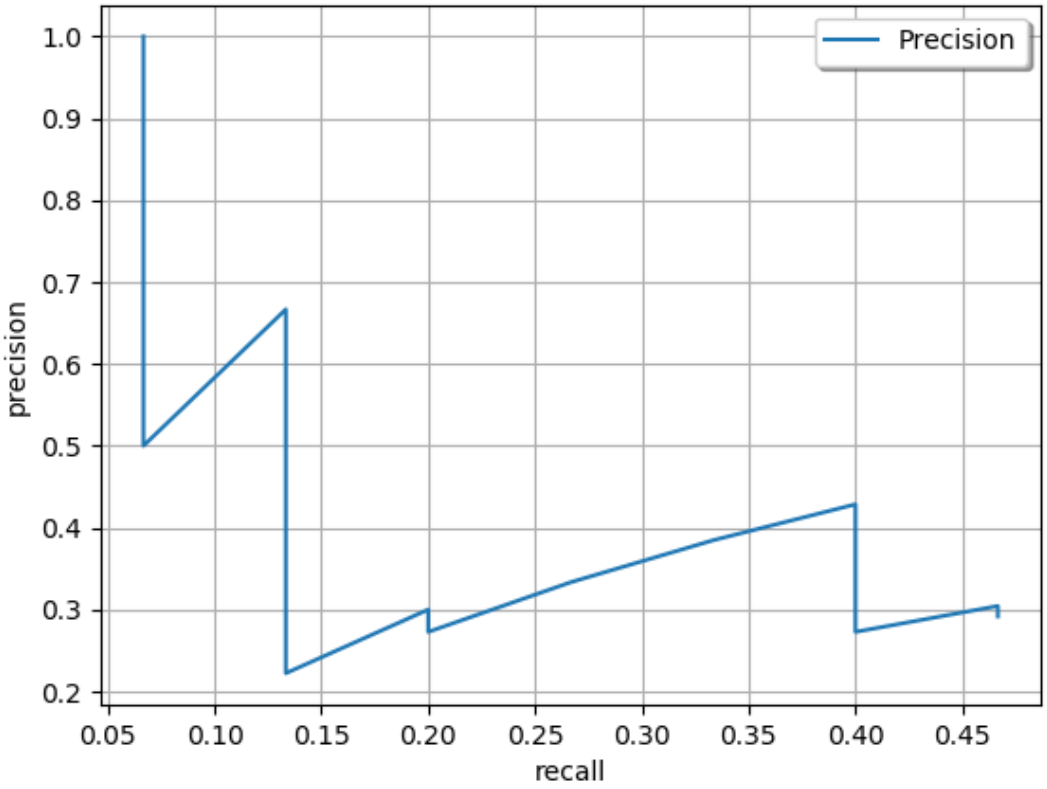


FIGURE 7.1. Precision Recall Curve

method was used in this project. The idea is to interpolate all the n points in the curve, using the equation (7.4), such that r takes the maximum precision whose recall value is greater than or equal to $r + 1$ [32]. Here the $\rho(\hat{r})$ is the precision measured at recall $\hat{r}$.

$$\sum_{n=0}(r_{n+1} - r_n)\rho_{interp}(r_{n+1}) \tag{7.4}$$

$$\rho_{interp}(r_{n+1}) = \max_{\hat{r}.\hat{r} \leq r_{n+1}}$$

Plotting the values calculated from the equation (7.4) on the plot shown in figure 7.1 , we can have four rectangles as shown in figure 7.2. Summing the area of rectangles (**A1, A2, A3, A4**) we can calculate the AUC, which will be the Average Precision.

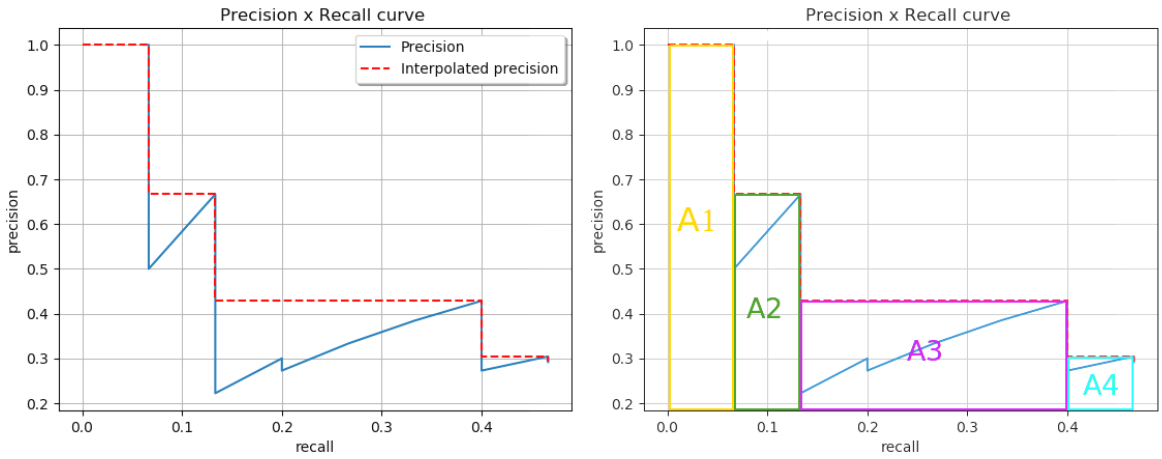


FIGURE 7.2. **Left**: Interpolated data points on Precision x Recall curve. **Right**: Rectangles produced from Interpolated data to calculate AUC [1]

7.1.1 Precision x Recall curve of models

To calculate the Precision x Recall curve of the models used in this project the code developed (by Rafael Padilla) [1] was used. Each model was run on 100 images, whose ground truth values were known, these ground truth values were stored in .txt format, where each row corresponds to the class and its bounding box coordinates. The predicted bounding box coordinates and the confidence yielded from each image applying the model, was also stored in a .txt format, with the file name same as image name. 100 txt files were created, each corresponding to the image, predictions were made on. Example of these txt files are shown below:

```
Ground Truth .txt                Predictions .txt
player 3274 849 3330 940         player 0.65330 3171 1016 3214 1086
player 3515 749 3550 833         player 0.57810 2988 800 3034 905
player 2654 822 2715 942         player 0.54516 3894 1366 3965 1487
```

Using PASCAL VOC evaluation metric on these txt files, Precision x Recall Curves were generated to see the performance of the models. These are depicted in the plots presented in figure 7.3 along with their model information. Since the GMM and Deep Simple Real time

Tracking (SORT) can not work on static images, the plot for both methods are not included in the figures below

Viewing the results shown in figure [7.3], it can be seen that the custom trained model reached a much higher accuracy than the rest of the models.

7.2 Frames Per Second (FPS)

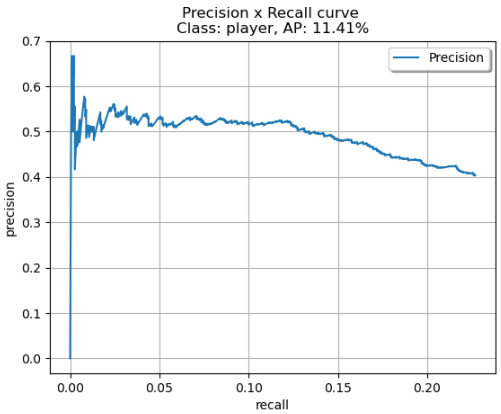
As this project aims to translate the predicted detections on smaller screens in real time, the FPS of the model needs to yield higher through put for better performance. Since all the models were tested on CPU for bench marking a very low through put was observed when using deep learning models. Background subtraction method on the other hand, produced a higher throughput rate comparatively. This is shown in the table 7.1. Figures shown in the table are collected by loading a 2 minute 30 second clip of $\sim$ 442 MB size.

Frame Per Second Throughput	
Model	**FPS**
Image-Subtraction	$\sim$ 5.622
GMM BS	$\sim$ 1.130
YOLO pre-trained YOLO weights	$\sim$ 0.892
Faster R-CNN pre-trained COCO weights	$\sim$ 0.262
SSD pre-trained COCO weights	$\sim$ 1.362
Faster R-CNN custom trained	$\sim$ 0.104
Deep SORT	$\sim$ 0.2824

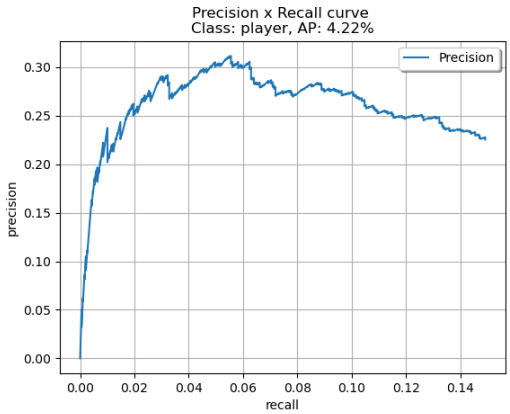
TABLE 7.1. The models were run between 30 seconds to 60 seconds, to calculate FPS

7.3 Visual Outputs

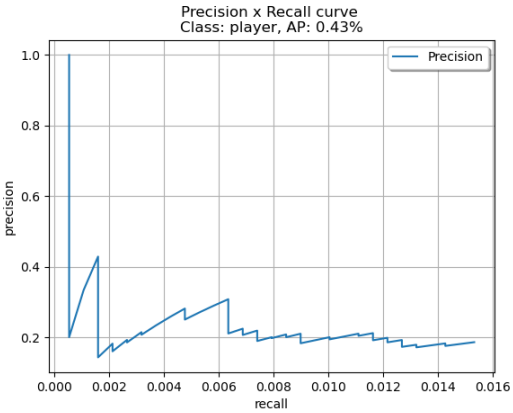
For the purpose of visualization, outputs were generated using different models. Since GMM background Subtraction depends on history of frames to calculate foreground, video stream was used to generate the output for it. For the rest of the models, single image was used to generate results. The outputs contain the frame on which model is being applied on, at the top and a small screen representation at the bottom, with players shown as colored circle (i.e.



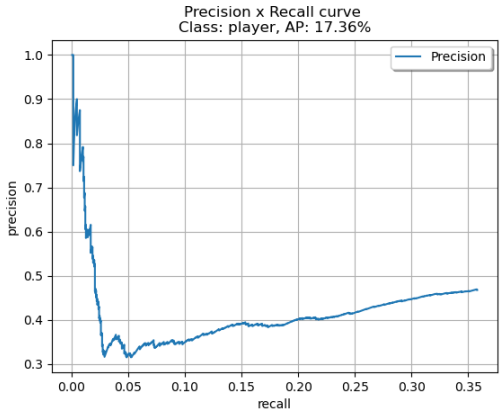
(a) YOLO on pre-trained YOLO weigths



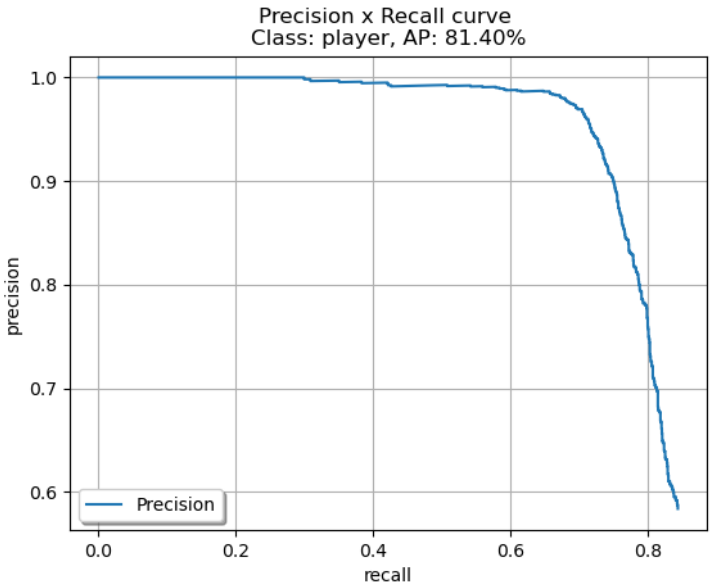
(b) Faster R-CNN on pre-trained COCO weights



(c) SSD on pre-trained COCO weights



(d) Background subtraction, Absolute difference



(e) Custom trained model on Faster R-CNN

FIGURE 7.3. Precision x Recall Curve with Average precision of models used

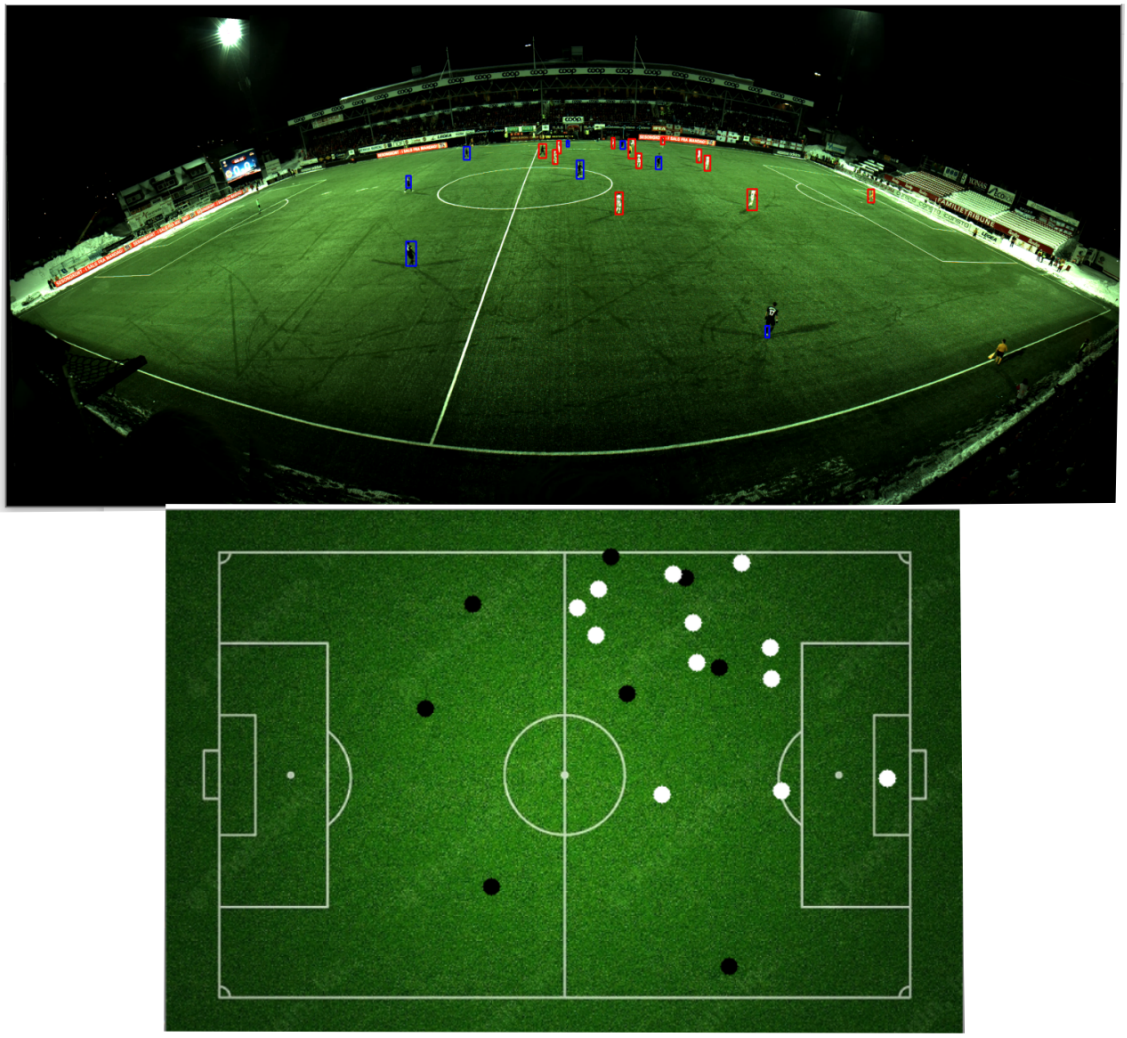


FIGURE 7.4. Result using GMM model

white for team with white jersey and black for team with black jersey).

The result shows that the goal keeper of the black team was not detected. Also at the background where players are highly occluded the model was unable to detect the players with high accuracy. The model also often makes mistakes while recognizing the team of the player. This happens due to the fact that the players are occluded and the bounding box contains more white pixels than the black or vice versa. Other than these limitations the model performed good accuracy in detecting the players on the pitch and showing them on the smaller screen.

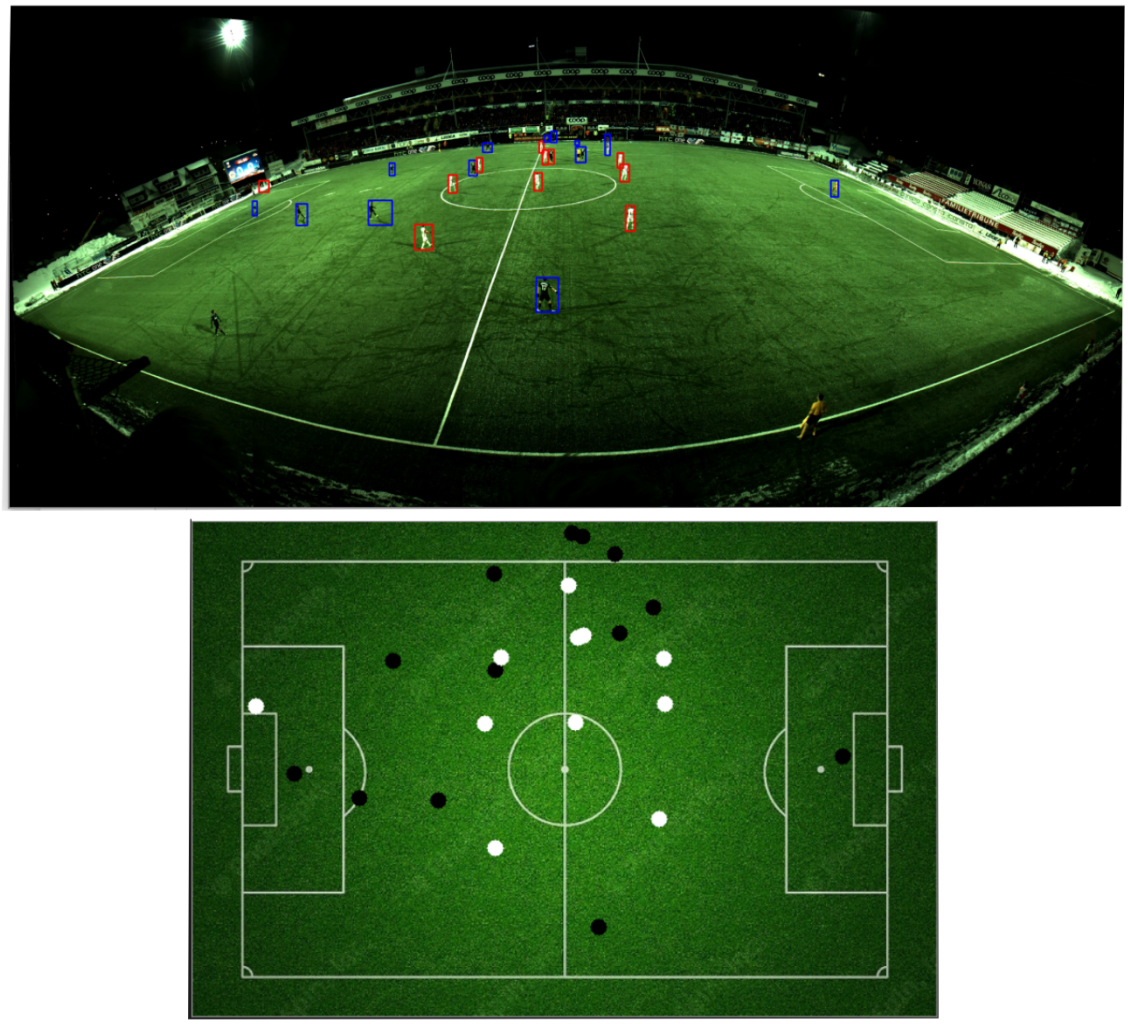


FIGURE 7.5. Result using Image subtraction model

As it can be viewed that there were several False detections generated by the model. This was mainly because, a lot of noise was being produced by the model after generating the foreground. Dedicated checks were performed to eliminate false detections, however, all the false detections could not be removed completely.

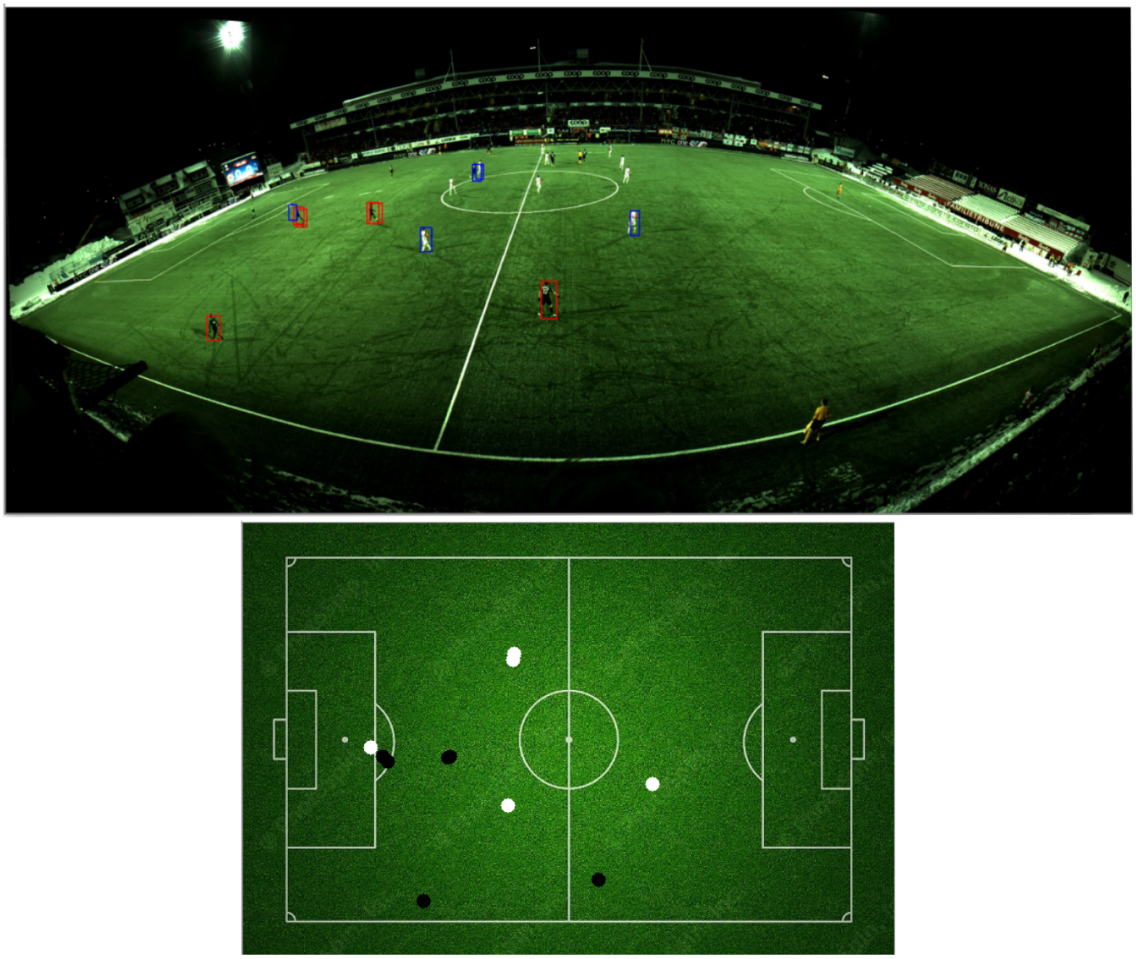


FIGURE 7.6. Result using Faster R-CNN model with pre-trained COCO weights

The Faster R-CNN model trained on the COCO data set, was unable to yield high accuracy at detecting the players. It can be viewed that most of the players on the pitch are not being detected. Since the COCO data set are the images of various objects with out any distortion in the camera angle, the trained weights did not perform very good.

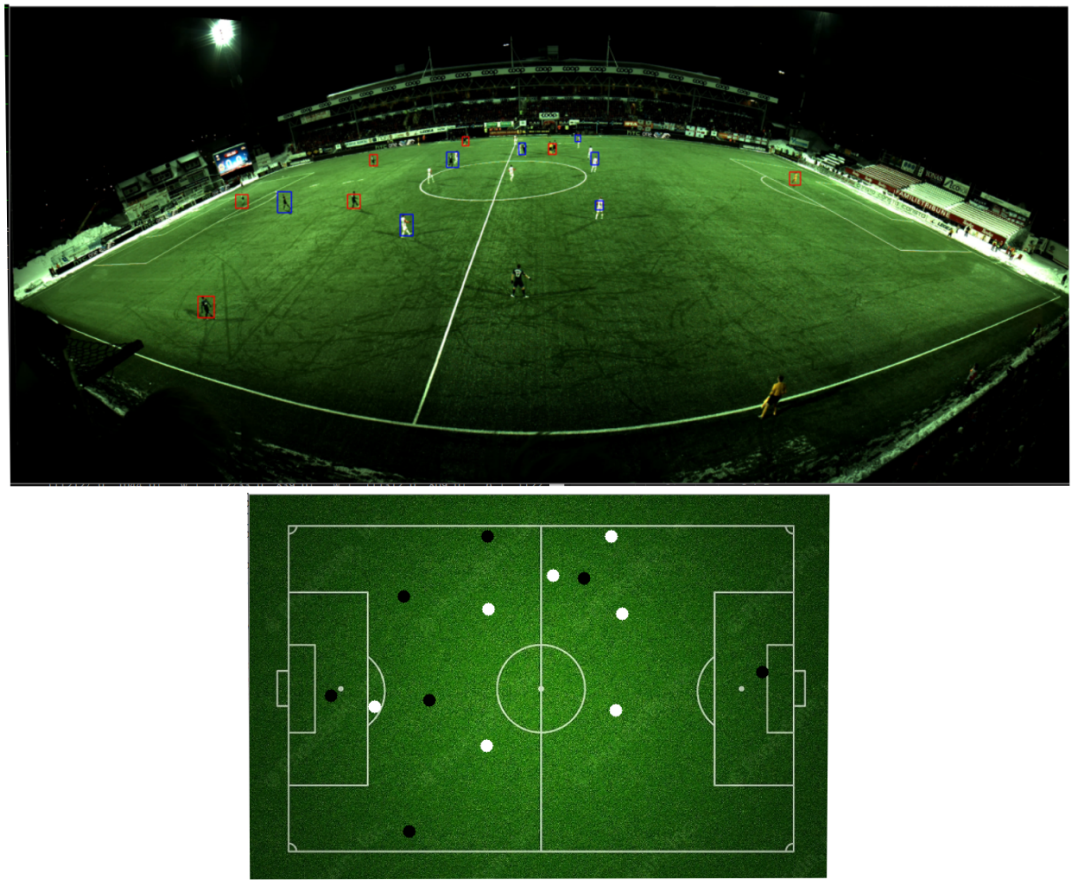


FIGURE 7.7. Result using YOLO model with pre-trained COCO weights

YOLO model trained on the COCO data set, was able to yield better accuracy than Faster R-CNN. YOLO's performance in general is better than rest of the models, due to its architecture. It can be seen that both goal keeper are marked as belonging to the same team. This is due to the fact that script created to detect the strength of the color considers the cropped images inside the bounding box, these cropped images contains the background as well which contains white lines on the football ground, hence adding to the strength of color, while not being part of player jersey.

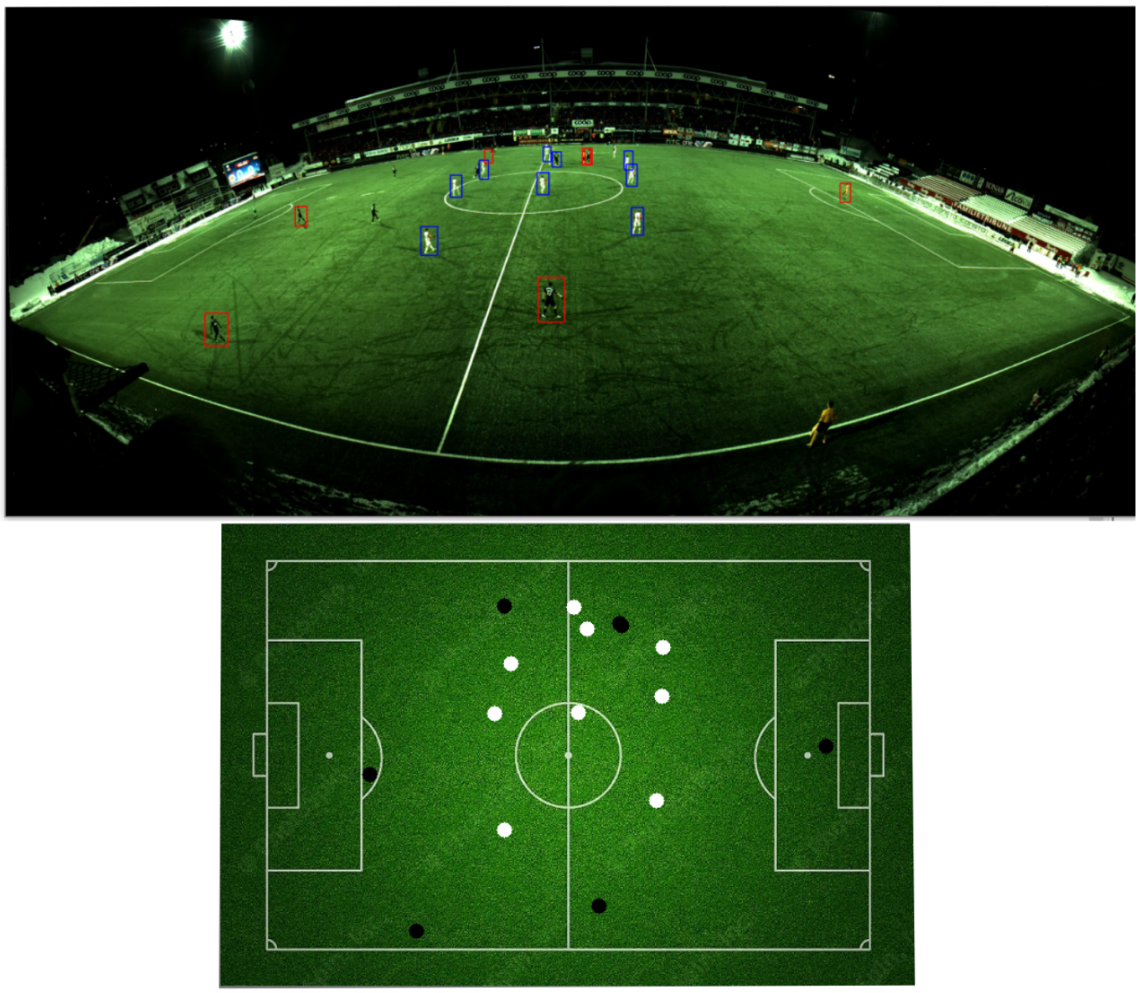


FIGURE 7.8. Result using Faster R-CNN model with custom trained weights

The custom trained model using Faster R-CNN was able to yield higher performance than rest of the deep learning models. It can be seen, the model performed a good job in detecting the white players, but could not detect player from the black team properly. It was anticipated that since the model has an Accuracy of 81%, it was not able to detect all the players in all the frames.

CHAPTER 8

Conclusion

The aim of this project was to retrieve the position of players on the football field, from a video stream and illustrate it on a smaller screen. Several models were created and tested against each other to localize the position of the players. Each model was evaluated based on the accuracy yielded and the throughput rate. These positions were then mapped out on to the smaller screen as an aerial view. Position of each player was represented on a 2D image using a circle, from each frame of a video. To make the system real time, a cloud integration along with a native application was developed to show the positions on to a mobile phone.

Extracting the positions of the players using background subtraction methods or the deep learning models, were not able to produce high accuracy at detecting the players. The GMM background subtraction method worked best when the players were not occluded, as it was able to detect the players present at the back (away from the camera). A custom trained deep learning model was able to perform much better when players were occluded, compared to Background subtraction model. Deep learning models were also computationally very expensive, which made their run time very slow as compared to Background subtraction methods. The process time per frame of each model is shown in table 7.1, which are calculated on a CPU and can be increased with the introduction of a GPU in the system.

It was hard to quantify the representation of the players on the smaller screen, due to the presence of distortion in the camera angle of video stream. It can not be stated with certainty, whether the players being shown on the smaller 2D image (425×640) were represented accurately regarding their position in the video frame. The players were represented based on the position of their feet, using a circle, having a radius of 3 pixels.

8.1 Future outlook

There is room for further improvements in this project as well as the positional data extracted can be used to create predictive machine learning models. These models will play a vital role in improving the capabilities of players and also creating new strategies for teams.

8.1.1 Model Improvement

The data set created to train the deep learning model for this project, will be available and can be used to train new models. At the time of writing this project, new models like YOLOv4, YOLOv5 [33] have been introduced. These models are claimed to be better and faster than rest of the CNN models. If these models are implemented on the video stream, it is anticipated that better accuracy can be achieved.

Object tracking algorithms like Kalaman filter, centroid tracking can be used to track the movement of an individual player. These tracking techniques do not work best when objects under consideration have similar colors, which in this case was true, considering the jerseys of the players. However, newer techniques can be developed for object tracking, which do not rely on the color of the object but its position and momentum. Introducing the tracking algorithm, ID can be assigned to each detection. This would help in calculating the stats of the players on individual level, without the use of sensor technology.

8.1.2 E2E models

End to End deep learning models like MuZero [34] can be used on the positional data collected. These models can be used to devise new strategies for the game of football. As these models can simulate different scenarios during training and can come up with better strategies than a human, doing so the whole paradigm of sports analysis can be shifted and an Artificial Intelligence (AI) coach can be developed. This AI coach if trained up to some extent, can provide with real-time analysis and strategies.

8.1.3 Smart Watches

Being low in size, in terms of storage, the data can be viewed on to much smaller screens like smart watches. Since some smart watches has a rectangular dial, e.g. Apple watch, positional data can be displayed on it using small dots This would allow more mobility, hence providing a better and faster way to analyze the data.

Bibliography

[1] Sergio Lima Netto Rafael Padilla and Eduardo A. B. da Silva. Survey on performance metrics for object-detection algorithms. 2020.

[2] Simen Sægrov, Alexander Eichhorn, Jørgen Emerslund, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Bagadus an integrated system for soccer analysis. In *2012 Sixth International Conference on Distributed Smart Cameras (ICDSC)*, pages 1–2. IEEE.

[3] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, Oystein Landsverk, Carsten Griwodz, Pål Halvorsen, Magnus Stenhaug, and Dag Johansen. Bagadus: An integrated real-time system for soccer analytics. *Transactions on Multimedia Computing, Communications and Applications*, 10.

[4] S.A. Pettersen, D. Johansen, H. Johansen, V. Berg-Johansen, V.R. Gaddam, A. Mortensen, R. Langseth, C. Griwodz, H.K. Stensland, and P. Halvorsen. Soccer video and player position dataset". In *Proceedings of the International Conference on Multimedia Systems (MMSys*, page 18–23, Singapore.

[5] FFmpeg Developers. Ffmpeg tool [sofware], 2016.

[6] Svein Arne Pettersen, Dag Johansen, Håvard Johansen, Vegard Berg-Johansen, Vamsidhar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, Håkon Kvale Stensland, and Pål Halvorsen. Soccer video and player position dataset. In *Proceedings of the 5th ACM Multimedia Systems Conference*, MMSys '14, page 18–23, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327053. doi: 10.1145/2557642.2563677. URL https://doi.org/10.1145/2557642.2563677.

[7] Tzutalin. *LabelImg. Git code*. URL https://github.com/tzutalin/labelImg.

[8] Ahmed Elgammal, David Harwood, and Larry Davis. Non-parametric model for background subtraction. In David Vernon, editor, *Computer Vision — ECCV 2000*, pages 751–767, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45053-5.

[9] Helly M Desai and Vaibhav Gandhi. A survey: Background subtraction techniques. *International Journal of Scientific & Engineering Research*, 5(12):1365, 2014.

[10] Ravi Srisha and Am Khan. Morphological operations for image processing : Understanding and its applications. 12 2013.

[11] George Green. An essay on the application of mathematical analysis to the theories of electricity and magnetism. which appears on pages 10–12 of his Essay.

[12] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] Goran Nakerst, John Brennan, and Masudul Haque. Gradient descent with momentum — to accelerate or to super-accelerate? *arXiv e-prints*, art. arXiv:2001.06472, January 2020.

[14] Karen Simonyan Andrew Zisserman. Very deep convolutional networks for large-scale image recognition.

[15] J. Heikkila and O. Silven. A real-time system for monitoring of cyclists and pedestrians. In *Proceedings Second IEEE Workshop on Visual Surveillance (VS'99) (Cat. No.98-89223)*, pages 74–81, 1999.

[16] Schwarz Gideon et al. Estimating the dimension of a model. *The annals of statistics*, 6 (2):461–464, 1978.

[17] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. *CVGIP*, 30 1:32–46.

[18] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[19] Jan Hosang, Rodrigo Benenson, and Bernt Schiele. Learning non-maximum suppression. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4507–4515, 2017.

[20] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[21] Christian Szegedy, Scott Reed, Dumitru Erhan, Dragomir Anguelov, and Sergey Ioffe. Scalable, High-Quality Object Detection. *arXiv e-prints*, art. arXiv:1412.1441, December 2014.

[22] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[23] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[24] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[25] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[26] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press.

[27] Laurence Moroney. The firebase realtime database. In *The Definitive Guide to Firebase*, pages 51–71. Springer, 2017.

[28] Bonnie Eisenman. *Learning react native: Building native mobile apps with JavaScript*. " O'Reilly Media, Inc.", 2015.

[29] Hammad Ali. Footy-redar app code. 2020. URL https://github.com/hmdall/mob-viewer. Available at:.

[30] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, January 2015.

[31] Ivan Dokmanic, Reza Parhizkar, Juri Ranieri, and Martin Vetterli. Euclidean Distance Matrices: Essential theory, algorithms, and applications. *IEEE Signal Processing Magazine*, 32(6):12–30, November 2015.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv e-prints*, art. arXiv:2004.10934, April 2020.

[34] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *arXiv e-prints*, art. arXiv:1911.08265, November 2019.

Appendix A

A1 Background Subtraction Implementation code

A1.1 Extracting Background from Video

```
def extract_background(videoFile):
        """Receives a video filename(with extension) and
        saves the background image in specified folder"""

    video = cv2.VideoCapture(videoFile)
    c = 0
    while video.isOpened():
        _,img = video.read()

        average_ = img
        fc = float(c)
        average_ = (fc*avg_img + img)/(fc+1)
        c += 1

    cv2.imwrite('path_of_file', average_)
```

A1.2 Binary Masking

```
bg = cv2.imread('img/background.jpg')
bg_img = cv2.cvtColor(bg, cv2.COLOR_BGR2GRAY)

def track_player(video_file):
    cap = cv2.VideoCapture(videoFile)

    while cap.IsOpened():
        ret, frame = cap.read()
        if not ret:
            break
        gray_img = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        bg_delta = cv2.absdiff(bg_img, gray_img)

        threshold = cv2.threshold(bg_delta, 30, 255,
                cv2.THRESH_BINARY)[1]

        threshold = cv2.erode(threshold, (13,13), iterations= 1)
        threshold = cv2.dilate(threshold, (10, 10), iterations=1)

        threshold = cv2.erode(threshold, (11,11), iterations= 1)
        threshold = cv2.dilate(threshold, (8, 8), iterations= 1)

        contours, _ = cv2.findContours(threshold.copy(),
                    cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

A1.3 Finding n-components for GMM

```
import cv2
from sklearn.mixture import GaussianMixture as GMM
```

```
frame = cv2.imread('img/0.png')
img = frame.reshape((-1,3))

n_components = np.arange(1, 10)
gmm_model = [GMM(n, covariance_type='tied')
             .fit(img) for n in n_components]

plt.plot(n_components, [m.bic(img) for m in gmm_model],
                 label="BIC")
plt.xlabel('n_components')
```

A1.4 Gaussian Mixture Model

```
import cv2
back_sub = cv2.bgsegm.createBackgroundSubtractorMOG(history = 5, nmixtu

vid_cap = cv2.VideoCapture(video_file_path)

while vid_cap.isOpened():
    ret, img = vid_cap.read()
    if not ret:
        break
    fgMask = back_sub.apply(img)
    fgMask = cv2.erode(fgMask, (13,13), iterations = 1)
    fgMask = cv2.dilate(fgMask, (2,2), iterations = 1)
    fgMask = cv2.erode(fgMask, (13,13), iterations = 1)
    fgMask = cv2.dilate(fgMask, (2,2), iterations = 2)
```

```
    contours, _ = cv2.findContours(fgMask.copy(),
    cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

A1.5 Finding Contours

```
# Finding Contours on each frame inside while loop
contours, _ = cv2.findContours(fgMask.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
player_pos = []
# Iterating over contours:
for cn in contours:
    (x, y, w, h) = cv2.boundingRect(cn)
    feet_coord = [float(x + int(w/2.0)), float(y + h)]
    feets = Point(feet_coord[0], feet_coord[1])
    rect_area = cv2.contourArea(cn)

    # Checking if detection is inside the polygon defined
     if not field_polygon_points.contains(feets):
                continue

    # Performing Several checks for player detection
    if not h > w * 1.5:
        continue

    if rect_area < 100:
        continue

    if w > h:
        continue
```

```
    if y > 1000 and 600 < x < 2900 and rect_area < 1000:
        continue

    if 800 < y < 1000 and 600 < x < 2900 and rect_area < 500:
        continue

    if 600 < y < 800 and 600 < x < 2900 and rect_area < 300:
        continue

    if y > 1000 and x > 2900 and rect_area < 200:
        continue

    player.append([x, y, x+w, y+h])
```

A2 Deep Learning Implementations code

A2.1 Faster R-CNN implementation

```
from object_detection.utils import ops as utils_ops
from object_detection.utils import label_map_util
from collections import defaultdict
import tensorflow as tf
import numpy as np
import cv2

def load_model(model_name):
  model_dir = pathlib.Path(model_name)/"saved_model"
  model = tf.saved_model.load(str(model_dir), None)
```

```
  model = model.signatures['serving_default']
  return model

PATH_TO_LABELS = 'FasterRCNN/label_map.pbtxt'
category_index = label_map_util
            .create_category_index_from_labelmap(
                                PATH_TO_LABELS,
                                use_display_name=True)
# Downloaded model
model_name = 'faster_rcnn_resnet50_coco_2018_01_28/'
# Loading model
detection_model = load_model(model_name)

def run_inference_for_single_image(model, image):
  image = np.asarray(image)
  # The input needs to be a tensor,
  # convert it using `tf.convert_to_tensor`.
  input_tensor = tf.convert_to_tensor(image)
  # The model expects a batch of images,
  # so add an axis with `tf.newaxis`.
  input_tensor = input_tensor[tf.newaxis,...]

  # Run inference
  output_dict = model(input_tensor)

  # All outputs are batches tensors.
  # Convert to numpy arrays,
  # and take index [0] to remove the batch dimension.
```

```
  # We're only interested in the first num_detections.
  num_detections = int(output_dict.pop('num_detections'))
  output_dict = {key:value[0, :num_detections].numpy()
                 for key,value in output_dict.items()}
  output_dict['num_detections'] = num_detections

  # detection_classes should be ints.
  output_dict['detection_classes'] =
                output_dict['detection_classes']
                .astype(np.int64)

  # Handle models with masks:
  if 'detection_masks' in output_dict:
    # Reframe the the bbox mask to the image size.
    dt_mask = utils_ops.
            reframe_box_masks_to_image_masks(
              output_dict['detection_masks'],
              output_dict['detection_boxes'],
               image.shape[0], image.shape[1])
    dt_mask = tf.cast(dt_mask > 0.5, tf.uint8)
    output_dict['dt_mask'] = dt_mask.numpy()
#   print(output_dict)
  return output_dict

cap = cv2.VideoCapture(vid_file_path)
_, image = cap.read()
h = image.shape[0]
w = image.shape[1]
```

```
confThreshold = 0.3
nmsThreshold = 0.6
player_pos = []

while True:
    boxes = []
    confidence = []
    ret, frame = cap.read()
    if not ret:
        break
    image = np.array(frame)
    out_duct = run_inference_for_single_image(detection_model, image)
    for i in range(0, len(out_duct['detection_boxes'])):
        ymin = int(out_duct['detection_boxes'][i][0] * h)
        xmin = int(out_duct['detection_boxes'][i][1] * w)
        ymax = int(out_duct['detection_boxes'][i][2] * h)
        xmax = int(out_duct['detection_boxes'][i][3] * w)
        confidence.append(out_duct['detection_scores'][i])
        boxes.append([xmin, ymin, xmax, ymax])

    indices = cv2.dnn.NMSBoxes(boxes,
                               confidences,
                               confThreshold,
                               nmsThreshold)
    for i in indices:
        i = i[0]
        box = boxes[i]
        left = box[0]
```

```
        top = box[1]
        width = box[2]
        height = box[3]
        player_pos.append([ confidence[i],
                            left,
                            top,
                            left + width,
                            top + height])
```

A2.2 YOLO Implementation

```
import cv2 as cv
import numpy as np
import imutils


total_frames = 0
confThreshold = 0.3
nmsThreshold = 0.3
inpWidth = 416
inpHeight = 416
outputFile = 'detected_file.mp4'
classesFile = 'coco.names'
classes = None


def postprocess(frame, outs):
    frameHeight = frame.shape[0]
    frameWidth = frame.shape[1]
```

```
classIds = []
confidences = []
boxes = []
# Scan through all the bounding boxes output
# from the network and keep only the
# ones with high confidence scores.
# Assign the box's class label as
# the class with the highest score.
for out in outs:
    for detection in out:
        scores = detection[5:]
        classId = np.argmax(scores)
        confidence = scores[classId]
        if confidence > confThreshold:
            center_x = int(detection[0] * frameWidth)
            center_y = int(detection[1] * frameHeight)
            width = int(detection[2] * frameWidth)
            height = int(detection[3] * frameHeight)
            left = int(center_x - width / 2)
            top = int(center_y - height / 2)
            classIds.append(classId)
            confidences.append(float(confidence))
            boxes.append([left, top, width, height])

# Perform non maximum suppression to
# eliminate redundant overlapping boxes with
# lower confidences.
```

```
    indices = cv.dnn.NMSBoxes(boxes,
                              confidences,
                              confThreshold,
                              nmsThreshold)
    for i in indices:
        i = i[0]
        box = boxes[i]
        left = box[0]
        top = box[1]
        width = box[2]
        height = box[3]
        predicted.append(classIds[i],
                         confidences[i],
                         left,top,left + width, top + height)
```

A3 Deep SORT implementation

```
# run_inference_for_single_image is used from section A2.1

import dlib

trackers = []
count = 0
while cap.isOpened():
    ret, frame = cap.read()
    image = np.array(frame)

    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
```

```
out_duct = run_inference_for_single_image(detection_model, image)

if len(trackers) < 23 & count % 30 == 0:
    for i in out_duct['detection_boxes']:
        ymin = int(i[0] * h)
        xmin = int(i[1] * w)
        ymax = int(i[2] * h)
        xmax = int(i[3] * w)

        feet_coord = [float((xmin+xmax)/2), float(ymax)]
        feets = Point(feet_coord[0], feet_coord[1])

        if not field_polygon_points.contains(feets):
            continue

        t = dlib.correlation_tracker()
        rect = dlib.rectangle(xmin, ymin, xmax, ymax)
        t.start_track(rgb, rect)

        trackers.append(t)
    fps.update()

else:
    for t in trackers:
        # update the tracker and grab the position of the tracked
        # object
        t.update(rgb)
        pos = t.get_position()
```

```
            # unpack the position object
            startX = int(pos.left())
            startY = int(pos.top())
            endX = int(pos.right())
            endY = int(pos.bottom())
```

A3.1 Field Polygon

```
# Select the polygon points on the field.
# Takes a video path as input and resizes it to show on the screen
# Saves the points in .txt format to be used later
def select_points(vid_filepath):

    points_file = '../txt/field_polygon.txt'
    posList = []
    cap = cv2.VideoCapture(vid_filepath)
    ret, frame = cap.read()
    factor = 2
    def mouse_points(event, x ,y, flags, param):
        global posList
        if event == cv2.EVENT_LBUTTONDOWN:
            print('cordinate x: {}, y:{} '.format(x, y))
            posList.append([x, y])

    while True:
        frame = cv2.resize(frame, (frame.shape[0]//factor,
                                   frame.shape[1]//factor))
        cv2.imshow('image', frame)
```

```
        cv2.setMouseCallback('image', mouse_points)
        if cv2.waitKey(0) & 0xFF == 27:
            break

    print('final points collected {} '.format(posList))
    np.savetxt(posList, points_file)
```

A3.2 Point Checker

```
from shapely.geometry import Point
from shapely.geometry.polygon import Polygon

points = np.loadtxt(points_file)
points = points.astype(int)
field_polygon_points = Polygon(points)

for cn in contours:
    feet_coord = [float(x + int(w/2.0)), float(y + h)]
    feets = Point(feet_coord[0], feet_coord[1])

    if not field_polygon_points.contains(feets):
            continue
```

A3.3 HSV Color picker

```
import cv2
import numpy as np



cap = cv2.VideoCapture(vid_filepath)
```

```
def nothing(x):
    pass
# Creating a window for later use
cv2.namedWindow('result')

# Starting with 100's to prevent error while masking
h,s,v = 100,100,100

# Creating track bar
cv2.createTrackbar('h', 'result',0,179,nothing)
cv2.createTrackbar('s', 'result',0,255,nothing)
cv2.createTrackbar('v', 'result',0,255,nothing)

while(1):

    _, frame = cap.read()

    #converting to HSV
    hsv = cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)

    # get info from track bar and appy to result
    h = cv2.getTrackbarPos('h','result')
    s = cv2.getTrackbarPos('s','result')
    v = cv2.getTrackbarPos('v','result')

    # Normal masking algorithm
    lower_blue = np.array([h,s,v])
```

```
    upper_blue = np.array([180,255,255])

    mask = cv2.inRange(hsv,lower_blue, upper_blue)

    result = cv2.bitwise_and(frame,frame,mask = mask)

    cv2.imshow('result',result)

    k = cv2.waitKey(5) & 0xFF
    if k == 27:
        break

cap.release()

cv2.destroyAllWindows()
```

A3.4 Color Pixel Calculator

```
class DetectionWithColor:
    def __init__(self):
        self.lower_white = np.array([60,0,204])
        self.upper_white = np.array([179,38,255])

        self.lower_black = np.array([0,0,0])
        self.upper_black = np.array([360,100,50])

        self.lower_yellow = np.array([18, 182, 130])
        self.upper_yellow = np.array([67, 255, 255])
    def detectPixelCount(self, player_img, x , y):
```

```
player_hsv = cv2.cvtColor(player_img, cv2.COLOR_BGR2HSV)

mask3 = cv2.inRange(player_hsv,
                self.lower_yellow, self.upper_yellow)

res3 = cv2.bitwise_and(player_img, player_img, mask=mask3)
res3 = cv2.cvtColor(res1,cv2.COLOR_HSV2BGR)
res3 = cv2.cvtColor(res1,cv2.COLOR_BGR2GRAY)
yellowCount = cv2.countNonZero(res1)

if yellowCount > 50:
    return 'r'

mask1 = cv2.inRange(player_hsv, self.lower_white,
                    self.upper_white, mask=mask1)
res1 = cv2.bitwise_and(player_img, player_img, mask=mask1)
res1 = cv2.cvtColor(res1,cv2.COLOR_HSV2BGR)
res1 = cv2.cvtColor(res1,cv2.COLOR_BGR2GRAY)
whiteCount = cv2.countNonZero(res1)

mask2 = cv2.inRange(player_hsv,
                self.lower_black, self.upper_black)
res2 = cv2.bitwise_and(player_img, player_img, mask=mask2)
res2 = cv2.cvtColor(res2,cv2.COLOR_HSV2BGR)
res2 = cv2.cvtColor(res2,cv2.COLOR_BGR2GRAY)
blackCount = cv2.countNonZero(res2)

if whiteCount > blackCount:
```

```
            return whiteCount, "w"
        else:
            return blackCount, 'b'
```

A3.5 Calculating coordinates for small screens

```
# input_pts are the feet coordinates of all the players in a vector
    def homography(self, input_pts):
        pts = np.matrix(np.zeros(shape=(len(input_pts),3)))
        c = 0
        for i in input_pts:
            x,y = i[0][0], i[0][1]
            pts[c,:] = np.array([x,y,1], dtype = "float32")
            c+=1

        player_top_points = list()
        newPoints = np.empty([len(input_pts),3], dtype = "float32")
        c = 0
        for i in pts:
            newPoints = self.hg_matrix*(i.T)
            x = int(newPoints[0]/float(newPoints[2]))
            y = int(newPoints[1]/float(newPoints[2]))

            player_top_points.append([[x, y], input_pts[c][1][0]])
            c +=1
```

A3.6 Cloud Integration

```
# Intialization of variables
 def __init__(self):
```

```
        self.firebase = firebase.FirebaseApplication(
                    'https://footy-redar.firebaseio.com', None)
# data is the array of scaled detections
 def putData(self, data):
        new_dict = {}
        for i in data:
            if str(i[1]) not in new_dict.keys():
                new_dict[i[1]] = []
                new_dict[i[1]].append(i[0])
            else:
                new_dict[i[1]].append(i[0])
        print('dictionary :', new_dict)
        if "b" in new_dict and "w" in new_dict:
            result = self.firebase.put('/points/',
                    '/-M7-fgFoB4YiVX4LVQQ8/', new_dict)
```