

# Deploying Multi Camera Multi Player Detection and Tracking Systems in Top View

Thesis submitted in partial fulfillment  
of the requirements for the degree of

*Master of Science in **Computer Science and Engineering** by Research*

by

Swetanjali Murati Dutta

20171077

swetanjali.dutta@research.iiit.ac.in



INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

HYDERABAD

International Institute of Information Technology, Hyderabad

Hyderabad 500032, India

November 2023

Copyright © Swetanjali Murati Dutta, 2023  
All Rights Reserved

International Institute of Information Technology  
Hyderabad, India

## **CERTIFICATE**

It is certified that the work contained in this thesis, titled “Deploying Multi Camera Multi Player Detection and Tracking Systems in Top View” by Swetanjal Murati Dutta, has been carried out under my supervision and is not submitted elsewhere for a degree.

---

Date

---

Advisor: Prof. Vineet Gandhi

*To my Family*



## **Acknowledgments**

Firstly, I would like to thank my advisor, Prof. Vineet Gandhi, for his constant support, guidance, and help. He provided me with inspiration and motivation to do groundbreaking research. I really found his way of explaining complex topics in a simple and intuitive way really amazing. I consider myself really fortunate to have got an opportunity to work under an advisor like Dr. Vineet, and I wish to continue being involved with him in solving such kind of challenging research problems in the future. I would also like to thank him for giving me the opportunity to work on this project.

A huge shout out to Jeet Vora and Kanishk Jain for all the help in deploying the system in Asia Cup 2022. I must mention that just before our system was about to go live, I faced severe challenges with my health, and at that time, Jeet and Kanishk stepped in and did the heavy lifting to ensure the system was deployed successfully. This thesis would not have been possible without their help in such a critical situation.

I would also like to thank Shyamgopal Karthik for all the guidance and advice. The constant discussions with Shyam really had a profound impact on me and my research work. A lot of my understanding in the field of Computer Vision has been tuned through discussions with him. I really admire his way of doing research and wish to continue working with him in the future. A huge thanks to Sayar Ghosh Roy for giving me an interesting perspective on various research topics and providing me with motivation when I needed it the most.

Finally, I would like to thank my family for their constant support, motivation, and prayers which enabled me to complete the thesis. I must say this journey was full of ups and downs (with multiple health-related challenges), and finally, I am so glad that I was able to successfully complete my thesis with the best wishes of my well-wishers and God's blessings.

## Abstract

Object Detection and Tracking computer vision algorithms have remarkably progressed in the past few years, owing to the rapid progress in the field of deep learning. These algorithms find numerous applications in surveillance, robotics, autonomous driving, sports analytics, and human-computer interaction. They achieve near-perfect performance on the benchmark datasets they have been evaluated on. However, deploying such algorithms in real-world poses a large number of challenges, and they do not work as well as we expect them to work by looking at their performance on benchmark datasets.

In this thesis, we present details of deploying one such Multi Camera Multi-Object Detection and Tracking system to track players in the bird's eye (top) view in a sports event, specifically in the game of cricket. Our system is able to generate the top-view map of the fielders from cameras placed on crowd stands of a stadium, making it extremely cost-effective compared to using spider cameras. We present details on the challenges and hurdles we faced while deploying our systems and the solutions we designed to overcome these challenges. Ultimately, we tailored a neatly engineered end-to-end system that went live in the Asia Cup of 2022.

Deploying such a multi-camera detection and tracking system poses many challenges. The first of them is related to camera placement. We had to devise strategies to place cameras optimally so that all the objects of interest could be captured by one or more of the cameras placed, at the same time ensuring that they do not appear so small that it becomes difficult for a detector to localize them. Constructing the bird's eye view of the detected players required camera calibration. In the real world, we may not find reliable point correspondences to calibrate a camera. Even if we might find point correspondences, sometimes they may be really hard to see to accurately calibrate a camera. Detecting players in a setup like this proved to be really challenging because the objects of interest appeared very small in the camera views. Moreover, since the detections were coming from multiple cameras, algorithms had to be devised to associate them correctly across camera views. Tracking in a setting like this was even harder because we had to rely only on motion cues to track the objects of interest. Appearance features did not add any value because of the fact all the players being tracked wore jerseys of the same color. Each of these challenges became even more, harder to solve because of the real-time requirements of our use case.

Finally, setting up the hardware architecture to receive the real-time live feeds from each camera in a synchronized manner and implementing a fast communication protocol for transmitting data between the various system components required careful design choices to be made, all of which have been presented in detail in this thesis.

# Contents

Chapter	Page
1 Introduction . . . . .	1
1.1 Problem Statement . . . . .	1
1.2 Trivial Solution . . . . .	2
1.3 Challenges . . . . .	3
2 Camera Calibration and Homography . . . . .	6
2.1 Camera Models and Image Formation . . . . .	6
2.2 Planar Homography . . . . .	9
2.3 Computing Homography using Direct Linear Transform Method . . . . .	11
2.4 Singular Value Decomposition for Total Least Squares . . . . .	12
2.5 Creating the top-view map of players from the camera view images using Homography	13
2.6 Method 1: Computing Homography Transformations using crease points . . . . .	15
2.7 Method 2: Computing Homography Transformations using intersection points of crease lines . . . . .	15
2.8 Method 3: Computing Homography Transformations using crease lines and middle stump points . . . . .	15
2.8.1 Extending the Direct Linear Transform Method to compute Homography using points as well lines . . . . .	16
2.9 Method 4: Computing Homography Transformations by calibrating one camera using crease points in a zoomed-in view and calibrating the other cameras with respect to the first view using player footpoints . . . . .	17
2.9.1 Introduction . . . . .	17
2.9.2 Algorithm . . . . .	18
2.9.3 Proof . . . . .	18
2.9.4 Calibrating the other cameras with respect to the calibrated camera . . . . .	19
3 Multi View Multi Object Detection . . . . .	21
3.1 Introduction . . . . .	21
3.2 Current State of the Art Single View Object Detectors . . . . .	22
3.2.1 Two stage object detectors . . . . .	22
3.2.2 One stage object detectors . . . . .	23
3.3 Detection Metrics . . . . .	25
3.4 Issues with current off-the-shelf state of the art detectors . . . . .	26
3.5 You Only Look Once (YOLO) . . . . .	27
3.6 Optimising inference time using TensorRT framework . . . . .	27

3.6.1	Layer and Tensor Fusion . . . . .	27
3.6.2	Precision Calibration . . . . .	29
3.6.3	Kernel Autotuning . . . . .	31
3.7	Custom Data Collection and Annotation . . . . .	31
3.8	Improving accuracy: Finetuning the existing model . . . . .	32
3.9	Training on synthetic data and testing on real data . . . . .	32
3.10	End to End Deep Multi-view Detection Models . . . . .	33
4	Merging Three Top-View Maps into the Final Top-View Map . . . . .	36
4.1	Modelling as a Graph Optimisation Problem . . . . .	36
4.2	Solving the problem as a Linear Programming Optimisation . . . . .	38
5	Tracking . . . . .	40
5.1	Introduction . . . . .	40
5.2	Why is Tracking necessary? . . . . .	41
5.3	Different Tracking Metrics . . . . .	42
5.4	State of the Art Trackers: A Review . . . . .	45
5.5	Our Proposed Algorithm: MergeAndTrack . . . . .	46
5.5.1	Detection of Players and Projection of Detected Players to Top-View . . . . .	47
5.5.2	Merging the three top-views into a single combined top-view . . . . .	47
5.5.3	Estimating the future (next frame) location of existing players in the top-view using Kalman Filters . . . . .	48
5.5.4	Associating top-view detections from each camera with the existing top-view tracks . . . . .	49
5.5.4.1	Modelling the association problem as a Graph Optimisation Problem . . . . .	49
5.5.4.2	Integer Programming Formulation . . . . .	51
5.5.4.3	Hungarian Algorithm . . . . .	51
5.5.4.4	Intuition of the Hungarian Algorithm . . . . .	53
5.5.5	Creation and deletion of tracks . . . . .	54
6	Hardware Architecture . . . . .	55
6.1	Using BlackMagic Decklink Cards . . . . .	55
6.1.1	Architecture . . . . .	56
6.1.2	BlackMagic Decklink SDK . . . . .	57
6.2	Network Device Interface (NDI) cameras and tools . . . . .	58
6.2.1	Architecture using Single Network Interface Card . . . . .	58
6.2.2	Architecture using Multiple Network Interface Cards . . . . .	60
6.3	Communicating with Graphics Server . . . . .	61
7	Software Architecture . . . . .	62
7.1	A Streaming, Event-Driven Pipeline with a Modular Architecture . . . . .	63
7.2	Communication between the various Components . . . . .	63
7.3	How is the Data stored? . . . . .	64
8	Results and Conclusion . . . . .	65
8.1	Future Work . . . . .	65
8.2	Results from Asia Cup 2022 . . . . .	66

*CONTENTS*

ix

Bibliography . . . . . 72

## List of Figures

Figure	Page
1.1 An example of the bird’s eye view graphic showing the position of fielders on the field as white dots. The generation of such a graphic from cameras placed on stands is one of the focuses of the thesis. . . . .	2
1.2 The top-view tracking information can be projected onto any other broadcast camera views, and players can be highlighted as shown. This is one of the many visualizations which can be shown. . . . .	3
1.3 The crease lines (along with the dimensions) on a typical cricket pitch . . . . .	4
1.4 Point correspondences in the camera view (a) and top view (b) which can be used to compute the homography projection matrix projecting points from the camera view to the top-view. . . . .	4
2.1 (a) Shows the setup in which the mathematics behind image formation is derived. (b) Derived expression for the y-coordinate of the image of a 3D point, $\chi$ located at $(X, Y, Z)$ in the real world. The derivation comes from the fact that triangles $\Delta xCP$ and $\Delta \chi CZ$ are similar, and hence the length of their corresponding sides are in the same ratio. . .	7
2.2 The camera coordinate system and the world coordinate system may not always be aligned. In order to align the world coordinate system with the camera coordinate system, points in the world coordinate system have to undergo a translation followed by a rotation. . . . .	8
2.3 Homography matrix $H$ relating the transformation of a plane in the world with the image plane . . . . .	10
2.4 Creating the top view map of players is simply a planar transformation of the camera image plane (containing player foot points) to the actual cricket field plane as shown in the diagram. . . . .	13
2.5 The template top-view map used as the cricket ground plane in all our experiments. Camera image planes are transformed to this cricket ground plane. . . . .	14
2.6 Camera view image showing that parts of crease lines are somewhat visible, which can be used for camera calibration. . . . .	14
2.7 Zoomed-in image: This is the zoomed-in image of the original image. We can compute homography from this view to the top view map easily as the crease lines are clearly visible. . . . .	19
2.8 Original image: This is the image from which we want to compute homography to the top view map, but we cannot see any point correspondences that can be used to do so. .	20
2.9 Top view map of the original image showing the umpire and fielder positions as red dots.	20

3.1 The Yolo and the SSD architectures illustrated. . . . . 24

3.2 Diagram showing the various optimizations done by the TensorRT framework. . . . . 28

3.3 An illustration of layer fusion optimization done by the TensorRT inference engine. We can see that the conv, bias, and Relu layers are fused into a single layer called CBR (also shared horizontally). . . . . 29

3.4 An illustration of how the TensorRT inference engine performs Quantisation optimization to reduce the precision of the weights and activations in a trained model without compromising on the performance. . . . . 30

3.5 A sample synthetic camera view image we created using the Unity 3D game engine to train our detector model. We generated multiple sequences with different configurations by varying the number and position of cameras placed, the number and the 3D model of players spawned on the cricket ground. Each sequence typically contained around 200 frames. . . . . 33

3.6 End-to-End Deep Multi-view detection: The proposed GMVD architecture takes in any number of camera view images and directly predicts the bird’s eye view occupancy map. The Perspective Transformation stage requires well-calibrated camera matrices to project the camera view feature maps to the top view. . . . . 34

4.1 The above is a sample graph constructed where the number of unique objects when all three views are merged is 3. The three distinct colors represent nodes in the three different views. The diamonds represent hidden nodes, and the circles are the normal nodes corresponding to detections in a top view. The lines represent edges that connect every pair of nodes except pairs where both the nodes belong to the same view. . . . . 37

5.1 Figure illustrating true positive, false positive, and false negative detections during the course of object tracking. The large circles represent the ground truth object detections, while the small circles denote the predicted object detections. The same color shapes denote that the detections belong to the same identity. The arrows indicate the correspondences between the observed and the predicted detections.[1] . . . . . 42

5.2 Figure demonstrating identity switches during the course of object tracking. Identity switches are marked with blue ellipses with the label 'Mismatch' above them. The large shapes represent the ground truth object detections, while the small circles denote the predicted object detections. Object detections that belong to the same identity are denoted with the same color of the shapes.[1] . . . . . 43

5.3 Figure illustrating the False Positive Associations (FPAs), False Negative Associations (FNAs), and True Positive Associations (TPAs) for a true positive (TP) of interest[2]. . . . . 44

5.4 An illustration showing a motion model predicting the locations of players *A* and *B* at time instant  $t + 1$  by taking into consideration the velocities and positions of *A* and *B* at time instant  $t$ . . . . . 48

5.5 Graphical illustration of the data association step in the top-view assigning identities to incoming detections (from a particular camera) at time instant  $t$  using the consolidated information of top-view tracks at time instant  $t - 1$ . . . . . 50

6.1 BlackMagic architecture: Hardware architecture showing how cameras are connected to a BlackMagic Decklink 8K Pro capture card in our processing server for delivering multiple 4K streams at realtime speeds using the BlackMagic technology. . . . . 56

6.2 Network Device Interface (NDI) architecture: Hardware architecture showing how NDI cameras are connected to a Network Interface Card (NIC) in our processing server for delivering multiple 4K streams using the NDI technology. . . . . 59

6.3 Network Device Interface (NDI) architecture: Hardware architecture showing how NDI cameras are connected to multiple Network Interface Cards (NICs) in our processing server to increase the bandwidth for delivering multiple 4K streams at realtime speeds using the NDI technology. Note that the media converters are not shown to improve clarity. . . . . 60

7.1 Software Architecture: This figure illustrates our data-processing pipeline highlighting the major system components and showing the data flow through the pipeline using arrows. The figure also illustrates the transformation of the data through various stages of the pipeline. Note: Only one camera is shown in this figure for clarity however in reality there are multiple video streams entering the server from multiple cameras. . . . 62

8.1 A Graphical Visualisation of certain players being tracked in the camera view is a simple extension of our pipeline which is left as a work to be explored in the future. . . . . 66



## List of Tables

Table		Page
3.1	Table showing the performance of various detectors on the MS COCO dataset[3] . . .	24
3.2	Table showing the number of layers before and after layer fusion and elimination of unused layers in some popular Deep Neural Network Architectures. . . . .	28
5.1	Table showing the tracking performance of SORT on the Wildtrack dataset [4] using detections from various models. These results show how heavily the detection metric, MODA influences the tracking metrics, IDF1 and MOTA. Improving the MODA score directly improves the tracking performance by a similar margin. . . . .	46

## *Chapter 1*

### **Introduction**

In this chapter, we explain the problem statement in detail and why solving the problem is very interesting to the sports community. We discuss the challenges involved in solving the problem in the given setup. We present discussions on object detectors, multi-object multi-camera tracking, camera calibration, Hungarian Matching, Kalman Filters, and linear programming optimization methods. These concepts form the crux on which the work is built.

#### **1.1 Problem Statement**

People in India are so enthusiastic about cricket that it is often said they “eat”, “drink”, “talk”, “worship” and “live” cricket. It is a sport that brings everyone together irrespective of their age, religion, culture, etc. The amount of enthusiasm generated when matches are played is truly remarkable. This excitement has led to many people playing the sport resulting in the development of world-famous cricketers. Not only the people playing the sport but also people watching it are so enthusiastic about the game that they try to understand the game, the strategies, etc. Broadcasting companies like Star Sports have been putting a lot of resources into improving the understanding of the game by presenting interesting data visualizations and graphics so that the audience watching the game is more engaged. These graphic presentations not only help the players and the coaches in analyzing the performance but also young kids who want to be players someday. Commentators can further use these graphic presentations to explain the non-trivial strategies of teams.

One such graphic is the bird’s eye view map of the field, showing the location of fielders as dots. This graphic is demonstrated in Figure 1.1. Such a visualization conveys precise information regarding the field placement a particular bowler is using. We know this information is of great interest to teams as batters can gauge what type of ball to expect depending on the field placement. It would also help batters in picking the right gaps so that they can score boundaries and minimize the chances of getting out. Bowlers can use this type of data to understand what type of field a particular batter is weak playing at. So, a lot of useful analysis can be done if this data is recorded for every ball in the match.



Figure 1.1: An example of the bird's eye view graphic showing the position of fielders on the field as white dots. The generation of such a graphic from cameras placed on stands is one of the focuses of the thesis.

Currently, broadcasting companies like Star Sports use systems that consist of drone cameras. Not only are the current systems costly, but they also require manual intervention. Hence, an automatic cost-effective solution can benefit many broadcasters.

Once the bird's eye view map is generated, we can do a lot of useful things, like tracking players in the top view. The top view tracking data can further be used to show the identity of players in any calibrated camera view (as shown in Figure 1.2). We can also find out statistics like how much distance a particular fielder has run in a match, the speed with which a player is running, etc., using this top-view map.

## 1.2 Trivial Solution

We know that a cricket pitch contains crease markings. A typical pitch would have crease lines as shown in Figure 1.3. One could take a top-view map like this and compute the homography to project ground plane points from the image view to the top-view map. We can easily find 8-point correspondences. For example, Figure 1.4 shows the point correspondences in the top-view map and the image view, which can be used to compute the homography from the camera view ground plane to bird's eye view. Once the homography is computed, all we need to do is to run a pedestrian detector in the camera view, take the bottom center point of detected bounding boxes and project them to the top view. The bottom center point of a bounding box is precisely a pixel in the camera view, which corresponds to a point on the ground plane ( $Z = 0$ ) in the world coordinate system. This transformation

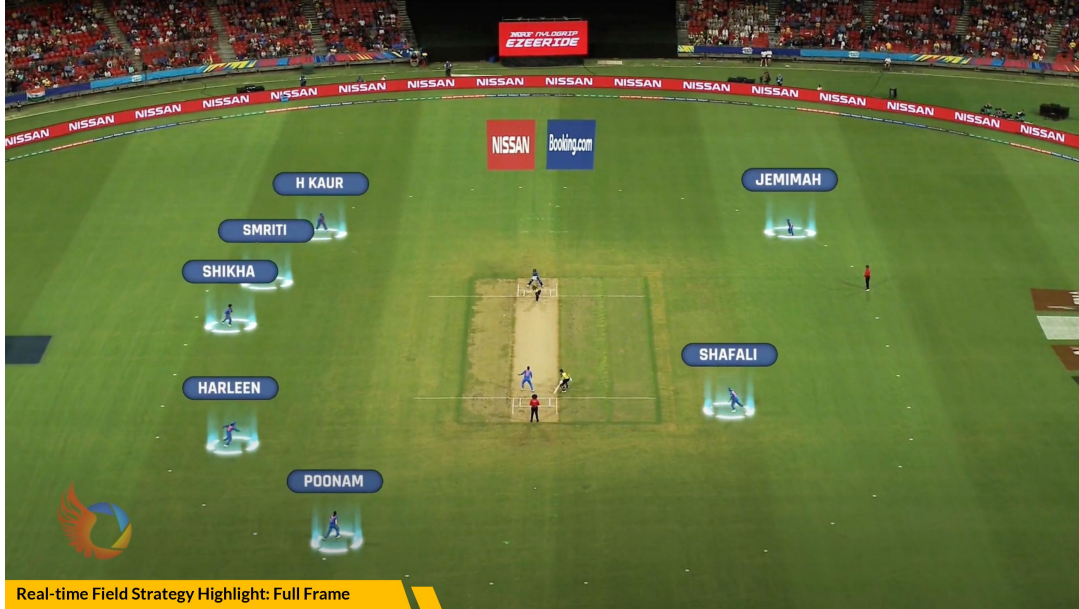


Figure 1.2: The top-view tracking information can be projected onto any other broadcast camera views, and players can be highlighted as shown. This is one of the many visualizations which can be shown.

can be achieved as follows:

$$P = s \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} = H \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (1.1)$$

where  $s$  is the scaling factor,  $H$  is the  $3 \times 3$  homography projection matrix projecting points from the camera view to the top view,  $(x, y)^T$  is the pixel coordinates in the camera view, and  $(X, Y)^T$  is the corresponding pixel coordinates in the top-view.

This would give a decent-looking top-view map showing the locations of fielders. However, in practical scenarios, we find that there are multiple challenges, and such a simple, straightforward solution does not work.

### 1.3 Challenges

- **Camera Placement:** The moment we place cameras on the stadium stands, it is impossible to see the crease markings. Hence, calibration cannot be done so easily. Zooming the camera to see the crease markings would decrease the coverage drastically, and we won't be able to see most of the players from the zoomed-in field of view.
- **Multiple Cameras:** The coverage of typical cameras is not enough to get all the players on the field, even from the topmost vantage point on the stands. This makes it a compulsion for us to

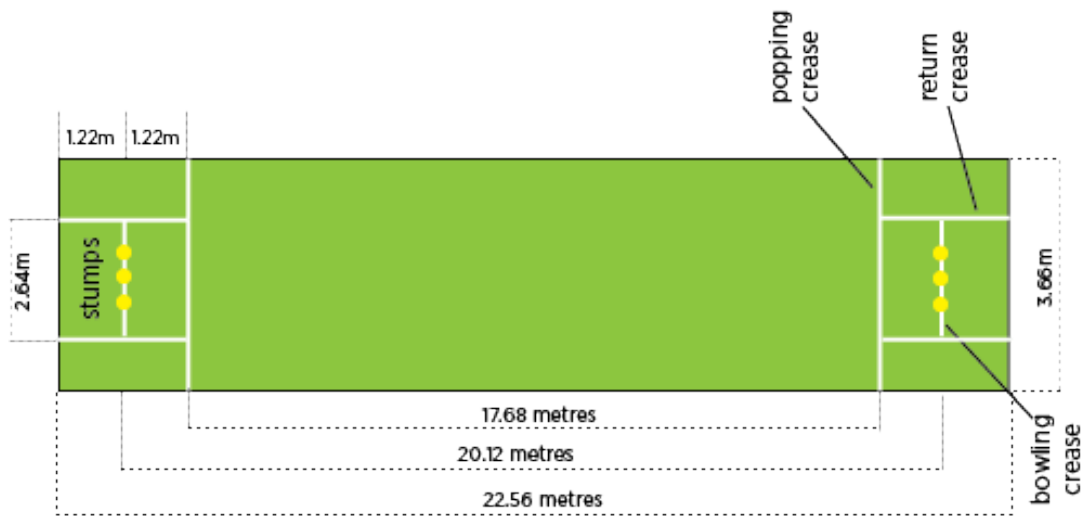


Figure 1.3: The crease lines (along with the dimensions) on a typical cricket pitch

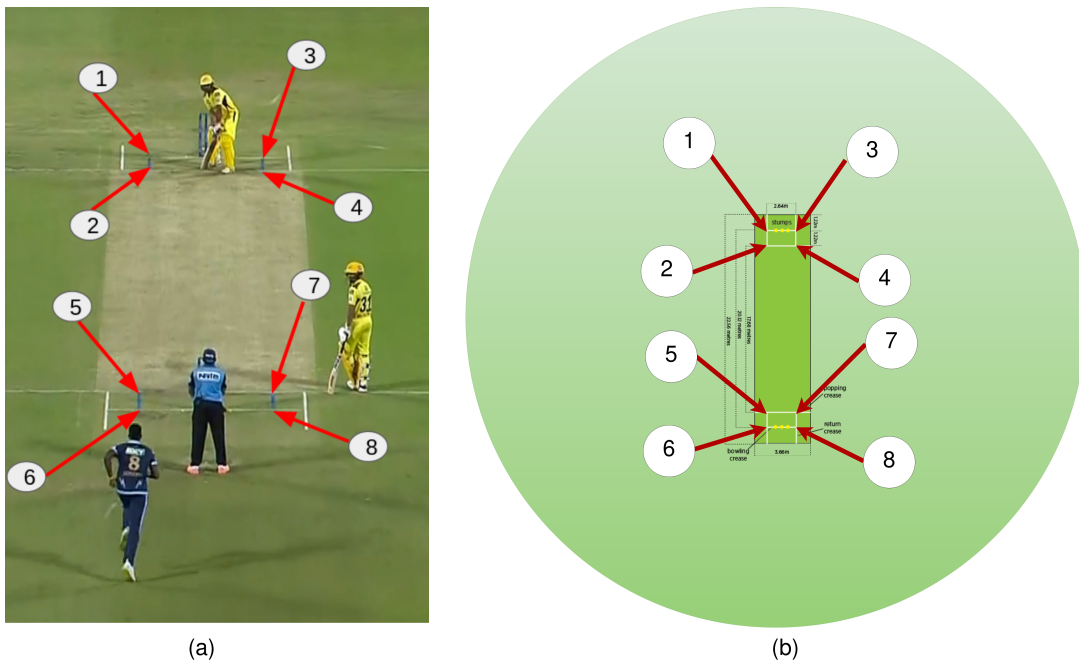


Figure 1.4: Point correspondences in the camera view (a) and top view (b) which can be used to compute the homography projection matrix projecting points from the camera view to the top-view.

use a multi-camera setup. For our purpose, we observe that three cameras placed at an angle of approximately 120 degrees from each other are an optimal cost-effective setup.

- Camera Synchronization: The moment we move to a multi-camera setup, synchronization of the feeds becomes very important. Since cameras will be located at different distances from the server, there may be slight delays between cameras. We have to account for such delays so that the algorithm does not give incorrect results.
- Data association between cameras: In a multi-camera setup associating players across cameras become a challenge, especially with weak calibration. We have to come up with algorithms that are robust to the calibration errors as well as associate players across cameras correctly.
- Player detection: Most pedestrian detectors are trained on a very different distribution of data in the sense that they cannot detect players from such a far distance. We had to fine-tune these detectors so that they could detect players in this setup. Especially, the bright lighting conditions make it very hard to detect players along with the white jersey color (in Test matches).
- Real-time Operations: One of the requirements of the project is to operate in real time. This makes things even more difficult. Even if 4K cameras could have been used in order to improve crease marking visibility and player detection, this becomes a problem now as none of the current pedestrian detectors can process 4K images even at 5 FPS. Our requirement was to get at least 20-25 FPS.
- GPU Heating: Detecting players using deep learning models at real-time speeds would require high-end GPUs. Running such a system for the entire duration of even a T20 match would heat up the GPUs considerably. Hence, placing the server in a cool room is of utmost importance. Bringing back the feeds to the broadcaster office and placing the server in the office seem to be the ideal thing to do. However, bringing back 4K feeds over long distances is extremely expensive. The bandwidth required would be extremely high, and it would cost nearly 4-5 Lakhs every match.

## Chapter 2

### Camera Calibration and Homography

The link between the real physical world and the image world is established through camera calibration. Each 3D point in the real world would be imaged onto a certain 2D point on the image plane. This mapping is determined by many internal parameters of the camera being used, for example, the focal length of the camera (intrinsic parameters). It is also dependent on where the camera is placed in the 3D world (extrinsic parameters). Once we know the intrinsic and extrinsic parameters, we will know exactly where any 3D world point would form its image in the image plane. Camera calibration is the process of finding out these intrinsic and extrinsic camera parameters. Moreover, it turns out that these parameters can be represented compactly using a  $3 \times 4$  matrix, commonly referred to as the  $P$  matrix.

#### 2.1 Camera Models and Image Formation

In this section, we briefly take a look at the mathematics behind image formation. Let us assume that a camera  $C$  is placed at  $(0, 0, 0)$  in the world coordinate system, and the coordinate system of the camera perfectly aligns with the world coordinate system, i.e., the camera is not rotated along any axis. The focal length of the camera, i.e., the distance between the camera center and the image plane, is  $f$ . Figure 2.1 shows this setup. Note that although in the real world, the image plane or the CCD array is located behind the camera center, the math works out in an exactly similar fashion when we mirror it and bring it in front of the camera center. Bringing the image plane in front of the camera center has a few advantages, one of the notable being we get an erect image.

Suppose we have a 3D point,  $\chi$ , in the world whose coordinates are  $(X, Y, Z)$ . We are interested in finding the image of this 3D point on the image plane. Precisely, we want to know the 2D point,  $x$ , on the image plane where the 3D point forms its image. Using the concepts of similarity of triangles, we can work out that the x-coordinate of the imaged 2D point( $x$ ) is given by the expression  $f \frac{X}{Z}$ . Similarly, the y-coordinate of the imaged 2D point( $x$ ) is given by the expression  $f \frac{Y}{Z}$ .

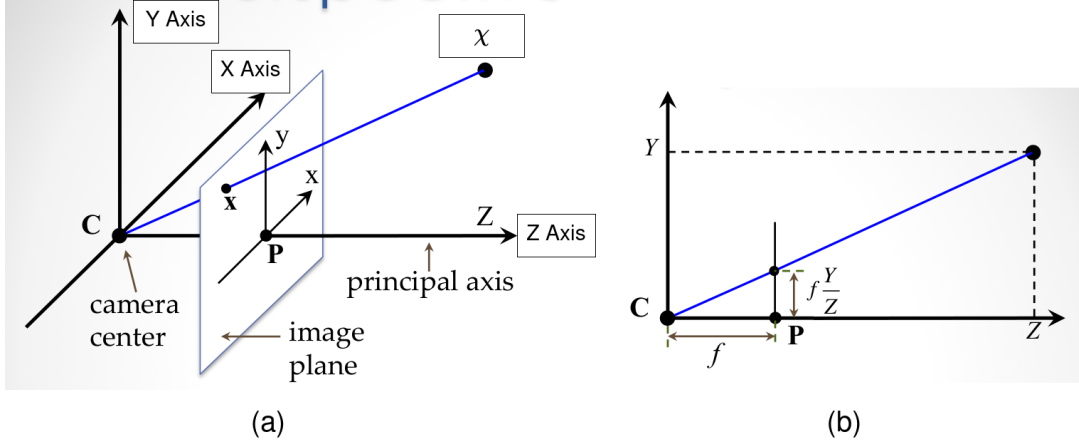


Figure 2.1: (a) Shows the setup in which the mathematics behind image formation is derived. (b) Derived expression for the  $y$ -coordinate of the image of a 3D point,  $\chi$  located at  $(X, Y, Z)$  in the real world. The derivation comes from the fact that triangles  $\Delta xCP$  and  $\Delta \chi CZ$  are similar, and hence the length of their corresponding sides are in the same ratio.

In matrix form using homogenous coordinates, the expression for the image of the 3D point,  $(X, Y, Z)$ , is as follows:

$$\tilde{x} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = P\bar{\chi} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \quad (2.1)$$

where the tilde over the  $x$  indicates homogenous coordinates and the bar over the  $\chi$  indicates an augmented vector. In order to get  $x$  which is the 2D point on the image plane, we divide by the scale factor as follows:

$$x = \begin{pmatrix} \frac{u}{w} \\ \frac{v}{w} \end{pmatrix} \quad (2.2)$$

We could re-write 2.9 as follows:

$$\tilde{x} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = K [I \mid 0] \bar{\chi} \quad (2.3)$$

where  $K$  is the internal camera matrix, also known as camera intrinsics.

Usually, the camera origin and the image origin are not the same and do not coincide. When referring to the coordinates of image points, we would want them to be in the image coordinate frame. In order



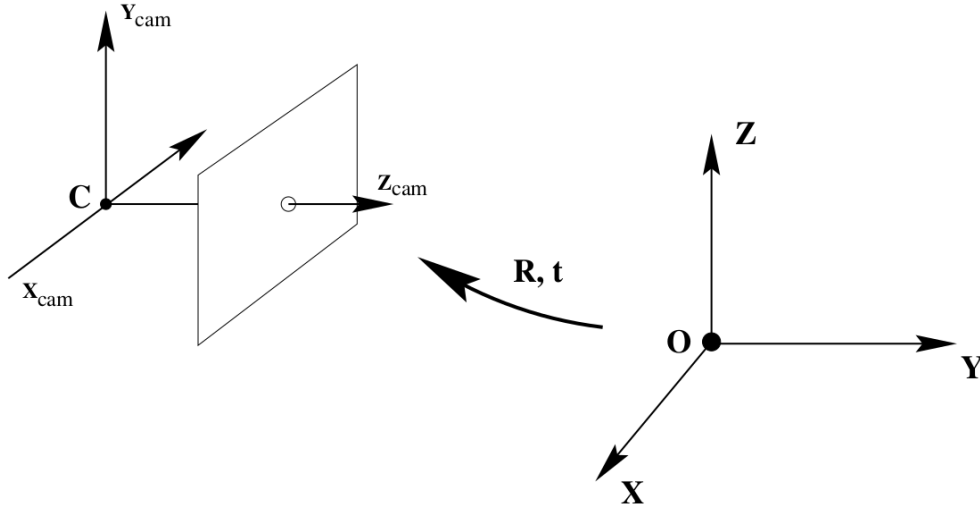


Figure 2.2: The camera coordinate system and the world coordinate system may not always be aligned. In order to align the world coordinate system with the camera coordinate system, points in the world coordinate system have to undergo a translation followed by a rotation.

to move the camera origin to the image origin, equation 2.3 can be re-written as below:

$$\tilde{x} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = K \begin{bmatrix} I & | & 0 \end{bmatrix} \bar{x} \quad (2.4)$$

where  $p_x$  and  $p_y$  are the offsets to be added to make the camera and image origin coincide.

We assumed that the camera coordinate system perfectly aligns with the world coordinate system and that the camera is placed at  $(0, 0, 0)$  in the world. However, in real life, this may not be the case always. The camera may be located at some coordinate  $(X_C, Y_C, Z_C)$  in the world, and it may be rotated along the X, Y, and Z axes of the world coordinate system. In such cases, it is necessary for us to align the world coordinate system and the camera coordinate system so that the math works out as derived above.

In general, we always refer to points, vectors, and everything in the world coordinate system. Let us assume a 3D point  $X$  in an inhomogeneous representation (referenced from the world coordinate frame). That same point would have different x, y, and z coordinate values when referenced from the camera coordinate frame (when the camera and world coordinate systems are not aligned). Let  $X_{cam}$  be the same 3D point (in an inhomogeneous representation, again) but referenced from the camera coordinate system. Let us assume that the camera is placed at  $C$  in the world coordinate system. The camera coordinates are  $(X_C, Y_C, Z_C)$ .

Let us take a concrete example where  $X$  is  $(X_C, Y_C, Z_C)$ . Now, we want to transform this point in such a way that its coordinates read  $(0, 0, 0)$  instead of  $(X_C, Y_C, Z_C)$ . In other words, we want to reference the point in the camera coordinate system (instead of in the world coordinate system)

because that's what the camera needs for the math to work out. Hence, a translation of  $-C$  is required to transform points from the world coordinate system to the camera coordinate system. When points undergo this translation, the world coordinates frame, and the camera coordinate have the same origin.

However, another transformation is required to align the coordinate axes of both the coordinate frames. Precisely, we want the vector  $(1, 0, 0)$  to align with the X-axis of the camera coordinate system. Similarly, we want the vectors  $(0, 1, 0)$  and  $(0, 0, 1)$  to align with the Y-axis and Z-axis of the camera coordinate frame, respectively. A 3x3 rotation matrix whose columns are vectors representing the camera coordinate axes will orient the world coordinate system with the camera coordinate system. Points undergoing these two sequences of transformations will rightly be transformed from the world coordinate frame to the camera coordinate frame, which is what we require for the math to work out as derived above. The following relation holds:

$$X_{cam} = R(X - C) \quad (2.5)$$

This sequence of transformations is shown visually in Figure 2.2.

We can write the same equation 2.5 using homogenous coordinates and in vectorized form as below:

$$\tilde{X}_{cam} = \begin{bmatrix} R & -RC \\ 0 & 1 \end{bmatrix} \bar{X} \quad (2.6)$$

Hence, when the camera is not placed at the world origin and is rotated by a certain orientation, we have to transform the world points to a camera coordinate system, following which we can use the camera matrix  $P$  to get the image coordinates of any 3D point in the world.

Putting everything together, we get the following equation:

$$\tilde{x} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{bmatrix} R & -RC \\ 0 & 1 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = K \begin{bmatrix} R & | & -RC \end{bmatrix} \bar{x} \quad (2.7)$$

where  $\bar{x}$  is the 3D world point in homogeneous coordinates,  $K$  is the 3x3 camera intrinsic matrix,  $R$  is 3x3 3D rotation matrix,  $-RC$  is the 3x1 translation vector.  $\begin{bmatrix} R & | & -RC \end{bmatrix}$  is the 3x4 camera extrinsic matrix.

## 2.2 Planar Homography

Planar homography, also known as projective transformation, is a class of transformation relating two planes (these planes need not only be image planes, homography can relate a plane in the real world with the image plane as well as shown in Figure 2.3).

Homography is a 3x3 matrix with 8 degrees of freedom. In other words, homography is defined only up to scale.  $h_{33}$  is usually fixed to one, and the entries of the homography matrix are normalized, i.e.,  $h_{11}^2 + h_{12}^2 + h_{13}^2 + h_{21}^2 + h_{22}^2 + h_{23}^2 + h_{31}^2 + h_{32}^2 + h_{33}^2 = 1$ .

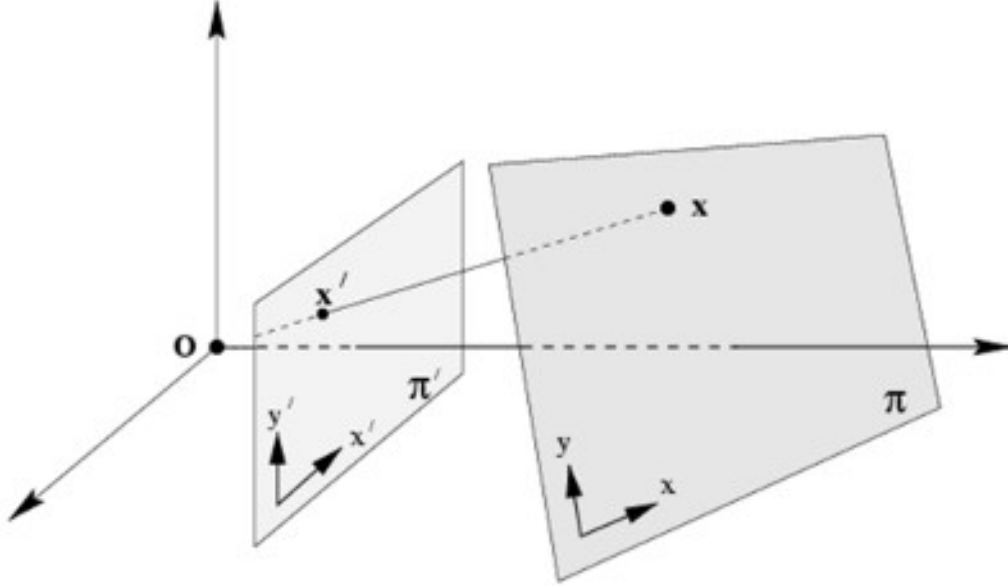


Figure 2.3: Homography matrix  $H$  relating the transformation of a plane in the world with the image plane

Suppose we are interested in establishing a transformation between the ground plane (i.e., 3D world points whose  $Z$ -coordinates are zero) and the image plane. In other words, we are interested in finding where all the 3D world points on the ground plane will form their image in the image plane. If we know the camera calibration matrix, we can easily do this:

$$\tilde{x} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = P\bar{\chi} = \begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ 0 \\ 1 \end{pmatrix} \quad (2.8)$$

where  $\chi$  is the 3D world point on the ground plane in inhomogeneous coordinates,  $P$  is the camera calibration matrix and  $\tilde{x}$  is the image coordinates in homogeneous system. Note that the  $Z$ -coordinate of  $\chi$  is set to 0 because the 3D world points we are interested in are situated on the ground plane. We can see that the third column of the camera matrix does not contribute to the value of  $u$ ,  $v$ , and  $w$  because  $Z = 0$ . Removing the third column from the camera matrix, we get the homography matrix  $H$  which transforms the 3D world points on the ground plane to the image plane:

$$\tilde{x} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = H\bar{\lambda} = \begin{pmatrix} p_{11} & p_{12} & p_{14} \\ p_{21} & p_{22} & p_{24} \\ p_{31} & p_{32} & p_{34} \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \quad (2.9)$$

where  $\lambda$  is the vector  $(X, Y)$ .

Camera calibration is a hard process, especially in a multi-camera setup where all cameras must be calibrated with a common frame of reference. Using a checkerboard, as suggested by [5], is practically impossible because the cameras are placed very far apart, and it will be impossible for multiple cameras to see the common checkerboard from such a great distance. Similarly, on the cricket ground, we do not have enough points whose exact 3D coordinates we can infer; hence we cannot use the 3D reference object-based calibration method (3D world point and 2D image point correspondences are used to calibrate the camera).

It turns out that we can compute the homography matrix  $H$  using point correspondences on the two planes. In other words, if we give the coordinates of points on the ground plane (in the world coordinate system) and their corresponding image points, then we can compute  $H$  as discussed in the next section.

## 2.3 Computing Homography using Direct Linear Transform Method

We create a set of point correspondences  $p_i, p'_i$  from the two planes. Let us assume that  $p_i$  is the source point, and  $p'_i$  is the destination point. In the example mentioned in the previous section,  $p_i$  is the ground plane point coordinates, and  $p'_i$  is the image of  $p_i$  on the image plane. We want to find the best estimate of  $H$  such that:

$$p'_i = H.p_i \quad (2.10)$$

where for all the correspondences  $i$ . Since the homography matrix has 8 degrees of freedom, we would need at least 4 point correspondences to establish  $H$  (each point correspondence gives two equations, one for x coordinate and one for y coordinate). Writing the linear equations for each correspondence:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \alpha \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2.11)$$

Expanding the matrix multiplication above:

$$x' = \alpha(h_1x + h_2y + h_3) \quad (2.12)$$

$$y' = \alpha(h_4x + h_5y + h_6) \quad (2.13)$$

$$1 = \alpha(h_7x + h_8y + h_9) \quad (2.14)$$

Substituting  $\alpha = \frac{1}{(h_7x+h_8y+h_9)}$  from Equation 2.14 into Equations 2.12 and 2.13, we get:

$$x'(h_7x + h_8y + h_9) = (h_1x + h_2y + h_3) \quad (2.15)$$

$$y'(h_7x + h_8y + h_9) = (h_4x + h_5y + h_6) \quad (2.16)$$

Expanding and rearranging the terms in Equations 2.15 and 2.16, we get:

$$-h_1x - h_2y - h_3 + h_7xx' + h_8x'y + h_9x' = 0 \quad (2.17)$$

$$-h_4x - h_5y - h_6 + h_7xy' - h_8yy' + h_9y' = 0 \quad (2.18)$$

Rewriting Equations 2.17 and 2.18 in matrix form:

$$A_i h = 0 \tag{2.19}$$

$$A_i = \begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & xx' & yx' & x' \\ 0 & 0 & 0 & -x & -y & -1 & xy' & yy' & y' \end{bmatrix} \tag{2.20}$$

$$h = [h_1 \ h_2 \ h_3 \ h_4 \ h_5 \ h_6 \ h_7 \ h_8 \ h_9]^T \tag{2.21}$$

We stack all the  $A_i$ s for all the point correspondences into a large matrix  $A$  and solve for  $h$  in  $Ah = 0$ . This is the classic homogeneous linear least squares problem which can be solved using Singular Value Decomposition.

## 2.4 Singular Value Decomposition for Total Least Squares

Writing the Linear Least Squares problem as an optimization problem:

$$\begin{aligned} & \text{minimize } \|Ah\|^2 \\ & \text{subject to } \|h\|^2 = 1 \end{aligned} \tag{2.22}$$

The 'subject to' constraint is important to prevent the solution from collapsing to the trivial solution of all 0s.

Using singular value decomposition, we can write

$$A = U\Sigma V^T \tag{2.23}$$

Assuming  $n$  point correspondences are stacked up to form  $A$ , the dimensions of matrix  $A$  is  $n \times 9$ . The dimensions of matrices  $U$ ,  $\Sigma$  and  $V^T$  are  $n \times n$ ,  $n \times 9$  and  $9 \times 9$  respectively.  $\Sigma$  is a diagonal matrix of singular values.  $U$  and  $V^T$  are orthonormal matrices, i.e., their rows/columns have a magnitude of 1, and the inner product between their rows/columns is 0.

Substituting Equation 2.23 into the formulation in 2.22,

$$\begin{aligned} & \text{minimize } \|U\Sigma V^T h\|^2 \\ & \text{subject to } \|h\|^2 = 1 \end{aligned} \tag{2.24}$$

Due to orthonormality of  $U$ , we can write:

$$\begin{aligned} & \text{minimize } \|\Sigma V^T h\|^2 \\ & \text{subject to } \|h\|^2 = 1 \end{aligned} \tag{2.25}$$

This is due to the property that  $\|Ux\|_2^2 = (Ux)^T Ux = x^T U^T Ux = x^T x = \|x\|_2^2$  when  $U$  is orthonormal. Geometrically, multiplying a vector by an orthonormal matrix just rotates the vector and has no impact on its norm.

Substituting  $y = V^T h$  into Equation 2.25:

$$\begin{aligned} & \text{minimize } \|\Sigma y\|^2 \\ & \text{subject to } \|h\|^2 = 1 \end{aligned} \quad (2.26)$$

Again, due to the orthonormality of  $V^T$ , we can write  $\|y\|_2^2 = \|V^T h\|_2^2 = \|h\|_2^2$ . Updating the constraints in Equation 2.26 in terms of  $y$ :

$$\begin{aligned} & \text{minimize } \|\Sigma y\|^2 \\ & \text{subject to } \|y\|^2 = 1 \end{aligned} \quad (2.27)$$

Since the diagonals in  $\Sigma$  are sorted in decreasing order, the solution to the optimization problem in Equation 2.27 is  $y = [0, 0, \dots, 1]^T$ . Since,  $y = V^T h$  and  $V^T = V^{-1}$  for orthonormal matrices, hence  $h = (V^T)^{-1} y = V y$ . Therefore,  $h$  is the last column of  $V$ , which by definition is the eigenvector of  $A^T A$  with the smallest eigenvalue.

## 2.5 Creating the top-view map of players from the camera view images using Homography

A cricket field is a plane in the real world. Players standing on the field have their feet grounded on the field plane. Points on this field plane form their images on the image planes of each of the 3 cameras capturing the scene as shown in Figure 2.4. We assume that the location of a player on the cricket field plane is given by the midpoint of the line joining both the footpoints of that player.

Given the images captured by the cameras, creating the bird's eye view is simply a planar transformation of the camera image plane to the cricket ground plane as shown in Figure 2.4. This can be achieved by computing the homography matrix, as discussed in the previous sections.

We can create the top-view map indicating player positions by applying a homography transformation to the bottom center point of the detected bounding boxes of players in the camera view. Taking the

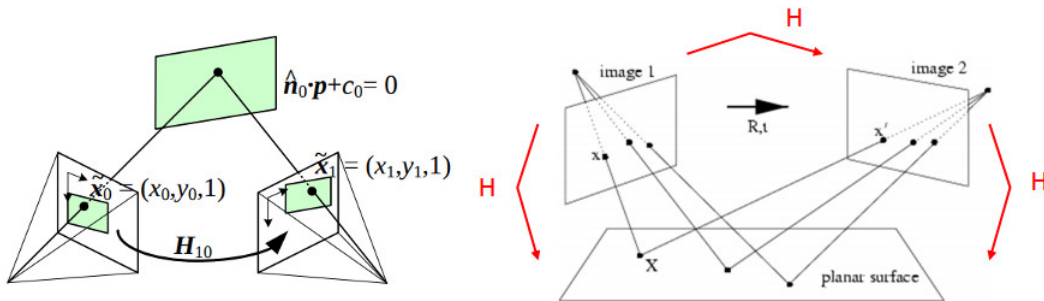


Figure 2.4: Creating the top view map of players is simply a planar transformation of the camera image plane (containing player foot points) to the actual cricket field plane as shown in the diagram.

bottom center point of a detected bounding box is a fairly good estimate of the image of the midpoint of the line joining both the footprints of a player.

In order to compute the homography matrix  $H$ , we need to find at least 4 point correspondences in the camera image and the cricket ground. It turns out that finding such correspondences is extremely difficult in our setting. Although there are crease markings on the cricket pitch, we cannot use them as they are not visible on the camera images due to the large distance the cameras are placed from the pitch.

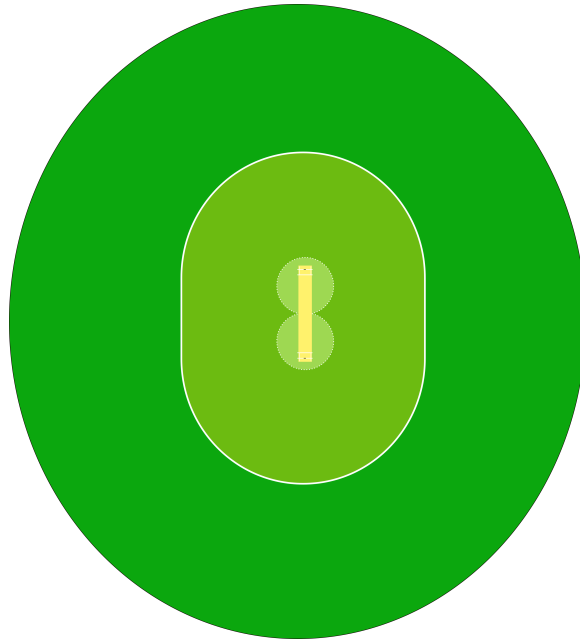


Figure 2.5: The template top-view map used as the cricket ground plane in all our experiments. Camera image planes are transformed to this cricket ground plane.



Figure 2.6: Camera view image showing that parts of crease lines are somewhat visible, which can be used for camera calibration.

We present qualitative results of calibrating cameras using four methods we engineered. The cricket ground plane to which the camera images are transformed is shown in Figure 2.5. This top-view template is used in all the results we present.

## **2.6 Method 1: Computing Homography Transformations using crease points**

The most basic approach to finding correspondences that we tried was by using the crease points shown in Figure 1.4. Although these points are very clearly visible on the cricket ground plane, finding correspondences of these 8 points in the camera images are extremely hard. In most situations, they are not at all visible. We do our best and approximately mark them to calibrate the cameras using this method. The results of calibrating the three cameras using this method on a Ranji Trophy match are shown in Figures (x, y, z).

## **2.7 Method 2: Computing Homography Transformations using intersection points of crease lines**

We observed that often the crease points are very hard to see in the camera images, but parts of the crease lines whose intersection points are the 4 crease points are somewhat visible (shown in Figure 2.6). Hence, we mark as many points as visible on each of these partly visible crease lines, estimate the equation of these crease lines in the camera images and then find the intersection points of these lines to get the 8 crease points in the camera views.

However, in some cases, we find that even these crease lines, are very difficult to see.

## **2.8 Method 3: Computing Homography Transformations using crease lines and middle stump points**

In the previous method, we saw that although all four crease lines may not be visible, but two of them are indeed visible. Similarly, although all crease points shown in method 1 may not be visible, but there are definitely a couple of point correspondences we could use, for example, the bottom of the middle stumps on both ends.

Now, we extend the DLT Method for computing homography to using point as well as line correspondences as shown in [6].



### 2.8.1 Extending the Direct Linear Transform Method to compute Homography using points as well lines

Let the equation of a line( $l$ ) on a plane  $P$  be given by  $ax + by + c = 0$ . We can represent this line as a vector,  $(a, b, c)$ . A 2-dimensional point  $X$  with coordinates  $(u, v)$  lies on the line  $l$ . Hence, we can write:

$$au + bv + c = 0 \quad (2.28)$$

Writing Equation 2.28 in vectorised form:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix}^T \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = l^T X = 0 \quad (2.29)$$

Let the correspondence of point  $X$  be  $X'$  on some other plane  $P'$ . Similarly, let the correspondence of the line  $l$  be  $l'$  in plane  $P'$ . Let homography matrix  $H$  transform the points on a plane  $P$  to plane  $P'$ . From the previous discussion, the following relation holds:

$$X' = HX \quad (2.30)$$

Since the correspondence of  $X$  is  $X'$ , and that of  $l$  is  $l'$ , the fact that  $X$  lies on  $l$  would imply  $X'$  lies on  $l'$ . Assuming the coordinates of  $X'$  is  $(u', v')$  and the line  $l'$  is represented by the vector  $(a', b', c')$ , we can write:

$$\begin{pmatrix} a' \\ b' \\ c' \end{pmatrix}^T \begin{pmatrix} u' \\ v' \\ 1 \end{pmatrix} = l'^T X' = 0 \quad (2.31)$$

Substituting the value of  $X'$  from Equation 2.30 into Equation 2.31, we get:

$$l'^T X' = l'^T HX = (H^T l')^T X = 0 \quad (2.32)$$

From equation 2.29, we know that  $l^T X = 0$ . Comparing this result with the result derived in Equation 2.32, we can conclude:

$$(H^T l')^T X = l^T X = 0 \quad (2.33)$$

$$l = H^T l' \quad (2.34)$$

Let  $H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix}$ . We normalize the equations of lines  $l$  and  $l'$  so that the constant term becomes 1. We then write Equation 2.34 as:

$$\begin{bmatrix} p \\ q \\ 1 \end{bmatrix} = \alpha \begin{bmatrix} h_1 & h_4 & h_7 \\ h_2 & h_5 & h_8 \\ h_3 & h_6 & h_9 \end{bmatrix} \begin{bmatrix} p' \\ q' \\ 1 \end{bmatrix} \quad (2.35)$$

where  $p' = \frac{a'}{c'}$ ,  $q' = \frac{b'}{c'}$ ,  $p = \frac{a}{c}$  and  $q = \frac{b}{c}$ .

Expanding the matrix multiplication above:

$$p = \alpha(h_1p' + h_4q' + h_7) \quad (2.36)$$

$$q = \alpha(h_2p' + h_5q' + h_8) \quad (2.37)$$

$$1 = \alpha(h_3p' + h_6q' + h_9) \quad (2.38)$$

Substituting  $\alpha = \frac{1}{(h_3p' + h_6q' + h_9)}$  from Equation 2.38 into Equations 2.36 and 2.37, we get:

$$p(h_3p' + h_6q' + h_9) = (h_1p' + h_4q' + h_7) \quad (2.39)$$

$$q(h_3p' + h_6q' + h_9) = (h_2p' + h_5q' + h_8) \quad (2.40)$$

Expanding and rearranging the terms in Equations 2.39 and 2.40, we get:

$$-h_1p' - h_4q' - h_7 + h_3p'p + h_6pq' + h_9p = 0 \quad (2.41)$$

$$-h_2p' - h_5q' - h_8 + h_3p'q - h_6q'q + h_9q = 0 \quad (2.42)$$

Rewriting Equations 2.41 and 2.42 in matrix form:

$$A_i h = 0 \quad (2.43)$$

$$A_i = \begin{bmatrix} -p' & 0 & p'p & -q' & 0 & pq' & -1 & 0 & p \\ 0 & -p' & p'q & 0 & -q' & q'q & 0 & -1 & q \end{bmatrix} \quad (2.44)$$

$$h = [h_1 \ h_2 \ h_3 \ h_4 \ h_5 \ h_6 \ h_7 \ h_8 \ h_9]^T \quad (2.45)$$

## 2.9 Method 4: Computing Homography Transformations by calibrating one camera using crease points in a zoomed-in view and calibrating the other cameras with respect to the first view using player foot-points

### 2.9.1 Introduction

Getting the 4-point correspondences from a plane for computing the homography may be really difficult in certain cases. Our problem of interest here is computing the homography to generate the top view map from the broadcast (side) view. To find point correspondences on the cricket field plane, one needs to be able to see the crease markings on the pitch, as there are no other reliable point correspondences that can be used on the field. Moreover, cameras are set up without zooming in too much, as this would decrease the field of view drastically, thereby missing capturing many players. In such a setting, it is very difficult to see the crease markings, especially from side views. Here, we propose an idea so

that one can compute the homography from the camera view to the top view by zooming in, capturing a frame, and then zooming out back to desired broadcast view (so that the majority of players can be captured). Note, while zooming in and zooming out, the camera position and orientation should strictly be left untouched.

### 2.9.2 Algorithm

1. Let us call the frame captured in Zoomed in view as *zoom\_img* and the frame captured in the desired broadcast view as *original\_img*.
2. Compute the Affine transform from *original\_img* to *zoom\_img*. Let us represent this transformation by  $T$ . To compute the affine transform, one needs three-point correspondences in both the views. However, the constraint of finding points on the plane is relaxed. This is a very useful relaxation as we can use holes in letters visible on stands to get accurate point correspondences. We may also use other point clues like flags at the boundaries, or we may ask someone to hold some large pattern for calibration on the opposite stand.
3. Compute homography using points in *zoom\_img*. Let us denote the computed homography matrix as  $H'$ .
4. Compute homography( $H$ ) for *original\_img* using the formula  $H = H'.T$  where  $.$  represents matrix multiplication.

Source code is available here on request, as this is a private repository currently.

### 2.9.3 Proof

Let us denote the required homography matrix in *original\_img* as  $H$ . Let us denote the homography matrix in *zoom\_img* as  $H'$ . Let us denote a point  $x$  in *original\_img*, which maps to the point  $y$  in the top view of the map. The point corresponding to  $x$  in *zoom\_img* is denoted by  $x'$ . Let us denote the affine transformation from *original\_img* to *zoom\_img* as  $T$ .

Since, *original\_img* and *zoom\_img* are related by the Affine Transform,  $T$ ; therefore,

$$x' = T.x \tag{2.46}$$

Now, we know that point  $x'$  maps to point  $y$  in top view by the Homography  $H'$ . Mathematically,

$$y = H'x' \tag{2.47}$$

Also, we know that point  $x$  will map to point  $y$  by the Homography matrix  $H$ .

$$y = Hx \tag{2.48}$$

Substituting 2.46 in 2.47, we get:

$$y = H'T.x \quad (2.49)$$

Comparing 2.48 and 2.49, we get:

$$H = H'T \quad (2.50)$$

This proves why the idea works.

#### 2.9.4 Calibrating the other cameras with respect to the calibrated camera

Once we calibrate any one of the cameras by zooming in as mentioned above, we can calibrate the other cameras with respect to this camera by using the bottom center points of player detections as correspondences. First, we draw bounding boxes around 4 players in the already-calibrated camera view and then project the bottom center points of the drawn bounding boxes to the top view using the computed homography matrix. These 4 points in the top view serve as a reference to calibrate the other cameras. All we need to do is find correspondences of these 4 reference top-view points in the camera view we want to calibrate. To do so, bounding boxes are drawn around the same 4 players in the to-be-calibrated camera view, whose bottom center points have correspondences in the top view. These 4 point correspondences are used to calibrate another camera with respect to the already calibrated camera. Calibrating this way means we are assuming that the bottom center point of bounding box detections coincides with the same  $3D$  world point. It turns out that this is a perfectly valid assumption to make in our setting because the players are detected from a far distance and appear very small.

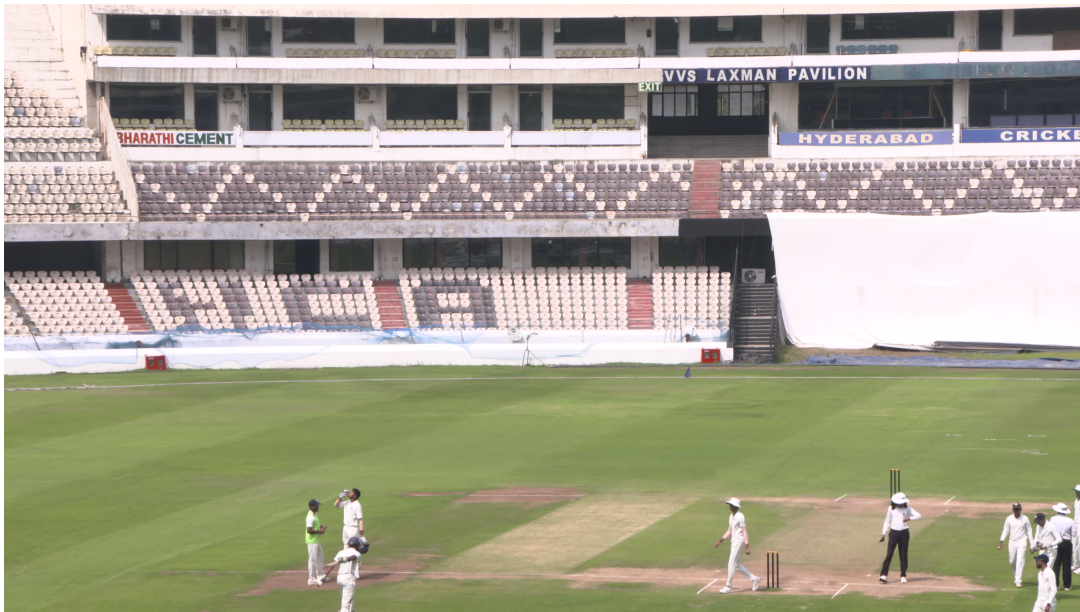


Figure 2.7: Zoomed-in image: This is the zoomed-in image of the original image. We can compute homography from this view to the top view map easily as the crease lines are clearly visible.



Figure 2.8: Original image: This is the image from which we want to compute homography to the top view map, but we cannot see any point correspondences that can be used to do so.

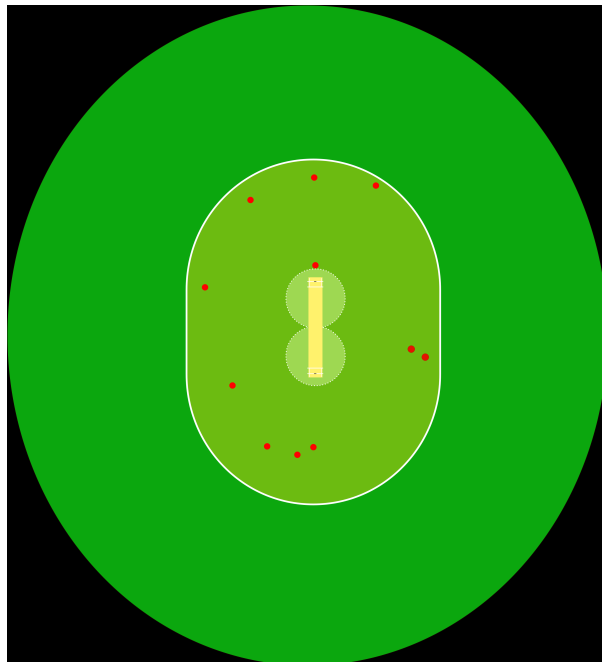


Figure 2.9: Top view map of the original image showing the umpire and fielder positions as red dots.

## *Chapter 3*

### **Multi View Multi Object Detection**

#### **3.1 Introduction**

Object Detection is one of the fundamental tasks in Computer Vision wherein rectangular bounding boxes are drawn around objects to indicate their presence and location in an image. The challenge in object detection comes because of occlusion, i.e. when something else blocks the visibility of an object. To improve the accuracy of object detectors, often multiple cameras/views are deployed. This would help in solving the problem arising due to occlusion. Even if an object is occluded in one view, it can be visible in some other camera view leading to successful detection of the object in the scene. This domain where multiple cameras are deployed to solve the detection problem is called Multi-View Detection, and when more than one object is being detected in the scene, it is called Multi-View Multi-Object Detection.

In Multi-View Detection, the most challenging problem we face is establishing the correspondence of detections across different views. In other words, a detection in one view corresponds to which detection in another view (if at all there is a correspondence). The most popular way of doing this is by using a Probabilistic Occupancy Map, but with this comes an additional challenge of calibrating the cameras (almost) perfectly.

In our use case, deploying a Multi-View detection setup has both advantages and disadvantages. The advantage is being able to tackle occlusions. Additionally, since our cameras are placed in the stands, we have to detect players who appear extremely small. Players standing diametrically opposite to a camera appear even smaller making them extremely hard to detect. Placing multiple cameras would help us solve such problems as the hope is there will be at least another camera that is nearer to the diametrically opposite player, and even if one camera misses it, that other camera (nearer to the player) will detect it.

Deploying a Multi-View detection setup poses certain challenges. Firstly, it increases the cost of the technical infrastructure a lot. Secondly, in our setup, camera calibration is very hard due to the placement of cameras, as discussed in Chapter 2. Poor calibration makes the establishment of detection correspondences across camera views even harder. Although we propose a robust algorithm to correctly

establish correspondences of detections across multiple camera views in Chapter 4, we find it difficult to run it at real-time speeds.

We proposed solutions for both single-view as well as multi-view use cases. This chapter focuses on the engineering we applied to existing state-of-the-art detectors to make them work well for our use case.

## 3.2 Current State of the Art Single View Object Detectors

### 3.2.1 Two stage object detectors

Two-stage object detectors are object detectors which consist of two main stages in their object detection pipeline:

1. **Region Proposal:** In this stage, a number of region proposals are generated which can potentially contain objects. Region proposals are typically in the form of bounding boxes. One of the most common region proposal algorithms is Selective Search. Selective Search is a bottom-up algorithm that starts with an initial segmentation of the image. Using an algorithm like what is mentioned in [7] works quite well to get the initial segmentation. It then iteratively combines/merges similar regions into larger regions based on a similarity score threshold. The similarity score takes into account various factors like size, texture, color, and shape of the merging regions so that multiple grouping criteria are taken into consideration in deciding what separates an object. Finally, the segmented patches are used to come up with bounding box proposals.
2. **Object Classification:** In this stage, each of the proposed regions from the Region Proposal stage is classified as whether they contain an object or not. If at all a region proposal contains an object, this stage classifies the type of object (it contains) and also refines the bounding box of the proposed region to better fit the object contained in it.

We present a brief review of some of the very popular two-stage object detectors below:

1. **Region-based Convolutional Neural Networks (R-CNN)[8]:** R-CNN is a two-stage object detector introduced by Girshik et al. in 2014. This method generates a large number of region proposals (approximately 2000) using a region proposal generation algorithm like Selective Search[9]. Each of the generated region proposals is passed through a CNN like ResNet[10], AlexNet[11], or VGG[12] to extract features. These extracted features of region proposals are then classified whether it contains a particular class of object or not. This is achieved by training multiple linear Support Vector Machine[13] classifiers, one for each class (popularly known as One vs All Classification in Literature). The class with the highest confidence of classification is said to be contained in the corresponding region proposal. A bounding box regressor is trained along with the classification head to predict a bounding box that better fits the object contained.

2. Fast R-CNN[14]: Fast RCNN improves on R-CNN by making the feature extraction process more efficient. In this method, instead of passing each region proposal through a CNN separately, the entire image is passed through the CNN. This leads to a massive reduction in the amount of computation because, typically, region proposals have large overlaps, and computation for overlapping regions can be shared. A unique idea that this algorithm proposes is the concept of Region of Interest (RoI) pooling. Region proposals can be of different dimensions, but the extracted feature of each region should be of the same dimensions. Each proposal has a corresponding mapping on the shared convolutional features map, which can be different. In order to transform them into embeddings of fixed dimensions, RoI pooling is used. No matter what the dimensions of the region proposal are, RoI pooling will always give an embedding of fixed dimensions.
3. Faster R-CNN[15]: Faster R-CNN further advances the Fast R-CNN algorithm by introducing a region proposal network (RPN). Instead of using the Selective Search algorithm to generate region proposals, which is quite slow, Faster R-CNN makes use of a region proposal network to come up with the proposals. This algorithm makes the object detection pipeline end-to-end learnable. Designing end-to-end learning pipelines is a key step toward improving the overall accuracy of a system. The RPN includes a Convolutional Neural Network, which extracts features from the input image. The outputted feature map is then passed through a  $3 \times 3$  convolutional layer with a padding of 1. This converts each pixel in the outputted feature map to a vector of dimension  $c$ , which is then simultaneously passed through a classification layer and a bounding box regression layer (these are  $1 \times 1$  convolutional layers) outputting  $2k$  and  $4k$  entries respectively (where  $k$  is the number of anchor boxes). The classification layer outputs two entries for each anchor box, indicating the score of the presence or absence of an object in that anchor box. Similarly, the regression layer takes as input  $k$  anchor boxes and transforms each of the anchor boxes to better fit the object contained, thereby outputting 4 values (denoting the coordinates and dimensions of a bounding box) for each of the  $k$  anchor boxes. The region proposals are determined by taking a Non-maximal suppression of the anchor boxes, which were classified to contain an object. These region proposals are then passed through the RoI pooling layer to make them of fixed dimensions, after which they are fed through a classification and regression layer simultaneously (exactly like Fast R-CNN).

### 3.2.2 One stage object detectors

Single-stage object detectors do not rely on region proposals to perform object detection. This class of detectors can predict the bounding boxes and their class probabilities in a single forward pass, making them much faster compared to two-stage object detectors.

We present a brief overview of two of the most popular single-stage object detectors:

1. You Only Look Once (Yolo) [21]: This single-stage object detector proposed by Redmon et al. in the year 2016 can detect objects at real-time speeds. Yolo divides the input image into a grid



Detector	Inference time (in FPS)	Mean Average Precision (mAP-0.5)
Fast R-CNN [14]	0.5	39.9
Faster R-CNN VGG-16 [15][12]	7	42.7
Faster R-CNN ResNet [15][10]	5	45.3
SSD300 [16]	46	41.2
SSD500 [16]	19	46.5
Yolo v2 [17]	40	44.0
Yolo v3 [18]	35	55.3
Yolo v4 [19]	38	62.8
Yolo v7 [20]	<b>75</b>	<b>69.7</b>

Table 3.1: Table showing the performance of various detectors on the MS COCO dataset[3]

of  $S \times S$  cells. Each cell outputs  $B * 5 + C$  values where  $B$  is the number of predefined anchor boxes, and  $C$  is the number of classes. The 5 predicted values for a bounding box in a cell consist of  $x, y, w, h$  and the confidence denoting the presence of an object. Note that the  $x, y$  coordinates are relative to the grid cell. The  $C$  predicted values for a grid cell represent the scores of the  $C$  classes (contained in it). The grid cell, which contains the center of an object, is responsible for detecting that object. The input image is passed through a series of convolutional and max pool layers to extract a feature map of dimensions  $S \times S \times (B * 5 + C)$ . Finally, the predicted bounding boxes (which have confidence greater than some set threshold) are passed through a Non-Maximal Suppression algorithm to remove duplicates/overlapping bounding boxes.

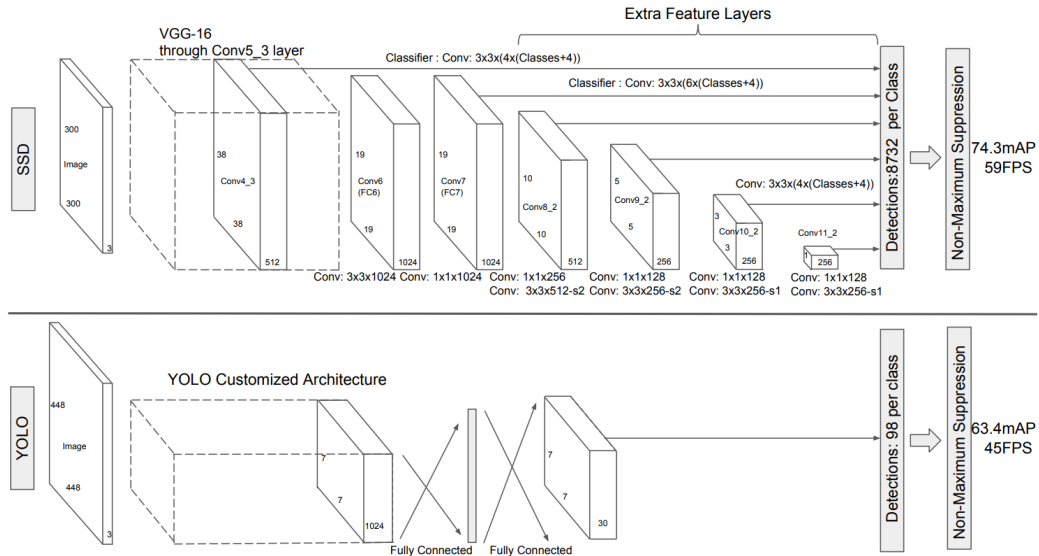


Figure 3.1: The Yolo and the SSD architectures illustrated.

2. Single Shot Multibox Detector (SSD) [16]: SSD is another very popular one-stage object detector that can detect objects of different scales and aspect ratios. The pipeline is very similar to Yolo, except the fact that SSD uses anchors on multiple feature maps at different resolutions to come up with the predicted bounding boxes. The width and height of the anchor boxes employed are relative to the resolution of their corresponding feature map. The SSD architecture (compared with Yolo architecture) is shown in Figure 3.1. As we can see, from each feature map, a classification head (comprising of  $3 \times 3$  Convolutional filters) comes out (denoted by the long arrows). This head is responsible for turning each pixel in the feature map into a fixed-size vector of dimensions  $x * (\text{classes} + 4)$ , where  $x$  is the number of predefined anchor boxes corresponding to that layer. The additional 4 elements outputted denote the  $x, y, w, h$  offsets of the corresponding anchor to better fit the object contained. The remaining *classes* values denote the scores for each class. Just like Yolo, the final step is passing the predicted bounding boxes through a Non-Maximal Suppression algorithm to rule out duplicates/overlapping bounding boxes of lower confidence.

### 3.3 Detection Metrics

1. Intersection Over Union (IoU): It is a commonly used metric in object detection and tracking tasks to measure the accuracy of object localization. It computes the amount of overlap between the predicted bounding box and the ground truth bounding box. Mathematically,

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}} \quad (3.1)$$

2. False Positives (FP): This metric computes the number of erroneous predicted bounding boxes, i.e., it gives the number of detections predicted by the algorithm but, in reality, does not exist.
3. True Positives (TP): This metric gives the number of predicted detections which has a corresponding ground truth detection. We say that the predicted detection is successfully matched with a ground truth detection. We must note that a match is successful only if the predicted detection and the ground truth detection have a certain amount of overlap measured by the IoU metric.
4. False Negatives (FN): This metric computes the number of misses by an algorithm. In other words, it gives the number of detections present in the ground truth but missed by the algorithm.
5. Precision: Precision measures how precise the predicted detections are. In other words, it tells us, out of all the detections predicted by the detector, how many are actually true positive detections. Mathematically,

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{\text{Positive detections predicted by the detector}} \quad (3.2)$$

6. Recall: Recall measures how good the detector was in capturing the positive detections in the ground truth. Mathematically,

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{Positive detections in the ground truth}} \quad (3.3)$$

7. Average Precision: It is the average of the precision values calculated at various recall values and is computed by taking the area under the Precision-Recall curve.

8. Mean Average Precision (mAP): Typically, Average Precision is computed for every object class separately. Mean Average Precision is computed by taking the mean of Average Precision values across the different object categories. mAP-0.5 (in table 3.1) represents the Mean Average Precision computed using an IoU threshold of 0.5.

9. Multi-Object Detection Accuracy (MODA): It measures the accuracy of the detection system by taking into consideration missed detections (FNs) and hallucinated detections (FPs). Mathematically,

$$\text{MODA} = 1 - \sum_t \frac{\text{False Positives} + \text{False Negatives}}{\text{Number of GT detections}} \quad (3.4)$$

10. Multi-Object Detection Precision (MODP): It measures the localization error of the detection system by taking into consideration how much overlap the predicted detections have with their corresponding (actual) ground truth detections. Mathematically,

$$\text{MODP} = \frac{\sum_t \sum_i \text{IoU}_i^t}{\text{Total number of frames}} \quad (3.5)$$

where  $\text{IoU}_i^t$  is the Intersection Over Union score of the  $i^{\text{th}}$  predicted detection with its matched ground truth detection in the  $t^{\text{th}}$  frame.

### 3.4 Issues with current off-the-shelf state of the art detectors

State-of-the-art (SOTA) in Object Detection has advanced quite a bit in the past few years. However, we still could not use any of them off-the-shelf (OTS) because of their limited generalization capabilities to unseen or uncommon types of objects or environments. Moreover, OTS object detectors are not robust and are very sensitive to changes in scale, lighting conditions, background clutter, and image quality. Ours was a use case where we had to detect extremely small persons on a green field. Also, we were not really interested in the detection of any other objects apart from the 15 players on the field.

Additionally, these state-of-the-art detectors are not trained on  $4K$  images, but we were dealing with  $4K$  feed. This caused the detectors to run at a much slower speed than what was reported. Usually, high-performing object detectors are computationally expensive, whereas lightweight fast detectors are not very accurate. We had to strike the right balance between performance and speed.

The above reasons clearly demanded a customized approach, the details of which we present in the subsequent sections of this chapter. We will first look at how we optimized the inference time of the detector using the TensorRT framework. Next, we will take a look at how we built a customized detector only to detect players using data that we captured and manually annotated from live matches. Finally, we throw some light on using synthetic data for building this customized detector, as the process of data collection and annotation is extremely expensive.

### **3.5 You Only Look Once (YOLO)**

Since we required a detector with real-time inference speeds for our use case, it is clear from the review in Section 3.2 that we had to use a One Stage Object Detector. Yolo and SSD were the two options we had to choose from. We chose Yolo over SSD. SSD is very good at detecting objects of various scales because it employs feature maps at multiple scales. This makes them slower to Yolo. However, the multiscale feature maps are something we do not require in our use case because we are interested in detecting objects of uniform scale. In other words, all the players appear at the same scale in the camera view. Yolo, on the other hand, typically divides the image into a grid of  $7 \times 7$  elements. Hence, it is not very good at detecting very small-scale objects. In our case, the players in the camera view appear pretty small. Hence, we had to customize Yolo to split the input image into a grid of size  $136 \times 136$  instead. This meant Yolo could not run as fast as reported. Hence, we had to make the implementation more efficient, and this was done by implementing Yolo in C++ using Nvidia's TensorRT framework (covered in the next section 3.6). We further optimized Yolo by finetuning it to detect objects of a single class instead of detecting objects of multiple classes(covered in section 3.8). Both these optimizations allowed us to run the Yolo detector at around 30 frames per second.

### **3.6 Optimising inference time using TensorRT framework**

Even after using the most efficient implementation of Yolo, we could achieve only 10 frames per second on the 4K feeds. We had to search for better alternatives, and we found TensorRT, an SDK developed by Nvidia for high-performance, low-latency deep learning inference on Nvidia GPUs. TensorRT is known to speed up the inference by almost 36 times. TensorRT takes as input a trained neural network, optimizes it by applying a number of optimization techniques, and finally outputs the optimized inference engine (as shown in Fig. 3.2). The subsequent subsections present details on each of the optimizations applied by the TensorRT inference optimizer.

#### **3.6.1 Layer and Tensor Fusion**

The TensorRT compiler analyzes the computational graph of the neural network and looks for ways to optimize the structure of it without changing the functional behavior of the network. One of the

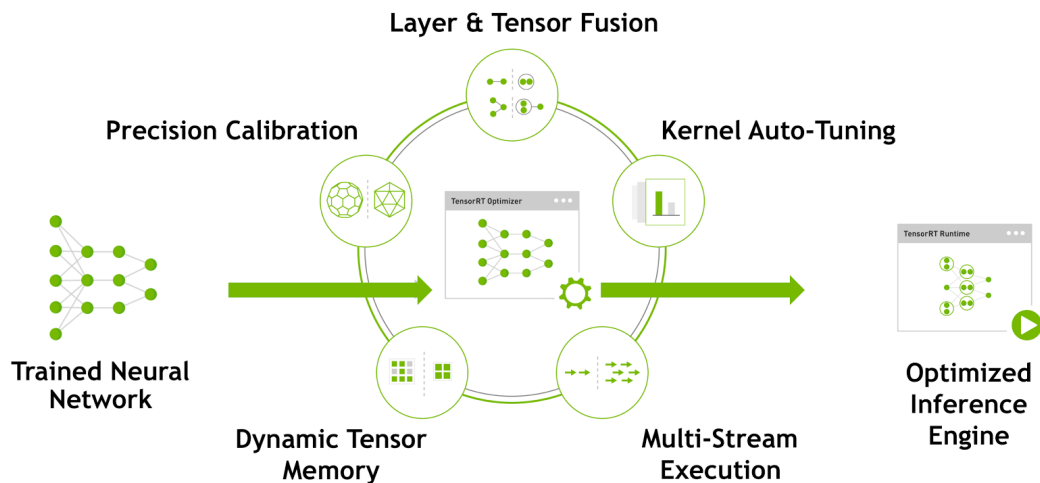


Figure 3.2: Diagram showing the various optimizations done by the TensorRT framework.

Network	Original number of Layers	Number of Layers after fusion
VGG19	43	27
Inception V3	309	113
ResNet-152	670	159

Table 3.2: Table showing the number of layers before and after layer fusion and elimination of unused layers in some popular Deep Neural Network Architectures.

optimization techniques is called layer fusion, where the compiler fuses multiple consecutive layers into a single layer. This transformation reduces the size of the neural network and improves its inference time. During the execution of a computational graph, a layer typically involves multiple function calls. Since the computation of the layer happens in the GPU, this means launching additional CUDA kernels. Launching a CUDA kernel is a much bigger overhead compared to the computation done by the kernel. Also, having multiple layers leads to increased GPU memory transfer, which is much slower compared to the fast on-chip memory access. Fusing layers mean the intermediate results from one layer remain in the cache and can be reused in the computation of the next layer without the need for data transfer to and from the GPU memory. This also leads to a reduction in the GPU memory footprint because a lesser number of layers need to be stored in the GPU memory.

Layer fusion is illustrated in Figure 3.3. For example, the Convolutional, bias, and ReLU layers in the un-optimized network are fused together into a single kernel called the CBR layer. This type of fusion is called vertical fusion. TensorRT also fuses layers horizontally that share the same input data and have the same filter size but different weights. This is depicted by the large horizontal 1x1 CBR block. TensorRT could have got rid of the "Concat" layer by preallocating output buffers and writing into them in a strided fashion.

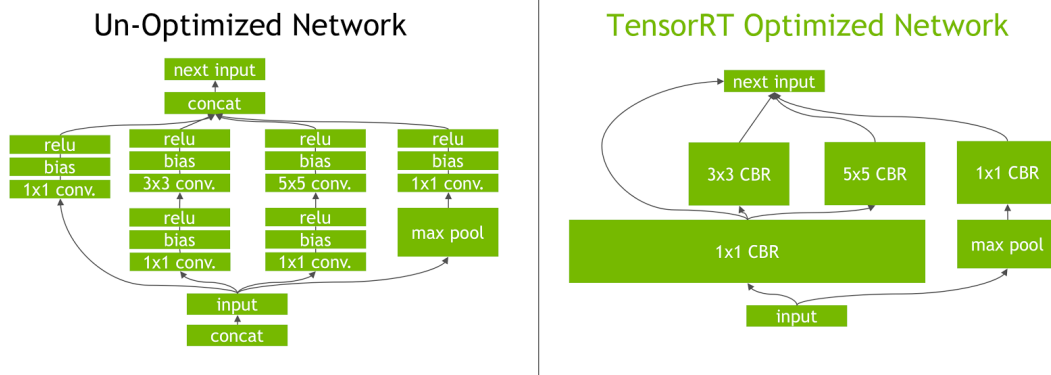


Figure 3.3: An illustration of layer fusion optimization done by the TensorRT inference engine. We can see that the conv, bias, and ReLU layers are fused into a single layer called CBR (also shared horizontally).

In order to understand layer fusion intuitively, consider the analogy where we want to buy three items from the supermarket. We have the option of making multiple trips to the supermarket and buying the three items. A more optimized option is to purchase all three items in a single trip. Now instead of purchasing three items, if we had to purchase 100 items, we would have been forced to make multiple trips because it is not possible to carry that many items back home in a single trip. This explains intuitively why we cannot fuse the entire network into a single layer.

### 3.6.2 Precision Calibration

Deep Neural Networks are typically trained using 32-bit floating point numbers (FP32). However, during inference (in many cases), there is a possibility of using FP16 or INT8 precision (because no backpropagation is involved). Precision calibration is an optimization technique that can be enabled in the TensorRT compiler on Deep Neural Networks to quantize floating point numbers to a lower precision without compromising on the accuracy of the model. This optimization leads to not only a reduction in memory footprint but also low latency and higher throughput during inference. Moreover, there might be hardware accelerators that can perform operations on half (or mixed) precision matrices very quickly. For example, Tensor cores innovated by Nvidia are specialized to perform the operation  $A * B + C$ , where  $A, B$  are 4x4 matrices containing FP16 (or INT8) elements and  $C$  is a 4x4 matrix with FP32 (or INT32) elements.

Reducing the precision is possible because, typically, the weights and activations in a neural network are usually concentrated in a small range. For example, as shown in figure 3.4, although the range of FP32 is from  $-3.4e + 38$  to  $3.4e + 38$ , we observe that the activations and weights lie in the range of  $-6$  to  $+6$ . And if the number of unique entries in this range is less than 255, we can indeed map it to INT8 (which has a range of  $-127$  to  $+127$ ) without loss in any information. This is exactly what is demonstrated in figure 3.4. Intuitively, we can think of this quantization as encoding the same

information using a different set of symbols. For example, 2.76 will be mapped to 58 in the new (quantized) encoding scheme.

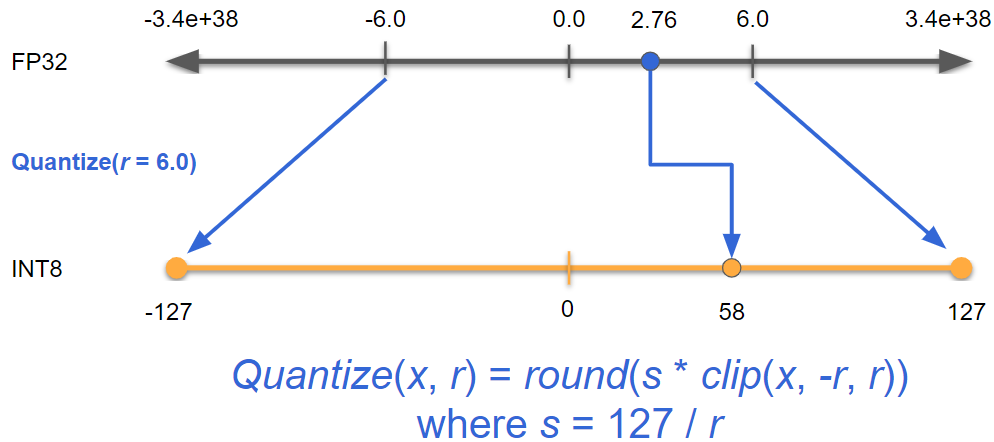


Figure 3.4: An illustration of how the TensorRT inference engine performs Quantisation optimization to reduce the precision of the weights and activations in a trained model without compromising on the performance.

In order to come up with this new quantized encoding scheme, TensorRT must get an idea of the values of weights and activations (produced by the network) so that it can accurately represent them with the precious 8 bits in INT8. This is achieved by feeding TensorRT with a mini dataset, often called the calibration dataset. This calibration dataset can be an extremely small subset of the original data, but it should be from the same distribution as the network would actually see. TensorRT would forward pass the images from the calibration dataset through the trained neural network and plot the activation values and the weights from the neural network on a graph as shown in Figure 3.4. It then decides the scaling factor ( $s$ ) and the clip value  $r$ . The clipping value ( $r$ ) is the threshold value above which values are clipped to  $r$ , resulting in saturation. Once the new encoding scheme is decided, the activation and weights can be plotted again using the new representation (encoding). TensorRT can now compute the information lost when it is trying to approximate the original distribution using the reference distribution with the help of the famous Kullback-Leibler (KL) Divergence[22] formula:

$$D_{KL}(Q, P) = \sum_i Q[i] \log \left( \frac{Q[i]}{P[i]} \right)$$

where  $Q$  is the probability distribution of the original encoding, i.e., FP32 in this case;  $P$  is the probability distribution of the new encoding scheme, i.e., INT8 in our case. If both the probability distributions are exactly the same, the information lost is 0 (this is because if  $Q[i] = P[i]$  then  $\log \left( \frac{Q[i]}{P[i]} \right) = \log(1) = 0$ ). TensorRT decides the new encoding in such a way that the information loss between the two probability distributions is minimized. If  $D_{KL}(Q, P) = 0$ , it means that quantization has no impact on the accuracy (which is the most desirable outcome).

In our use case, precision calibration was really easy as it was completely unsupervised. We just had to capture a few frames from the already placed cameras at the start of the game and feed them to TensorRT, and TensorRT would generate a highly optimized model within a few minutes, which was used for the entire match. This worked really well in our use case, and we observed zero drop in the accuracy of the quantized model because of the fact that we were interested in detecting very specific objects.

### **3.6.3 Kernel Autotuning**

Kernel Autotuning is an optimization technique done by the TensorRT framework where the TensorRT compiler chooses the most efficient implementation of a kernel (from hundreds of kernel implementations) depending on which Nvidia GPU is being used. For example, convolutions can be implemented in a variety of ways, but TensorRT chooses the best algorithm for computing convolutions depending on the hardware. Similarly, for matrix multiplication, TensorRT may try various loop unrolling factors and tiling sizes and choose the parameters such that the computation is most efficient on the specific GPU architecture.

The very first step of Kernel Autotuning is to generate the various types of kernels with different configurations and optimizations. As mentioned earlier, for matrix multiplication, various loop unrolling factors and tiling sizes would be generated. The next step, called profiling, executes the different generated kernels on the available GPU architecture and collects various metrics measuring the performance of each kernel (for example, the execution time, the memory footprint, etc.). The TensorRT compiler then analyzes the profile for each kernel and selects the most optimal one to be used in the model. It uses a variety of techniques like random search, Genetic algorithms, and Machine Learning techniques to perform autotuning.

## **3.7 Custom Data Collection and Annotation**

As mentioned earlier, we are dealing with a very different type of data distribution compared to what is used to train state-of-the-art detectors. Hence, we had to finetune existing detectors using training images from our dataset. We recorded multiple video sequences from Ranji Trophy and Tamil Nadu Premier League matches and curated a set of 400 training images which we used to finetune Yolov5.

The training images were chosen at random from video sequences of approximately an hour. The frame rate of the videos was around 30 FPS. We developed our own tool, which randomly chose a frame from these videos, ran the finetuned Yolov5 model on it, and showed the detections that this model detected. The annotator then had the option of removing any detection or adding any other detection that the (finetuned) model missed. These new annotations were further used to finetune the model, after which the newly tuned model was used to generate the detections for the next to-be-annotated frames. This helped speed up the process of annotation as the newly-tuned model performed better in



matching the ground truth detections compared to the old one. In other words, every time the model was finetuned, it gave better and better detections to the annotator making the job of the annotator easier with time. After finetuning a couple of times, we observed that the finetuned model was fairly accurate, and the annotator had to remove/add only a couple of hard detections. We found that it was most optimal to re-finetune the model after every ten frame annotations. This sped up things a lot, and a single annotator could annotate close to 300 frames in a couple of hours.

We have released the code for the tool we built here.

### **3.8 Improving accuracy: Finetuning the existing model**

We customized and finetuned the standard Yolo model. The train and test splits were specified in the appropriate YAML file. We initialized our model with the weights of the yolov5l model. For training, we used 3 GTX 1080Ti Nvidia GPUs. The final layer of the Yolo model had to be modified since we were changing the number of classes from 80 to 1. We also removed anchors with a horizontally elongated orientation because we were not interested in detecting objects with that kind of orientation. Further, all Yolo v5 models employ multi-scale support at strides of 8, 16, and 32. We disabled this multiscale support and used only the feature map corresponding to stride 8. We used a batch size of 15 images and trained the detector for 10 epochs. The images were resized to  $1088 \times 1088$  before passing them through the model. This meant the image was split into  $136 \times 136$  grid (since we were using the feature map, which downsamples the input image by a factor of 8). The default optimizer, Stochastic Gradient Descent, was used with an initial learning rate of 0.01. During the training process, the loss, precision, recall, and mAP metrics were monitored on the validation set, and the checkpoint which gave the best Mean Average Precision (mAP) score on the validation set was used for deployment. The commands to annotate and finetune the Yolov5 model are detailed in the README of the repository.

### **3.9 Training on synthetic data and testing on real data**

Collecting the training data was extremely challenging. It was really difficult to get permission from the authorities to access the stadium during domestic matches, and it was even harder to get their permission to set up cameras to record. COVID made things even more difficult. We were lucky to get a few sequences from a Ranji Trophy match and a TNPL match which was used for training the detector (as mentioned earlier). Moreover, even after collecting the data, annotation was a costly and time-consuming process. To overcome these challenges, we tried creating synthetic data to train our detector and deploy it on real images. We used the Unity 3d game engine to spawn PersonX [23] models (representing synthetic players) at random locations in this 3d model of a cricket stadium. We finetuned the Yolo detector with this synthetically generated data. When we evaluated the performance of this detector on real data, we found a substantial drop in the performance of the model. This was because the 3d models of the players and the stadium used were not realistic enough. Also, the lighting

conditions and the camera parameters were not realistic enough to get a photo-realistic rendering of the camera view, which could simulate real-world data distribution.



Figure 3.5: A sample synthetic camera view image we created using the Unity 3D game engine to train our detector model. We generated multiple sequences with different configurations by varying the number and position of cameras placed, the number and the 3D model of players spawned on the cricket ground. Each sequence typically contained around 200 frames.

### 3.10 End to End Deep Multi-view Detection Models

End-to-end deep learning methods have been shown to be immensely effective. MVDet [24] was the first method to propose a novel end-to-end learnable anchor-free architecture for multi-view multi-object detection. SHOT [25] and MVDeTr [26] are architectures that extended the MVDet idea. These anchor-free methods pushed the State of the Art by a huge margin by taking the Multi-Object Detection Accuracy from 74.1% to 90.2% on the Wildtrack dataset [4]. In fact, the MVDet paper proposed a synthetic dataset that it used to train the detector and then test on the real Wildtrack dataset. One issue with all these methods is that they do not generalize well. When we say generalize, we are saying generalization in three ways:

1. Varying number of cameras: MvDet, MVDeTr, and SHOT; all three of these methods do not generalize well when the number of cameras is changed. In other words, they overfit to a certain number of cameras. These architectures do not support training on some number of cameras and testing on different numbers of cameras. Generalization across a number of cameras is an important requirement when we want to deploy Multi-View Multi-Object detection models in

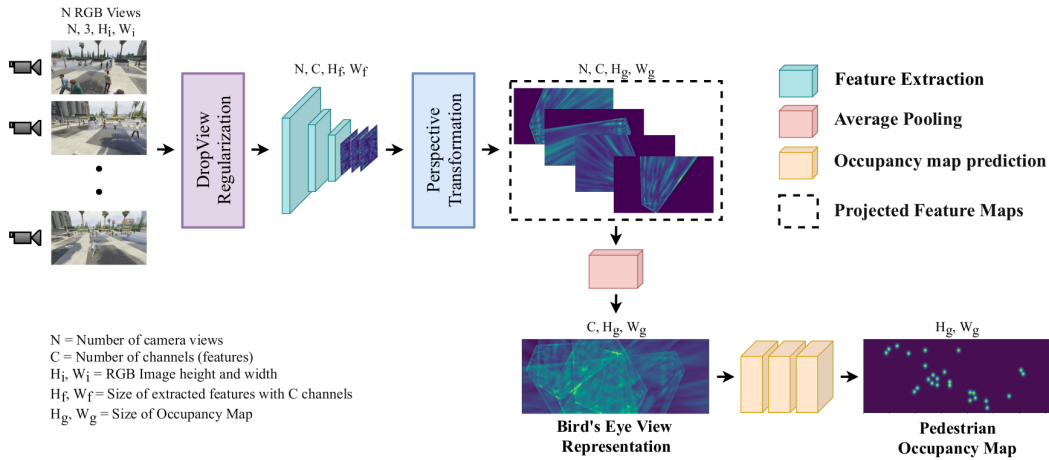


Figure 3.6: End-to-End Deep Multi-view detection: The proposed GMVD architecture takes in any number of camera view images and directly predicts the bird’s eye view occupancy map. The Perspective Transformation stage requires well-calibrated camera matrices to project the camera view feature maps to the top view.

the real world. It might so happen that one of the cameras stops working. In such cases, these methods would completely stop working as well, even though the feed is coming from the other cameras.

2. Varying camera configuration: All three of these anchor-free Multi-View Multi-Object detection methods do not generalize well to changing camera configurations. Even if the number of cameras remains constant between the train and test set, we find that these methods perform extremely poorly if either of the camera positions is slightly changed. In fact, just changing the order of cameras in the train set and test set makes these models fail.
3. Varying scenes: These models perform very poorly when trained on a certain scene and deployed on a different scene. In fact, they do not generalize even to changing weather conditions or slight deviations in the lighting of the scene on which it was trained.

Due to these limitations, although the state of the art in object detection was pushed by a huge margin, it was not usable in the real world. We propose a barebone architecture called GMVD [27] (short for Generalised Multi-View Detection), along with a feature-rich synthetically generated dataset to overcome these limitations. Our dataset can be used to benchmark the generalization performance of Multi-view detectors. We perform multiple experiments to demonstrate the invariant nature of our proposed model (trained on the proposed GMVD dataset) to changing number of cameras, changing scenes, and changing camera configurations.

The GMVD model architecture is shown in Figure 3.6. We pass each of the camera view images through a backbone network, typically ResNet-18 [10]. This extracts the features from each of the camera views. The extracted feature maps for each camera are then projected to the top view (also known as

the bird's eye view) using the homography matrix for that camera. The perspective-transformed feature maps from multiple camera views are then aggregated together using a simple average pooling operation. This way of aggregating the feature maps is what makes it highly generalizable compared to the prior methods. Finally, the aggregated top-view feature maps are passed through a series of convolution layers to predict the occupancy map in the top-view.

We did not use this architecture for our use case because it was slow due to the usage of deep neural networks. The GMVD model, after a number of optimizations using the TensorRT framework, could achieve an inference speed of 14.2 frames per second on Full-HD images. Hence, this did not meet our real-time requirements of achieving inference speeds of 30+ FPS.

## Chapter 4

### Merging Three Top-View Maps into the Final Top-View Map

Once we have calibrated a camera, we can create the bird's eye view of the players visible in that camera. However, in our setup, we have three cameras, and an algorithm needs to be designed to combine the information from the three individual cameras' top-view maps and merge them into a final top-view map. Merging multiple top-view maps is a very challenging problem, primarily due to the fact that the same player in multiple top-views may not occupy the exact same pixel location. This is primarily due to inaccurate calibrations and, more importantly, due to the fact that we are projecting the bottom center point of detections. The bottom center point of detections of a player from different views does not necessarily indicate the same 3D point in the world.

Hence, our algorithm needs to identify the detections which belong to the same player from more than one top-view map and handle them accordingly while generating the output so that the final output produced does not contain duplicate identities. Similarly, there may be players who are visible only in one top view, and we should be able to identify them correctly so that we don't miss them while generating the final output. A good merging algorithm should be able to reason out whether a player detection is unique to a view or is present in one or more other views and associate accordingly.

The merging algorithm we present in this chapter is extremely robust to inaccurate calibrations and is able to associate detections across the three top-view maps with near-perfect accuracy when provided with the exact number of unique identities in the final output map. For example, in cricket, if all the players are detected at least in one view, then there are 15 unique identities in the final top-view output map.

#### 4.1 Modelling as a Graph Optimisation Problem

We model this data association problem across three top views as a graph. Detections in each bird's eye view are represented as nodes of the graph. If the number of nodes (detections) present in a top-view map is less than the given unique number of identities present in the combined final output( $N$ ), then we add 'hidden' nodes to that view. As a result, each view has exactly  $N$  nodes.

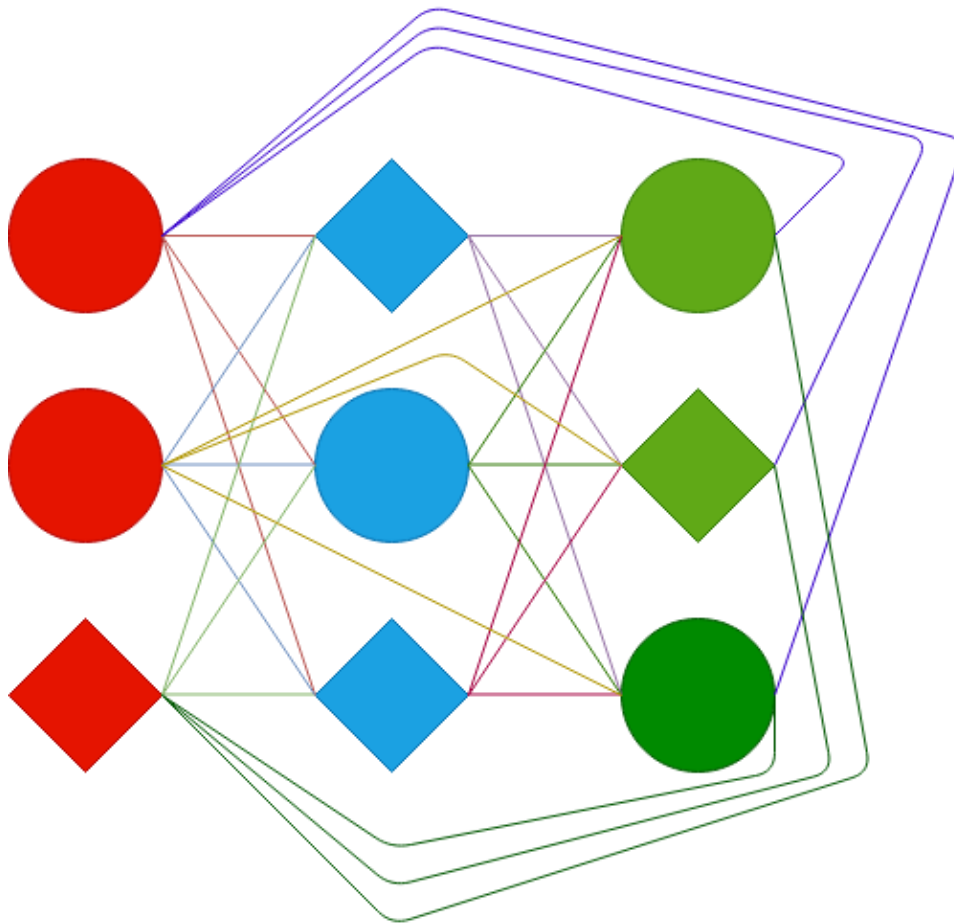


Figure 4.1: The above is a sample graph constructed where the number of unique objects when all three views are merged is 3. The three distinct colors represent nodes in the three different views. The diamonds represent hidden nodes, and the circles are the normal nodes corresponding to detections in a top view. The lines represent edges that connect every pair of nodes except pairs where both the nodes belong to the same view.

There exists undirected edges connecting every pair of nodes except pairs where both the nodes of the pair belong to the same top-view map. Each edge has a cost associated with it. This cost is simply the Euclidean distance separating the 2D detection points corresponding to the nodes being connected by this edge. If an edge connects a pair of nodes where at least one node is a hidden node, then the cost of that edge is 0.

The cost of an edge connecting a pair of nodes determines the penalty that would incur on associating both nodes in this pair. Higher cost means a greater penalty will be incurred in linking the two nodes. Lower cost means a lesser penalty will be incurred in linking the two nodes. Our goal is to minimize the overall cost of linking these nodes subject to certain constraints, and this optimization problem is the subject of the next section. A sample graph construction is shown in Figure 4.1.

**Note:** If two nodes are linked in the final solution, then it implies that the detections corresponding to the linked nodes refer to the same player identity. If two nodes are not linked in the final solution, then it implies that the detections corresponding to the unlinked nodes do not refer to the same player identity.

## 4.2 Solving the problem as a Linear Programming Optimisation

We formulate the above Optimization problem as an Integer Programming problem. Let us denote the set of edges connecting nodes of view 1 and view 2 with  $E_1$ . Similarly, let us denote the set of edges connecting nodes of view 2 and view 3 with  $E_2$  and the set of edges connecting nodes of view 1 and view 3 with  $E_3$ . Let  $C_1(e_1)$  denote the cost of edge  $e_1$  in  $E_1$ . Let  $C_2(e_2)$  denote the cost of edge  $e_2$  in  $E_2$ . Let  $C_3(e_3)$  denote the cost of edge  $e_3$  in  $E_3$ .

The following is the objective of the optimization problem:

$$\min \sum_{e_1 \in E_1} C_1(e_1) \cdot x_{e_1} + \sum_{e_2 \in E_2} C_2(e_2) \cdot y_{e_2} + \sum_{e_3 \in E_3} C_3(e_3) \cdot z_{e_3} \quad (4.1)$$

where  $x_{e_1}, y_{e_2}, z_{e_3} \in \{0, 1\}, \forall e_1 \in E_1, \forall e_2 \in E_2$  and  $\forall e_3 \in E_3$ .

$x_{e_1}$  is 1 if the pair of nodes connected by  $e_1$  in  $E_1$  is linked and 0 otherwise.  $y_{e_2}$  is 1 if the pair of nodes connected by  $e_2$  in  $E_2$  is linked and 0 otherwise.  $z_{e_3}$  is 1 if the pair of nodes connected by  $e_3$  in  $E_3$  is linked and 0 otherwise.

The structure of the problem demands certain constraints to be enforced while solving the optimization problem. Each node in a view has to link to exactly one other node in the other two views. Even though a detection may be unique to the view, note we added hidden nodes to take care of this situation. No node can be linked to two or more nodes in another view.

Additionally, to ensure consistency, we introduce a constraint that says that if node  $i$  in view 1 is linked with node  $j$  in view 2 and node  $j$  in view 2 is linked with node  $k$  in view 3, then node  $i$  in view 1 has to link with node  $k$  in view 3. The accuracy of association improved drastically upon adding this constraint.

Hence, the optimization problem needs to be solved subject to the following constraints:

$$\sum_{e_1 \in E_1; v \in e_1} x_{e_1} = 1 \quad (4.2)$$

for all nodes( $v$ ) that belong to view 1. Similarly,

$$\sum_{e_2 \in E_2; v \in e_2} y_{e_2} = 1 \quad (4.3)$$

for all nodes( $v$ ) that belong to view 2. And,

$$\sum_{e_3 \in E_3; v \in e_3} z_{e_3} = 1 \quad (4.4)$$

for all nodes( $v$ ) that belong to view 3.

Finally, to enforce the consistency as described above; for every edge  $e_3$  connecting node  $i$  in view 1 with node  $k$  in view 3, the following constraint is added:

$$x_{e_1} + y_{e_2} \geq 2 * z_{e_3} - m_j * B \quad (4.5)$$

$$\sum_{l=1}^{l=N} m_l \leq (N - 1) \quad (4.6)$$

$\forall e_3, j | e_3 \in E_3, 1 \leq j \leq N, j \in e_1, j \in e_2. m_i$  can only take the values 0 or 1.

**Note: B is a large number(say 1000)**

The interpretation of the above constraint is as follows: For every edge  $e_3$  connecting node  $i$  in view 1 with node  $k$  in view 3, we want at least one node  $j$  in view 2 such that the edge connecting node  $i$  to node  $j$  is selected and the edge connecting node  $j$  to node  $k$  is also selected.

Let us try to understand the constraints in 4.5 and 4.6 better. When an edge  $e_3$  is selected, connecting node  $i$  in view 1 to node  $k$  in view 3, the variable  $z_{e_3}$  takes the value 1 in 4.5. Now, when  $m_j$  takes the value 1, the constraint in 4.5 is trivially satisfied and does not depend on values of  $x_{e_1}$  and  $y_{e_2}$ . The optimization problem would have set all the  $m_{j_s}$  to 1 had the constraint in 4.6 not been present. The constraint in 4.6 ensures that at least one node  $j$  is present where  $m_j$  is set to 0, and the constraint in 4.5 is not trivially satisfied. Hence, when  $m_j$  is 0 (and  $z_{e_3}$  is 1), the constraint in 4.5 reads as  $x_{e_1} + y_{e_2} \geq 2$ . This can only be satisfied when both  $x_{e_1}$  and  $y_{e_2}$  are set. And, since  $i, j \in e_1$  and  $j, k \in e_2$ , we have a node  $j$  (in view 2), such that when node  $i$  in view 1 is linked with node  $k$  in view 3, then, node  $i$  in view 1 is linked with node  $j$  in view 2 and node  $j$  in view 2 is linked with node  $k$  in view 3, thereby enforcing consistency.

The above problem is a Mixed Integer Programming problem and hence NP-Hard. In order to solve the problem efficiently, we used the Gurobi[28] solver. This solver could find exact solutions within a second or two on a standard CPU.



## Chapter 5

### Tracking

#### 5.1 Introduction

Object Tracking is a fundamental problem in the field of Computer Vision wherein the system not only localizes one or more objects in the scene by drawing bounding boxes around them but also assigns an identity to them. In other words, tracking helps us identify where a particular object at time instant  $t$  goes in the scene at time instant  $t + \alpha$ . This is a very challenging problem because we need to establish correspondence between objects in subsequent frames. It becomes even more challenging when an object visible in the current frame becomes occluded in the subsequent frames. In such situations, we have to guess its location based on past trajectory. Some object trackers use the appearance information of detected objects (Re-Identification Networks[29][30][31]) to establish this correspondence. However, ReId networks are computationally expensive and extremely slow to run. Also, they are unable to distinguish objects which are similar looking. For example, in our use case, the players on the field wearing the same jersey color appear really small for ReId networks to identify which player is which in subsequent frames. Some object trackers rely heavily on the past trajectory motion of a tracked object. Using the past motion, these trackers try to predict where a particular object goes in the next frame, and when new detections come for the next frame, they try to associate one of the detections, which is very close to the predicted location. Some object trackers combine both motion and appearance cues to track objects. Tracking has numerous applications, some of them being surveillance, robotics, autonomous driving, and Human-Computer Interaction. In Cricket, tracking finds numerous applications because once we track a player or the ball, we can compute a lot of useful information like how much distance a player ran throughout the match, how many runs did he give away, how many runs did he score, where did he field, what was the fastest speed he ran at and many more. In this chapter, we look at how the problem of tracking was solved in our use case. Ours was a use-case where we could not rely on appearance features for tracking because all players wore the same color jersey. Moreover, we required a tracker which could do inference at realtime speeds. We present details of how we tailored a custom tracker for our use case, which ran at realtime speeds as well as tracked players with a high accuracy.

## 5.2 Why is Tracking necessary?

Since we are creating a top-view map, it seems like only detections are enough to construct it, and there is no need for tracking. However, tracking is necessary in our use case because of a couple of reasons. Our system is realtime, and hence the map should update instantaneously as players in the field move. Therefore, when a bowler is taking a run-up, we should be able to see this on the top view clearly. It turns out that simply projecting the detections to the top-view results in a non-smooth trajectory. A moving average filter is necessary to smoothen the trajectory. In order to apply this moving average filter, we need to know where a particular player was located at previous time instants. This requires tracking because we are essentially trying to retrieve where a particular player was located across various time instants. Once we have this info, we can simply take an average of all these values and get the current location. Plotting the location at the current time instant in this way for every player would result in a smooth trajectory, and there wouldn't be any jumps in trajectory between successive timestamps.

The other and more important reason why we require the identity information is that we need to eliminate the umpires and the batters from the top view map. In other words, we would only want to show the location of players from the fielding team on the top-view map. Doing this would require us to know in each frame which detections are of batters and umpires and which detections are for the remaining fielders. We thought of using the average color information from detections; however, in Cricket matches, there is no guarantee that both the teams and the umpires would all wear jerseys of contrasting colors. This is an even bigger problem during test matches when everyone is wearing white.

Using positional cues does not help much either because a fielder may be standing very close to the leg-side umpire. Similarly, the umpire standing near the stumps can be confused at times with the bowler because, during the run-up, the bowler comes very close to this umpire. Moreover, the positions of the umpires and batters are constantly changing. Hence, only position cues are not sufficient to identify the umpires/batters.

The only feasible way to figure out who are the umpires and the batters is by marking them at a particular time instant and then relying on the tracking algorithm to hold onto the assigned identity. The better the tracking algorithm the longer would be the duration we will be holding onto the correct identity. As long as the correct identity is held onto, for each frame, we would know correctly which detections are those of the umpires and which are those of the batters. If there is an identity swap in one of the frames, then a wrong identity would be assigned to the detection(s), and we might start to wrongly identify the batters/umpires. Especially when an umpire/batter is swapped identity with one of the fielders, it is extremely expensive because it would result in an undesirable outcome wherein the umpire/batter would be started to be shown on the generated top-view map. We should try to avoid these swaps as much as possible, if not completely. In case such swaps occur, we need to reinitialize the tracking system and mark the umpires and batters again.

### 5.3 Different Tracking Metrics

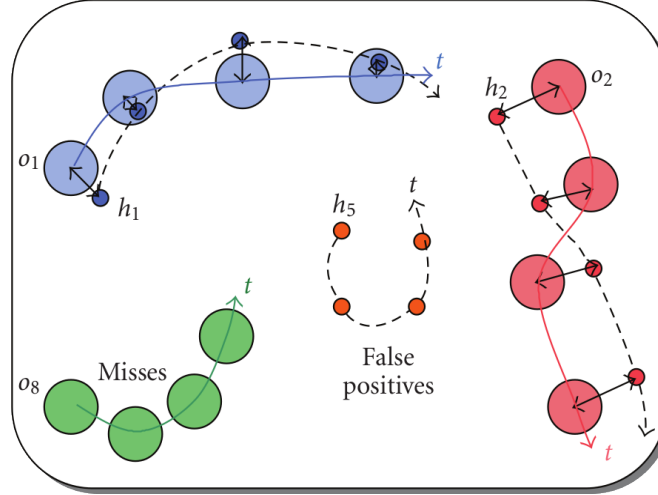


Figure 5.1: Figure illustrating true positive, false positive, and false negative detections during the course of object tracking. The large circles represent the ground truth object detections, while the small circles denote the predicted object detections. The same color shapes denote that the detections belong to the same identity. The arrows indicate the correspondences between the observed and the predicted detections.[1]

Some of the detection metrics like the Intersection Over Union (IoU), False Positives (FP), True Positives (TP), and False Negatives (FN) defined in 3.3 are relevant for tracking as well.

1. Identity Switches (IDSw): This metric quantifies the number of times the identity of an object changes/switches during the course of tracking. In other words, during the tracking process, if a ground truth track gets assigned  $n$  different identities assigned by the tracking algorithm, then the number of identity switches for that track is  $n - 1$ .
2. Multi-Object Tracking Precision (MOTP): This metric measures how precisely the tracking algorithm tracks objects. It measures how well a tracked object aligns with respect to its corresponding ground truth or reference data. It is equal to the average IoU score over the set of True Positives (TPs). Mathematically,  $MOTP = \frac{1}{|TP|} \sum_{TP} IoU$ .
3. Multi-Object Tracking Accuracy (MOTA): Multi-Object Tracking Accuracy tries to measure the overall accuracy of the detector by taking into account the number of mistakes, i.e., number of False Positives, Number of False Negatives and Number of Identity Switches in the predictions. Mathematically,

$$MOTA = 1 - \frac{|FP| + |FN| + |IDSw|}{\text{Number of Ground Truth Detections}} \quad (5.1)$$

4. Identification F1 (IDF1) score[32]: IDF1 calculates a one-to-one mapping from the set of predicted trajectories to the set of ground truth trajectories. Identity True Positives (IDTPs) are

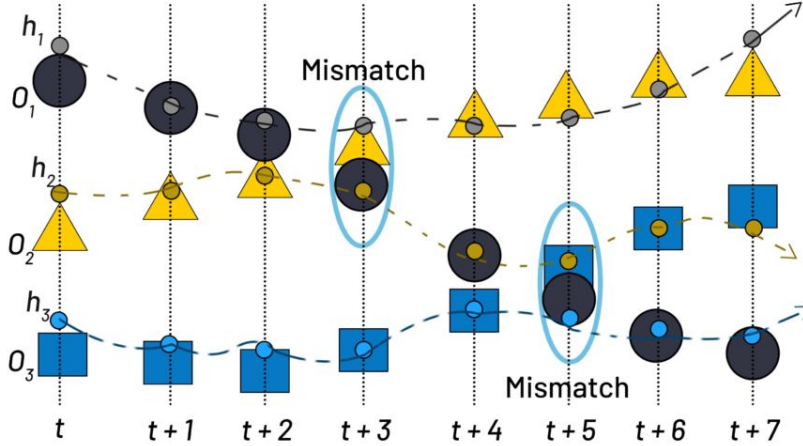


Figure 5.2: Figure demonstrating identity switches during the course of object tracking. Identity switches are marked with blue ellipses with the label 'Mismatch' above them. The large shapes represent the ground truth object detections, while the small circles denote the predicted object detections. Object detections that belong to the same identity are denoted with the same color of the shapes.[1]

matched detections from the overlapping regions of the predicted and the ground truth trajectories. IDFPs (Identity False Positives) are detections that remain in the predicted trajectories from the non-overlapping regions of the matched trajectories. IDFNs (Identity False Negatives) are detections which are remaining in the ground truth trajectories from the non-overlapping regions of the matched trajectories. We must note that detections from unmatched predicted and ground truth trajectories also contribute to IDFPs and IDFNs, respectively. Mathematically:

$$IDF1 = \frac{|IDTP|}{|IDTP| + 0.5 * |IDFP| + 0.5 * |IDFN|} \quad (5.2)$$

The one-to-one mapping between the predicted and ground truth trajectories is computed using the Hungarian Algorithm[33] so that the IDF1 score is maximized.

5. Mostly Tracked (MT): An object is said to be Mostly Tracked in a video sequence if it has been correctly identified and localized for at least 80% of the time of its presence in the video.
6. Mostly Lost (ML): An object is said to be Mostly Lost in a video sequence if it has been correctly identified and localized for less than 20% of the time of its presence in the video.
7. True Positive Association (TPA)[2]: True Positive Association set for a given True Positive  $c$  is the set of True Positives which have the same ground truth ID and the predicted ID as TP  $c$ .

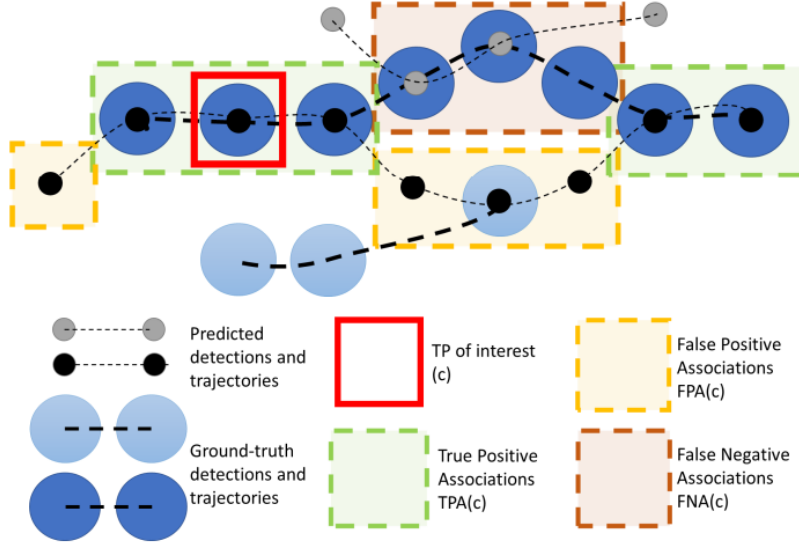


Figure 5.3: Figure illustrating the False Positive Associations (FPAs), False Negative Associations (FNAs), and True Positive Associations (TPAs) for a true positive (TP) of interest[2].

8. False Negative Association (FNA)[2]: False Negative Association set for a given True Positive  $c$  is the set of ground truth detections which have the same ground truth ID as TP  $c$ , but were either assigned a different predicted ID or were not assigned any predicted ID at all.
9. False Positive Association (FPA)[2]: False Positive Association set for a given True Positive  $c$  is the set of predicted detections that have the same predicted ID as TP  $c$  but were either assigned a different ground truth ID or does not correspond to any ground truth object.
10. Higher Order Tracking Accuracy (HOTA)[2]: HOTA provides a unified metric to measure both the detection and the association accuracy together. While the Detection Accuracy (DetA) score measures how accurately an object was localized, Association Accuracy (AssA) score measures how well the detected objects were associated with the same identity over time. Mathematically Detection Accuracy is defined as follows,

$$\text{DetA}_\alpha = \frac{|TP|}{|TP| + |FP| + |FN|} \quad (5.3)$$

Mathematically, Association Accuracy is defined as follows,

$$\text{AssA}_\alpha(c) = \frac{1}{|TP|} \sum_{c \in \{TP\}} A(c)$$

$$A(c) = \frac{|TPA(c)|}{|TPA(c)| + |FPA(c)| + |FNA(c)|}$$

Higher Order Tracking Accuracy can be mathematically written as below,

$$\text{HOTA}_\alpha = \sqrt{\text{DetA}_\alpha * \text{AssA}_\alpha} = \sqrt{\frac{\sum_{c \in \{TP\}} A(c)}{|TP| + |FP| + |FN|}} \quad (5.4)$$

## 5.4 State of the Art Trackers: A Review

1. Simple Online and Realtime Tracking (SORT)[34]: SORT is a very simple and bare-bone tracker implementation that can run at realtime speeds of 250 frames per second. It uses the tracking-by-detection paradigm to track objects in an online fashion, i.e., it uses only the current and the previous frame to track objects. The tracker has the following major components in its pipeline:
  - (a) An **object detector** like Faster RCNN [15] which detects the objects in the frames and passes on these detections to the main tracker. We must note that the performance of SORT is largely based on the performance of the detector used. Hence, this component is a crucial component in determining the performance of the SORT tracker in real-world scenarios.
  - (b) A state estimation algorithm like **Kalman Filters**[35] which estimates the current location of each object from the previous frame. Kalman Filters is extremely robust to noisy measurements, and it optimally predicts the true state of an object by combining the noisy measurements and motion model.
  - (c) An association algorithm like the **Hungarian Algorithm** [33] (described in section 5.5.4.3) which associates the existing tracks (having the predicted detections from the previous frame) with the detections in the current frame. This step assigns identities to the current detections. A good association step is critical to a tracker maintaining correct identities and minimizing swaps throughout the tracking process. In SORT, the assignment cost matrix is simply the Intersection over Union value of the corresponding bounding boxes.
  - (d) A **track management algorithm**, which updates the existing tracks with the newly associated detections. The track management algorithm is also responsible for heuristically determining the creation and deletion of tracks (when detections go unmatched).
2. Deep SORT[36]: The Deep SORT tracker extends SORT by integrating the appearance features of an object into the tracking pipeline. As a result, Deep SORT is able to better track objects which occlude for a long period of time and then reappear. All the steps in Deep Sort are exactly the same as SORT presented above, except for the additional computation of the appearance vector for each object detection and the assignment cost matrix computation in the association step. In Deep SORT, the assignment cost matrix is computed by taking the weighted sum of the appearance distance[37] and the Mahalanobis distance (of the Kalman states) between the predicted detections of the existing tracks and the new (to-be-associated) detections.
3. ByteTrack[38]: Most of the tracking algorithms consider only the high confidence (above a certain threshold) object detections as plausible candidates for the association step. This, however, may not be true say for an object which is partially occluded. Partially occluded objects may have significantly low confidence, and throwing away such detections leads to track fragmentations. ByteTrack extends SORT and Deep SORT trackers by keeping all the detections, including

Detections used	MODA	IDF1	MOTA	MT	ML	FP	FN	IDS <sub>w</sub>
Ground Truth Detections	100%	94.4%	99.1%	41	0	0	0	9
GMVD Model[27]	80.1%	81.3%	72.4%	29	2	124	116	23
MVDet Model[24]	88.2%	85.4%	86.2%	34	2	62	48	21
MVDeTr Model[26]	91.5%	90.3%	91.0%	36	2	33	43	10

Table 5.1: Table showing the tracking performance of SORT on the Wildtrack dataset [4] using detections from various models. These results show how heavily the detection metric, MODA influences the tracking metrics, IDF1 and MOTA. Improving the MODA score directly improves the tracking performance by a similar margin.

the low-confidence ones, for the association step. To this end, it classifies detections as low-confidence and high-confidence ones. The high-confidence detections are first associated with the predicted detections of the existing tracks exactly like it is done in SORT/Deep-SORT. It then matches the low-confidence detections with the predicted detections of the remaining existing tracks, which were not matched with the high-confidence detections (in the first association step). Using these low-confidence detections, often, true objects are retrieved while filtering out background objects.

4. Observation-Centric SORT[39]: Kalman filters assume that the object being tracked has linear motion. Although this assumption is fine for tracking objects which become occluded for small instants of time, it breaks if objects are occluded for a slightly longer duration of time. Since Kalman filters use the priori state estimate to come up with the posteriori state, errors can keep on accumulating during the period of occlusion. This leads to divergence of the predicted and actual trajectory of an object. OC-SORT shows how SORT can achieve state-of-the-art performance if the accumulated errors arising from the occlusion period are corrected in the corresponding Kalman Filter state by interpolating a virtual trajectory from the time of last seen to the time of re-association of the object.

## 5.5 Our Proposed Algorithm: MergeAndTrack

Table 5.1 shows how heavily the quality of tracking by SORT is influenced by the quality of detections. Our detection merging algorithm in the top-view (presented in Chapter 4), when combined with Simple Online Realtime Tracking, gives nearly perfect tracking results even when the calibrations are extremely noisy. However, such a method is extremely slow. The rate-limiting step in the pipeline is the merging step. Although the tracker runs at 250 frames per second(FPS), the merging step runs at barely 0.5 – 1 FPS. In this chapter, we modify the entire pipeline so that we not only achieve (nearly) perfect tracking results but are also able to run the entire end-to-end pipeline at approximately 24 FPS.

The main motivation behind the idea of the algorithm we propose is doing the expensive merging step as rarely as possible, only once in a while when things can go wrong. At other times, associate the

current detections from each individual camera to the detections in the already existing merged top-view map. Once the associations are done, we update the existing merged top-view map with the values of the newly associated detections.

All the tracking happens only on the top view using the projected 2D points. Hence, this tracking method is purely based on motion and does not use the appearance features of players at all. The deployment of our system in test cricket matches was another reason why we were motivated not to use appearance cues at all in the tracking algorithm. The following are the main components of our tracking algorithm:

- Detection of players and projection of players to the top view.
- Merging the three top views into a single combined view (done only once at the start and from then onwards as rarely as possible).
- Estimating the future (next frame) location of existing players in the top-view using Kalman Filters.
- Associating the new detections from each individual camera's top-view with the existing top-view tracks.
- Creation and deletion of tracks

The details of each of the mentioned components are presented in the subsequent sub-sections.

### **5.5.1 Detection of Players and Projection of Detected Players to Top-View**

Our tracking algorithm follows the tracking-by-detection paradigm, where tracking follows after the detection step. In order to detect players from each individual camera, we run the Yolo algorithm optimized using the TensorRT framework in C++ (details of which were covered in Chapter 3). We get realtime speeds of 25 FPS while running the detector on 4K images with this optimization. Since each camera feed is processed on a separate GPU, we do not suffer a drop in speed with multiple cameras.

Once the players are detected in a camera view, we take the bottom center point of each of the detected player's bounding boxes and project them to the top view using the computed homography matrix for that camera view (details of which were covered in Chapter 2). Then, we get 3 top-view maps, one for each camera, indicating the player locations as 2D points.

### **5.5.2 Merging the three top-views into a single combined top-view**

This is the slowest step in the pipeline running at around 0.5 – 1 FPS. As a result, we would want to do this step as rarely as possible. Possibly, we would want to do this step once at the very beginning when we are starting to track. The other times we may want to do this step is possibly after an over change or after a wicket has fallen when there is sufficient clutter for our proposed algorithm to fail.



Also, during those times, additional players can come into the field bringing drinks. It is always safe to have a new merged map after such events to be sure that the tracking follows perfectly. The details of this step were covered in Chapter 4.

### 5.5.3 Estimating the future (next frame) location of existing players in the top-view using Kalman Filters

Once we have the single merged top view, we can proceed with the tracking from the subsequent frames. At each time instant, detections from the bird's eye view of each camera (which are 2D points) need to be associated with the already existing tracks in the existing (merged) top-view map (from the previous time instant). In order to improve the quality of the associations, the tracks from the previous time instant must be extrapolated to the current time instant using motion models. If the propagation of an existing target's track is not done to the next (current) frame, then the association of the new detections with the old tracks does not work out well. We use a simple linear motion model known as the Kalman Filter to predict the new location of existing tracks. Kalman Filters takes into account the velocity of a player while predicting the new location. This greatly helps in the data association step.

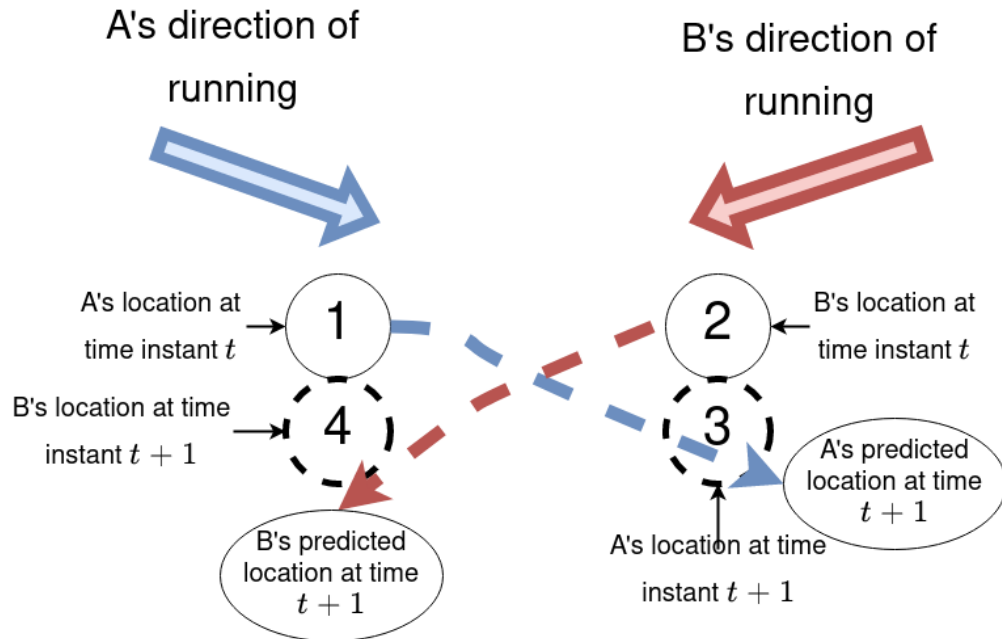


Figure 5.4: An illustration showing a motion model predicting the locations of players  $A$  and  $B$  at time instant  $t + 1$  by taking into consideration the velocities and positions of  $A$  and  $B$  at time instant  $t$ .

In order to intuitively understand the role a motion model plays in the data association step, let us imagine two players whose identities are  $A$  and  $B$  (represented as 2D points) crossing each other.  $A$  is running in the direction indicated by the blue arrow, and  $B$  is running in the direction indicated by the red arrow, as shown in Figure 5.4. At time instant  $t$ ,  $A$  is located at 1 and  $B$  is located at 2. At time

instant  $t + 1$ ,  $A$  is located at 3 and  $B$  is located at 4. This is a classic example of a crossover. Now, if we were to associate the tracks at time instant  $t$  with the new detections at time instant  $t + 1$  without using a motion model, clearly  $A$  would have been associated with the detection at 4 and  $B$  would have been associated with the detection at 3. This is an incorrect association and indicates an identity switch in the tracking.

However, when a motion model that takes the velocity of tracks into account is used to extrapolate the tracks to time instant  $t + 1$ , we get the following results:

1.  $A$ 's track is extrapolated to a location somewhere very close to 3
2.  $B$ 's track is extrapolated to a location somewhere very close to 4

When these propagated tracks are used in the association step, the association takes place correctly, i.e.,  $A$ 's track will be associated with the detection at 3, and  $B$ 's track will be associated with the detection at 4.

In the Kalman Filter algorithm, we keep a record of the following states for an object's track, which helps us do the extrapolation of tracks to future time instants:

- $x$  indicating the horizontal pixel location of the object.
- $y$  indicating the vertical pixel location of the object.
- $\dot{u}$  indicating the velocity of the object along the X-axis.
- $\dot{v}$  indicating the velocity of the object along the Y-axis.

#### 5.5.4 Associating top-view detections from each camera with the existing top-view tracks

Once we predict the location of each fielder at time instant  $t$  (using the merged top-view map at time instant  $t - 1$ ), we have to associate (match) the points in this predicted top-view map with the original top-view map (of each camera) at time instant  $t$ . This association step allows us to assign identities to the detections at time instant  $t$ , and it is a critical step in the tracking pipeline. Incorrect associations would lead to undesirable identity swaps. Figure 5.5 shows this process graphically for a single camera. The exact process is repeated for multiple cameras in our pipeline. Once the identities are transferred to the top-view map of each camera (at time instant  $t$ ), we can update the merged top-view map by taking the average value of coordinates for each identity across all the cameras.

In the remainder of this subsection, we will present more details on how the associations were done using the Hungarian algorithm.

##### 5.5.4.1 Modelling the association problem as a Graph Optimisation Problem

We model the above problem as a Graph optimization problem. The graph consists of two sets of nodes/vertices, where each node represents a top-view detection (2D point). One set contains points

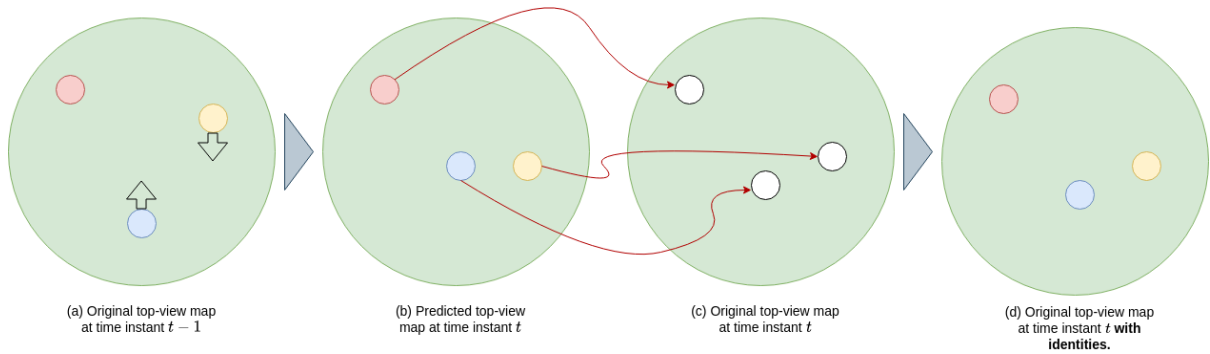


Figure 5.5: Graphical illustration of the data association step in the top-view assigning identities to incoming detections (from a particular camera) at time instant  $t$  using the consolidated information of top-view tracks at time instant  $t - 1$ .

from the predicted top-view map, and the other set contains points from the original top-view map of a particular camera. Note that we model and solve the optimization problem for each camera individually and then combine the results to update the merged top-view map (as mentioned earlier).

An edge connects every vertex in set 1 to every other vertex in set 2. There exists no edge between vertices of the same set. Such an undirected graph can be formally identified as a complete bipartite graph in Graph Theory Literature. Formally,

**Definition 1.** An undirected graph  $G = (V, E)$  is **bipartite** if the vertices in set  $V$  can be partitioned into two disjoint sets  $V_1$  and  $V_2$  such that for every edge  $(u, v) \in E$ ,  $u \in V_1$  and  $v \in V_2$ .

**Definition 2.** A **complete bipartite graph**  $G = (V_1, V_2, E)$  is a bipartite graph where every vertex in  $V_1$  is connected to every other vertex in  $V_2$  i.e  $\forall v_1 \in V_1$  and  $\forall v_2 \in V_2, \exists (u, v) \in E$ .

We can assume that the number of vertices in  $V_1$  is equal to the number of vertices in  $V_2$ . If not, we can add **dummy nodes** in the set with the lesser number of nodes.

There is a cost associated with each edge in the bipartite graph. In our use case, this cost is equal to the Euclidean distance between the two points connected. If a node is connected to a **dummy node**, the cost associated with that edge is 0. The problem we are trying to solve is to match/associate every vertex in  $V_1$  with a vertex in  $V_2$  by selecting edges such that every vertex is incident on exactly one edge and the total cost incurred on selecting the edges is as small as possible. This can be identified as finding a perfect matching with minimal cost. Formally,

**Definition 3.** A **matching** in a graph  $G = (V, E)$  is a subset  $M \subseteq E$  such that  $\forall v \in V$ , there is at most one edge in  $M$  which is incident on  $v$ .

**Definition 4.** A matching  $M$  in a graph  $G = (V, E)$  is said to be a **perfect matching** if  $\forall v \in V$ ; there is exactly one edge in  $M$  which is incident on  $v$ .

We transform the problem of finding a perfect matching with minimum cost to a problem of finding a perfect matching with maximum cost by setting the edge weights as follows:  $w(u, v) = M - w(u, v)$ , where  $M = \max_{(u,v) \in E} w(u, v)$ .

#### 5.5.4.2 Integer Programming Formulation

Let us introduce a variable  $x_{ij} \in [0, 1]$  such that  $(i, j) \in E$ ,  $i \in V_1$  and  $j \in V_2$ . It denotes whether the edge connecting vertex  $i$  to vertex  $j$  has been selected in the matching  $M$  or not.  $x_{ij} = 0$  denotes that the edge has not been included in the matching, whereas  $x_{ij} = 1$  denotes that the edge has been selected in the matching.

As discussed earlier, we want to select a perfect matching  $M$  such that the cost of the matching is maximized. Hence, the objective of our optimization problem is as follows:

$$\text{maximise } \sum_{(i,j) \in E} x_{ij} * w_{ij} \quad (5.5)$$

The problem is solved under the following constraints:

$$\begin{aligned} x_{ij} &\in [0, 1], \forall (i, j) \in E \text{ such that } i \in V_1 \text{ and } j \in V_2 \\ \sum_{j \in V_2} x_{ij} &= 1, \forall i \in V_1 \\ \sum_{i \in V_1} x_{ij} &= 1, \forall j \in V_2 \end{aligned}$$

We can use a solver like Gurobi[28] to solve the above Integer Programming problem. However, Integer Linear Programming being an NP-hard problem, is computationally expensive. Hence, we cannot use this approach in our pipeline because association is something we would want to do as many as 60 times in one second to run our tracking pipeline at realtime speeds.

#### 5.5.4.3 Hungarian Algorithm

The Hungarian Algorithm [33] can be used to find the maximum weighted matching in bipartite graphs at very fast speeds. It has a runtime complexity of  $O(N^3)$ , where  $N$  is the number of vertices in the graph. This algorithm is also known as the Kuhn-Munkres algorithm.

Before presenting the details of the Hungarian algorithm, we will define a few terms and prove some lemmas that would be useful for understanding the algorithm.

**Definition 5.** A vertex in a graph  $G = (V, E)$  is said to be **matched** if it is incident on an edge  $e \in M$ , where  $M \subseteq E$ . It is said to be **unmatched** or **free** if it is not incident on any edge in matching  $M$ .

**Definition 6.** An **alternating path** in a graph  $G = (V, E)$  is a path such that the edges alternate between  $M$  and  $E - M$ .

**Definition 7.** An *augmenting path* in a graph  $G = (V, E)$  is an alternating path starting and ending with a free/unmatched vertex.

**Definition 8.** An *alternating tree* in a graph  $G = (V, E)$  is a tree rooted at some free vertex  $u \in V$  such that all the paths starting at  $u$  are alternating.

**Definition 9.** A *vertex labeling* is a function  $l : V \rightarrow R$ . It assigns a label to each vertex  $v \in V$ .

**Definition 10.** A *feasible labeling* is a vertex labeling satisfying the condition:  $l(x) + l(y) \geq w(x, y), \forall x \in V_1, y \in V_2$ .

**Definition 11.** An *Equality Subgraph* (with respect to a labelling  $l$ ) is a subgraph  $G_l = (V, E_l)$  of a graph  $G = (V, E)$ , such that  $l(x) + l(y) = w(x, y), \forall (x, y) \in E_l$ . In other words, an equality subgraph  $G_l$  contains only the perfectly feasible edges within graph  $G$ .

**Definition 12.** A *vertex neighborhood* for a vertex  $v \in V$  in a graph  $G = (V, E)$  can be defined as a set of vertices  $S(v) \subseteq V$  wherein  $S(v) = \{u : (v, u) \in E\}$ . In other words, it is the set of vertices in  $V$  which share an edge with  $v$ .

**Definition 13.** A *set neighborhood* of a set of vertices  $X \subseteq V$  is  $S(X) = \bigcup_{x \in X} S(x)$ .

Next, we prove a few theorems on which the Hungarian Algorithm is based.

**Theorem 1.** If a graph  $G = (V, E)$  with a feasible labeling  $l$  has a perfect matching  $M$  in its equality subgraph  $G_l = (V, E_l)$ , then  $M$  is also the maximum-weighted matching.

*Proof.* Let  $W(M')$  be the weight of any perfect matching  $M'$  in  $G$  (not necessarily in  $E_l$ , i.e., the equality subgraph).

From definition 4 of perfect matching, we know every vertex  $v \in V$  is covered exactly once by an edge in perfect matching  $M'$ . Hence, we can write:

$$\sum_{(x,y) \in M'} l(x) + l(y) = \sum_{v \in V} l(v) \quad (5.6)$$

From definition 10 of feasible labelling, we can write:

$$W(M') = \sum_{(x,y) \in M'} w(x, y) \leq \sum_{(x,y) \in M'} l(x) + l(y) \quad (5.7)$$

Substituting equation 5.6 in equation 5.7, we get:

$$W(M') = \sum_{(x,y) \in M'} w(x,y) \leq \sum_{(x,y) \in M'} l(x) + l(y) = \sum_{v \in V} l(v) \quad (5.8)$$

$$W(M') \leq \sum_{v \in V} l(v) \quad (5.9)$$

Now, let  $W(M)$  be the weight of a perfect matching  $M$  in the Equality Subgraph  $E_l$ . Hence, we can write the following from the definition 11

$$W(M) = \sum_{(x,y) \in M} w(x,y) = \sum_{(x,y) \in M} l(x) + l(y) = \sum_{v \in V} l(v) \quad (5.10)$$

$$W(M) = \sum_{v \in V} l(v) \quad (5.11)$$

Substituting equation 5.11 in equation 5.9, we get:

$$W(M') \leq W(M) \quad (5.12)$$

Hence, proved that  $M$  is the maximum weighted matching. □

---

**Algorithm 1:** Hungarian Algorithm

---

**Data:** bipartite graph  $G = (V, E)$ , feasible labelling  $l$ , equality subgraph  $G_l = (V, E_l)$ , some matching  $M$  in  $E_l$

**Result:**  $M$  : Maximum weighted matching in  $G$

**while**  $M \neq$  perfect matching **do**

1. Try to find an augmenting path for  $M$  in  $E_l$ ; this increases the size of  $M$  ;
  2. If no augmenting path exists, improve the labeling  $l$ ; this increases the size of  $E_l$  ;
- 

#### 5.5.4.4 Intuition of the Hungarian Algorithm

From the previous subsection 5.5.4.3, we noticed that the problem of finding the maximum weighted matching in a graph  $G$  boiled down to finding any perfect matching in the Equality Subgraph of  $G$ . Hence, we have turned an optimization problem into a combinatorial problem.

The Hungarian algorithm starts with a trivial equality subgraph of  $G$  where vertex labels on one side of the bipartite graph have a value of 0 and vertex labels on the other side of the bipartite graph have a value equal to the maximum weight value of edges incident on that vertex. The edge set of the Equality graph  $E_l$  is constructed by including those maximum-weighted edges which contributed to vertex labels

of the non-zero label side of the bipartite graph. Initially, we start with any matching  $M$ . In every iteration, the Hungarian algorithm either tries to find an augmenting path for matching  $M$  in  $E_l$ , which increases the size of the matching. Otherwise, it improves the labeling  $l$  by including at least one more edge into  $E_l$ , which was not a part of  $E_l$ . The existing labeling is improved using the lemma presented below.

**Lemma 1.** *Let  $X$  and  $Y$  be the two disjoint vertex sets of the bipartite graph  $G$  in which we want to find the maximum weighted matching. Let  $P \subseteq X$  and  $Q = S(P) \subset Y$ , where  $S(P)$  represents the set neighborhood of vertex set  $P$  from definition 13.  $P$  and  $Q$  contain the vertices in the augmenting alternating path. We set  $\delta_l$  as follows*

$$\delta_l = \min_{x \in P, y \notin Q} \{l(x) + l(y) - w(x, y)\} \quad (5.13)$$

where  $l(x)$ ,  $l(y)$  denote the labelling of vertices  $x$  and  $y$  respectively. We update the labeling of all the vertices in the graph as follows:

$$l'(v) = \begin{cases} l(v) - \delta_l, & \text{if } v \in P \\ l(v) + \delta_l, & \text{if } v \in Q \\ l(v), & \text{otherwise} \end{cases} \quad (5.14)$$

We can easily see that updating the labels this way preserves the labeling feasibility. Moreover, if an edge  $(x, y) \in E_l$ , then the edge  $(x, y) \in E_{l'}$  as well. Additionally, there will be one more edge which will get added to  $E_{l'}$ , i.e., the edge  $(x, y)$  such that  $x \in P, y \notin Q$  whose weight is equal to  $l(x) + l(y) - \delta_l$ .

The algorithm is guaranteed to terminate because, in each step, we are increasing the size of the matching or increasing the Equality Subgraph edge set size.

### 5.5.5 Creation and deletion of tracks

We implement a simple track management system where tracks are deleted if it has not been matched with any detection for more than some empirically decided threshold number of frames. Similarly, if a detection goes unmatched, we create and initialize a track that remains inactive for an empirically decided threshold number of frames before starting to appear. During the inactive period of the track, if it goes unmatched with a detection, the track is deleted.

## *Chapter 6*

### **Hardware Architecture**

This chapter outlines the various pipelines we tried to read the live video stream from the cameras and feed them into the server for further processing. We will look at the various hardware components involved in each of the pipelines starting from the camera, the wires/cables doing the actual transfer, and finally, the adapter, which takes these signals and feeds them into the computer doing the actual processing. We will also talk briefly about how we transferred the processed data to the Graphics server, which would actually show our output on the broadcast. This server was operated by AE Live, and it was important for us to decide on a common set of protocols to do the data transfer so that the encoding and decoding of the data could be performed consistently. At a very high level, we tried two pipelines:

1. BlackMagic Decklink Cards
2. Network Device Interface (NDI) cameras and tools

We will present the details of each of the above-mentioned pipelines in the subsequent sections. Figuring out these details we will be presenting was a major step towards deploying our work into production.

#### **6.1 Using BlackMagic Decklink Cards**

Decklink cards developed by Blackmagic are possibly the highest-performing cards to capture from cameras and live streams and get the feed into a Mac, Windows, or Linux server. These cards are the industry standard and are widely used in film productions, TV programs, music videos, Hollywood feature films, and commercials. They can support capture of resolution as high as 8K with frame rates up to 60 frames per second.

There is a wide variety of Decklink cards to choose from, depending on the use case. We chose the highest-performing one, namely the Decklink 8K Pro card. Decklink 8K Pro is a Gen3 8-lane PCI Express capture card designed for high-resolution 8K workflows. It features 4 12G Serial Digital Interface (SDI) connections allowing up to 4 different video streams of any format to be connected and captured on the server using the free Blackmagic SDK.



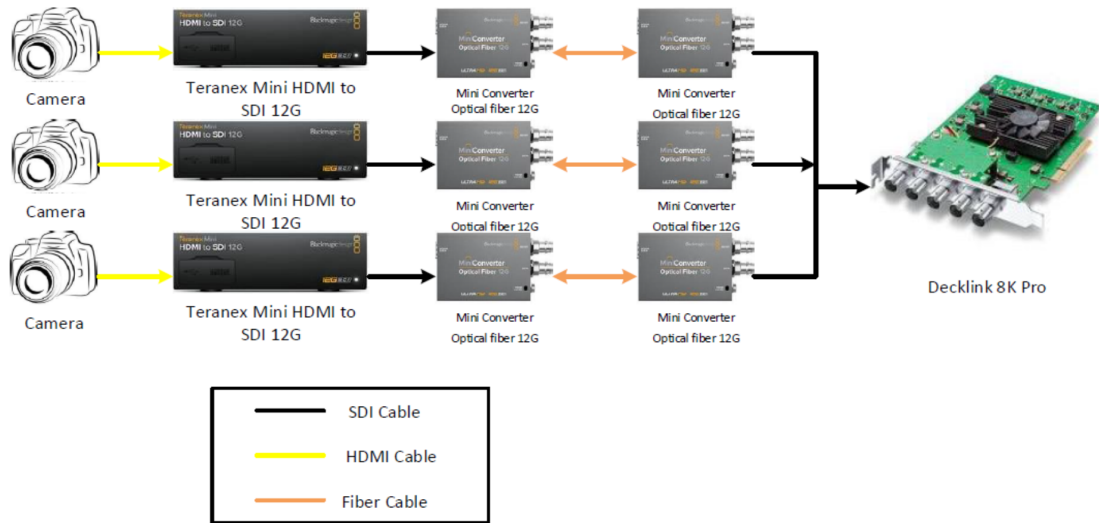


Figure 6.1: BlackMagic architecture: Hardware architecture showing how cameras are connected to a BlackMagic Decklink 8K Pro capture card in our processing server for delivering multiple 4K streams at realtime speeds using the BlackMagic technology.

### 6.1.1 Architecture

The camera captures the feed and transmits the video and audio signals via the High Definition Multimedia Interface (HDMI). This comes out as an HDMI cable, as shown by the yellow line in figure 6.1. Since we are transmitting 4K feed, we had to use HDMI cables labeled as "high-speed". Using any other HDMI cable would not work. Also, these cables labeled as "high-speed" must necessarily be less than 5 meters in length; otherwise, it leads to a loss in signal quality.

Had our server been placed really close to the capturing cameras, HDMI cables could have been an option for data transmission. However, this is not possible given the gigantic size of the cricket ground and the fact that we have to place our cameras all around the ground to maximize the coverage. Moreover, even if we were using a single camera, we could not have placed the server very close to the camera because this would have meant setting it up outdoors on the stands, which can lead to overheating of the Graphics Processing Unit (GPU) inside the server (used at full capacity in our pipeline).

Additionally, the BlackMagic Decklink card takes as input a Serial Digital Interface (SDI) cable. There are multiple reasons for such a design choice by BlackMagic, the primary one being distance. While HDMI cables can carry signals for short distances (5m or less), 12G SDI cables can transmit signals for distances as long as 100m without degrading the quality of the signals. The other reason for choosing SDI input over HDMI input is the risk of interruption of a stream associated with HDMI cables. HDMI cables do not have a locking mechanism, and there is a danger of the stream getting interrupted/disconnected by a slight pull or if someone's footfalls on the cable. SDI cables, on the other hand, lock tightly and remain fastened to the connector even if there is some tension pulling it.

This setup clearly demands a conversion from HDMI to SDI which is done by the Teranex Mini HDMI to SDI 12G converter. However, this is not enough! As mentioned earlier, SDI cables can carry signals for distances up to 100 meters, but we require to carry it up to a few kilometers. Our server is placed inside the broadcast room, which is usually located at some corner of the ground. In order to transmit signals up to a few kilometers without any degradation in the quality, we have to use a fiber cable. Fiber cables are much more expensive than SDI cables; however, they can transmit signals over longer distances (around 100 kilometers) without any signal loss. Hence, a second conversion is necessary from SDI to Fiber optic. Blackmagic provides the Mini Converter Optical Fiber 12G for this purpose. One advantage of this converter is that the same converter can be used in both directions. We place one such converter near the camera, which converts the 12G SDI to Fiber optic, and another converter near the server which converts the fiber optic back to 12G SDI. Finally, the signal enters the server through one of the 4 SDI ports in the BlackMagic Decklink 8K Pro Card and is made available for further processing.

The exact similar setup can be replicated for other cameras.

### **6.1.2 BlackMagic Decklink SDK**

BlackMagic provides a Software Development Kit called the DeckLink SDK<sup>1</sup>, which allows developers to integrate BlackMagic hardware into their software applications with ease. The SDK provides high-level as well as low-level interfaces and libraries to perform common functionalities like capturing a video stream entering any of the Decklink card ports, playing back a stream, performing realtime video processing, and controlling BlackMagic hardware through software. The SDK is written in C++ and is supported on Mac, Linux, and Windows. The SDK provides a very efficient implementation to read the frames from the incoming stream. Typically, a 4K stream is read at 25 frames per second using the OpenCV imread() function in Python. However, using the Hardware acceleration techniques and optimized SDK functionalities provided by Blackmagic, we can read 4K streams at speeds up to 60 FPS. There are multiple factors that make reading a 4K stream so efficient using the BlackMagic suite:

1. Direct Memory Access (DMA): This is a mechanism that allows the transfer of data from the Decklink card memory to the compute server memory (RAM) without the involvement of the Central Processing Unit (CPU). As a result of this, the CPU is free to do other useful tasks. DMA bypasses the traditional CPU usage for data transfer by delegating the transfer process to a dedicated controller called the DMA controller, which results in efficient and high-speed data transfer.
2. Asynchronous processing: The BlackMagic SDK provides support for asynchronous processing wherein developers can process frames and keep reading them in parallel. In other words, when one frame is being read, the previous frames can be processed simultaneously. This feature

---

<sup>1</sup>The BlackMagic SDK along with it's documentations can be found here.

increases the throughput of object detection greatly. The BlackMagic SDK provides this functionality by invoking a callback when a video frame arrives from the Decklink card. This callback function is called 'DeckLinkCaptureDelegate::VideoInputFrameArrived()', and the developer can put in any code to perform custom processing on the arrived frame. In our case, we pass the arrived frame through the Yolo object detector, and the output of this processing is bounding boxes indicating the players in the frame. Each time the callback is invoked, a new thread is created, and the main thread need not wait for the newly created thread to complete processing. It can continue reading the next frames, making the implementation highly parallel.

3. **Hardware Acceleration:** It is the process wherein computationally expensive tasks are offloaded to the hardware instead of burdening the CPU with them. This enhances performance greatly. For example, BlackMagic has specialized hardware in their Decklink cards to handle complex video encoding and decoding algorithms efficiently. Otherwise, the CPU had to be involved in doing such processing. Some other examples of image processing tasks that are offloaded to specialized hardware in order to reduce the burden of CPU are: a) Color space conversion, b) Scaling and resizing, c) deinterlacing and interlacing of broadcast feeds

We cannot afford to store the frames anywhere on the disk nor pass them to some other process using sockets, as this would increase the latency by many folds. Hence, we had to design our detection pipeline in C++ so that the frame which is present in the RAM can be shared by the detection module as well. Since both the reading of frames and computing the detections are part of the same process, the pointer of the frame is passed to the detection module. This implementation does not make a copy anywhere in the memory and hence, makes the implementation extremely efficient.

## **6.2 Network Device Interface (NDI) cameras and tools**

NDI, short for Network Device Interface, is a technology developed by NewTek that enables sharing of video and audio content over the Ethernet using Internet Protocol. The main advantage of NDI is the low setup cost. Since NDI operates over IP, there is no requirement for specialized video cables and cards for video transmission. NewTek offers a wide variety of PTZ (Pan-Tilt-Zoom) cameras which can capture high-quality content and transmit over Ethernet using NDI protocols. These cameras can be easily integrated into the NDI workflow without the need for expensive capture cards and/or physical connections.

### **6.2.1 Architecture using Single Network Interface Card**

Each of the NDI cameras captures the feed and transmits the captured video signals in the form of User Datagram Protocol (UDP) packets (datagrams) through Cat6 Ethernet Cables. UDP is a connectionless, lightweight protocol sitting on top of Internet Protocol (IP) and is well-suited for realtime,

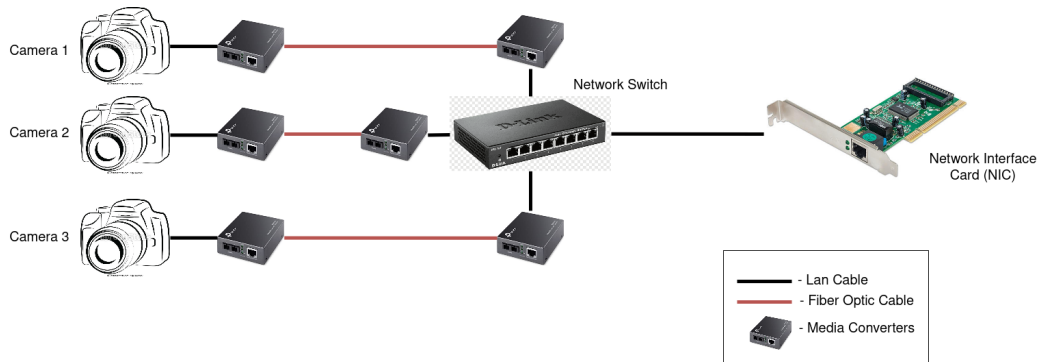


Figure 6.2: Network Device Interface (NDI) architecture: Hardware architecture showing how NDI cameras are connected to a Network Interface Card (NIC) in our processing server for delivering multiple 4K streams using the NDI technology.

low-latency streaming applications. All the datagrams arrive at a network switch sitting in front of the processing server. The network switch connects multiple networking devices together, and its responsibility is to forward the packets to the right destination device. In this setup (shown in 6.2), all the packets are forwarded to the Network Interface Card (NIC) of our processing server. The NIC decodes each of the arriving packets and stitches the three incoming streams separately. It is possible to put together the three streams separately even though the packets from all three streams are entering through a single channel (leading to an intermingling of packets from the three streams). This is because datagrams contain information about the source IP address, i.e., the IP address of the device generating the packet. All we need to do is bucket the incoming packets based on the source IP address of the packet, and we will be able to put together the stream captured by the individual cameras. This functionality is provided by the NDI SDK<sup>2</sup>. We, as developers, just need to specify the IP address of the NDI source we are interested in, and the NDI SDK has implemented functions with the help of which we would be able to get the frames from the desired source. Once we receive the frames, we can do any form of processing on them. In our case, it is sent to the detection module to get the bounding boxes of the players in that frame.

This setup uses 10 GigaBit per second Cat-6 Ethernet cables. Cat6 Ethernet cables can transmit data up to a distance of 100 meters without a loss in the quality of the transmitted signal. If we have to transmit signals for distances more than the recommended 100 meters length, we have to use fiber cable and deploy Media converters to convert Ethernet Connection to Fiber connection and vice versa. Fiber cables can transmit signals up to a distance of 40 km without degradation in signal quality. This entire setup is shown in figure 6.2.

Let us analyze the bandwidth required for this setup, and what is the maximum speed (in frames per second) we can achieve if we were transmitting uncompressed raw feeds. 1 Gigabit is equal to  $1e9$  bits. A 4K frame has  $3840 * 2160 * 3 = 24883200$  pixels. One pixel occupies 8 bits of storage. Using these back of the envelope calculations, a 4K frame occupies  $24883200 * 8 = 199065600$  bits. This is

<sup>2</sup>The NDI SDK can be downloaded here: <https://ndi.video/download-ndi-sdk/>

approximately 0.2 Gigabits. Since we are transmitting three streams through the same cable (from the switch to the NIC), we require a total of  $0.2 * 3 = 0.6$  Gigabits bandwidth for one frame. Our cable has a bandwidth of 10 Gigabit; hence we can achieve a maximum of  $10/0.6 = 16.7$  frames per second. This speed is clearly not enough for our use case, which demands realtime speeds.

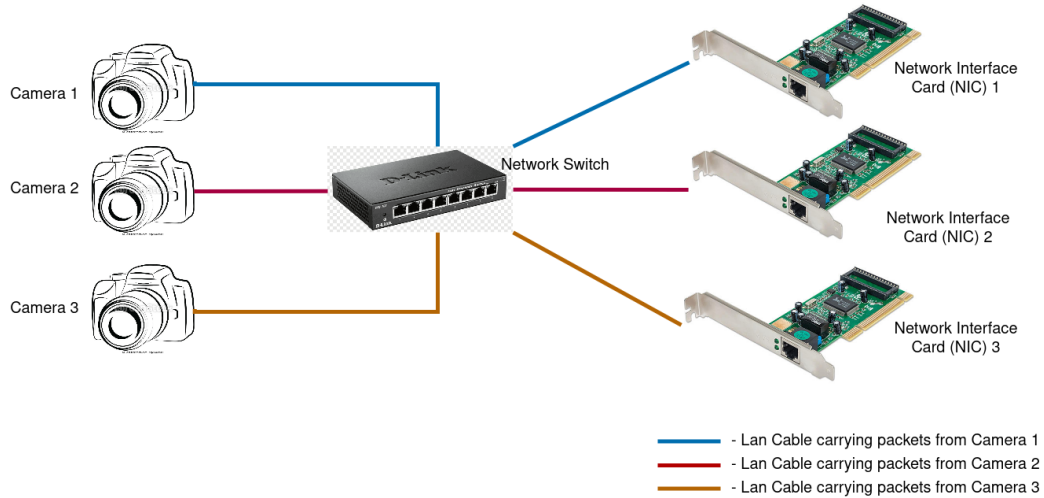


Figure 6.3: Network Device Interface (NDI) architecture: Hardware architecture showing how NDI cameras are connected to multiple Network Interface Cards (NICs) in our processing server to increase the bandwidth for delivering multiple 4K streams at realtime speeds using the NDI technology. Note that the media converters are not shown to improve clarity.

## 6.2.2 Architecture using Multiple Network Interface Cards

Careful analysis shows that the bandwidth of the network is throttled by the Ethernet cable from the switch to the Network Interface Card. This motivated us to build a customized server with multiple NICs. Specifically, if we had three NICs, then each camera could send its packets to exactly one NIC. Each NIC has a unique IP address. The cameras can be configured so that the packets generated from that camera use the destination IP address of the corresponding NIC. Figure 6.3 illustrates this setup. (Note that the converters were removed for clarity). Packets from camera one will have the IP address of NIC 1 as their destination IP. Similarly, packets coming out of cameras 2 and 3 will have the IP address of NIC 2 and 3, respectively, as their destination IP. When the packets from all three streams arrive at the switch, the switch will forward only the camera one packet to NIC 1, only the camera two packets to NIC 2, and only the camera three packets to NIC 3. Instead of getting a bandwidth of 10 Gigabits per second when we had a single NIC, we now have a bandwidth of 30 Gigabits per second, with each ethernet cable carrying almost the same number of packets from the switch to each of the NICs. In other words, this setup balances the load uniformly across the 3 NICs, and we can get speeds as high as  $16.67 * 3 = 50$  Gigabits per second.

Although this setup was more cost-effective, we leaned toward using the BlackMagic Decklink cards because of multiple reasons:

1. The overall speed we achieved using the BlackMagic setup was much higher than the setup using NDI tools. This was because BlackMagic uses specialized hardware in their capture cards to offload common processing tasks like encoding and decoding of signals from the CPU. NDI does not use any such specialized cards, and hence such hardware acceleration techniques can be deployed on the NDI setup. As a result, the CPU is overburdened with common processing tasks resulting in lower throughput.
2. Assembling a server with 3 Network Interface Cards is not very trivial. Moreover, maintaining it, in the long run, would become increasingly difficult. NDI does not support such a multi-NIC setup off the shelf (OTS), and we need to keep our code up to date to support it. In contrast, it is much easier to use Off the shelf BlackMagic card with three input ports. BlackMagic is very regular in rolling out updates for their software.
3. NDI primarily uses UDP, and we know that UDP is unreliable. It does not guarantee reliable delivery of packets, and packets can be lost. This is not very favorable in our setup. If we want to establish a reliable TCP connection, a separate network infrastructure needs to be established, which has a much higher maintenance cost.
4. NDI is heavily dependent on the Network infrastructure. Streaming multiple 4K feeds can lead to network congestion if not properly managed.

### **6.3 Communicating with Graphics Server**

Our processing server had to communicate with the graphics server by sending the details of the generated top-view map so that it could broadcast the graphics output on the screens of users. We implemented this communication service using socket programming. A socket is the basic building block of network communication. We created a socket object configured to use IPv4 addressing and the User Datagram protocol to send data to the Graphics server, which was a part of the same local area network (LAN) as our processing server. The data we wanted to send was encoded in binary using the 'ASCII' encoding scheme. The data we send is a list of  $2 * n$  floating point numbers, which represents the  $(x, y)$  coordinates of the  $n$  detected players on the top view. Every odd number in the transmitted list contains the  $x$  coordinates of the detected players on the top view. Similarly, every even number in that list contains the  $y$  coordinates of the detected players. The  $y$  coordinate of a detected player immediately followed the  $x$  coordinate of that player in the list.

## Chapter 7

### Software Architecture

In this chapter, we take a deeper look into the software architecture of the built system. We already took a look at each of the individual components that constitute this system. This chapter will dive into the low-level implementation details and design choices in bringing up the various components of this real-time data processing streaming pipeline. These design choices make the system extremely efficient, real-time, extensible, maintainable and easy to deploy in production settings.

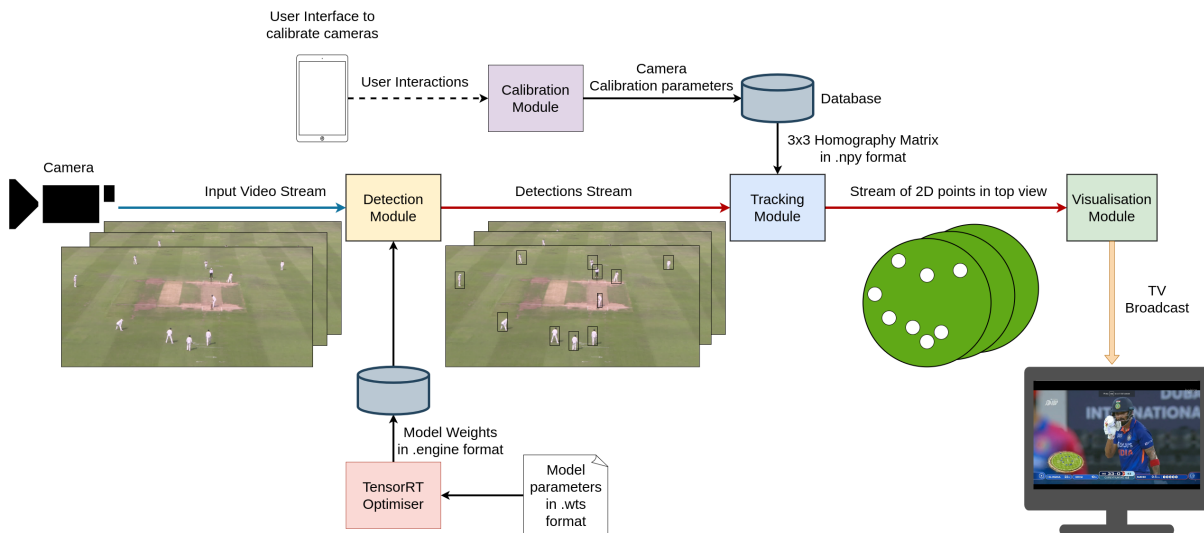


Figure 7.1: Software Architecture: This figure illustrates our data-processing pipeline highlighting the major system components and showing the data flow through the pipeline using arrows. The figure also illustrates the transformation of the data through various stages of the pipeline. Note: Only one camera is shown in this figure for clarity however in reality there are multiple video streams entering the server from multiple cameras.

## 7.1 A Streaming, Event-Driven Pipeline with a Modular Architecture

Our pipeline is a streaming pipeline since the source of input are video streams from multiple cameras. It is also an event driven architecture since different modules in the pipeline are executed when events trigger. For example, the arrival of a video frame into the processing server triggers the detection module which in turn triggers the Tracking Module and finally the output is communicated to the Visualisation Module.

The architecture is highly modular with each key functionality like detection, tracking, visualisation and calibration implemented as a separate modular component. This helped us implement different modules in different programming languages. We were also able to deploy each component on it's own. This made the development easier and each of the modules could be managed much more effectively, independent of the other modules. Moreover, decoupling the implementation into modules ensured that if some module was undergoing changes the other modules were unaffected as long as the communication between modules were consistent.

The Detection module reads video frames from cameras using the BlackMagic SDK and is responsible for predicting the bounding boxes around players for each frame. We implemented the Detection Module in C++ and optimised it with the TensorRT framework as mentioned earlier. It was implemented using C++ because of performance reasons. We bench-marked the inference speeds on 4K images and found that the same TensorRT implementation in Python was 5 times slower than the C++ implementation.

The Camera Calibration Module helps in calibrating the cameras which is useful for projecting the player positions from the camera view to the bird's eye view. It was implemented in Python and the User Interface to mark the point correspondences was implemented as an OpenCV Graphical User Interface. The GUI shows the user the camera view image first where they mark atleast 4 points by double clicking on the relevant pixels in the GUI. Next, the user is shown the top view map where they mark the same corresponding points as marked in the camera view (in the same order).

The Tracking Module gets as input the detections from each frame and is responsible for merging the detection results from all the cameras and tracking the players and projecting them to the top-view. It was implemented in Python because we made use of some libraries which were available only in Python, an example being filterpy, a library which helped us implement Kalman Filters.

The Visualisation Module responsible for broadcasting the output to millions of viewers is owned and operated by the Graphics Partner.

## 7.2 Communication between the various Components

The communication protocol between the detection and the tracking modules and the tracking and the visualisation modules is implemented using raw socket programming. Each camera has a corresponding port associated to it through which it's detection results are communicated to the tracking module by



the detection module. The detection module just pushes a list of 5 value tuples for each camera frame - 4 values representing the bounding box and 1 value representing the confidence score of the detection. The list data is encoded in binary using ASCII encoding.

The tracking module listens to the sockets to which the detection module is writing and consumes the detection results immediately when available. It then uses this information to come up with the trajectory for each player in the top-view. The trajectory information in the form of top view points is written to another socket to which the Visualisation Module is listening to.

Implementing the communication protocol using sockets this way proved to be extremely efficient. The data formats agreed between modules communicating was designed so optimally that there was no loss of data during communication between modules. We ensured that we communicate as minimal information as possible at the same time ensuring sufficient information is communicated to the next module to perform its operation.

### 7.3 How is the Data stored?

The different modules persist different types of data to the disk. These stored data is used by some other module in the pipeline. For example, the detection module consumes the detector model weights which were written by the TensorRT optimiser into the File System. Similarly, the Camera Calibration Module writes the camera calibration parameters to the disk which is consumed by the tracking module. In this section, we take a look at the various data formats used to store the detection and calibration data in the disk.

Each camera has a calibration matrix associated with it, which is a  $3 \times 3$  matrix of floating point numbers. This calibration matrix is used to do a transformation of points from that camera view to the top view. These calibration matrices (one for each camera) is stored in the .npy (short for Numpy) format. Any Numpy data structure can be stored in the disk in this format. Numpy provides functions to write and read these formats in the Python code easily.

The Yolo detection model is stored in .wts format, which is a plain text file format. The first line in the file is a number indicating how many lines the file has excluding itself. Next, each line contains the layer name (a String), followed by an integer ( $x$ ) denoting the number of weights in that layer.  $x$  hexadecimal numbers follow in the same line which represent the layer weights.

The TensorRT optimiser converts the .wts file to a serialised .engine file by applying a series of optimisation techniques. The .engine file contains the optimised deep learning model as a sequence of bytes, ready for deployment. The Detection Module loads this .engine file into the main memory during inference. The model weights when stored as .wts file or .engine file are versioned. The files are suffixed with the date of creation. For example, detector-20231116.wts. Additionally the weights files contain tags like 'Golden', 'Latest' etc.

## *Chapter 8*

### **Results and Conclusion**

In spite of having a stable algorithm to do the tracking and generate the top-view map, we faced certain challenges when we deployed our algorithm to production. The multi-camera setup required a custom assembly of the server with three GPUs. It was difficult to find a vendor to assemble such a server with three slots. Even though such a server could have been assembled, transporting it from one place to another was becoming extremely difficult, and this was an absolute necessity as tournaments were played across multiple venues. We were also affected by the pandemic, as there were massive cost cutoffs and reductions in the workforce when our system was about to go live.

When we went to deploy our system, we faced multiple challenges. Detections were becoming very hard because of spider cameras on the field. There were players standing exactly at the diametrically opposite end of the placed camera. This made them appear even smaller and almost impossible to detect. Then, we had to tackle occlusions. These reasons meant we had to handle missing detections in our tracking pipeline, no matter how much we improved our detectors.

We know from the existing literature that poor detections drastically degrade the tracking results. Hence, our tracking algorithm had to improve. At the same time, in order to meet the realtime requirements, we could not afford to implement some heavy tracker. Modifying SORT seemed to be the only viable option under these constraints.

#### **8.1 Future Work**

The pipeline we built can be extended to other sports and this is something which we plan to explore in future works. For example, in sports like Kabaddi the tracking is much more challenging because there is a lot of clutter and occlusions. As a future work, we plan to explore some novel real-time tracking ideas in sports like Kabaddi.

We plan to extend this pipeline to build other kinds of Graphical visualisations, which show players being tracked in camera views. Figure 8.1 shows one such sample graphical visualisation we have thought of.

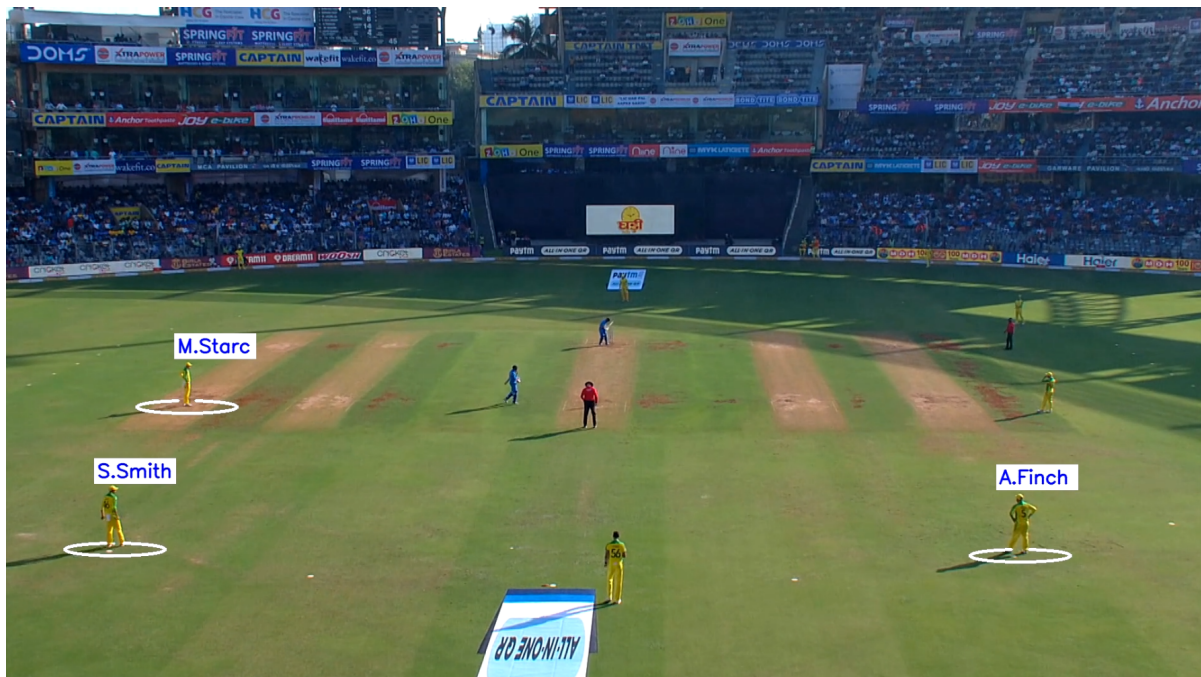


Figure 8.1: A Graphical Visualisation of certain players being tracked in the camera view is a simple extension of our pipeline which is left as a work to be explored in the future.

Furthermore, we can use our existing pipeline to compute some statistical analytics like the distance ran by a player, how fast a player is running, distance between two fielders etc.

Improving the detector performance so that it can generalise better to unseen stadiums and unseen camera positions is another future work which we plan to do as currently we need to finetune the detector on a few images (around 50) from the camera vantage point in the stadium where we are deploying.

## 8.2 Results from Asia Cup 2022

Our system was successfully deployed in production during the Asia Cup of 2022 in Dubai. We present a few qualitative results of our system going live multiple times below. This folder contains a few videos of our system going live on the screen of spectators.

hotstar 33L LIVE

ACC ASIAN CRICKET COUNCIL

SL **90-5** 12.3  
PROJECTED 144

HASNAIN 0-24 2.3 100

BOUNDARIES FOURS 12 SIX 0

SL LIVE Cricket

Press **esc** to exit full screen

hotstar 15L LIVE

ACC ASIAN CRICKET COUNCIL

BAN **90-4** 11.5  
CURRENT RUN RATE 7.6

MADUSHANKA 0-17 2.5 10000

AFIF 8 8 ▶ MAHMUDULLAH 2 4

Bangladesh Cricket Board

SL LIVE Cricket

Press **esc** to exit full screen

hotstar  
33L LIVE

SL **35-2** P 4.5  
CURRENT RUN RATE 7.2

GUNATHILAKA 0 2 de SILVA 24 15 HASNAIN 0-18 1.5 1 1LB 4 1 0

SL

OF WORLD  
ASIA CUP

hotstar  
22L LIVE

SL **104-4** 15.1  
TARGET 122

NISSANKA 53 44 SHANAKA 14 12 HASAN 0-22 2.1 DSC END

SL



ACC ASIAN CRICKET COUNCIL

hotstar 21L LIVE

Press **esc** to exit full screen

INTER INTERNATIONAL S

SHREY

fairplay

18



SL **93-4** 14.2  
NEED 29 OFF 34

NISSANKA 52 43 ▶ SHANAKA 4 8

RAUF 2-9 2.2

★

ACC ASIAN CRICKET COUNCIL

hotstar 21L LIVE

NISSANKA 31 25 ▶ RAJAPAKSA 15 13

SL **58-3** 9.1  
NEED 64 OFF 65

QADIR 0-9 1.1

DSC END

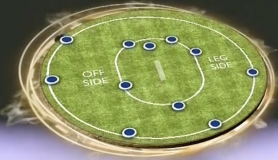

★



Press **esc** to exit full screen

hotstar  
40L LIVE

OF WORLD  
ASIA CUP

IND **152-2** 16.4  
CURRENT RUN RATE 9.1

PANT 14<sup>13</sup> KOHLI 69<sup>44</sup> FAREED 2-30 2.4 1 1 1 1

Press **esc** to exit full screen

hotstar  
40L LIVE

OF WORLD  
ASIA CUP




IND **33-0** P 4.5  
CURRENT RUN RATE 6.8

KL RAHUL 22<sup>19</sup> KOHLI 10<sup>10</sup> FAREED 0-5 0.5 1 0 0 0 0

## Related Publications

- Vineet Gandhi, Swetanjal Murati Dutta. **System And Method For Optical Tracking Of One Or More Players In A Sports Event.** Indian Patent App. 202041040579
- Jeet Vora, Swetanjal Dutta, Kanishk Jain, Shyamgopal Karthik, Vineet Gandhi. **Bringing Generalization to Deep Multi-View Pedestrian Detection.** Accepted at WACV 2023 IEEE/CVF Winter Conference on Applications of Computer Vision - 3rd Workshop on Real-World Surveillance, Applications and Challenges.



## Bibliography

- [1] K. Bernardin and R. Stiefelhagen, “Evaluating multiple object tracking performance: The clear mot metrics.” *EURASIP J. Image Video Process.*, vol. 2008, 2008. [Online]. Available: <http://dblp.uni-trier.de/db/journals/ejivp/ejivp2008.html#BernardinS08>
- [2] J. Luiten, A. Osep, P. Dendorfer, P. Torr, A. Geiger, L. Leal-Taixé, and B. Leibe, “Hota: A higher order metric for evaluating multi-object tracking,” *Int. J. Comput. Vision*, vol. 129, no. 2, p. 548–578, feb 2021. [Online]. Available: <https://doi.org/10.1007/s11263-020-01375-2>
- [3] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft coco: Common objects in context,” in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 740–755.
- [4] T. Chavdarova, P. Baqué, S. Bouquet, A. Maksai, C. Jose, T. Bagautdinov, L. Lettry, P. Fua, L. Van Gool, and F. Fleuret, “Wildtrack: A multi-camera hd dataset for dense unscripted pedestrian detection,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5030–5039.
- [5] Z. Zhang, “A flexible new technique for camera calibration,” in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, 2000, pp. 1330–1334.
- [6] E. Dubrofsky and R. J. Woodham, “Combining line and point correspondences for homography estimation,” in *Advances in Visual Computing*, G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, F. Porikli, J. Peters, J. Klosowski, L. Arns, Y. K. Chun, T.-M. Rhyne, and L. Monroe, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 202–213.
- [7] P. F. Felzenszwalb and D. P. Huttenlocher, “Efficient graph-based image segmentation,” *International Journal of Computer Vision*, vol. 59, no. 2, pp. 167–181, 2004.

- [8] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 00, June 2014, pp. 580–587. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/6909475/>
- [9] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders, “Selective search for object recognition.” *Int. J. Comput. Vis.*, vol. 104, no. 2, pp. 154–171, 2013. [Online]. Available: <http://dblp.uni-trier.de/db/journals/ijcv/ijcv104.html#UijlingsSGS13>
- [10] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR ’16. IEEE, June 2016, pp. 770–778. [Online]. Available: <http://ieeexplore.ieee.org/document/7780459>
- [11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS’12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.
- [12] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: <http://arxiv.org/abs/1409.1556>
- [13] C. Cortes and V. Vapnik, “Support vector networks,” *Machine Learning*, vol. 20, pp. 273–297, 1995.
- [14] R. Girshick, “Fast r-cnn,” *CoRR*, vol. abs/1504.08083, 2015. [Online]. Available: [http://www.cv-foundation.org/openaccess/content\\_iccv\\_2015/papers/Girshick\\_Fast\\_R-CNN\\_ICCV\\_2015\\_paper.pdf](http://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Girshick_Fast_R-CNN_ICCV_2015_paper.pdf)
- [15] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS’15. Cambridge, MA, USA: MIT Press, 2015, p. 91–99.

- [16] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “Ssd: Single shot multibox detector,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 21–37.
- [17] J. Redmon and A. Farhadi, “Yolo9000: Better, faster, stronger,” *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, 2016.
- [18] —, “Yolov3: An incremental improvement,” *ArXiv*, vol. abs/1804.02767, 2018.
- [19] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, “Yolov4: Optimal speed and accuracy of object detection,” *ArXiv*, vol. abs/2004.10934, 2020.
- [20] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, “YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors,” *arXiv preprint arXiv:2207.02696*, 2022.
- [21] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.
- [22] S. Kullback and R. A. Leibler, “On information and sufficiency,” *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.
- [23] X. Sun and L. Zheng, “Dissecting person re-identification from the viewpoint of viewpoint,” in *CVPR*, 2019.
- [24] Y. Hou, L. Zheng, and S. Gould, “Multiview detection with feature perspective transformation,” in *ECCV*, 2020.
- [25] L. Song, J. Wu, M. Yang, Q. Zhang, Y. Li, and J. Yuan, “Stacked homography transformations for multi-view pedestrian detection,” in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021, pp. 6029–6037.
- [26] Y. Hou and L. Zheng, “Multiview detection with shadow transformer (and view-coherent data augmentation),” in *Proceedings of the 29th ACM International Conference on Multimedia (MM ’21)*, 2021.

- [27] J. Vora, S. Dutta, K. Jain, S. Karthik, and V. Gandhi, “Bringing generalization to deep multi-view pedestrian detection,” in *2023 IEEE/CVF Winter Conference on Applications of Computer Vision Workshops (WACVW)*, 2023, pp. 110–119.
- [28] Gurobi Optimization, LLC, “Gurobi Optimizer Reference Manual,” 2023. [Online]. Available: <https://www.gurobi.com>
- [29] K. Zhou and T. Xiang, “Torchreid: A library for deep learning person re-identification in pytorch,” *arXiv preprint arXiv:1910.10093*, 2019.
- [30] K. Zhou, Y. Yang, A. Cavallaro, and T. Xiang, “Omni-scale feature learning for person re-identification,” in *ICCV*, 2019.
- [31] —, “Learning generalisable omni-scale representations for person re-identification,” *TPAMI*, 2021.
- [32] E. Ristani, F. Solera, R. S. Zou, R. Cucchiara, and C. Tomasi, “Performance measures and a data set for multi-target, multi-camera tracking,” in *ECCV Workshops*, 2016.
- [33] H. W. Kuhn, “The Hungarian Method for the Assignment Problem,” *Naval Research Logistics Quarterly*, vol. 2, no. 1–2, pp. 83–97, March 1955.
- [34] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, “Simple online and realtime tracking,” in *2016 IEEE International Conference on Image Processing (ICIP)*, 2016, pp. 3464–3468.
- [35] R. E. Kalman and Others, “A new approach to linear filtering and prediction problems,” *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.
- [36] N. Wojke, A. Bewley, and D. Paulus, “Simple online and realtime tracking with a deep association metric,” in *2017 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2017, pp. 3645–3649.
- [37] N. Wojke and A. Bewley, “Deep cosine metric learning for person re-identification,” in *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2018, pp. 748–756.
- [38] Y. Zhang, P. Sun, Y. Jiang, D. Yu, F. Weng, Z. Yuan, P. Luo, W. Liu, and X. Wang, “Bytetrack: Multi-object tracking by associating every detection box,” 2022.

- [39] J. Cao, X. Weng, R. Khirodkar, J. Pang, and K. Kitani, “Observation-centric sort: Rethinking sort for robust multi-object tracking,” *arXiv preprint arXiv:2203.14360*, 2022.