2026

SRAnalytics

ANALYSING VOLLEYBALL PLAYER DATA USING AI AND COMPUTER VISION

ALEKSANDER BUK (N00221532)

SUPERVISOR: MOHAMMED CHERBATJI

SECOND READER: CYRIL CONNOLLY

CREATIVE COMPUTING DL836

YEAR 4

2022/2026

DECLARATION OF OWNERSHIP

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Programme Chair.

WARNING: Take care when discarding program listings lest they be copied by some-one else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by others. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Computing (Hons) course handbook. Please read carefully and sign the declaration below.

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

Declaration

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.
Signed:

X ______________________________

Alekander Buk
Student

Date: ________29/04/2026________

Failure to complete and submit this form may lead to an investigation into your work.

ABSTRACT

The research conducted in this paper will delve into the prospect of using Artificial Intelligence in analysing the game of volleyball. Statistics can shape the strategies, training and individual development of a volleyball team and players.
It will extensively document the research, design and development of the application, which aims to provide reception statistics to teams and volleyball players alike.
Machine learning (ML) and computer vision (CV) can be utilized to detect, track and record ball and player movements. Furthermore, the data provided by these models can be extracted and filtered, to produce statistics in a significantly shorter period than manually reviewing footage.
The resulting application utilizes these models to produce and display reception statistics, giving access to results many average players don't have the time to find.

ACKNOWLEDGMENTS

I would like to thank Mohammed Cherbatji (AKA TheRealMoChe AKA The GOAT) because we always had great weekly meetings, and he let me speak about ideas instead of just completely shutting them down instantly. During this project he pushed me to explore AI models and ways to implement them, along with challenging me to think of more than one way of doing something. His expertise and input guided this project and

Like he famously quotes from an undisclosed source of material “just be yourself”. That is something that has stuck with me for the past three years. Mo has served as an inspiration to me from the first time we had class, how he seems to be knowledgeable in everything, and the way he teaches, letting you explore topics by yourself but steering you ever so slightly just worked super well for me.

I want to thank my girlfriend Anastasija Culkova, who is a proud supporter and always listens to me rant about random programming stuff even though she has no idea what any of it means. She is always there for me and it’s lovely to have someone in your corner (also makes good banana bread)

I want to thank Adam Doyle, who is one of my close friends and finishing his thesis this year, we have similar basis of projects, and brainstorming ideas. He is a huge inspiration to me also, as he works full time, has a lovely daughter named Lilith who is just the sweetest, and still manages to output some banger projects, and constantly keep up with learning new things. His work ethic and desire to be a good person really keeps me motivated.

I want to thank Filip Mishevski, he’s been there to support me throughout this project, always pushing me to keep working when at times I felt stuck.

I want to thank Kaho Yeung (AKA KHAOS) just because I’ve known him for 10 years now. He introduced me to volleyball, and this project wouldn’t be possible with some input from him about what things to track. Additionally, I still haven’t been to his house so IM CALLING HIM OUT RIGHT NOW.

Contents

DECLARATION OF OWNERSHIP 1

ABSTRACT 2

ACKNOWLEDGMENTS 3

1.Introduction 6

2. Research 7

2.1 What is computer vision? 7

2.1.1 Common ML Models used in CV 8

2.1.1.1 Convolutional neural networks (CNN) 8

2.1.1.2 Recurrent neural networks (RNN) 8

2.2 Image recognition 8

2.3 Image classification 8

2.4 Object detection 9

2.5. Object Tracking 10

2.5.1 Object Tracking Techniques 10

2.5.1.4 Kernel Based Tracking 10

2.5.1.5 Contour Based Tracking 11

2.5.1.6 Optical-Flow Based Tracking 11

2.5.1.7 Machine Learning Based Tracking 11

2.6 Volleyball and sports analysis 12

2.6.1 The sport of volleyball 12

2.6.1.1 The positions in volleyball 12

2.6.1.2 Common terms in volleyball 12

2.6.1.4 How volleyball works 13

2.6.1.5 Statistics gathered in volleyball 13

2.6.2 Techniques for statistics analysis in volleyball 14

2.6.2.1 Techniques for tracking players and the ball 14

2.6.2.2 Models for player and ball tracking 14

2.7 Computing Ball speed and Trajectory 15

3. Requirement Analysis 17

3.1 User 17

3.2 System requirements 17

3.3 Functional 17
3.4 Non-functional 17
4. Design 19
4.1 System Architecture 19
4.1.1 Frontend 20
4.1.2 API orchestrator 20
4.1.3 Pipeline 1 – Ball and Action classifiers using Ultralytics YOLOV8 20
4.1.4 Pipeline 2 – Player Detection using SambaMOTR 21
4.1.5 Docker 21
4.1.6 Data exchange and communication 21
4.1.7 Workflow 22
4.2 Workflow Diagram 23
5. Implementation 24
5.1 Pipeline1- Ball and Actions tracking 24
5.2 Pipeline2 – SambaMOTR tracking 40
5.3 API orchestrator 52
5.4 Frontend 55
5.4.2 App.jsx 58
5.5 Docker 78
6. Testing 84
6.1 User Testing 84
6.2 Model Testing and training 85
6.2.1 Volleyball Model training 85
6.2.1.2 Train 26 – Results and comments,25 epochs 87
Conclusions, Limitations and Future Work 87
7.1 Conclusions 87
7.2 Limitations 88
7.3 Future work 88
References 89

1.Introduction

The analysis of sports performance has become increasingly important both in professional sports and amateur settings, providing teams and players with valuable insights into improving decision-making and overall performance.
In volleyball, key actions such as receptions play a crucial role in determining the outcome of a rally, yet analysing these actions manually from video footage is time-consuming and subjective.
This project aims to address and solve this challenge by developing an automated system for volleyball video analysis that can detect, track and evaluate reception events.

The system combines computer vision and deep learning techniques to process match footage and extract meaningful data. It uses object detection and tracking models to identify players and the ball and applies custom logic to associate receptions with specific players based on spatial proximity and visual characteristics such as jersey colour. In addition, the system allows users to define a perfect zone on the video, enabling the evaluation of reception quality based on where the ball is located.

To support scalability and modularity, the system is built using a containerised architecture with Docker, separating the frontend, backend API and processing pipelines into independent services. A web-based frontend provides an intuitive interface for uploading videos, configuring analysis parameters, and viewing results.
Overall, this project demonstrates how modern machine learning techniques and software architecture can be combined to automate sports analytics and provide actionable insights from video data

2. Research

This chapter introduces the background research conducted to aid choosing the relevant models for the project.

2.1 What is computer vision?

Computer vision is a subfield of artificial intelligence that gives machines the ability to process, analyse and interpret visual inputs such as images and videos. It uses machine learning (ML) to help computers and other systems take out meaningful information from visual data. (Szeliski, 2022)

Some examples of tasks that CV algorithms are trained for are:

- Image recognition
- Image classification
- Object detection
- Image segmentation
- Object tracking
- Scene understanding
- Facial recognition
- Pose estimation
- Optical character recognition
- Image generation
- Visual inspection

2.1.1 Common ML Models used in CV

This section covers common machine learning architectures used in Computer Vision

2.1.1.1 Convolutional neural networks (CNN)

Commonly known as CovNets, they are used for classification and computer vision tasks. They have three main layers (convolutional, pooling and FC (fully connected). The convolutional later is where majority of the processing happens. It needs data, filter and feature map. The feature detector or kernel moves across the image checking if a feature is present, for example a certain RGB arrangement from a matrix of pixels in 3D. This is known as a convolution. A feature detector is a small 2D filter (usually 3x3) that scans the image. It computes a dot between the filter and image region, producing a value. As the filter moves across the image, these values form an array called a feature map. (LeCun et al., 2015)

2.1.1.2 Recurrent neural networks (RNN)

An RNN processes data step by step, keeping track of information as it goes, which makes it especially good for things like understanding videos or generating descriptions. Unlike regular neural networks that look at one input at one time, RNNs are "temporally deep" meaning they build understanding across time and can connect information from earlier in a sequence to something much later

(Donahue et al., 2015)

2.2 Image recognition

Image recognition encompasses the identification of people, places, objects and is the foundation of image classification, object detection and image segmentation. (Krizhevsky et al., 2012)

2.3 Image classification

Image classification involves categorising the images involved into classes or groups. From there, it predicts the most fitting label for an image or objects within that image. (Krizhevsky et al., 2012)

Types of image classification

2.3.2 Rule based image classification

This method relies on strictly developed process of image collection and labelling to match the specific task of the classification wanted. This process is manual and usually involves extensive labour labelling and selecting key pixels of the image that provide the most information. Role based image classification groups pixels in pixel clusters if they are similar enough into classes using set rules. It allows greater flexibility in customisation of classes and classification without using extremely complex ML models. An example of such rule can

be, for “cars” the image should contain tires, side mirrors, doors, and windows. This form of image classification includes techniques like thresholding and template matching.

2.3.3 Statistical Image classification

This method of classification is designed to automatically learn and recognise patterns in image frames. To classify them effectively, it relies on large, labelled datasets and powerful models like CNNs. These CNNs use up to three layers, each more complex to identify parts of the image. As the pixel data moves along the layers a bigger number of parts become recognised until the image is finally classified (Rawat & Wang, 2017)

2.4 Object detection

Object detection identifies where in an image a specific object is by drawing bounding boxes around it. Then, the image classification distinguishes which class or group it belongs to.

2.4.1 Image processing

Images are expressed as continuous functions represented as f (x,y) on a 2D coordinate plane and then digitised into a grid using sampling and quantisation. These in turn convert the function into a grid structure of pixel elements. The computer then segments these images into regions based on visual similarity of pixels.

By labelling images through annotation tools, objects are marked as regions defend by specific pixel characterises like area, intensity or colour. During training, the model learns these visual patterns. When given a new image, it looks for these regions that match the leaned patterns, so not the object itself just associated features. In this sense object detection is a form of pattern recognition, the models classify regions on learned combinations of parameters, rather than truly “understanding” the object it’s looking for. (*IMAGE SAMPLING and QUANTIZATION - File Exchange - MATLAB CentralFile Exchange - MATLAB Central*, 2024)

2.5. Object Tracking

Object tracking is a fundamental task in computer vision that allows systems to identify and continuously follow objects as that move through a sequence of frames

Before moving onto techniques, the concept of **Gaussian noise** must be introduced

Gaussian noise – is a normally distributed noise (like a bell curve) added to data/model weights, to make models generalise better and simulate real world imperfections.

2.5.1 Object Tracking Techniques

2.5.1.1 Point Based Tracking

Point based tracking methods represent an object using a set of feature points and track their movements over time. These points are chosen because they are easy to detect and track, such as corners and edges.

2.5.1.2 Kalman Filter

The Kalman filter is a linear quadratic estimator used to predict an object future position based on its previous state. It corrects the predicted state of the object in the next frame and corrects it using measurements from sensors or detections. It assumes the noise is Gaussian in motion and measurement.

2.5.1.3 Particle Filter

Otherwise known as Sequential Monte Carlo methods, these filters use a set of random particles (samples) to represent the possible states of the object.

Each particle is assigned a weight based on similarity, and then resampling (repeatedly drawing samples from the dataset to handle imbalanced data) to retain higher probability particles.

2.5.1.4 Kernel Based Tracking

Kernel based tracking treat the object as a region or kernel instead of individual pixels/points. Tracking is based on similarity measures like RGB colours or colour histograms

2.5.1.4.1 Mean shift

Mean shift tracks objects by iteratively shifting a search window towards the highest density of similar pixels.

2.5.1.4.2 CAMshift

Continuously Adaptive Mean Shift extends the idea of mean shift by adapting the size and orientation of the windows according to the current scale of the object.

2.5.1.5 Contour Based Tracking

2.5.1.5.1 Active Contours

Active contours commonly known as snakes, are deformable curves that evolve to fit object boundaries like tumours in medical images. It uses internal energy (Einternal) to keep the contour smooth and continuous and external energy (Eexternal) to pull the contour towards features of interest like edges and lines.

2.5.1.6 Optical-Flow Based Tracking

This method tracks the motion of every pixel between the consecutive frames based on intensity changes. It estimates the motion of these by analysing changes in intensity or colour. It focuses on tracking motion patterns across the whole image. It's defined as the apparent velocity of brightness patterns in an image sequence. This model assumes the intensity of a moving pixel remains constant across frames.

2.5.1.6.1 Lucas-Kanade Method

Lucas Kanade method is a spare flow method, which tracks a set of distinct points instead of every pixel. It assumes constant flow in a small neighbourhood around each point.

2.5.1.6.2 Horn-Schunck Method

Horn Schunk method is a dense flow method, using every available pixel to compute flow vectors (mathematical representations of motion in an image) using a global smoothness constraint. It makes a dense flow field which is great for full scene motion tracking.

2.5.1.7 Machine Learning Based Tracking

This method uses data driven models to learn the appearance, motion or behaviour of object in past frames, instead of relying solely on mathematics or handmade features.

2.5.1.7.1 Traditional ML Trackers

Firstly, the features of an image are extracted from the object such as colour histograms (RGB (red, green, blue values), HSV (hue,saturation,value)). It then uses a supervised classifying algorithm to distinguish the object from the background (such as random forests (combines the output of multiple decision trees to reach a single result)). Lastly it detects the object in the first frame, then slides a search window in the subsequent frames, the classifier scores **candidate regions** (specific area which is identified as potentially holding an object of interest) and then chooses the best matching ones.

2.5.1.7.2 Deep Learning Trackers

This approach lets the model automatically learn the robust features and can track objects under conditions like occlusion, scale changes and multiple object backgrounds. They use neural networks to learn automatically, allowing them to track object despite appearance changes (volleyball spinning, being disfigured due to blur).

2.5.1.7.3 Tracking by detection

Detects objects in each frame using a **CNN (**see 2.1.1.1**)** based detector to associate detected objects using similarity scores.

2.5.1.7.4 Siamese Network Trackers

Compares a template of the object from the first frame with candidate regions in the current frame and the network outputs a similarity score to locate the object.

2.6 Volleyball and sports analysis

In this section will introduce volleyball and the statistics gathered within the sport, along with explaining how these figures can help the team adjust strategy, compute player statistics and provide a numeric insight into a ball sport.

2.6.1 The sport of volleyball

Volleyball is mainly a 6 versus 6 player sport, where the aim of the game is to put the ball down on the other side of the opponent's court. Each team is allowed a maximum of 3 touches of the ball, with all body parts. The play starts with a serve, with a team sending the ball from outside of the court into the opponent's side and ends then the ball lands on the floor on either side.

2.6.1.1 The positions in volleyball

Setter: Always gets the second touch of the ball to set (overhand pass the second touch to the middle, outside or opposite hitters)

Outside hitter: This is a player who passes the serve and mainly attacks the ball from the left side of the net. They assist the libero with passing.

Opposite: The opposite takes its name from attacking opposite the outside hitter on the right side of the net and blocking opposite the enemy teams outside hitter. They do not participate in passing the enemies serve (unless the squad is running 4 passers)

Middle blocker: The middle blocker attacks and blocks from the middle of the court, they only play in the front row. They do not pass the enemies serve

Libero: The libero subs for the middle in the backrow, it is a defensive and passing heavy position. They participate in the passing the enemies serve

2.6.1.2 Common terms in volleyball

Spike: the action of hitting the ball towards the enemy's court in a whip like motion of the arm.

Tip: the action of gently tapping the ball to make it fall short into the enemy's side of the court

Serve: The action of hitting the ball from outside the back line of your own side of the court, over the net into the opponent’s side of the court.

Block out: hitting the ball at the opponents block and making the ball bounce out of bounds therefor scoring a point.

Ace: Scoring a point from a serve. The ball drops on the floor before anyone from the opposing team can pass it or keep it up.

Receive/Pass: The act of passing the ball when the opposing team serves towards your side of the court

Set: The act of passing the second touch of the ball to one of the attackers (outside, middle or opposite)

2.6.1.4 How volleyball works

There are multiple ways to score a point including but not limited to, spiking, tipping, block out, scoring an ace from a serve.

When one side is serving, the other side is receiving. This is the action of passing the opponents serve to one of your players preferably the **setter**.

2.6.1.5 Statistics gathered in volleyball

Attack conversion rate %

Points scored

Spike height

Spike speed

“Perfect” reception within a zone

2.6.2 Techniques for statistics analysis in volleyball

In this section, the models and techniques that will be used in this project for tracking the players and the ball will be covered. The use of these models in combination will allow statistics calculations from the results.

2.6.2.1 Techniques for tracking players and the ball

2.6.2.1.1 Kalman-Based Filtering

(See 3.1.1.1) This method predicts the players next position based on previous observations and correct it once new detections are available. This is good for smooth object tracking but falls short when players or the ball is occluded as it relies on constant detections for updating the observations.

2.6.2.1.2 DeepSORT

DeepSORT is a multi-object tracking method that extends upon traditional techniques. In addition to motion prediction using a Kalman Filter, it incorporates appearance feature maps extracted by a deep neural network. This helps distinguish between players with similar movement patterns. It handles occlusion and rapid movements, however it relies on a large dataset.

2.6.2.2 Models for player and ball tracking

Various models for player and ball tracking have been research, including YOLO family, Faster R-nn and SportsMOT

2.6.2.2.1 YOLO Family

You Only Look Once family of models is widely used for real time player detection. They are single stage detectors that preform object localization and classification in a single pass of the network, going only forwards. (Redmon et al., 2016)

2.6.2.2.2 Faster R-CNN

This is a two-stage detection model that firstly generates region proposals and then classifies them. This results in highly accurate player detection especially in cluttered scenes/occluded scenes. This is more suited towards offline detection, for example post-match analysis.

2.6.2.2.3 SportsMOT

SportsMOT is s a large-scale multi-object tracking dataset and framework designed specifically for sports scenes, where all players and objects in play are tracked across video frames. The dataset contains 240 video sequences totalling over 150,000 frames with dense bounding box annotations in basketball, volleyball, and football contexts,

and is intended to support development and evaluation of object tracking models under fast, variable motion and visually similar appearances. (Cui et al., 2023)

2.7 Computing Ball speed and Trajectory

Volleyball requires analysis of ball speed and trajectories. Frame by frame detection and parabolic curves are commonly used

2.7.1 Ball Speed with frame-by-frame detection

The balls position is detected in each frame as pixel co-ordinates $(xi, yi$). The displacement of these pixels between frames is calculated using the Euclidian Distance formula $di = (xi + 1 - xi)2 + (yi + 1 - yi)2$.

Pixel distances can be converted to meters using the length of a volleyball court (18m) and using a scale factor multiplication equal to the real length of the court($dm, i = di \times \frac{18}{Lp}$) where dm, i is the displacement in meters between frames, di is ball displacement between frames, Lp is the length of the court in pixels and 18 is the real world meters.

Then the peak speed of a serve or spike is computed using the time interval between frames, which was determined by the video frame rate ($\Delta t = \frac{1}{FPS}$)

where Δt is the interval of time between each frame (fps) eg Δt=1/60 (60 frames per second) = 0.0167 seconds per frame.

2.7.2 Modelling Passed Ball Trajectories using a Parabolic Curve

A parabolic curve is a symmetrical, curved shape defined by a quadratic equation ($y = ax^2 + bx + c$

It represents an objects path under uniform acceleration, like our ball travelling through the air.
Since a volleyball pass follows projectile motion under gravity, the ball moves along a path that can be approximated using a 2D parabola, where x is the horizontal position in meters, y is the vertical position in meters and a, b, c are coefficients (numbers that multiply the variables in the equation) determined by the balls motion. Using the same formulae as before, converting pixels to meters using the court scale factor, and collecting all points during the pass, we get an array of points.

Fitting the parabola requires using quadratic regression (is a type of regression analysis used to model the relationship between a dependant variable and independent variable when the relationship is **curved**) to find the coefficients a, b and c to minimize the error between the measured points and the parabola.

2.7.3 Setting a “perfect pass” zone for passing percentage calculation

Setters typically set from the 2.5m point into the length from the centre line of the court, and about 6 or 7 meters across. Using the real-world pixel to meters conversion and mapping the perfect pass zone, we can then use the landing coordinates from the parabolic curve to determine the accuracy of the pass.

3. Requirement Analysis

This chapter analyses the requirements for the application. It will touch on what is required from the user, system and both functional and non-functional requirements.

3.1 User

The user should be able to analyse volleyball footage and get performance insights User must be able to upload a video of a match, process it and view the results in an interactive interface. They should easily be able to identify their reception events, filter their jersey colour, define a perfect pass zone and navigate to specific moments within the video.

3.2 System requirements

- System should have a frontend interface
- System should include an API for communication
- System should include a pipeline for ball tracking
- System should include a pipeline for player tracking

3.3 Functional

1. Allow users to upload video files through the frontend interface
2. Process uploaded videos using two backend pipelines
3. Detect and track players and balls across the video
4. Identify and classify play events, in particular receptions
5. Match players to receptions based on jersey colour
6. Allow users to input RGB values and adjust tolerance for filtering players
7. Enable users to draw a zone on the video to evaluate reception quality
8. Determine whether receptions fall into that zone
9. Display processed results, including statistics and event lists
10. Allow users to navigate to specific plays in the video

3.4 Non-functional

- The system should process videos within a reasonable time.
- The system should be able to be easily scaled up.
- The frontend should be intuitive.
- The system should handle errors gracefully and report them for easy fixing
- The system should be dockerised to ensure each component is isolated and easy to update and debug
- The system should run consistently across different environments

- Uploaded files should be handled safely, and services should be isolated, in case of further scale and monetization later down the line

4. Design

In this chapter, the system architecture and overall design philosophy will be covered, including the frontend, API orchestrator, and both pipelines.

4.1 System Architecture

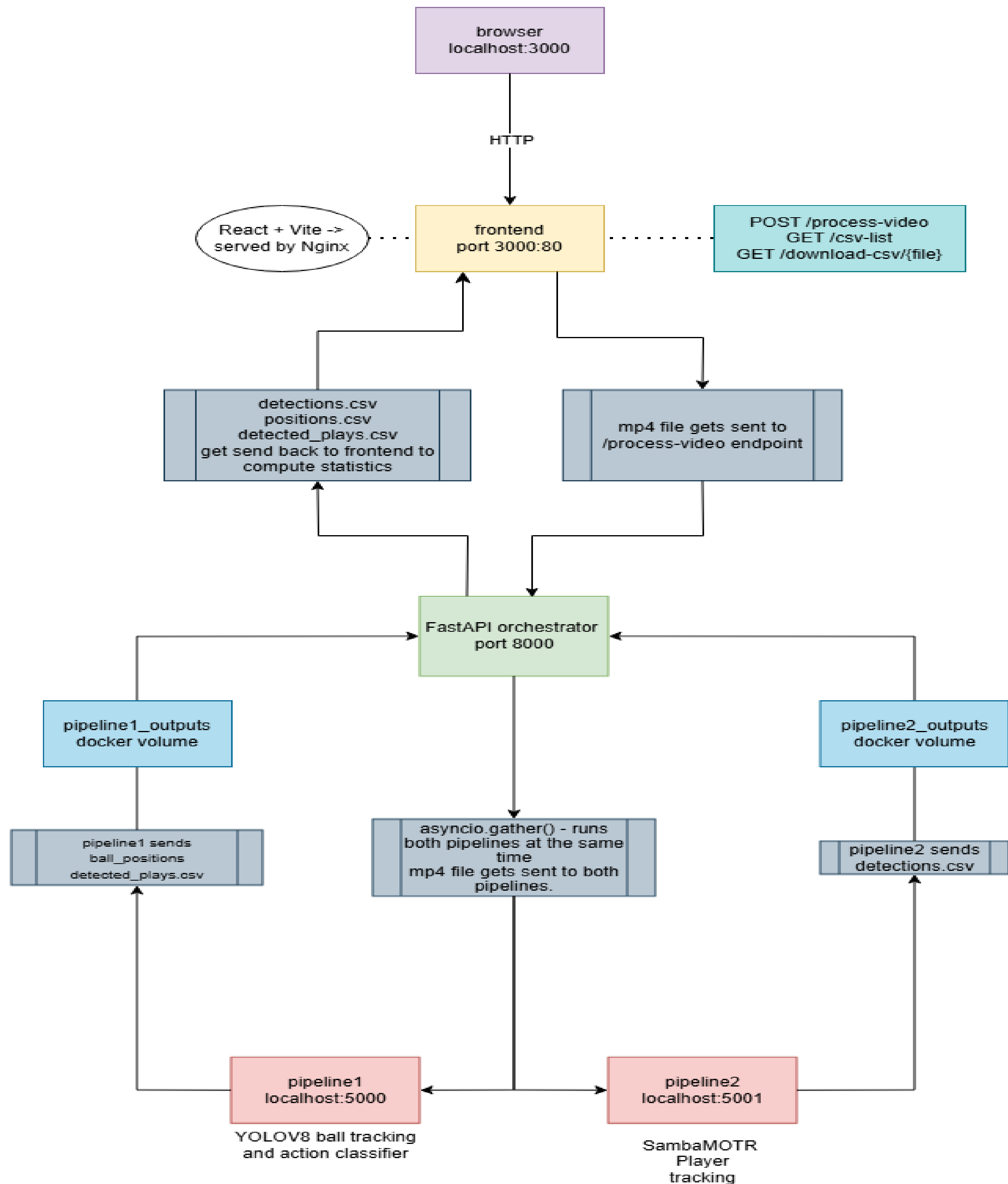


Figure 1 (System Architecture Diagram)

The SRAnalytics app utilizes a React/Nginx frontend combined with a FastAPI orchestrator and both YOLO pipelines and SambaMOTR model pipelines for ball tracking, action classification and player tracking.

It is containerized with Docker compose for local development and deployment. The architecture consists of four main services

4.1.1 Frontend

Technologies: React + Vite, served by Nginx
Role: Provides an interactive User Interface in the form of a website for users to upload volleyball videos, configure analysis parameters and visualise results.
Features:
Video upload and management.
Interactive filtering and searching of analysed results
Visualising tracking data and detected events
Deployment: Runs in its own docker container, which exposes port 3000

4.1.2 API orchestrator

Technologies: FastAPI (python)
Role: Acts as the central coordinator between the frontend and both pipelines
Features:
Receives video uploads and the user requests from the frontend.
Initiates processing jobs by communicating with both processing pipelines.
Serves processed CSV files back to the frontend for statistic calculations.
Handles error reporting and status updates.
Deployment: Runs in its own docker container, exposes port 8000

4.1.3 Pipeline 1 – Ball and Action classifiers using Ultralytics YOLOV8

Technologies: Python, YOLOv8 + custom PyTorch Models for object detection
Role: Pipeline1 is responsible for detecting the ball and recognizing actions in the uploaded video frames.
Features:
Uses a YOLO model to detect the ball in each frame of the video, tracks the ball positions and logs them into a csv file, with frame number, timestamp, ball_id and x, y (position) for each ball.
It uses another custom trained YOLOv8 model to detect specific actions in each frame, e.g. “reception”. Outputs another csv file with frame number, timestamp, action type, confidence, bounding box and ball position.
Deployment: Runs in its own docker container, exposes port 5000

4.1.4 Pipeline 2 – Player Detection using SambaMOTR

Technologies: Python, PyTorch

Role: Tracks players on the court using multi object detection and tracking.

Features: Utilizes deep learning models (DETR) for multi-object tracking, tracking players positions across the video and assigning them ids. Outputs a csv with frame_id, timestamp, bounding box positions, and mean RGB value of each detection.

Deployment: Runs in its own docker container, exposes port 5001

4.1.5 Docker

Technologies: Docker, Docker compose

Role: Provides isolated, reproducible environments for each service in the system, ensuring consistency in the deployments and simplifying dependency management. For example, the two pipelines use different versions of the NumPy package, without docker, this would have caused a conflict and neither of the pipelines would run.

Features:

Encapsulates each component (frontend, API, processing pipelines) in its own container.

Defines service configurations, networks and shared volumes using a single docker-compose.yml file.

Enables easy scaling, restarting and updating of individual services.

Makes sure local development, testing and production deployment use the same setup.

Deployment: Each service is built from its own Dockerfile, with its own dependencies and startup commands. Docker compose runs the startup and interconnection of all containers and exposing necessary ports. Shared docker volumes are used for efficient data storage and exchange between containers.

4.1.6 Data exchange and communication

Technologies: RESTful APIs, Docker Shared Volumes, HTTP, JSON, CSV

Features:

The frontend communicates with the API orchestrator via HTTP REST endpoints for video uploads, status checks and result retrieval.

The Api triggers processing jobs and exchanges data with the pipelines using endpoints and shared file paths.

CSV and mp4 files are transferred between containers using docker shared volumes.

Processed results are stored as the CSV files which are then accessed by the frontend for visualisation and filtering.

4.1.7 Workflow

User uploads a video through frontend -> API orchestrator receives the file and stores it in a shared volume -> orchestrator tell the processing pipelines to analyse the videos, passing the file path from the shared volume -> the pipelines writes the output csv's and videos to the shared volume -> frontend retrieves and displays the results by accessing these files via the Api

4.2 Workflow Diagram

Figure 2 (Workflow Diagram)

5. Implementation

This section will cover the implementation of all the technologies and insight into some of the code and how it all ties together to make the full working app.

5.1 Pipeline1- Ball and Actions tracking

This pipeline implementation features a two-stage computer vision system for analysing volleyball video footage. It detects and tracks a ball across frames, identifies gameplay actions such as receptions, or defence moves, and computes metrics such as speed. The system combines deep learning-based object detection with a lightweight tracking approach and frame by frame processing.
This is designed to operate as a web API using FastAPI but can also be run standalone.

Models & training used in Pipeline1:
Ultralytics Yolov8.
YOLO (you only look once) is a single stage object detection algorithm that uses a convolutional neural network to predict bounding boxes and class probabilities from an entire image in one forward pass (process of feeing input data through a neural networks layers from input to output) which makes it good for fast real time application such as ball tracking.
(Ball tracking train21 stats)
The model was trained on a dataset with over 3000 images

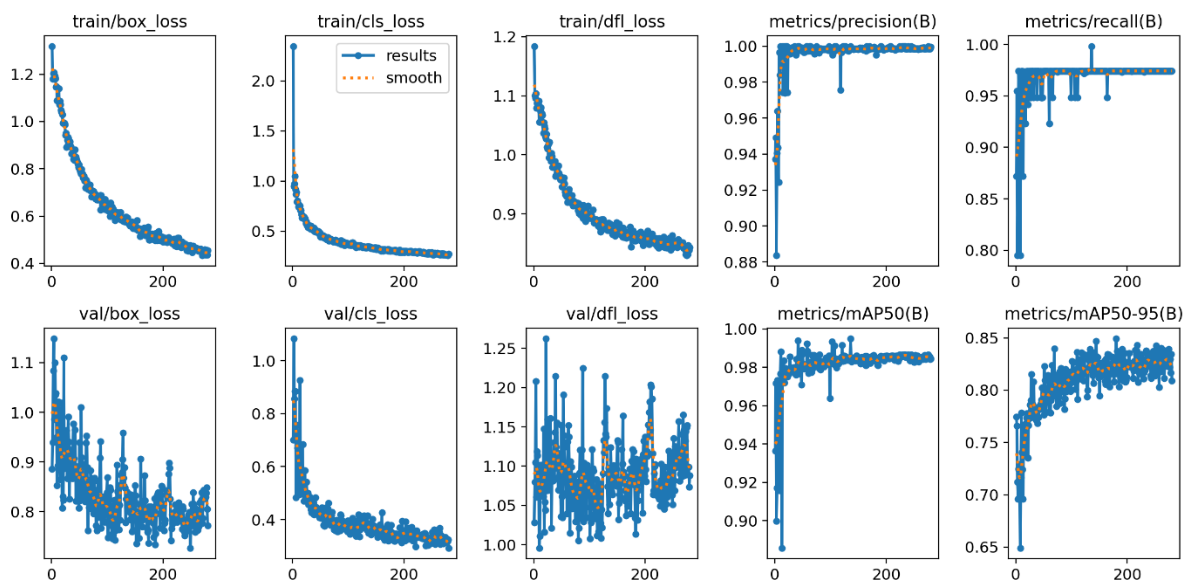


Figure 3 (Evaluation of Yolo Model)

Here are the results of the training, looking at this all the loss curves steadily decrease,

meaning the model is learning effectively and converging well. Precision rises quickly and stabilises at 0.99+ so the model produces very few false positives, along with recall being around 0.97+ which means most actual balls are being detected consistently. While the mAP@50-95 (mean average precision) is around 0.80-85 it is a much stricter but realistic metric and its very reasonable for a small fast object like a volleyball.

Pipeline processing workflow:

The system processing video data as frames sequentially in two stages:

1. Ball detection and tracking
2. Action detection and trajectory analysis

The pipelines functions read the video frame by frame using this line

```
cap = cv2.VideoCapture(video_input)
```

Figure 4

which appears in both functions, process_ball_tracking and process_trajectory
Each frame is then passed through both functions, and the results are stored and later written to outputs files, including annotated videos and CSV data.

PyTorch Compatibility handling

Before any processing occurs, this code modifies how PyTorch loads models. This ensures that all pretrained YOLO models load correctly as there was an issue with that when dockerizing the application.

```
_original_torch_load = torch.load
def _patched_torch_load(*args, **kwargs):
    kwargs['weights_only'] = False
    return _original_torch_load(*args, **kwargs)
torch.load = _patched_torch_load
```

Figure 5

1.Ball detection and Tracking:

The process_ball_tracking function implements an end-to-end pipeline for detecting and tracking a ball in a video using a deep learning object detection model called YOLO (YOU ONLY LOOK ONCE) combined with a centroid based tracking algorithm. It produces a processed output video and a CSV file containing the tracking positions over time (each frame).

Short 1-6 step overview:

1. Loads a trained YOLO model for object detection
2. Reads the input frame by frame
3. Detects the ball in each frame
4. Tracks the detected object across frames using a centroid tracker
5. Records spatial and temporal position data
6. Writes a video and a CSV log of positions

In depth function explanation

Model initialisation

```
from ultralytics import YOLO
```

Figure 6

```
model = YOLO('runs/detect/train26/weights/best.pt')
```

Figure 7

This pretrained YOLO model is loaded from a specified path within the docker container. As mentioned previously, this model has been trained to detect a volleyball using my own annotated dataset on Roboflow.

Video input and output setup

```
cap = cv2.VideoCapture(video_input)
if not cap.isOpened():
    raise ValueError(f"Cannot open video: {video_input}")
```

Figure 8

The input video is opened using the OpenCV library. This function checks whether the video stream is successfully initialised, if not it raises an error to prevent processing from happening.
Frame properties like width, height and frames per second (FPS) are extracted.

```
frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
actual_fps = cap.get(cv2.CAP_PROP_FPS)
```

Figure 9

If the video gives a valid fps value it overrides the default parameters, this is to make sure the timestamps are accurate in accordance with the other model.

```
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
out = cv2.VideoWriter(output_video, fourcc, fps, (frame_width, frame_height))
```

Figure 10

This initialises the output video writer.

Initialising the tracking

```
ct = CentroidTracker()
positions = []
frame_count = 0
```

Figure 11

A centroid based tracking algorithm is used to keep the same ids of objects across the frame, in this case 1 object, the volleyball. (Rosebrock, 2018)

Frame processing loop and timestamp calculation

```
while True:
    ret, frame = cap.read()
    if not ret:
        break

    timestamp_s = round(frame_count / fps, 4) if fps > 0 else 0.0
```

Figure 12

Each frame is read sequentially until the video ends, and a timestamp is computed based on the frame index and fps. This allows the tracking data to be interpreted in real (world) time.

Object detection, tracking and Data Recording and Visualisation

```
    results = model(frame)
    detections = results[0].boxes.xyxy.cpu().numpy()

    centroids = ct.update(detections)
```

Figure 13

The YOLO model processes each frame and returns bounding box detections. They are extracted in (x1, y1, x2, y2) format and converted to a NumPy array for compatibility with the tracker.

The centroid tracker then updates object identities based on the detected bounding boxes. It computes the centroid of each bounding box and matches it to previously tracked objects using the Euclidean distance formula.

```
for obj_id, centroid in centroids.items():
    cx, cy = float(centroid[0]), float(centroid[1])
    positions.append({
        'frame': frame_count,
        'timestamp_s': timestamp_s,
        'ball_id': obj_id,
        'x': cx,
        'y': cy
    })
```

Figure 14

This extracts the centroid coordinates and then a record is appended to the positions list, including

frame

timestamp

ball_id

x and y coordinates.

Output frame writing, resource cleanup and CSV export

```
        out.write(frame)
        frame_count += 1

    cap.release()
    out.release()

    with open(output_csv, 'w') as f:
        f.write('frame,timestamp_s,ball_id,x,y\n')
        for pos in positions:
            f.write(f"{pos['frame']},{pos['timestamp_s']},{pos['ball_id']},{pos['x']},{pos['y']}\n")
```

Figure 15

After processing all frames, the writer and video capture objects are released to free up

resources.
A csv file is generated containing the tracked positions, with each row corresponding to a detected object at a specific frame, structured as:
frame,timestamp_s,ball_id,x,y
Finally, the whole process is wrapped in a try-except block, so if any error occurs the function reports the error and returns false, otherwise returning True on successful completion

2. Action and trajectory analysis

process_trajectory() combines two sources of information; the action detection model being run on the frames and then ball trajectory data from the ball_positions.csv file.

```
def process_trajectory(video_input, csv_file, video_output, model_path):
```

Figure 16

The function takes in four inputs, although model_path isn't used because the model is hardcoded

```
model = YOLO('runs/detect/train29/weights/best.pt')
```

Figure 17

This function uses the same opening loading and getting video properties, creating the output video writer.

Loading ball position data

```
ball_positions = {}
```

Figure 18

A dictionary is created to store ball coordinates by frame number.

```
if os.path.exists(csv_file):
    with open(csv_file, 'r') as f:
        for line in f.readlines()[1:]:
            parts = line.strip().split(',')
            frame = int(parts[0])
            x, y = float(parts[3]), float(parts[4])
            ball_positions[frame] = (x, y)
```

Figure 19

This reads the ball_positions.csv file, and the first line is skipped because it has column headers.

Each row is split by commas, and the following values are extracted

```
frame = int(parts[0])
x, y = float(parts[3]), float(parts[4])
ball_positions[frame] = (x, y)
```

Figure 20

This assumes the csv structure is like this (frame, timestamp_s,ball_id,x,y) (which it is) then the ball position is stored like this

```
ball_positions[frame] = (x, y)
```

Figure 21

Initialising storage variables

```
detected_plays = []
frame_count = 0
prev_ball_pos = None
prev_frame = None
```

Figure 22

detected_plays stores all results that will be written to another CSV file
frame_count tracks current frame index
prev_ball_pos stores the previous known ball coordinate
prev_frame stores the frame number of the previous known ball position

Running YOLO action detection

Another yolov8 model was trained for this task, with a dataset of 15000 images including all the volleyball actions, like spike, set , defence move, reception, block and stand. This project focuses on reception, so that's the class that will be tracked.

```
results = model(frame, conf=0.10)
boxes = results[0].boxes
```

Figure 23

The current frame is passed into the YOLO model. Since the reception class didn't have many images, a low confidence threshold is used to increase the chances of detecting a reception. However, this can also increase false positives.
YOLO outputs bounding boxes, class IDs and confidence scores, that get stored in the "boxes" variable

Retrieving ball position for the current frame

```
ball_x, ball_y = None, None
if frame_count in ball_positions:
    ball_x, ball_y = ball_positions[frame_count]
```

Figure 24

The function checks whether the current frame has ball coordinates in the CSV obtained dictionary. If the ball was detected in this frame, its x and y coordinates are assigned to **ball_x** and **ball_y**, if not both **remain None**.

Tracking receptions and saving detection information

```
frame_logged = False
if boxes is not None and len(boxes) > 0:
    for box in boxes:
        cls_id = int(box.cls.cpu().numpy()[0])
        if cls_id != 3:  # Only log 'reception' (class 3)
            continue
        conf = float(box.conf.cpu().numpy()[0])
        x1, y1, x2, y2 = box.xyxy.cpu().numpy()[0]
        action = model.names[cls_id]

        detected_plays.append({
            'frame': frame_count,
            'timestamp_s': round(timestamp_s, 4),
            'action': action,
            'confidence': round(conf, 4),
            'x1': round(float(x1), 2),
            'y1': round(float(y1), 2),
            'x2': round(float(x2), 2),
            'y2': round(float(y2), 2),
            'ball_x': round(ball_x, 2) if ball_x is not None else '',
            'ball_y': round(ball_y, 2) if ball_y is not None else '',
            'speed_ms': speed_ms if speed_ms is not None else '',
            'speed_kmh': speed_kmh if speed_kmh is not None else '',
        })
        frame_logged = True
```

Figure 25

The variable **frame_logged** is set to false initially to track whether an action detection has been recorded for the current frame, ensuring that every frame is logged even if no action is detected. The function then checks whether YOLO has produced any detections using if boxes is not None and **len(boxes) > 0**, and if so, iterates through each detection, where box represents a potential object or action.

For each detection, the **class ID** is extracted using **cls_id = int(box.cls.cpu().numpy()[0]),** which corresponds to the predicted action label based on the models training, in this case "**reception**" is "**3**". Filtering detections to include only class "3" using **if cls_id != 3: continue**, focusing exclusively on reception and ignoring all other actions.

If the detection is valid, the confidence and bounding box coordinates are extracted, where **(x1,y1) and (x2,y2)** represent the top left and bottom right corners of the box respectively.

The action label is retrieved via **model.names[cls_id]**
This information is then appended to the **detected_plays** list, storing frame number, timestamp, action label, bounding box coordinates, ball position and estimated ball speed.
Finally, **frame_logged** is set to True to show that a valid reception has been recorded on that frame

```
if not frame_logged:
    detected_plays.append({
        'frame': frame_count,
        'timestamp_s': round(timestamp_s, 4),
        'action': '',
        'confidence': '',
        'x1': '',
        'y1': '',
        'x2': '',
        'y2': '',
        'ball_x': round(ball_x, 2) if ball_x is not None else '',
        'ball_y': round(ball_y, 2) if ball_y is not None else '',
        'speed_ms': speed_ms if speed_ms is not None else '',
        'speed_kmh': speed_kmh if speed_kmh is not None else '',
    })
```

Figure 26

Even if no reception action has been detected in each frame, the function still appends a record to the list, to keep a complete temporal record of the video rather than only storing frames where detections occur. This is a crucial thing as the statistical analysis

later depends on all the frames being recorded and lined up.

```
        out.write(frame)
        frame_count += 1

    cap.release()
    out.release()

    # Write detected_plays CSV
    plays_csv = str(video_output).replace('_tracked.mp4', '_detected_plays.csv')
    with open(plays_csv, 'w') as f:
        f.write('frame,timestamp_s,action,confidence,x1,y1,x2,y2,ball_x,ball_y,speed_ms,speed_kmh\n')
        for play in detected_plays:
            f.write(f"{play['frame']},{play['timestamp_s']},{play['action']},{play['confidence']},"
                    f"{play['x1']},{play['y1']},{play['x2']},{play['y2']},"
                    f"{play['ball_x']},{play['ball_y']},{play['speed_ms']},{play['speed_kmh']}\n")

    return True
except Exception as e:
    print(f"Trajectory processing error: {e}")
    return False
```

Figure 27

The final section of this is similar to the previous function, looping through each play in detected plays and writing it into a csv.

/Process Endpoint

This endpoint handles the full video processing workflow through the FastAPI backend.

Creating post endpoint

```
@app.post("/process")
async def process(video: UploadFile = File(...)):
```

Figure 28

This creates a post endpoint called /process. It accepts an uploaded video file from the user. The async keyword allows the server to handle file uploads and reads without blocking the server.

Creating tmp folder

```
upload_dir = Path("/tmp/uploads")
upload_dir.mkdir(parents=True, exist_ok=True)
```

Figure 29

This creates a temporary folder where uploaded videos are stored. **Parents=True** allows missing parent folders to be created, while **exist_ok = true** prevents an error if the folder already exists.

```
    with open(video_path, "wb") as f:
        f.write(await video.read())
```

Figure 30

saves the uploaded video with a unique filename using the current timestamp and the original filename, the video is then saved in binary mode using "**wb**" (write binary)

Defining file paths

```
csv_output = output_dir / f"{video_path.stem}_positions.csv"
video_output = output_dir / f"{video_path.stem}_tracked.mp4"
plays_csv_output = output_dir / f"{video_path.stem}_detected_plays.csv"
```

Figure 31

csv output stores ball position data
video output stores annotated tracked video
plays csv output stores detected action/ trajectory results.

Running ball tracking

```
success = process_ball_tracking(str(video_path), str(csv_output), str(video_output))
```

Figure 32

the uploaded video is first passed into the ball tracking function. This detects the ball, tracks its centroid across the frames, saves the ball positions to CSV and produces a video with annotations.

Running trajectory/action processing

```
if success:
    traj_success = process_trajectory(str(video_path), str
    (csv_output), str(video_output), None)
```

If ball tracking succeeds, the second stage runs. **Process_trajectory** uses the original video and ball position CSV to detect reception actions and estimate trajectory related values such as ball speed.

Returning output paths

```
return JSONResponse(content={
    "status": "success",
    "positions_csv_path": str(csv_output),
    "detected_plays_csv_path": str(plays_csv_output) if
    traj_success else None,
    "video_path": str(video_output)
})
```

if processing is successful, the API returns a JSON response containing the paths to the generated files. If trajectory processing fails, the detected plays CSV path is returned as None.

Handling exceptions

```
        else:
            raise HTTPException(status_code=500, detail="Processing
            failed")

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

Any unexpected error is caught and returned as an HTTP 500 response with the error message

Centroid Tracker Algorithm
The centroid tracker was written from a tutorial at
https://pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/
Imports and class setup

```
from collections import OrderedDict
import numpy as np

class CentroidTracker:
```

OrderedDict: stores objects in the order they were added
numpy: used for mathematical operations

class **CentroidTracker:**
Defines a tracker that follows objects based on their centroids (center points)

Initialization

```
def __init__(self, max_lost=20):
    self.next_object_id = 0
    self.objects = OrderedDict()
    self.lost = OrderedDict()
    self.max_lost = max_lost
```

max_lost: how many frames an object can disappear before being removed
next_object_id: assigned unique IDs to objects
objects: stores object ID corresponds to bounding box
lost: counts how many frames an object has been missing

Registering and Deregistering a new Object

```
def _register(self, bbox):
    self.objects[self.next_object_id] = bbox
    self.lost[self.next_object_id] = 0
    self.next_object_id += 1

def _deregister(self, object_id):
    del self.objects[object_id]
    del self.lost[object_id]
```

def _register *adds a new object when detected*
stores the bounding box
sets lost count to 0
increments ID for next object

def_deregister *removes an object when its gone too long*

Main update function

```
def update(self, rects):
    if len(rects) == 0:
        for object_id in list(self.lost.keys()):
            self.lost[object_id] += 1
            if self.lost[object_id] > self.max_lost:
                self._deregister(object_id)
        return self.objects

    input_centroids = np.zeros((len(rects), 2), dtype="int")
    for i, (x1, y1, x2, y2) in enumerate(rects):
        cX = int((x1 + x2) / 2.0)
        cY = int((y1 + y2) / 2.0)
        input_centroids[i] = (cX, cY)

    if len(self.objects) == 0:
        for bbox in rects:
            self._register(bbox)
    else:
        object_ids = list(self.objects.keys())
        object_centroids = np.array([
            ((bbox[0] + bbox[2]) / 2, (bbox[1] + bbox[3]) / 2)
            for bbox in self.objects.values()
        ])

        D = np.linalg.norm(object_centroids[:, np.newaxis] - input_centroids, axis=2)
        rows = D.min(axis=1).argsort()
        cols = D.argmin(axis=1)[rows]

        used_rows, used_cols = set(), set()
        for row, col in zip(rows, cols):
            if row in used_rows or col in used_cols:
                continue
            object_id = object_ids[row]
            self.objects[object_id] = rects[col]
            self.lost[object_id] = 0
            used_rows.add(row)
            used_cols.add(col)

        unused_rows = set(range(D.shape[0])) - used_rows
        for row in unused_rows:
            object_id = object_ids[row]
            self.lost[object_id] += 1
            if self.lost[object_id] > self.max_lost:
                self._deregister(object_id)

        unused_cols = set(range(D.shape[1])) - used_cols
        for col in unused_cols:
            self._register(rects[col])

    return self.objects
```

The code computes **input_centroids**, **handles len(rects)==0** by updating **self.lost** and calling _deregister, registers new object **if len(self.objects)==0,** otherwise computes distance **d=linealg.norm(object_centroids[:,np newaxis] – input_centroids, axis=2**) to match objects via rows, cols ,updates **self.objects[object_id] = rects[col],** increments/removes **unused_rows,**
registers **unused_cols**, and finally returns **self.objects.**

5.2 Pipeline2 – SambaMOTR tracking

"The SAMBA model (Segu et al., 2024) was used as the primary tracking method, with minor modifications made to extend the implementation by exporting tracking results to CSV format for further processing."

SambaMOTR is an end-to-end multiple object tracking method that models all object trajectories jointly rather than independently. It introduces a set of sequences representation (Samba) that captures long term temporal dependencies and interaction between objects.
In practice, each object is represented by a track query is updated across frames using a shared temporal memory. This allows the model to handle occlusion, maintain id's and track objects consistently over time without relying on manual association rules.

Demo.py takes in a video, runs the object tracking model on each frame, draws tracking boxes and saves the results into a CSV file

How the model runs

Loading libraries and model

```
import cv2
import time
import os
import numpy as np
import torch
```

These are the core tools, cv2 is for video handling + drawing, torch is for the deep learning model and numpy is for maths and arrays.
Here is how the model is loaded

```
config = yaml_to_dict(config_path)
model = build_model(config)
load_checkpoint(model=model, path=model_path)
model.eval()
print("Model loaded.")
```

Loading the build configuration, building the model structure, loading the trained weights, and set sets it to inference mode using **model.eval()**

Timer class

The timer class is a small but important utility that helps measure how long different parts of the program take to run, specifically how long each frame takes to process.

```
class Timer(object):
```

Defines the class

Variables inside of timer

```
def __init__(self):
    self.total_time = 0.
    self.calls = 0
    self.start_time = 0.
    self.diff = 0.
    self.average_time = 0.
    self.duration = 0.
```

Each variable has a purpose

total_time: the sum of all elapsed times across frames

calls: how many times the timer has been used (number of frames processed)

start_time: the timestamp when timing starts

diff: the time taken for the most recent frame

average_time: avg time per frame

duration: the value returned

these allow the timer to track instant performance and overall performance.

Starting and stopping the timer

```
def tic(self):
    self.start_time = time.time()

def toc(self, average=True):
    self.diff = time.time() - self.start_time
    self.total_time += self.diff
    self.calls += 1
    self.average_time = self.total_time / self.calls
    if average:
        self.duration = self.average_time
    else:
        self.duration = self.diff
    return self.duration
```

when tic runs **time.time()** returns the current time in seconds. This marks the start of the measurement. This happens right before processing a frame

toc calculates how long the operation took, and adds this frames time to the total, also increments the frame count.

Computing average time

self.average_time = self.total_time / self.calls calculates the average processing time per frame

Choosing what to return

If average = true return the average time, if not return just the latest frame time

Image preprocessing

Prepares each frame for the model to understand it.

Keep original image

```
ori_image = image.copy()
```

This creates a duplicate of the frame. The model needs a processed version, but later the demo needs to sample colours to output, so it keeps an original for that.

Get image size and compute scaling

```
h, w = image.shape[:2]
scale = 800 / min(h, w)
```

This gets the width and height and is used to calculate how much to resize the image. The second line fins the smaller side of the image, and resizes it so that side becomes 800 pixels. E.g. if the image is 1920x1080 the images gets scaled to 800 /1080

Limit max size and compute new dimensions and resize

```
if max(h, w) * scale > 1536:
    scale = 1536 / max(h, w)
target_h = int(h * scale)
target_w = int(w * scale)
```

Even after scaling, the image might be too large, so this makes sure no side exceeds 1536 pixels, preventing GPU memory issues, keeping inference fast and model inputs within expected boundaries

Computes the new target h and w with the scale

```
image = cv2.resize(image, (target_w, target_h))
image = F.normalize(F.to_tensor(image), [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
return image, ori_image
```

Resizes the image,

Convert image from NumPy (H x W x C) to PyTorch tensor (C x H x W)

Normalize pixel values (normalization transforms pixel values so they are centred and scaled. It subtracts the pixel – mean and then divides by standard deviation)

This ensures that most values are around 0, values typically fall in range [-2,2] and data becomes more consistent and stable.

Deep learning models need this normalized data because that is what most of them are trained on.

Each colour channel is normalized separately, it uses standard ImageNet statistics

Finally, return both images, using image for inference and **ori_image** for drawing boxes an extracting colours.

Filtering results

```
def filter_by_score(tracks: TrackInstances, thresh: float = 0.7):
    keep = torch.max(tracks.scores, dim=-1).values > thresh
```

Keeps only confident detections

```python
def filter_by_area(tracks: TrackInstances, thresh: int = 100):
    assert len(tracks.area) == len(tracks.ids), f"Tracks' 'area
    keep = tracks.area > thresh
    return tracks[keep]
```

Removes very small boxes (noisy detections)

Colour assignment

```python
def get_color(idx):
    idx = idx * 3
    color = ((37 * idx) % 255, (17 * idx) % 255, (29 * idx) % 255)
    return color
```

each object id gets a unique colour

Extracting RGB from bounding boxes

```python
def get_mean_rgb(frame_bgr, x1, y1, w, h):
    """Sample the mean RGB colour from the centre-third of a bounding box."""
    img_h, img_w = frame_bgr.shape[:2]
    # Use the middle vertical third to avoid ground / background at edges
    y_start = int(max(y1 + h / 3, 0))
    y_end   = int(min(y1 + 2 * h / 3, img_h))
    x_start = int(max(x1, 0))
    x_end   = int(min(x1 + w, img_w))
    if x_end <= x_start or y_end <= y_start:
        return 0, 0, 0
    crop = frame_bgr[y_start:y_end, x_start:x_end]  # BGR
    mean_bgr = crop.reshape(-1, 3).mean(axis=0)
    b_mean, g_mean, r_mean = mean_bgr  # cv2 is BGR
    return round(float(r_mean), 2), round(float(g_mean), 2), round(float(b_mean), 2)
```

This function calculates the average colour of an object inside a bounding box by focusing only on the relevant part of the box. It determines the image boundaries to ensure coordinates are valid, then selects only the middle vertical third of the box. This avoids adding background colours into the colour measurement. This is a key addition to the model as this enables player jersey matching in the frontend.

Drawing Tracking Results

```
def plot_tracking(image, tlwhs, obj_ids, scores=None, frame_id=0, fps=0., ids2=None, show_text=True):
    im = np.ascontiguousarray(np.copy(image))
    print(im.shape)
    im_h, im_w = im.shape[:2]

    text_scale = 2
    text_thickness = 2
    line_thickness = 3

    if show_text:
        cv2.putText(
            im,
            'frame: %d fps: %.2f num: %d' % (frame_id, fps, len(tlwhs)),
            (0, int(15 * text_scale)),
            cv2.FONT_HERSHEY_PLAIN,
            text_scale,
            (0, 0, 255),
            thickness=2
        )

    for i, tlwh in enumerate(tlwhs):
        x1, y1, w, h = tlwh
        intbox = tuple(map(int, (x1, y1, x1 + w, y1 + h)))
        obj_id = int(obj_ids[i])
        color = get_color(abs(obj_id))

        cv2.rectangle(im, intbox[0:2], intbox[2:4], color=color, thickness=line_thickness)
        cv2.putText(im, str(obj_id), (intbox[0], intbox[1] - 12),
                    cv2.FONT_HERSHEY_PLAIN, text_scale, color, thickness=text_thickness)

        if scores is not None and show_text:
            cv2.putText(im, 'score: %.2f' % scores[i],
                        (intbox[0], intbox[1] - 24),
                        cv2.FONT_HERSHEY_PLAIN,
                        text_scale, color, thickness=text_thickness)

    return im
```

key lines:

cv2.rectangle and **cv2.putText**

these lines draw: bounding boxes, object ids and optionally scores. Also draws frame info using ‘frame: %d fps: %.2f num: %d’

Demo processing () – Main function of the script

This is the main function of the script, it works in conjunction with the orchestrator API, but more on that later.

How it works:

After loading the model

```
config = yaml_to_dict(config_path)
model = build_model(config)
load_checkpoint(model=model, path=model_path)
model.eval()
print("Model loaded.")
```

It creates an output folder

```
save_folder = out_dir
if with_timestamp:
    ts = time.strftime("%Y_%m_%d_%H_%M_%S", time.localtime())
    save_folder = os.path.join(save_folder, ts)
```

if **with_timestamp** is true, it makes a new folder with the current date and time, avoiding overwriting previous results.

```
csv_path = os.path.join(save_folder, "detections.csv")
csv_file = open(csv_path, "w", newline="")
csv_writer = csv.writer(csv_file)

csv_writer.writerow([
    "frame_id",
    "timestamp",
    "track_id",
    "x1", "y1", "w", "h",
    "score",
    "r_mean", "g_mean", "b_mean",
])
```

then it creates the csv file which will store one row per tracked object per frame. Each row contains the frame number, timestamp, object id, bounding box position, confidence score and mean RGB.

```
if os.path.isdir(input_path):
    frames = read_frames_from_folder(input_path)
    height, width, _ = frames[0].shape
else:
    cap = cv2.VideoCapture(input_path)
    frames = []
    while True:
        ret, frame = cap.read()
        if not ret:
            break
        frames.append(frame)
    fps = cap.get(cv2.CAP_PROP_FPS)
    height, width, _ = frames[0].shape
    cap.release()
```

The function loads the input, either a folder of images or a video file. Either way the result becomes a list of frames.

```
timer = Timer()
frame_id = 0
```

As mentioned previously, timer is used here to measure how long each frame takes to process. It also is later used to estimate the fps shown in the output video.
Frame id starts at 0 and increases by 1 for every frame so the script knows which frame its processing.

Initial track storage

```
tracks = [TrackInstances(
    hidden_dim=model.hidden_dim,
    num_classes=model.num_classes,
    state_dim=getattr(model.query_updater, "state_dim", 0),
    expand=getattr(model.query_updater, "expand", 0),
    num_layers=getattr(model.query_updater, "num_layers", 0),
    conv_dim=getattr(model.query_updater, "conv_dim", 0),
    use_dab=config["USE_DAB"]
).to("cuda")]
```

This creates an empty tracking container. It stores information about objects being tracked, such as their ids, boxes , scores, hidden features, and model state.
The values passed come in from the model in the first two lines, **hidden_dim** and **num_classes.**
.to("cuda") means this tracking data is moved to the GPU, because the model also runs there.

Runtime tracker

```
tracker = RuntimeTracker(
    det_score_thresh=0.45,
    track_score_thresh=0.35,
    miss_tolerance=60,
    use_motion=True,
    motion_min_length=2,
    motion_max_length=4,
    visualize=False,
    use_dab=config["USE_DAB"],
)
```

This creates the object tracker. Settings to note:
det_score_thres = a detection must have at least this confidence score to be considered
track_score_thresh = an existing track must stay above this score to continue being trusted (tracked)
miss_tolerance = if an object disappears for X amount of frames, the tracker can keep it alive, this helps with occlusion in certain situations like volleyball players crossing each other
use_motion = allows the tracker to use motion history, allowing the tracker to estimate

where an object might move based on previous frames

Disabling gradient tracking

```
with torch.no_grad():
```

This tells PyTorch that the model is only being used for prediction, not training. Making it faster and using less memory

Preprocessing and looping through frames

```
for frame in frames:

    image, ori_image = process_image(frame)
    frame_tensor = tensor_list_to_nested_tensor([image]).to("cuda")
```

image and **ori_image** are the same as explained previously.
frame_tensor = This wraps the image into the format expected by the model and moves it to the GPU

Running and timing the model

```
timer.tic()
frame_id += 1

res = model(frame=frame_tensor, tracks=tracks)
previous_tracks, new_tracks = tracker.update(
    model_outputs=res,
    tracks=tracks
)
```

timer.tic starts measuring processing time for this frame.
frameid +=1 increments each time a frame is processed, from frame 0 to 1 to 2 etc etc.
res = model(frame=frame_tensor, tracks= tracks)
This sends the current frame and existing track information into the model
the model uses the current image, previous tracking memory and existing tracks to predict object locations and ids for the current frame.

Update the tracker

```
previous_tracks, new_tracks = tracker.update(
    model_outputs=res,
    tracks=tracks
)
```

The tracker decides which detections match existing objects, which detections are new objects, which old tracks are missing and which tracks should remain active.
the output is split into previous tracks (are objects known from earlier frames) and new tracks (newly detected objects).

Postprocess tracking results and move them to CPU

```
tracks = model.postprocess_single_frame(
    previous_tracks, new_tracks, None, intervals=[1]
)
```

This combines and cleans the tracking results for the current frame. It updates tracks, which will be used again in the next frame. The tracking depends on memory, so the model needs to know what already existed before. This line prepares the tracking state for the next loop iteration

```
tracks_result = tracks[0].to("cpu")
```

since the model and tracker run on the GPU, but writing csv data and doing python processing is easier on the CPU, this line copies the current frames tracking results back to the CPU.

Scaling, filtering and converting bounding box areas

```
ori_h, ori_w = height, width
```

getting original image size to scale bounding boxes as they are output in normalized values

```
tracks_result.area = (
    tracks_result.boxes[:, 2] * ori_w *
    tracks_result.boxes[:, 3] * ori_h
)
```

Because the boxes are normalized, width and height are multiplied by ori_h and ori_w

```
tracks_result = filter_by_score(tracks_result)
tracks_result = filter_by_area(tracks_result)
```

filter out tracks by size and score (low score detectons + tiny boxes)

```
tracks_result.boxes = box_cxcywh_to_xyxy(tracks_result.boxes)
tracks_result.boxes *= torch.tensor([ori_w, ori_h, ori_w, ori_h], dtype=torch.float)
```

converts boxes from centre x and centre y to x1,y1,x2,y2

scales boxes to real pixel coordinates by multiplying boxes by image original sizes.

```
online_tlwhs, online_ids, online_scores = [], [], []
```

prepares lists for drawing

these lists collect data needed by plot_tracking: **online tlwhs (top left x and y, width, height format)**, online_ids: object ids and finally online_confidence.

Looping through each detected/tracked object

```
for i in range(len(tracks_result)):
    x1, y1, x2, y2 = tracks_result.boxes[i].tolist()
    w, h = x2 - x1, y2 - y1
```

boxes get converted to x1, y1, w, h for **plot_tracking()**

```
track_id = tracks_result.ids[i].item()
score = torch.max(tracks_result.scores[i]).item()
```

extracting **track_id** and score. **Track_id** is the unique object ID assigned by the tracker

```
# Sample mean RGB from the bbox region on the original frame
r_mean, g_mean, b_mean = get_mean_rgb(ori_image, x1, y1, w, h)
```

take RGB colour from middle of the bounding box. The processed image was resized and normalized so it's not good for RGB analysis

```
online_tlwhs.append([x1, y1, w, h])
online_ids.append(track_id)
online_scores.append(score)
```

store data for drawing

```
csv_writer.writerow([
    frame_id,
    round(timestamp, 4),
    track_id,
    round(x1, 2), round(y1, 2),
    round(w, 2), round(h, 2),
    round(score, 4),
    r_mean, g_mean, b_mean,   # <-- RGB added
])
```

write detection to csv

```
timer.toc()
```

Stop timing

```
if len(online_tlwhs) > 0:
    online_im = plot_tracking(
        ori_image,
        online_tlwhs,
        online_ids,
        scores=online_scores,
        frame_id=frame_id,
        fps=1. / max(timer.average_time, 1e-6),
        show_text=show_text
    )
else:
    online_im = ori_image

vid_writer.write(online_im)
```

If there is tracked objects, draw them, or if none are detected just use the original image

Close video and CSV and copy CSV to pipeline output folder

```
vid_writer.release()
csv_file.close()
print(f"CSV written to: {csv_path}")

pipeline2_dir = "/outputs/pipeline2"
os.makedirs(pipeline2_dir, exist_ok=True)
dest_csv_path = os.path.join(pipeline2_dir, "detections.csv")
shutil.copy(csv_path, dest_csv_path)
print(f"CSV copied to: {dest_csv_path}")
return save_path
```

It is important that the writer and csv file is closed as they may not save properly if they aren't closed.

Finally copy the csv to pipeline output folder. This copies them to a hardcoded location **/outputs/pipeline2** which makes the Api access them easier.

```
# Remove the raw video output (only CSV is needed)
if os.path.exists(output_path):
    os.remove(output_path)
```

The returned video path is deleted because only the csv is needed

/Process endpoint for pipeline2

The process endpoint for pipeline2 is much the same as pipeline1, with the same code snippet for making the relevant directories

```
upload_dir = Path("/tmp/uploads")
upload_dir.mkdir(parents=True, exist_ok=True)

video_path = upload_dir / f"{datetime.now().timestamp()}_{video.filename}"

with open(video_path, "wb") as f:
    f.write(await video.read())

output_dir = Path("/tmp/outputs/")
output_dir.mkdir(parents=True, exist_ok=True)
```

The most important part is the command

```
cmd = [
    "python","-u", "/app/demo/demo.py",
    str(video_path),
    str(output_dir),
    CONFIG_PATH,
    MODEL_PATH
]
```

This launches the tracking script

"python -u /app/demo/demo.py <video_path> <output_dir> <config_path> <model_path>"

These arguments match the argparse section in demo.py

```
parser = argparse.ArgumentParser()
parser.add_argument('in_video_path')
parser.add_argument('output_dir')
parser.add_argument('config_path')
parser.add_argument('model_path')
parser.add_argument('--fps', type=int, default=30)
parser.add_argument('--no_timestamp', action='store_true')
parser.add_argument('--no_text', action='store_true')
parser.add_argument('--player_id', type=int, default=None)
```

Then the command is started as a subprocess

```
process = subprocess.Popen(cmd,stdout=subprocess.PIPE,stderr=subprocess.STDOUT, text=True, bufsize=1)
```

This means the API starts another python program from inside of itself. Stdout captures the output from demo.py, and stderr combines errors with normal logs for debugging.

This loop prints the tracking script logs live, then waits for the process to finish

```
for line in process.stdout:
    print(line, end="", flush=True)

exit_code = process.wait()
print(f"\nProcess has stopped here (exitcode {exit_code})")
```

Then, it assumes these files exist, and finally returns a JSON response with the csv and video paths

```
# Expected output files
output_csv = output_dir / "detections.csv"
output_video = output_dir / "output_compressed.mp4"

return JSONResponse(content={
    "status": "success",
    "csv_path": str(output_csv),
    "video_path": str(output_video)
})
```

A quick summary of this, it’s similar to the **pipeline1 /process function** except it calls another python script inside of the request as a subprocess, and waits for that to finish.

5.3 API orchestrator

The API orchestrator is a python-based REST API (a rest Api is an application program interface architectural style that follows design principles of REpesentational State Transfer) using the FastAPI framework. It acts as the coordinator between the frontend and pipelines, receiving a video, assigns it a unique job ID, stores it temporarily, sends it to both services (pipelines) and retrieves the results.

App creation, setup and configuration

```
app = FastAPI(title="SR Analytics - Orchestrator API")

# Add CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

PIPELINE1_URL = os.getenv("PIPELINE1_URL", "http://localhost:5000")
PIPELINE2_URL = os.getenv("PIPELINE2_URL", "http://localhost:5001")

jobs = {}
```

The first line here makes it a web API that clients (in this case the frontend) can interact with. It is designed to be asynchronous meaning it doesn't block other actions while the processing happens.

The API allows requests from any frontend using CORS. This means any client can all the API.

The pipeline services are configured using environment variables, allowing flexibility to be run locally, on remote server, and in this case Docker containers

The orchestrator keeps tracks of jobs in memory using jobs = {}.

Each job looks like this {
status: “processing” or “error”
“pipeline1”: None,
“pipeline2”: None,
}
when the job status is done, pipeline1 and 2 should return the paths to the processed CSV. The next section talks about why jobs are used

Video upload and Job creation

```
@app.post("/process-video")
async def process_video(background_tasks: BackgroundTasks, video: UploadFile = File(...)):
    """Submit video for background processing by both pipelines."""
    job_id = str(uuid.uuid4())
    jobs[job_id] = {"status": "processing", "pipeline1": None, "pipeline2": None}
```

Creating the /process-video endpoint as an entry point. This endpoint takes in a video file, and then a unique **job_id** is created using **uuid** (is a function imported from the system allowing a 128 bit number to generated and assigned to identify information, in this case the job id)

the job is then initialised.

```
upload_dir = Path("/tmp/uploads")
upload_dir.mkdir(parents=True, exist_ok=True)
video_path = upload_dir / f"{job_id}_{video.filename}"
with open(video_path, "wb") as f:
    f.write(await video.read())
```

This code snippet then creates a upload directory, a unique file path and finally writes the file to disk.

```
background_tasks.add_task(run_pipelines, video_path, job_id, video.filename)
return {"job_id": job_id}
```

Instead of the processing happening immediately, a background task is created, then the Api responds instantly and allows the client to continue without waiting for processing to finish. This was an issue during development where the request would timeout and not wait for the pipelines to process.

Running the pipelines

```
def run_pipelines(video_path, job_id, orig_filename):
    async def process():
        timeout = aiohttp.ClientTimeout(total=None)
        async with aiohttp.ClientSession(timeout=timeout) as session:
            with open(video_path, "rb") as f:
                video_bytes = f.read()
```

Inside of this an async function sends the video to both pipelines.

The video is read using **with open(video_path)**

Requesting pipeline2

```
p2_form = aiohttp.FormData()
p2_form.add_field("video", video_bytes, filename=orig_filename, content_type="video/mp4")
```

This creates a form object, like how a browser sends a file upload. Then video field gets added, with **video_bytes** as the file content, filename is the original filename, and content type tells the server it's a video in the mp4 format

```
p2_resp = await session.post(f"{PIPELINE2_URL}/process", data=p2_form)
```

sends a post request to pipeline2, using the URL parameter set previously. This makes the post request send to http://localhost:5001/process. Await, waits for the response and data=p2_form sends the video as multipart/form-data

Response handling

```
if p2_resp.status == 200:
    p2_result = await p2_resp.json()
    jobs[job_id]["pipeline2"] = p2_result
else:
    jobs[job_id]["status"] = "error"
    return
```

If the response status is 200 or "OK"
then it awaits the JSON response from the pipeline, and stores it in the job dictionary
If it fails, stop processing completely.

Pipeline1 Repeats the same pattern for sending the video, with changes to the pipeline URL which is now http://localhost:5000.

These pipelines run sequentially, but with more resources like ram and storage, they can be run in parallel.

Design choices

A few points to touch on here.
Sending bytes instead of a file path, for scale. Pipelines might later run on different machines, which then wouldn't be able to access the local file system, so sending the file over HTTP is essential.

Using aiohttp FormData makes sure it works with FastAPI endpoints.

Endpoint for downloading csv files

```
@app.get("/download-csv/{filename}")
async def download_csv(filename: str):
    try:
        search_dirs = [
            Path("/tmp/pipeline1_outputs"),
            Path("/tmp/pipeline2_outputs"),
        ]
        candidates = []
        for d in search_dirs:
            if d.exists():
                candidates += [f for f in d.glob(f"*{filename}") if f.name.endswith(filename)]
        if not candidates:
            raise HTTPException(status_code=404, detail=f"File {filename} not found")
        # Pick the most recently created file
        latest = max(candidates, key=lambda f: f.stat().st_ctime)
        print(f"[download-csv] Serving: {latest}")
        return FileResponse(
            path=latest,
            media_type="text/csv",
            filename=filename
        )
    except HTTPException:
        raise
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

When a request is made, the function looks in both pipeline output folders to find files that match the requested name. It collects all the matching candidates, if none are found, returns a 404 (not found). If multiple candidates are found, it selects the most recently created file to ensure the latest tracked results are returned. The selected files are then sent back using **FileResponse**, which sends the CSV to the frontend with the filetype. This allows the frontend to request a filename without worrying about where it came from or where its stored and making sure the most up to date file is sent.

5.4 Frontend

This section will cover the frontend, which is built with React.js and Vite, served by Nginx. The frontend is responsible for user interaction and displaying the video, statistics, and calculation logic.

Following a classic React structure, it uses **App.jsx** as the central piece, managing state and data flow, with smaller components like **VideoPanel**, **StatsPanel** and **UploadRow** handle specific UI pieces and utility functions in **Utils.js** provide important logic such as CSV parsing and data grouping. This structure keeps the code organised. AI was used to assist in creating the layout and some functionality such as the parsing and perfect zone drawing.

Each of these pieces will be broken down, starting with:

5.4.1 Utils.js

Utils.js provides helper function for handling CSV data, formatting timestamps and organising detected plays. It helps the frontend transform raw data into a format that components can easily use.

```
import Papa from "papaparse";
```

Importing Papaparse, a library known for speed and reliability of handling large CSV files. It turns the CSV rows into JavaScript objects

Parsing CSV files

```
//uses papaparse libraru to parse the csv file.
export function parseCsv(file) {
  return new Promise((resolve, reject) => {
    Papa.parse(file, {
      header: true,
      skipEmptyLines: true,
      dynamicTyping: true,
      complete: (r) => resolve(r.data),
      error: reject,
    });
  });
}
```

What is a promise and why use it?

Papaparse is callback-based (but this app uses async/await) so using a promise (an object that represents a value that will exist in the future, it can be either resolved by a value or rejected with an error) allows the app to await the parsedCSV files without any error.

Grouping By Action

```
export function groupByAction(rows) {
  const groups = {};
  for (const row of rows) {
    const key = (row.action || "other").toString().toLowerCase().trim();
    if (!groups[key]) groups[key] = [];
    groups[key].push(row);
  }
  for (const key of Object.keys(groups)) {
    groups[key].sort((a, b) => (a.timestamp_s ?? 0) - (b.timestamp_s ?? 0));
  }
  return groups;
}
```

creates groups for each action. then it loops through each row of the input data and assigns it to a group based on the "**action**" property. if the action is missing or empty, it assigns the row to the "other". after grouping, it sorts each group by the "**timestamp_s**" property in ascending order and then returns the groups object. the groups object is a dictionary where the keys are the action types, and the values are arrays of rows that belong to that action type.

5.4.2 App.jsx

Imports and components

```
import { useState, useRef } from "react";
import { useEffect } from "react";
import { parseCsv, groupByAction } from "./utils";
import TopBar from "./components/TopBar";
import VideoPanel from "./components/VideoPanel";
import UploadRow from "./components/UploadRow";
import StatsPanel from "./components/StatsPanel";

function Notification({ message, onClose }) {
  useEffect(() => {
    const timer = setTimeout(onClose, 4000);
    return () => clearTimeout(timer);
  }, [onClose]);
  return (
    <div style={{
      position: "fixed",
      top: 20,
      right: 20,
      background: "#222",
      color: "#fff",
      padding: "16px 24px",
      borderRadius: 8,
      boxShadow: "0 2px 8px rgba(0,0,0,0.2)",
      zIndex: 1000,
      fontSize: 16
    }}>
      {message}
    </div>
  );
}
```

This code snippet sets up the tools and UI pieces used by the app. The react hooks (useState, useRef and useEffect) allow the app to store data, access the video player and run side effects.
The helper functions parseCSV and groupByAction are imported from utils.js and used after the backend processing finishes.
parseCSV turns downloaded csv files into usable JavaScript objects and groupByAction organises detected plays by action type.
The imported components split the interface into smaller parts.
Topbar: shows summary information
VideoPanel: displays the video
UploadRow: handles upload and process controls
StatsPanel: shows results

The notification component is a small temporary popup which was added after user testing. More on that later. It receives a message and onClose function. When it appears, useEffect starts a timer using setTimeout. After four seconds, onClose runs

and hides the notification. ClearTimout(timer) helps prevent bugs if the notification disappears before the timer finishes.

Main state inside APP

```
export default function App() {
  const videoRef = useRef(null);

  const [videoFile, setVideoFile] = useState(null);
  const [videoUrl, setVideoUrl] = useState(null);
  const [plays, setPlays] = useState([]);
  const [groups, setGroups] = useState({});
  const [openGroups, setOpenGroups] = useState({});
  const [activeFilter, setActiveFilter] = useState("all");
  const [activeFrame, setActiveFrame] = useState(null);
  const [loading, setLoading] = useState(false);
  const [activePlayer, setActivePlayer] = useState("all");
  const [rawDetections, setRawDetections] = useState([]);
  const [rawBallPositions, setRawBallPositions] = useState([]);

  // Zone state lives here so VideoPanel and StatsPanel share it
  const [zone, setZone] = useState(null);
  const [drawingZone, setDrawingZone] = useState(false);
  const [tempZone, setTempZone] = useState(null);

  const [showNotification, setShowNotification] = useState(false);
  const canProcess = videoFile && !loading;
```

This is the state for the whole frontend. State is data that can change over time. Each variable has its own state.

videoRef points to the video player element, which lets the app control the playback of the video manually (later used in the seek function)
videoFile stores the actual uploaded file, while videoURL stores a temporary browser preview URL so the user can watch the video before or during processing.
“plays” stores detected events from the processed CSV file and “groups” stores those events by action type (pass, set, reception) openGroups remembers which grouped sections are expanded or collapsed.

activeFilter, activeFrame and activePlayer control what the user is currently focusing on. For example, activeFrame highlights the selected frame.

rawDetections and rawBallPostions store CSV data from the pipelines. They are used for overlays, statistics and zone calculations.
The perfect pass zone or “zone” related states are kept in APP because both videopanel and statspanel need to access them
zone is the completed selected area, drawingZone tracks whether the uses is currently

drawing, and tempZone stores the temporary shape while drawing
canProcess only equals to true when a video has been selected and the app is not already processing, stopping double processing overloading the pipelines with requests.

Uploading a video and seeking to events

```
function handleVideoFile(file) {
  setVideoUrl((prev) => {
    if (prev) URL.revokeObjectURL(prev);
    return URL.createObjectURL(file);
  });
  setVideoFile(file);
}

function handleSeek(timestamp_s, frame) {
  if (videoRef.current && timestamp_s != null) {
    videoRef.current.currentTime = parseFloat(timestamp_s);
    videoRef.current.pause();
  }
  setActiveFrame(frame);
}
```

handleVideoFile is run when the user selects a video. Before creating a temporary URL, it checks whether one already exists and revokes it. It then creates a temporary URL, which lets the browser display a video without uploading it anywhere. This is important because objectUrls use browser memory.

After creating the preview URL, the function stores the file in videoFile. This file is sent to the backend which is explained in the next step.
handleSeek function is used when the user clicks on a detected play. If the video player exists and the timestamp is valid, it sets the videos current time using videoRef.current.currentTime. Then it pauses the video so the user can inspect that moment, along with updating activeFrame.

Sending video to the backend

```
async function handleProcess() {
  if (!canProcess) return;
  setShowNotification(true);
  setLoading(true);
  try {
    const form = new FormData();
    form.append("video", videoFile);

    const res = await fetch("http://localhost:8000/process-video", {
      method: "POST",
      body: form,
    });
    if (!res.ok) throw new Error("Failed to submit video");
    const { job_id } = await res.json();

    let jobStatus = null;
    for (;;) {
      const statusRes = await fetch(`http://localhost:8000/status/${job_id}`);
      if (!statusRes.ok) throw new Error("Failed to get job status");
      jobStatus = await statusRes.json();
      if (jobStatus.status === "done") break;
      if (jobStatus.status === "error") throw new Error("Processing failed");
      await new Promise((r) => setTimeout(r, 3000));
    }
```

This function is the processing function.
It first checks canProcess. If no video is selected or the app is already loading it stops.
Then it shows a notification and sets loading to true, which disables the process button and shows the user that processing is happening.
The video is packed into FormData under the field name “video”, matching what the FastAPI backend expects. The app then sends a POST request to the /process-video endpoint, if the file matches what is expected, the backend responds with a job_id. As mentioned previously, this is because the processing takes a long time and the frontend previously timed out due to this.
Afte that, the frontend uses a for loop to poll for the job status. If the status is done the loop breaks and the app moves on. Otherwise, it polls every three seconds using setTimeout wrapped in a promise. Lastly, it returns error if the status is error.
This keeps the frontend responsive while the pipelines process the users video.

Downloading and parsing outputs

```
const pipeline1 = jobStatus.pipeline1 || {};
const pipeline2 = jobStatus.pipeline2 || {};
const csvListData = {
  files: [
    pipeline2.csv_path,
    pipeline1.positions_csv_path,
    pipeline1.detected_plays_csv_path,
  ].filter(Boolean),
};

const allPlays = [];
csvListData.files = csvListData.files.map((f) => f.split("/").pop());
for (const filename of csvListData.files) {
  try {
    const csvRes = await fetch(
      `http://localhost:8000/download-csv/${filename}`
    );
    const csvBlob = await csvRes.blob();
    const parsed = await parseCsv(csvBlob);
    if (filename.includes("detections")) {
      setRawDetections(parsed);
    } else if (filename.includes("positions")) {
      setRawBallPositions(parsed);
    } else if (filename.includes("detected_plays")) {
      allPlays.push(...parsed);
    }
  } catch (err) {
    console.warn(`Failed to parse ${filename}:`, err);
  }
}

const grps = groupByAction(allPlays);
setPlays(allPlays);
setGroups(grps);
const open = {};
for (const key of Object.keys(grps)) open[key] = true;
setOpenGroups(open);
setActiveFilter("all");
} catch (err) {
console.error("Error processing files:", err);
alert("There was an error processing the files. Please try again.");
} finally {
setLoading(false);
}
}
```

Once the job is done, the frontend extracts the pipeline results. Extracting pipeline outputs from jobStatus, if either one is missing it uses an empty object instead of just crashing.

Then it creates a list of CSV paths from the results, and .filterBoolean removes any missing values like “null” or “undefined”.

allPlays holds all the detected play events, and then the code converts the full file paths

into just filenames because the download endpoint expects only that. The code uses f.split("/").pop(). The string get split and at the last available forward slash.
The app then loops over every CSV filename, downloads it from the endpoint, converts the response into a blob (short for binary large object, is a way of representing raw file data in the browser) and passes that into parseCSV. Based on the filename, the parsed data is stored in different places. Detection data goes into RawDetections, ball position goes into rawBallPositions and detected play rows go into allPlays.

After all CSVs are loaded, it groups the detected plays. Then it opens all groups by default.
Since the app only tracks detections, this only opens that group.

Final UI rendering

```
return (
  <div className="app">
    {showNotification && (
      <Notification
        message="Processing may take some time. Please wait..."
        onClose={() => setShowNotification(false)}
      />
    )}
    <TopBar totalEvents={plays.length} />

    <div className="workspace">
      <div className="left-col">
        <VideoPanel
          videoRef={videoRef}
          videoUrl={videoUrl}
          videoFile={videoFile}
          zone={zone}
          setZone={setZone}
          drawingZone={drawingZone}
          setDrawingZone={setDrawingZone}
          tempZone={tempZone}
          setTempZone={setTempZone}
        />
        <UploadRow
          videoFile={videoFile}
          loading={loading}
          canProcess={canProcess}
          onVideoFile={handleVideoFile}
          onProcess={handleProcess}
        />
      </div>

      <StatsPanel
        detectedPlays={plays}
        groups={groups}
        openGroups={openGroups}
        activeFilter={activeFilter}
        activeFrame={activeFrame}
        totalEvents={plays.length}
        onToggleGroup={handleToggleGroup}
        onFilterChange={setActiveFilter}
        onSeek={handleSeek}
        activePlayer={activePlayer}
        onPlayerFilter={setActivePlayer}
        detections={rawDetections}
        ballPositions={rawBallPositions}
        zone={zone}
        setZone={setZone}
        drawingZone={drawingZone}
        setDrawingZone={setDrawingZone}
        tempZone={tempZone}
        setTempZone={setTempZone}
      />
    </div>
  </div>
```

This render section brings everything together. If showNotification is true, the notification appears. This notification comes up whenever the user clicks process for a video. This was also added after user testing.
Topbar displays the total number of detected events
The layout is split into a left column and a stats area
Videopanel gets the video preview plus zone drawing state
UploadRow gets the selected video, loading state, callback functions for processing and uploading.
Statspanel receives the parsed results, filters, selected frame, player filter, raw detections, ball positions and zone data.
App acts a parent controller, it has all the important states, performs backend communications and passes down data to child components.

5.4.3 Components

This section introduces the components which app.jsx uses to render the User Interface

Videopanel
This component is responsible for showing the uploaded video and allowing the user to draw a zone/region on top of the video. It receives most of its important data as props (properties) from App.jsx (which also controls the state) while videoPanel handles visual interactions.

Receiving props

```
export default function VideoPanel({
  videoRef,
  videoUrl,
  videoFile,
  zone,
  setZone,
  drawingZone,
  setDrawingZone,
  tempZone,
  setTempZone,
}) {
  const overlayRef = useRef(null);
```

The most important are videoRef, which allows control of the video element, and videoURL, which is used to display the video.
The zone related props control how the drawing system behaves.
OverlayRef is a transparent layer placed over the video to capture mouse input

Coordinate Scaling

```
const getScaledCoords = (clientX, clientY) => {
  const rect = overlayRef.current.getBoundingClientRect();
  const renderedW = rect.width;
  const renderedH = rect.height;
  const videoEl = videoRef.current;
  const scaleX = videoEl ? videoEl.videoWidth / renderedW : 1;
  const scaleY = videoEl ? videoEl.videoHeight / renderedH : 1;
  return {
    x: (clientX - rect.left) * scaleX,
    y: (clientY - rect.top) * scaleY,
  };
};
```

This function converts mouse positions from screen space to video pixel coordinates. The displayed video size might not match the resolution, this scaling ensures the zone corresponds to the original video data. The browser gives positions relative to the whole page, not the video.

The function first gets the size and position of the video overlay, then calculates how much the displayed video has been scaled compared to its real resolution using videoWidth and Height. It then uses these scale factors to adjust mouse positions so it matches the original videos coordinate system. Subtracting the overlays top left positions and multiplying by the scale, this ensures the zone corresponds to the actual pixels in the video.

Drawing the “perfect” pass zone

```
const handleMouseDown = (e) => {
  if (!drawingZone) return;
  e.preventDefault();
  const { x, y } = getScaledCoords(e.clientX, e.clientY);
  setTempZone({ x1: x, y1: y, x2: x, y2: y });
};

const handleMouseMove = (e) => {
  if (!drawingZone || !tempZone) return;
  const { x, y } = getScaledCoords(e.clientX, e.clientY);
  setTempZone((prev) => ({ ...prev, x2: x, y2: y }));
};

const handleMouseUp = (e) => {
  if (!drawingZone || !tempZone) return;
  const { x, y } = getScaledCoords(e.clientX, e.clientY);
  const finalZone = { ...tempZone, x2: x, y2: y };
  setZone(finalZone);
  setTempZone(null);
  setDrawingZone(false);
};
```

This happens in three stages.
First, when the user clicks it starts a new rectangle by setting the starting point.
Second, when the user drags the mouse, the rectangle updates dynamically.
Lastly, when the zone is released, the zone is finalised and saved.

Reversing the scaling back to display coordinates

```
// Convert scaled (video pixel space) zone coords back to overlay display coords
const toDisplayCoords = (zone) => {
  if (!overlayRef.current || !videoRef.current) return null;
  const rect = overlayRef.current.getBoundingClientRect();
  const videoEl = videoRef.current;
  const scaleX = rect.width / (videoEl.videoWidth || rect.width);
  const scaleY = rect.height / (videoEl.videoHeight || rect.height);
  return {
    x1: zone.x1 * scaleX,
    y1: zone.y1 * scaleY,
    x2: zone.x2 * scaleX,
    y2: zone.y2 * scaleY,
  };
};
```

Zones are stored in real video coordinates, but to display them visually they must be converted back to screen coordinates.

Managing temporary vs final zones

```
const activeZoneRaw = tempZone || zone;
const activeZone = activeZoneRaw ? toDisplayCoords(activeZoneRaw) : null;

// Store display coords for confirmed zone in state to avoid accessing refs during render
const [confirmedZoneDisplay, setConfirmedZoneDisplay] = useState(null);
```

Determines what should be displayed tempzone vs final zones. This lets the rectangle appear as its being drawn and remain on screen after.

Rendering video and overlay

```
{videoUrl ? (
  <>
    <video
      ref={videoRef}
      src={videoUrl}
      style={{ width: "100%", display: "block" }}
      controls={!drawingZone}
    />


    <div
      ref={overlayRef}
      style={{
        position: "absolute",
        top: 0,
        left: 0,
        width: "100%",
        height: "100%",
        cursor: drawingZone ? "crosshair" : "default",
        zIndex: 2,
        pointerEvents: drawingZone ? "all" : "none",
      }}
      onMouseDown={handleMouseDown}
      onMouseMove={handleMouseMove}
      onMouseUp={handleMouseUp}
    >
```

The video is displayed using the uploaded file, and controls are disabled during drawing so the mouse can be used for selecting a zone instead of interacting with playback Then the overlay is placed on top, and it captures mouse events only when the drawing mode is active. When drawing is off, pointerEvents : none allows normal video interactions like play and pause.

Drawing the perfect pass rectangle

```
{confirmedZoneDisplay && (
  <div
    style={{
      position: "absolute",
      left: `${Math.min(confirmedZoneDisplay.x1, confirmedZoneDisplay.x2)}px`,
      top: `${Math.min(confirmedZoneDisplay.y1, confirmedZoneDisplay.y2)}px`,
      width: `${Math.abs(confirmedZoneDisplay.x2 - confirmedZoneDisplay.x1)}px`,
      height: `${Math.abs(confirmedZoneDisplay.y2 - confirmedZoneDisplay.y1)}px`,
      border: "2px dashed #f87171",
      background: "rgba(252, 165, 165, 0.12)",
      pointerEvents: "none",
    }}
  />
)}
```

This visually renders the selected zone, using Math.min and math.abs ensures the rectangle is correct regardless of drag direction.

Fallback UI

```
  <div className="video-placeholder">
    <div className="vp-icon"></div>
    <div className="vp-text" style={{color: "#fff"}}>Upload a match video using the
    button below <br /> Using a video with the camera angle behind your team works
    best!</div>
  </div>
)}
```

just a placeholder with text when there is no video uploaded.

Statspanel.jsx

Is responsible for showing the reception analytics results. It takes data from the backend CSV files, matches receptions to player detection using jersey colour and ball distance, chekcs whether the ball is inside the selected “perfect” zone and displays the final filtered list in the UI

Matching RGB

```
function matchesRgb(r, g, b, userRGB, tolerance) {
  return (
    Math.abs(r - userRGB.r) <= tolerance &&
    Math.abs(g - userRGB.g) <= tolerance &&
    Math.abs(b - userRGB.b) <= tolerance
  );
}
```

This helper function checks whether a detected players average rgb is close enough to the users chosen jersey colour. The tolerance is used to measure a range from the user inputted value, aiding in detection matching.

Checking if the ball positon is inside the zone

```
// Helper: check if point is in zone (zone coords are in video pixel space)
function isInZone(ball_x, ball_y, zone) {
  if (!zone) return false;
  const { x1, y1, x2, y2 } = zone;
  const minX = Math.min(x1, x2),
    maxX = Math.max(x1, x2);
  const minY = Math.min(y1, y2),
    maxY = Math.max(y1, y2);
  return ball_x >= minX && ball_x <= maxX && ball_y >= minY && ball_y <= maxY;
}
```

The zone is stored in pixel coordinates, so the ball coordinates can be compared directly.

Grouping receptions if they are close to each other frame wise

```
function groupConsecutiveReceptions(receptions, frameGap = 1) {
  if (!receptions.length) return [];
  // Sort by frame just in case
  const sorted = [...receptions].sort((a, b) => a.frame - b.frame);
  const groups = [];
  let current = {
    ...sorted[0],
    frames: [sorted[0].frame],
    startFrame: sorted[0].frame,
    endFrame: sorted[0].frame,
  };

  for (let i = 1; i < sorted.length; i++) {
    const rec = sorted[i];
    if (rec.frame <= current.endFrame + frameGap) {
      // Extend current group
      current.endFrame = rec.frame;
      current.frames.push(rec.frame);
    } else {
      groups.push({ ...current });
      current = {
        ...rec,
        frames: [rec.frame],
        startFrame: rec.frame,
        endFrame: rec.frame,
      };
    }
  }
  groups.push({ ...current });
  return groups;
}
```

This groups receptions that happen across consecutive frames into a single event. This was added because the same reception appeared multiple times in the stats panel causing clutter.

useMatchedReceptions function

```
function useMatchedReceptions({
  detectedPlays,
  detections,
  ballPositions,
  userRGB,
  tolerance,
  distanceThreshold,
  perfectZone,
}) {
```

It uses useMemo to avoid recalculating everything unless the data input changes.
The detections and ball positions are grouped by frame.

```
const detByFrame = useMemo(() => {
  const map = new Map();
  for (const d of detections) {
    if (!map.has(d.frame_id)) map.set(d.frame_id, []);
    map.get(d.frame_id).push(d);
  }
  return map;
}, [detections]);

const ballByFrame = useMemo(() => {
  const map = new Map();
  for (const b of ballPositions) {
    if (!map.has(b.frame)) map.set(b.frame, { x: b.x, y: b.y });
  }
  return map;
}, [ballPositions]);
```

This makes matching faster because the code can immediately lookup detections and ball locations for a specific frame

Core matching algorithm

Step by step, this algorithm filters raw detections into meaningful reception events. It runs inside of useMemo, as mentioned previously.

```
if (!Array.isArray(detectedPlays)) return [];
const results = [];
```

starts by checking if the input is valid, if not return an empty result.
intialises an empty array for results.

Loop through plays

```
for (const play of detectedPlays) {
  if (play.action !== "reception") continue;
```

only keeps those with the label “reception”

Get ball positions

```
const frame = play.frame;
const ballX = play.ball_x ?? ballByFrame.get(frame)?.x;
const ballY = play.ball_y ?? ballByFrame.get(frame)?.y;
if (ballX == null || ballY == null) continue;
```

this makes sure every matched reception has a valid ball position.
For each reception, gets frame number, ball position from the play or from ballByFrame, if none exist, skip.

Get detections in that frame and find nearest player to ball

```
const frameDets = detByFrame.get(frame) ?? [];
let nearest = null,
  nearestDist = Infinity;
for (const d of frameDets) {
  const cx = d.x1 + d.w / 2,
    cy = d.y1 + d.h / 2;
  const dist = Math.sqrt((ballX - cx) ** 2 + (ballY - cy) ** 2);
  if (dist < nearestDist) {
    nearestDist = dist;
    nearest = d;
  }
}
```

Each detection is treated as a player
The bounding box is converted to a centre point. The distance from the ball to that player is then calculated using the Euclidian distance formula.
This assumes the closest player to the ball is the receiver.

Filtering rules and check if reception is perfect

```
if (!nearest) continue;
if (nearestDist > distanceThreshold) continue;
if (
  !matchesRgb(
    nearest.r_mean,
    nearest.g_mean,
    nearest.b_mean,
    userRGB,
    tolerance,
  )
)
  continue;
```

If no player is found, skip. If the player is too far, skip.
If the players jersey RGB doesn't match, skip.

```
const perfect = isInZone(ballX, ballY, perfectZone);
```

checks if the pass is perfect.

Store result

```
    results.push({
      ...play,
      frame,
      timestamp_s: play.timestamp_s,
      track_id: nearest.track_id,
      r_mean: nearest.r_mean,
      g_mean: nearest.g_mean,
      b_mean: nearest.b_mean,
      ball_x: ballX,
      ball_y: ballY,
      distance_px: nearestDist.toFixed(2),
      confidence: play.confidence,
      perfect,
    });
  }
  return results;
```

A new object is created combining original play data, matched player tracking info, ball position, distance from player to ball and whether it was perfect.
Return results.

Dependencies

```
}, [
  detectedPlays,
  detByFrame,
  ballByFrame,
  userRGB,
  tolerance,
  distanceThreshold,
  perfectZone,
]);
```

This function only reruns when these values change.

Zone notification function

```
// Notification popup component
function ZoneNotification({ visible }) {
  if (!visible) return null;
  return (
    <div
      style={{
        position: "fixed",
        top: 24,
        right: 24,
        background: "#222",
        color: "#fff",
        padding: "14px 22px",
        borderRadius: 8,
        boxShadow: "0 2px 8px rgba(0,0,0,0.2)",
        zIndex: 2000,
        fontSize: 16,
      }}
    >
      Drawing perfect zone<br />
      Click and drag on the video to draw your zone.<br />
      A rectangle with a red border will appear as you draw.<br />
      It is reccomnded draw the zone near the top of the net, giving space for the ball to trave
    </div>
  );
}
```

This function gives the users instructions on how to draw the zone

Drawing perfect zone
Click and drag on the video to draw your zone.
A rectangle with a red border will appear as you draw.
It is reccomnded draw the zone near the top of the net, giving space for the ball to travel into it after the reception.

StatsPanel state management and Data preparation

This block does three things:

sets user-controlled state (userRGB, Tolerance, etc)

prepares processed data (matchedReceptions, groupedReceptions)

defines how each row is rendered

```
export default function StatsPanel({
  detectedPlays = [],
  detections = [],
  ballPositions = [],
  openGroups,
  activeFrame,
  onToggleGroup,
  onSeek,
  zone,
  drawingZone,
  setDrawingZone,
}) {
```

The component receives data and functions from App.jsx**. detectedPlays** contain play event, detections contain player bounding boxes and RGB values and **ballPositions** contains ball coordinates. It receives UI controls such as **onSeek** to let the user jump to a play and zone related props.

Local settings defined by user controlled inside of StatsPanel

```
const [userRGB, setUserRGB] = useState({ r: 120, g: 120, b: 120 });
const [tolerance, setTolerance] = useState(30);
const [distanceThreshold, setDistanceThreshold] = useState(50);
const [showColorInput, setShowColorInput] = useState(false);
```

stores users jersey colour in **userRGB**

tolerance

distance (how close the ball needs to be for the player to be considered a passer)

and **showColourInput** controls whether the panel is open.

Calling custom matching hook

```
const matchedReceptions = useMatchedReceptions({
  detectedPlays,
  detections,
  ballPositions,
  userRGB,
  tolerance,
  distanceThreshold,
  perfectZone: zone,
});
```

It combines play data, player detections, ball positions, jersey colour settings, distance threshold, and the selected zone. The result is a filtered list of receptions that match the chosen jersey colour and are close enough to the ball.

Groups receptions and calculating perfect and total count of receptions

```
// Group consecutive receptions
const groupedReceptions = groupConsecutiveReceptions(matchedReceptions);

const perfectCount = groupedReceptions.filter((r) => r.perfect).length;
const totalCount = groupedReceptions.length;
```

Render row

```
// Custom row renderer: just show "Reception" and perfect label if needed
const renderRow = (row) => (
  <div
    style={{
      display: "flex",
      alignItems: "center",
      background: row.perfect ? "rgba(34,197,94,0.10)" : "transparent",
      borderLeft: row.perfect ? "4px solid #22c55e" : "4px solid transparent",
      paddingLeft: 8,
      marginBottom: 2,
    }}
  >
    <span>Reception</span>
    {row.perfect && (
      <span
        style={{
          background: "#dcfce7",
          color: "#16a34a",
          borderRadius: "4px",
          padding: "2px 8px",
          fontWeight: 600,
          marginLeft: 10,
          fontSize: 12,
          border: "1px solid #bbf7d0",
        }}
      >
        Perfect
      </span>
    )}
  </div>
);
```

Determines how the reception row should look in the UI. Every row displays reception, but perfect receptions get a green “perfect” label and a highlighted background.

Zone drawing UI

```
<div className="zone-section" style={{ marginBottom: 16 }}>
  <button
    onClick={() => setDrawingZone(true)}
    style={{
      background: drawingZone ? "#fee2e2" : "var(--card-bg, #1a1d23)",
      border: "1px solid var(--border-color, #333)",
      borderRadius: "6px",
      padding: "8px 14px",
      marginBottom: "8px",
      cursor: "pointer",
      fontWeight: "bold",
      color: "inherit",
      boxShadow: drawingZone ? "0 2px 8px #fecaca55" : "none",
    }}
  >
    {drawingZone
      ? "Drawing… (drag on video)"
      : zone
        ? "Redraw Perfect Zone"
        : "Draw Perfect Zone"}
  </button>
  {zone && (
    <div style={{ fontSize: 12, color: "#6b7280", marginTop: 4 }}>
      Zone active —{" "}
      <span style={{ color: "#16a34a", fontWeight: 600 }}>
        {perfectCount} perfect
      </span>{" "}
      of {totalCount} receptions inside
    </div>
  )}
</div>
```

Button onClick activates the drawing mode inside of **VideoPanel**. Once drawn, **statspanel** can calculate the number of perfect receptions.

```
    <PlayGroup
      key="reception"
      actionKey="reception"
      rows={groupedReceptions}
      isOpen={openGroups?.["reception"] ?? true}
      onToggle={() => onToggleGroup?.("reception")}
      onSeek={onSeek}
      activeFrame={activeFrame}
      showPerfectLabel
      renderRow={renderRow}
    />
  )}
```

This component either shows an empty message or renders the grouped detections.

5.5 Docker

Docker is basically a tool for put your app into containers and help them work together, even if they all require different versions of languages, dependencies and even operating systems.

This system would have trouble working without docker.
It separates the application into independent services that run together reliably. The system is split into four main containers:
API
PIPELINE1
PIPELINE2
FRONTEND
Each services has its own Dockerfile that defines the environment, dependencies and runtime, making sure everything runs the same way regardless of what machine its on.

API container and Dockerfile

```
FROM python:3.11

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
```

This Dockerfile builds the API orchestrator container. It starts from python3.11 sets /app as the working directory, installs the Python dependencies from requirements.txt, copies the API code into the container, exposes port 8000 and runs the FastAPI app using uvicorn (an asynchronous server gateway interface)

Frontend container and Dockerfile

```
FROM node:20 AS build

WORKDIR /app
COPY package.json package-lock.json* ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

This Dockerfile has two stages.
The first is a build stage, then a production stage.
In the first stage it uses a Node.js image to build the react app. It sets /app as the working directory, installs dependencies using npm install, copies the full project and then runs npm run build. This command complies the React app into static files (html, CSS and JavaScript)
The second stage uses a Nginx image to serve the build frontend. It copies the compiled files from the build stage into Nginx default web directory. It also replaces the config with a custom nginx config to handle routing for single page applications, like this one. The container exposes port 80 and the command starts Nginx server

Pipeline 2 Dockerfile

```
FROM nvidia/cuda:12.1.1-cudnn8-devel-ubuntu22.04

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y \
    python3.11 python3.11-dev curl \
    libsm6 libxext6 libxrender-dev libgomp1 ffmpeg \
    build-essential gcc g++ \
    && rm -rf /var/lib/apt/lists/*

# Install pip specifically for Python 3.11
RUN curl -sS https://bootstrap.pypa.io/get-pip.py | python3.11

# Make python3.11 the default python/pip
RUN ln -sf /usr/bin/python3.11 /usr/bin/python && \
    ln -sf /usr/bin/python3.11 /usr/bin/python3 && \
    ln -sf /usr/local/bin/pip3.11 /usr/bin/pip

WORKDIR /app

COPY requirements.txt .
RUN python -m pip install --no-cache-dir -r requirements.txt


COPY . .
ENV FORCE_CUDA=1

# Install correct PyTorch version and recompile MultiScaleDeformableAttention
RUN pip uninstall torch torchvision torchaudio -y && \
    pip install torch==2.5.1 torchvision torchaudio --index-url https://download.pytorch.org/
    whl/cu121

RUN cd models/ops && \
    pip uninstall MultiScaleDeformableAttention -y && \
    TORCH_CUDA_ARCH_LIST="7.5 8.0 8.6 8.7 8.9" sh make.sh

RUN mkdir -p /outputs/pipeline2

ENV SAMBAMOTR_MODEL=/app/sambamotr_sportsmot.pth
ENV SAMBAMOTR_CONFIG=/app/configs/sambamotr/sportsmot/def_detr/
train_residual_masking_sync_longer.yaml

EXPOSE 5001

CMD ["uvicorn", "api_server:app", "--host", "0.0.0.0", "--port", "5001", "--reload"]
```

This Dockerfile creates a GPU ready environment for SambaMOTR model, installs all video processing and deep learning dependencies, compiles CUDA (GPU) model operations, sets the model and config paths and prepares the service to run on port 5001

Pipeline1 Dockerfile

```
FROM python:3.11

WORKDIR /app

RUN apt-get update && apt-get install -y \
    libsm6 libxext6 libxrender-dev libgomp1 ffmpeg \
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt && \
    pip uninstall -y opencv-python opencv-contrib-python 2>/dev/null ; \
    pip install --no-cache-dir --force-reinstall opencv-python-headless>=4.10.0


COPY . .



RUN mkdir -p /outputs/pipeline1

EXPOSE 5000

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "5000", "--reload"]
```

This Dockerfile creates a CPU based service that installs video processing dependencies, runs a FastAPI backend and processes videos. An important thing to note is the uninstallation of opencv-python and reinstallation of opencv-python-headless as there is a GUI issue with that version.

Docker-Compose.yml

```yaml
services:
  api:
    build: ./api
    container_name: sr-analytics-api
    ports:
      - "8000:8000"
    environment:
      - PIPELINE1_URL=http://pipeline1:5000
      - PIPELINE2_URL=http://pipeline2:5001
      - PYTHONUNBUFFERED=1
    volumes:
      - ./api:/app
      - uploads:/tmp/uploads
      - pipeline1_outputs:/tmp/pipeline1_outputs
      - pipeline2_outputs:/tmp/pipeline2_outputs
    depends_on:
      - pipeline1
      - pipeline2
    networks:
      - sr-network

  pipeline1:
    build: ./srballandactions
    container_name: sr-pipeline1
    ports:
      - "5000:5000"
    environment:
      - PYTHONUNBUFFERED=1
    volumes:
      - ./srballandactions:/app
      - uploads:/tmp/uploads
      - pipeline1_outputs:/outputs/pipeline1
    networks:
      - sr-network

  pipeline2:
    build: ./sambamotr
    container_name: sr-pipeline2
    ports:
      - "5001:5001"
    environment:
      - PYTHONUNBUFFERED=1
      - NVIDIA_VISIBLE_DEVICES=all
      - NVIDIA_DRIVER_CAPABILITIES=compute,utility
    gpus: all
    volumes:
      - ./sambamotr:/app
      - uploads:/tmp/uploads
      - pipeline2_outputs:/outputs/pipeline2
    networks:
      - sr-network

  frontend:
    build: ./frontend
    container_name: sr-analytics-frontend
    ports:
      - "3000:80"
    depends_on:
      - api
    networks:
      - sr-network

volumes:
  uploads:
  pipeline1_outputs:
  pipeline2_outputs:

networks:
  sr-network:
    driver: bridge
```

This file connects all the services into a single working application. The Api service acts as the central orchestrator, built from the ./api directory and exposed on port 8000. It communicates with the two pipelines using environment variables, allowing it to send uploaded videos for processing.

Additionally it mounts shares volumes,

Name ↑
sranalytics_pipeline1_outputs
sranalytics_pipeline2_outputs
sranalytics_uploads

Enabling it to store incoming videos and access results generated by the pipelines.
The pipeline services are built from their respective folders, share the upload directory and its own output volumes so results can be accessed by the Api.
Pipelines2 service is unique, as it is configured for GPU acceleration using GPUs: ALL and Nvidia environment variables for heavier processing tasks, as it is a transformer model after all. The frontend is exposed on port 3000 and depends on the API, providing the user interface that interacts with the backend services.
All containers are connected through a shared bridge network (sr-network) which allows them to communicate using service names instead of localhost. The defined volumes ensure shared storage across containers and depends on ensures proper startup order. This configuration allows the system to run as a set of isolated but interconnected and coordinated services, making it scalable and easy to deploy.

6. Testing

6.1 User Testing

User testing was conducted on one candidate, a volleyball player.

Test format and questions:
The testing was conducted at the developer's house.
A notepad was given with basic instructions and the tester was sat in a chair in with the app open in front of them.

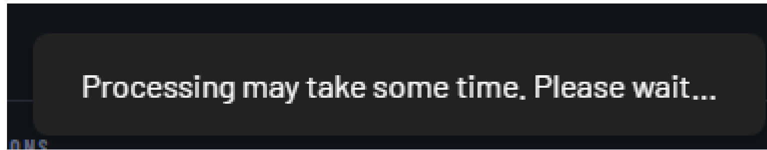
```
Welcome to the user test. Please get comfortable and open in front of you is the application, SR analytics.
Below are tasks, and you must try to complete each of them to the best of your ability. After each task is completed, write
your thoughts below the task!

upload a match video

success, upload video text should also be a button, or have explanation for what kind of video angle works with the program

process the video

progressing, bar would be good ts takes takes so long, get more ram, find a sponsor
add jersey colour
put in a colour dropper, maybe put in a colour wheel,

filter the receptions by pixel distance and rgb colour

I think there should be some instructions on what to do with the tolerance and ball distance, in the instructions explain
to adjust up if the ball/ jerseys aren't detected

draw perfect zone
put in instructions, I feel like there should be some kind of idiot proofing here, like instructions on where to draw the
box

comments regarding ui

maybe list what second it is as well as frame. group frames that are right beisde eachother, to reduce visual clutter, annd
because its the same play. all the icons and everything looks good. make matched and perfect instances icons bigger and
more visible.
```

Changes implemented from user feedback.

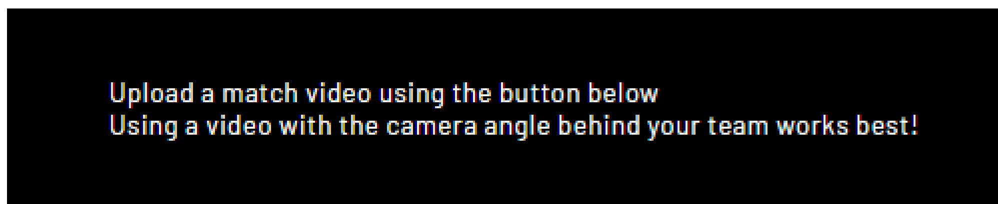
1. Notification for drawing zone with instructions

Drawing perfect zone
Click and drag on the video to draw your zone.
A rectangle with a red border will appear as you draw.
It is reccomnded draw the zone near the top of the net, giving space for the ball to travel into it after the reception.

2. Reception event grouping (link to that section somehow?)
3. Notification popup for processing



4. Updated placeholder text to suggest camera angle and point user towards the match video upload button



6.2 Model Testing and training

A YOLOv8 medium model was used for volleyball detection. It has 25b parameters and the mean average precision mAP@0.75 is 0.519 while YOLOv8 nano is 0.342. Retraining a model with a labelled dataset including test, train and valid images help the model understand and learn to detect the specific object you are trying to detect (in this project, it's a volleyball)
Common terms in model training

mAP@50-95 (mean average precision, a metric that takes average precision at 10 different thresholds 0.55-0.95 in 0.5 increments.)

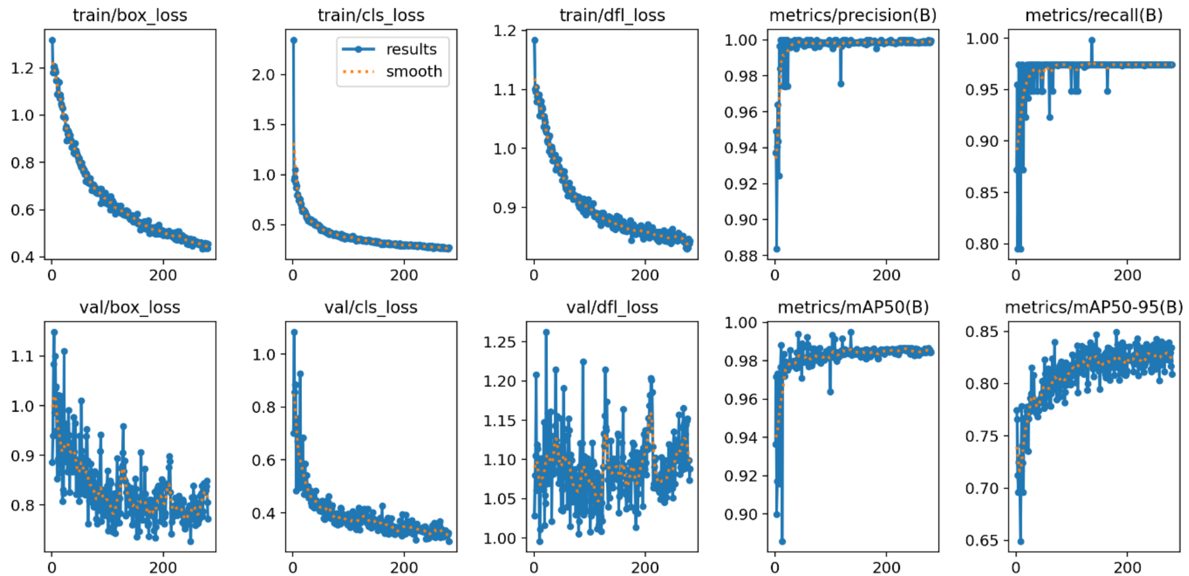
IoU - (intersection over union, a fundamental metric used in computer vision to measure the accuracy of an object detection model by comparing predicted boxes with ground truth, i.e. the model predicts the ball will be on the left side of the screen, but the truth is it was nearer the left middle.)

6.2.1 Volleyball Model training

The model was trained on a dataset annotated by myself, using Roboflow.com
The dataset consisted of 3000 images of annotated volleyballs, flying, occluded, out of focus etc.
Ultralytics YOLOs CLI was utilised for training, running the command below in a python terminal, and starting from a weighted checkpoint saved time.
yolo detect train data=data.yaml model=yolov8n.pt imgsz=640 epochs=250

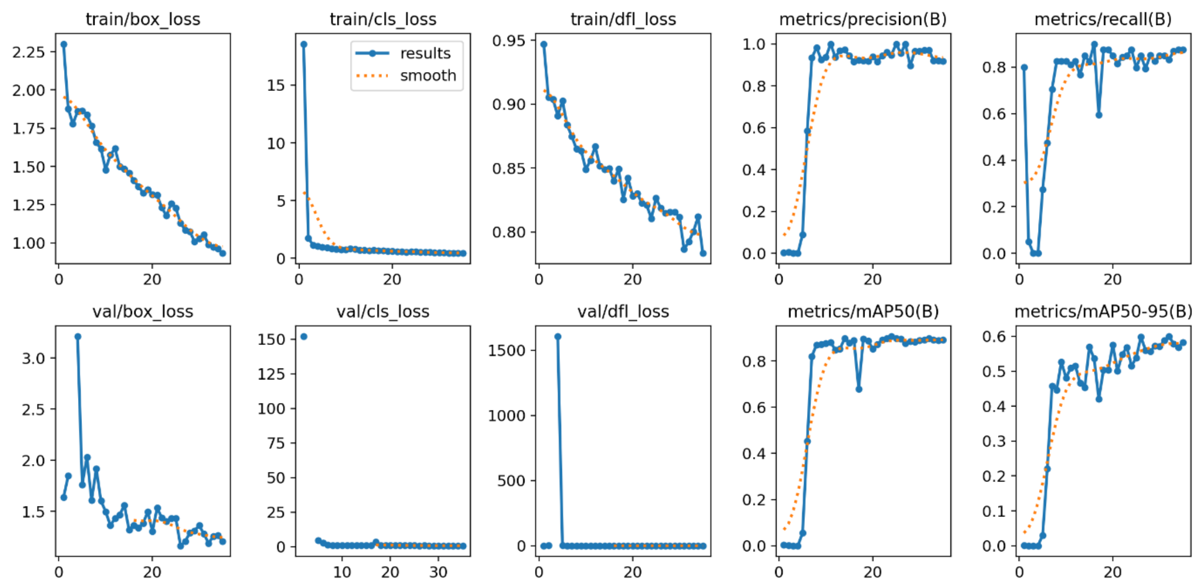
In total there was 26 trains, with each one changing various things, like images augmentation, epoch length (a pass through the entire dataset) and image size. The most notable two were train 21 and 26.

6.2.1.1 Train 21 – Results and Comments, 250 epochs



The main metrics to focus on are mAP@50-95, Recall and DFL_loss.
DFL loss is a function that focuses on hard to detect examples, like a stretched or occluded volleyball. Near the ending epochs, we can see this continues to go down and it means that the model is getting better at detecting volleyballs even if they are spinning fast or weirdly shaped in the frame.
Precision being 99+ means there is very little false positives, meaning that the actual volleyball is being tracked, not just any yellow object.
mAP@50-95 evening out at 0.83 shows very solid work for this model, with the longer training epochs showing higher accuracy and near perfect recall and precision, compared to train 26. This model was chosen for its high accuracy in occluded and hard to detect frames, as each user video will have different lighting, colour settings and motion blur.

6.2.1.2 Train 26 – Results and comments,25 epochs



This was a shorter train on a different, smaller 1000 image dataset. The dataset was taken from two videos; however, it was not augmented in any way. We can see the effects of shorter training time, the mAP@50-95 tops out at around 0.6, while the dfl loss shoots down to 0 almost instantly, due to all volleyballs being easy to detect as they are not augmented, shaped or resized in any way. This train was not chosen since all balls are clearly visible here, so the model would have difficulty detecting volleyballs in movement or occluded by the net or a player.

Conclusions, Limitations and Future Work

7.1 Conclusions

I want to start off by saying I'm very proud of the work I have done for this project and how much I've learned. Prior to starting this, I had 0 experience with Python, AI models, training them, applying them and adapting them for my own needs. This has been a learning experience and has really garnered my interest, as I am now planning to pursue a masters in AI.

Reflecting critically on this, I know it could have come out better if I had started off with a better understanding of python and had more computational resources. The RGB filtering isn't the greatest idea in terms of reliability, the frontend logic is flawed, and the UI is not the most interesting or intuitive. It's simple and serves a singular purpose, to upload the video and send it to the backend pipelines. Originally, there wasn't supposed to be a frontend but upon reflection and guidance from Mo, my lecturer, it was implemented. The models chosen were chosen because they could run on my machine, if I could choose other models I one hundred percent would. However, I think it's neat that I can use this project for my own gameplay. I probably could have completed a course on AI implementation or something during my time also working on

my project, but I enjoyed learning something new along each step of the way, following tutorials and reading documentation.
All in all, this project was a decent foundation to starting my career (hopefully) in AI engineering.

7.2 Limitations

Two main limitations affected this project

1. The lack of expertise of deep learning models

The use of RGB based filtering was the best idea at the time, but this can be affected by many things. Lighting shadows, camera quality, motion blur can alter perceived colours, making it difficult to match the correct player. This means users need to manually adjust tolerances and RGB values, which reduces automation and impacts usability greatly. Additionally, the zone-based detection isn't the greatest because it depends on manual input. Ideally the model would detect each volleyball position and then track how far the setter had to move to set the ball.

2. The sheer computational cost of video processing using deep learning models

The hardware available to me at home was nowhere near the needed scale or power to make this project anywhere near as good as I imagined it. The system worked with, even with a rtx 3080, Ryzen 5 5800x and 32gb of ram it struggled to run the Samba model and couldn't even retrain it. Given greater resources, larger models could have been used, and accuracy, detection and overall performance of this app would have increased multifold.

7.3 Future work

Future work would see improvements to the system, focusing on increasing accuracy, automation and performance. The key area is to replace the RGB based filtering with a larger model used for jersey number detection. Revamping the user experience and adding better player tracking, could mean this system even tracks the volleyball positions. This would enable the previously mentioned tracking how far the setter had to travel to pass the ball, also allowing the trajectories to be more accurately mapped and measured. Training a larger model on action recognition would also be beneficial, as this would provide the introduction of defensive move calculations.

References

Szeliski, R. *Computer Vision: Algorithms and Applications*. Springer, 2022.

Rosebrock, A. (2018, July 23). *Simple object tracking with OpenCV*. PyImageSearch. https://pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. Nature, 521(7553), 436–444. https://doi.org/10.1038/nature14539

Donahue, J., Lisa Anne Hendricks, Guadarrama, S., Rohrbach, M., Subhashini Venugopalan, Darrell, T., & Saenko, K. (2015). Long-term recurrent convolutional networks for visual recognition and description. Computer Vision and Pattern Recognition. https://doi.org/10.1109/cvpr.2015.7298878

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. Communications of the ACM, 60(6), 84–90.

Rawat, W., & Wang, Z. (2017). Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. Neural Computation, 29(9), 2352–2449. https://doi.org/10.1162/neco_a_00990

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 779–788. https://doi.org/10.1109/cvpr.2016.91

Cui, Y., Zeng, C., Zhao, X., Yang, Y., Wu, G., & Wang, L. (2023). SportsMOT: A Large Multi-Object Tracking Dataset in Multiple Sports Scenes. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.2304.05170

IMAGE SAMPLING AND QUANTIZATION - File Exchange - MATLAB CentralFile Exchange - MATLAB Central. (2024, February 7). Mathworks.com. https://uk.mathworks.com/matlabcentral/fileexchange/159131-image-sampling-and-quantization

Jahan, T. J., Aananthakrishnan, S., & Petrini, F. (2020). Online and Real-time Object Tracking Algorithm with Extremely Small Matrices. ArXiv.org. https://arxiv.org/abs/2003.12091

Segu, M., Piccinelli, L., Li, S., Yang, Y.-H., Schiele, B., & Gool, V. (2024). *Samba: Synchronized Set-of-Sequences Modeling for Multiple Object Tracking*. ArXiv.org. https://arxiv.org/abs/2410.01806