Development of Real-Time 5G-Enabled Video Analytics for Soccer Juggling Performance Evaluation

Rizan Bin Rudin

ELECTRICAL & ELECTRONICS ENGINEERING

UNIVERSITI TEKNOLOGI PETRONAS

JANUARY 2025

CERTIFICATION OF APPROVAL

Development of Real-Time 5G-Enabled Video Analytics for Soccer Juggling Performance Evaluation

by

Rizan Bin Rudin

19001584

A progress report/ project dissertation submitted to the

Electrical & Electronics Engineering Programme

Universiti Teknologi PETRONAS

in partial fulfilment of the requirement for the

BACHELOR OF ENGINEERING WITH HONOURS

(ELECTRICAL & ELECTRONICS ENGINEERING)

Approved by,

_____

(Ahmad Bukhari Aujih**)**

UNIVERSITI TEKNOLOGI PETRONAS

BANDAR SERI ISKANDAR, PERAK

January 2025

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

_____

RIZAN BIN RUDIN

ABSTRACT

This project successfully used 5G technology and video analytics to analyze soccer juggling skills in real time, making training more efficient and data-driven. The system accurately measured key performance metrics like the number of juggles, speed, and consistency. By applying machine learning, it provided valuable feedback to players of all skill levels, helping them improve their technique. The integration of 5G allowed for smooth, high-quality video streaming with minimal delay, ensuring real-time performance tracking. Throughout development, research and testing guided the implementation of a 5G-enabled CCTV system and a machine learning model for accurate assessment. The project followed a structured timeline, as outlined in the Gantt chart, and successfully met its goals. In the final phase, the system was tested and validated, proving to be a reliable and effective tool for soccer training and skill improvement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Soccer, with its immense global popularity, demands a high level of skill, agility, and coordination from its players. Fundamental techniques, such as juggling, play a crucial role in enhancing ball control, balance, and focus skills that are indispensable in competitive play. Traditionally, coaches and trainers have relied on subjective, visual assessments to evaluate players' juggling abilities. While effective to some degree, these evaluations are often inconsistent, prone to human bias, and limited in their capacity to deliver quantitative insights. Such limitations can hinder accurate performance tracking and slow down skill development, as players may not receive the precise feedback needed to improve in real-time.

Recent advancements in technology offer new possibilities for objective, data-driven performance evaluation in sports. With the integration of computer vision, machine learning, and 5G connectivity, real time video analytics systems have become feasible solutions for tracking and assessing player performance. This cutting-edge approach allows for the precise capture and analysis of complex movements like juggling, measuring aspects such as speed, consistency, and the number of juggles. By providing coaches and players with objective, real time feedback, this technology has the potential to transform soccer training, enhance skill development, and bring new levels of precision to player evaluations.

1.2 PROBLEM STATEMENT

Soccer training today still relies heavily on coaches' subjective visual assessments to evaluate key skills like juggling. While this approach is standard practice, it often leads to inconsistencies and biases, making it challenging to provide players with accurate and reliable feedback. Coaches' observations can vary widely, creating disparities in evaluations, which ultimately affect players' ability to track their progress or identify areas for improvement. Additionally, these methods are time-consuming and prone to human error, meaning players and coaches spend significant time on evaluations that may not be entirely precise. Without a more standardized and objective system, skill development can become inefficient and less impactful.

One of the most pressing issues is the absence of real-time feedback during training. Players often only receive insights after a session ends, which means they miss the chance to make immediate adjustments to their techniques. This delay in feedback slows down the learning process and makes it harder for players to achieve the rapid progress needed in competitive environments. As juggling and other foundational skills are critical to overall performance, the lack of immediate, actionable feedback hinders the effectiveness of training sessions.

A potential solution lies in developing an automated system powered by advanced technologies such as 5G and machine learning to evaluate juggling skills in real-time. However, implementing such a system comes with its own set of challenges. First, real-time evaluation requires processing large amounts of video data with minimal delay. While 5G networks are designed for high bandwidth and ultra-low latency, maintaining consistent performance across varied environments, such as outdoor fields or training centers with fluctuating network conditions, can be challenging. Ensuring the system works seamlessly during peak network usage times adds another layer of complexity.

Furthermore, designing machine learning algorithms capable of accurately analyzing juggling movements across different skill levels and environments is no small task. The algorithms need to adapt to a wide range of player techniques and handle challenges like poor lighting, busy backgrounds, or sudden changes in movement. Integrating 5G with edge computing hardware, such as NVIDIA Jetson Nano, requires careful optimization to ensure fast processing without overloading the system or consuming too much energy.

Cost and infrastructure are also significant concerns. Deploying 5G technology in training facilities involves high setup costs, compatibility issues with devices, and ongoing maintenance expenses. Equipping facilities with 5G-enabled cameras, edge processors, and other necessary tools may not be feasible for all organizations. Additionally, ensuring the security and privacy of player data during real-time processing is a critical challenge, especially given the growing importance of compliance with data protection regulations.

Despite these hurdles, the potential benefits of such a system are immense. A well-implemented 5G-enabled system could revolutionize soccer training, providing players and coaches with instant, accurate, and actionable feedback. By addressing the challenges associated with its implementation, this technology could make skill evaluation more precise, adaptive, and efficient, paving the way for a new era in sports training.

1.3 OBJECTIVES

1.3.1 Develop an Automated System for Real-Time Skill Assessment

The main aim is to design and build an automated system that can evaluate soccer juggling skills in real-time. This system will move beyond traditional methods that rely on coaches' subjective observations, ensuring more consistent and accurate feedback. By automating the process, the system removes potential human biases, providing players with reliable assessments to help them refine their techniques effectively.

1.3.2 Leverage 5G Technology for Rapid Data Processing

To make the system truly real-time, 5G technology will be used for rapid processing of large video data streams. The high speed and ultra-low latency of 5G will ensure that feedback is provided instantly, enabling players to adjust their techniques on the spot. This responsiveness will transform training sessions into dynamic and interactive experiences, bridging the gap between performance and immediate guidance.

1.3.3 Employ Advanced Video Analytics and Machine Learning

Advanced video analytics and machine learning algorithms will be at the core of the system, enabling it to detect and analyze juggling movements with incredible accuracy. These technologies ensure that evaluations are reliable, consistent, and adaptable to different skill levels and environments. By focusing on

metrics like juggling technique, rhythm, and consistency, the system will provide detailed insights tailored to each player's performance.

1.3.3 Provide Instant Feedback for Continuous Improvement

A standout feature of the system is its ability to provide instant, actionable feedback to both players and coaches. This immediate feedback loop allows players to fine-tune their techniques during training sessions, helping them make consistent progress. Over time, players can track their improvements and work toward achieving their full potential, making the system an invaluable tool for long-term skill development.

1.4 SCOPE OF WORK

This project aims to revolutionize soccer training by implementing advanced technologies to enhance skill assessment and feedback. A key component involves setting up 5G-connected CCTV cameras to capture high-quality video footage of soccer juggling sessions. These cameras will provide the foundational data for the system, ensuring clear, uninterrupted, and real-time video streams for accurate analysis.

To analyze the captured footage, machine learning algorithms will be developed to detect and evaluate key juggling metrics. These algorithms will automate the process of measuring performance, offering consistent and precise evaluations of skill levels. By focusing on critical metrics such as juggling frequency, accuracy, and technique, the system will eliminate subjective biases and provide a reliable method for assessing player performance.

Additionally, a user-friendly interface will be designed to display real-time performance data and feedback. This interface will prioritize accessibility and usability, enabling players and coaches to engage with the insights effortlessly. The feedback will be presented in a clear, actionable format, allowing users to make immediate adjustments during training sessions and track progress over time. Together, these elements will create an integrated system that streamlines skill evaluation and fosters continuous improvement in soccer training.

CHAPTER 2

LITERATURE REVIEW

2.1 Real-Time Pose Estimation and Motion Tracking for Performance Using Deep Learning Models

In recent years, real-time pose estimation and motion tracking have gained significant traction in performance analysis due to their ability to provide instant feedback, particularly in athletic and motion-intensive applications. Advanced deep learning models, such as OpenPose and DeepSORT, have emerged as state-of-the-art tools for this purpose, offering robust pose estimation and motion tracking capabilities.

OpenPose, developed by Cao et al. [8], is one of the most widely used models for multi-person 2D pose estimation. It employs a two-stage convolutional neural network (CNN) to simultaneously detect keypoints and generate Part Affinity Fields (PAFs). PAFs represent the spatial relationships between body parts, enabling OpenPose to track multiple individuals in a single frame effectively. This approach is particularly beneficial in dynamic environments, such as sports training, where athletes' movements are complex and overlapping. OpenPose's ability to capture fine-grained motion details makes it ideal for applications requiring precise body motion analysis.

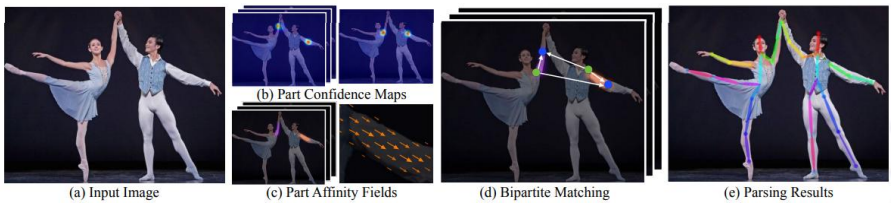


Figure 1: OpenPose Algorithm Architecture [6]

For real-time motion tracking, DeepSORT (Simple Online and Realtime Tracking with a Deep Association Metric), introduced by Wojke et al. [9], builds on the SORT algorithm by incorporating a deep convolutional neural network for appearance-based feature extraction. This enhancement allows DeepSORT to distinguish between visually similar individuals, a critical feature in scenarios where subjects frequently overlap or interact, such as athletic training. It uses a Kalman filter for motion prediction and a Hungarian algorithm for data association, ensuring smooth and reliable tracking across multiple frames.

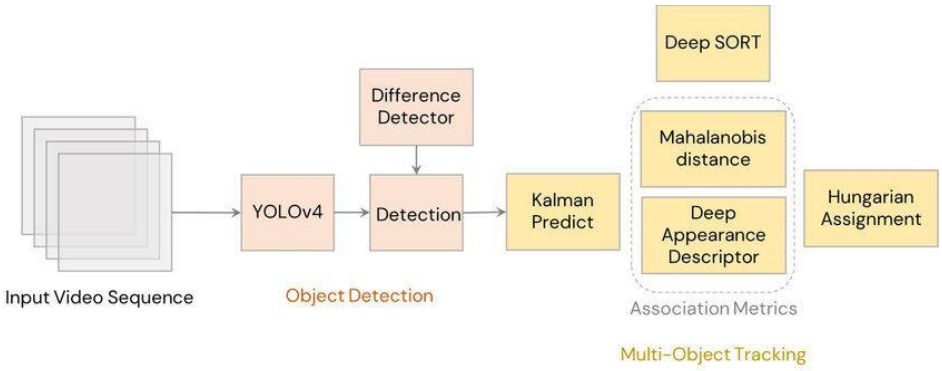


Figure 2: DeepSORT architecture [7]

The integration of OpenPose and DeepSORT has proven effective in scenarios requiring real-time tracking and pose estimation. OpenPose provides accurate detection of body poses, while DeepSORT ensures consistent motion tracking, even in dynamic and crowded scenes. This combination has been applied successfully in athletic performance evaluation, offering immediate and actionable

feedback to athletes and coaches. Experimental studies demonstrate that these models achieve robust tracking and reliable pose estimation under real-world conditions, though challenges such as handling full occlusions and scaling for large groups of individuals remain areas for improvement [8], [9].

Future developments in pose estimation and motion tracking could focus on addressing the limitations of current models. Enhancements in handling occlusions, reducing latency for large-scale applications, and improving scalability for multiple individuals would significantly expand the applicability of these models. Additionally, the integration of emerging technologies such as 5G and edge computing may further enhance real-time processing and feedback capabilities.

2.2 Learning Soccer Juggling Skills with Layer-wise Mixture-of-Experts

Soccer juggling is a fundamental skill in soccer training that involves precise footwork and balance to control the ball in the air. It plays a crucial role in developing a player's coordination and technique. With advancements in machine learning, automated systems for assessing and training soccer juggling have gained prominence, offering real-time feedback to enhance performance. The Layer-wise Mixture-of-Experts (LMoE) model, introduced as an extension of the Mixture-of-Experts (MoE) framework, has shown significant promise in skill-based tasks like soccer juggling by providing adaptive, context-aware feedback [2].

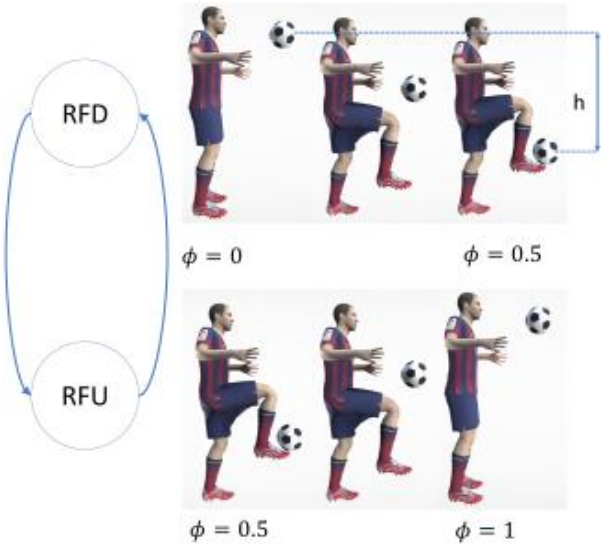


Figure 3: Stages of right foot juggling [2]

The Layer-wise Mixture-of-Experts (LMoE) architecture builds on the foundational principles of MoE, which employs specialized expert networks to divide complex tasks into smaller, manageable components. In the LMoE model, each layer of the neural network can selectively activate a subset of expert networks based on the input characteristics. This enables the model to dynamically allocate computational resources to focus on specific features of the task. For soccer juggling, the LMoE model can specialize in analyzing different aspects of juggling, such as ball trajectory, timing, and body posture. This modular approach allows the model to adapt to the unique styles and proficiency levels of individual players [1].

One of the key strengths of the LMoE model lies in its ability to provide real-time adaptation during skill learning. For example, the system can dynamically adjust its focus based on the height of the ball's bounce or the player's foot positioning during a juggling attempt. By tailoring feedback to these specific variables, LMoE facilitates a personalized learning experience, helping players identify and improve areas such as balance, timing, and coordination. This level of

adaptability creates a more engaging and efficient training process, as players can immediately internalize the feedback and refine their skills in real-time [1].

Studies highlight the importance of adaptive learning systems like LMoE in improving motor learning and skill retention. By providing structured, actionable insights during practice, the LMoE model helps accelerate the skill acquisition process. For instance, it can guide players through a step-by-step improvement plan, focusing on essential aspects of juggling while dynamically adjusting its feedback as the player's proficiency evolves. This ensures that players receive feedback that evolves with their performance, making the learning process both effective and sustainable [1].

In summary, the LMoE model represents a breakthrough in automated soccer training systems. Its ability to adaptively focus on different components of juggling movements, combined with its real-time feedback capabilities, makes it a powerful tool for enhancing soccer juggling skills. By bridging the gap between traditional training methods and machine learning, LMoE not only supports faster skill development but also creates a more interactive and tailored training experience for players at all skill levels [1].

2.3 Integrated 5G MEC System and Its Application in Intelligent Video Analytics

The rise of 5G networks, known for their ultra-low latency, high bandwidth, and ability to connect millions of devices, has revolutionized real-time applications like intelligent video analytics (IVA). When paired with Multi-Access Edge Computing (MEC), 5G networks bring computational tasks closer to the source of data generation, significantly reducing delays and improving efficiency in video processing. This integration has opened the door to a wide range of applications,

from urban surveillance to autonomous driving, providing faster and more reliable analytics compared to traditional cloud-based systems [3].

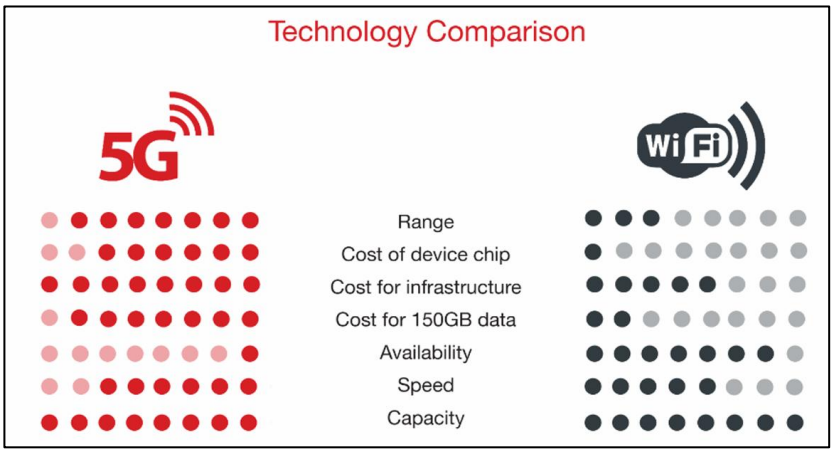


Figure 4: 5G vs WiFi technology comparison [5]

This figure highlights a comparison between 5G and WiFi across key aspects such as range, cost, speed, and capacity. 5G excels in areas like range, speed, and capacity, making it ideal for large-scale, high-demand applications such as real-time video analytics and smart city infrastructure. However, this advanced capability comes at a higher cost for device chips, infrastructure setup, and data usage, as compared to WiFi, which is more economical and widely accessible for smaller-scale deployments. WiFi remains a viable choice for localized environments due to its affordability and availability, though it falls short in performance when handling extensive data loads or providing ultra-low latency. This comparison illustrates how 5G's superior technical features make it better suited for cutting-edge, data-intensive projects, while WiFi continues to serve as a cost-effective alternative for less demanding use cases.

At the heart of an integrated 5G MEC system are three main components: the 5G Radio Access Network (RAN), edge servers, and an orchestration layer. The

RAN facilitates high-speed data transmission between devices and processing units. Edge servers, positioned near the data source, handle localized processing, drastically cutting down the time required to send raw data to a centralized cloud. Meanwhile, the orchestration layer ensures that resources are allocated efficiently and that data flows seamlessly between the edge nodes and the central cloud. This setup allows computationally heavy tasks, like analyzing video feeds, to be performed at the edge, ensuring faster and more responsive processing [3].

To enhance the capabilities of these systems, 5G MEC integrates artificial intelligence (AI) and machine learning (ML) frameworks. Deployed on edge servers, these frameworks perform complex IVA tasks such as object detection, activity recognition, and anomaly detection. By dynamically allocating resources based on network traffic and workload, MEC systems ensure that these models operate efficiently even in high-demand situations. This ability to adapt makes 5G MEC particularly suitable for applications that demand instant decision-making, like live surveillance or autonomous vehicles [3].

Studies have consistently highlighted the superiority of 5G MEC systems over traditional cloud-based setups. One major advantage is the dramatic reduction in latency. While traditional systems can experience delays of tens or even hundreds of milliseconds, 5G MEC systems reduce latency to as low as 1 millisecond by processing data locally. This improvement is critical for applications like autonomous driving, where even a slight delay could pose safety risks, or real-time security monitoring, where immediate action is often required. Additionally, because data is processed locally at the edge rather than sent to centralized servers, 5G MEC systems significantly enhance data privacy. Sensitive information remains closer to its source, reducing the risk of breaches. MEC also supports advanced encryption and anonymization methods, making it an ideal choice for privacy-focused applications like healthcare or smart city monitoring [3].

Another notable benefit of 5G MEC is its ability to optimize bandwidth usage. By processing video data locally, these systems eliminate the need to send large volumes of raw data over the network, reducing congestion and improving overall efficiency. This is especially valuable in densely populated areas, where network resources are often stretched to their limits. Furthermore, MEC's distributed architecture ensures scalability. For example, in smart cities, new edge nodes can be added to handle the growing number of connected devices and video streams, ensuring consistent performance even as data demands increase [3].

While the advantages of 5G MEC systems are clear, there are still challenges to overcome. Efficiently managing resources under dynamic workloads and addressing edge-node failures are areas that require further research. Additionally, advancements in AI and hardware optimization could further enhance the scalability and robustness of these systems. Future research should focus on improving resource allocation strategies and developing advanced AI models that can handle large-scale real-time applications, further unlocking the potential of 5G MEC systems [3].

In conclusion, the integration of 5G networks and MEC represents a significant leap forward for intelligent video analytics. By bringing computation closer to the source, these systems deliver unparalleled speed, efficiency, and security, making them indispensable for real-time applications like healthcare, autonomous driving, and urban surveillance. As the technology continues to evolve, it is expected to play an even more transformative role in the future of video analytics [3].

2.4 Collaborative Edge and Cloud Neural Networks for Real-Time Video Processing

The rapid advancement of Internet of Things (IoT) applications has fuelled a growing demand for efficient real time video processing, which is essential for technologies such as autonomous vehicles, augmented reality, and video surveillance. Traditional approaches to video processing face significant challenges: edge devices often lack the computational power required for complex tasks, while cloud-based systems suffer from latency and bandwidth constraints that can compromise performance. To overcome these limitations, Grulich and Nawab propose a collaborative edge-cloud framework that integrates neural networks to optimize real-time video processing [4].

The proposed framework intelligently distributes the computational workload between edge devices and cloud servers. The key innovation lies in three core techniques. First, splitting involves partitioning the neural network so that initial processing is performed locally on edge devices, while more computationally intensive tasks are offloaded to the cloud. This approach ensures that latency-sensitive operations, such as object detection or motion tracking, are handled immediately at the edge. Second, compression techniques are applied to reduce the size of data before it is transmitted to the cloud, conserving bandwidth without significantly compromising the quality of information. Finally, differential communication further minimizes data transfer by transmitting only the changes or differences between consecutive video frames. This method dramatically reduces the volume of data sent over the network, making the system highly efficient [4].

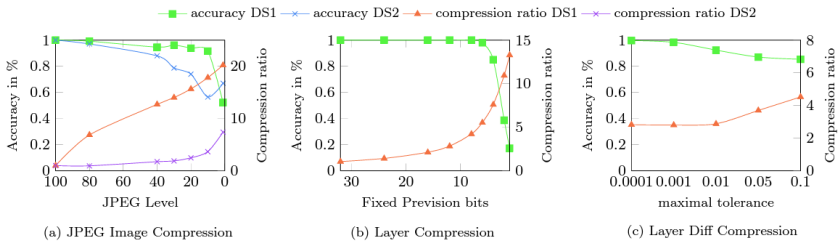


Figure 5: Evaluation of the single NN compression techniques [4]

By combining these strategies, the framework achieves a balance between computational efficiency, low latency, and minimal bandwidth usage, enabling real-time analysis of video streams. This is particularly advantageous in time-critical applications, such as autonomous driving, where delays in processing can lead to serious safety risks, or in surveillance systems, where immediate responses are essential to prevent potential threats. The collaborative nature of the system also ensures scalability, allowing it to handle increasing workloads as more devices and data streams are added to the network [4].

Grulich and Nawab provide a practical demonstration of their framework, enabling users to experiment with different neural network configurations and techniques. This hands-on approach allows for an in-depth understanding of how various methods such as splitting, compression, and differential communication impact key performance metrics, including accuracy, latency, and bandwidth usage. The results highlight the framework's ability to achieve significant reductions in latency and bandwidth consumption while maintaining high accuracy, demonstrating its viability for real-world applications [4].

A critical aspect of this framework is its adaptability to diverse operating conditions. For example, in low-bandwidth environments, edge devices can operate

independently to ensure uninterrupted functionality, while in high-bandwidth scenarios, additional computational tasks can be offloaded to the cloud to enhance processing capabilities. This dynamic allocation of resources ensures optimal performance regardless of the network environment, making the system highly versatile and robust [4].

While the framework offers numerous advantages, including scalability, adaptability, and efficiency, the authors acknowledge certain challenges that require further investigation. Achieving seamless integration and synchronization between edge and cloud resources is a complex task, particularly when dealing with large-scale deployments involving multiple devices. Additionally, ensuring data privacy and security during the transmission of video data between edge devices and the cloud is a critical concern that must be addressed. Future research should focus on refining these aspects, exploring advanced encryption techniques, and developing algorithms that can handle more diverse and dynamic workloads [4].

In conclusion, Grulich and Nawab's framework presents a groundbreaking solution to the challenges of real-time video processing in IoT applications. By leveraging the complementary strengths of edge and cloud computing and employing innovative data reduction strategies, the framework achieves a level of efficiency and scalability that traditional methods cannot match. Its ability to adapt to varying network conditions and handle diverse application scenarios positions it as a promising foundation for future developments in IoT and real-time analytics.

CHAPTER 3

METHODOLOGY

3.1 Phases of Methodology

Throughout this project, the following structured methodology was followed to ensure a systematic and efficient approach to developing a real-time 5G-enabled video analytics system for soccer juggling performance evaluation.

Figure 6 illustrates the structured methodology followed during the development process, comprising five key phases that ensure systematic progress and refinement of the project.

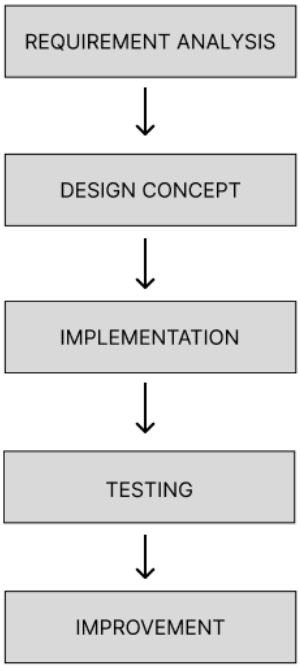


Figure 6: Phases of methodology

3.1.1 Requirement Analysis

The first phase of the project focused on thoroughly identifying the essential requirements, tools, and hardware needed to successfully implement the real-time video analytics system. This phase began with an in-depth exploration of the project's technical needs to ensure a clear and precise understanding of its scope. Research was conducted to document the necessary software, libraries, databases, communication protocols, and hardware components. This comprehensive study provided the foundation for selecting the tools and technologies best suited to achieve the project's objectives.

Key components identified during this phase included the NVIDIA Jetson platform, known for its high-performance AI capabilities and energy efficiency, and a 5G enabled CCTV camera to provide high-resolution video input with ultra-low latency. To support data management, robust database systems were chosen to ensure efficient storage and retrieval of large volumes of video data. Additionally, 5G connectivity was selected as the backbone for rapid and seamless data transmission, which is critical for real-time processing. Together, these components were integrated into the design framework to enable the system to process video streams in real time and deliver actionable insights.

3.1.2 Design Concept

In the design concept phase, a comprehensive framework was created to outline the system's architecture and workflow, ensuring seamless integration of all components for real-time video processing. The design centered on utilizing a 5G-enabled CCTV camera to capture high-resolution video footage, which would be transmitted wirelessly over a 5G network to a router. The router acted as the intermediary, relaying the video data to an NVIDIA Jetson device, which was

designated as the primary processing unit. The NVIDIA Jetson, equipped with advanced AI capabilities, was tasked with running neural network models to perform video analytics in real time. To complement this setup, a database system was integrated into the architecture to store the processed video data, ensuring easy retrieval and future reference.

This framework established a clear and efficient data flow between components. Video data captured by the CCTV camera would be transferred wirelessly to the router, which in turn communicated with the NVIDIA Jetson for immediate processing. Additionally, remote access to the NVIDIA Jetson was enabled via Secure Shell (SSH), allowing for flexible monitoring and system updates. By combining cutting-edge hardware with robust communication protocols, the design concept provided a solid blueprint for achieving the project's objectives of real-time video analytics.

3.1.3 Implementation

The implementation phase focused on setting up and configuring the system to ensure seamless data flow from video capture to real-time analysis. The hardware setup involved integrating the 5G CCTV camera, NVIDIA Jetson, router, and database, making sure all components worked together efficiently. On the software side, key tools and libraries were installed to enable smooth processing and AI-based analytics.

To support the system's deep learning capabilities, YOLO was installed for object detection, OpenCV for computer vision processing, and PyTorch for executing neural network models. Visual Studio Code (VS Code) was also set up as the main development environment, providing an intuitive interface for coding

and debugging. Additionally, Python pip was configured to manage package installations efficiently.

A virtual environment was created to keep the project's dependencies organized and prevent conflicts between different library versions. This ensured that the system remained stable, scalable, and easy to maintain. For data storage, the system was connected to Firebase, Google Sheets, or MongoDB, allowing performance analytics to be saved and accessed easily. By the end of this phase, the entire system was fully operational, with video data seamlessly processed and analyzed in real time, making it ready for accurate soccer juggling performance tracking.

3.1.4 Testing

In the testing phase, the system underwent a thorough evaluation to assess its performance, accuracy, and reliability in handling real-time video processing tasks. Latency testing was a key focus, measuring the time taken for data to be transmitted, processed, and analyzed across the system components to ensure real-time responsiveness. The accuracy of the video analytics models was rigorously tested to verify the system's ability to detect objects and estimate poses with precision. Furthermore, the system's reliability was evaluated under varying network conditions and data loads, ensuring it could maintain consistent performance even in challenging scenarios. The findings from this phase highlighted the system's strengths and revealed areas for improvement, enabling targeted adjustments to optimize accuracy, reduce latency, and enhance overall reliability, ensuring the system met its real-time processing objectives.

3.1.5 Improvement

In the final phase, the system was optimized to improve efficiency, accuracy, and real-time performance. By integrating CUDA and TensorRT, the YOLO models were converted into TensorRT format (.engine), allowing inference to run on the NVIDIA Jetson's GPU instead of the CPU. Below shows the conversion process from Pytorch model file (.pt) to TensorRT model engine (.engine)

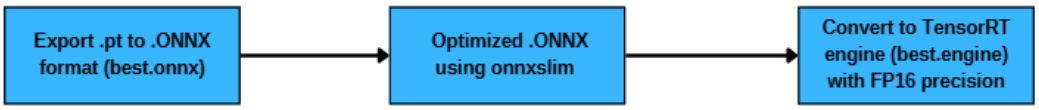


Figure 7: Conversion process from Pytorch model file (.pt) to TensorRT model engine (.engine)

With the latest improvements, the system now processes frames much faster, reducing latency to under 100ms per frame. This has significantly boosted the frame rate from 10 FPS to a range of 15-30 FPS. The system also detects CUDA availability automatically, ensuring it uses GPU acceleration whenever possible for better performance.

To speed things up even more, multiprocessing was added to distribute tasks across multiple CPU cores. One major improvement was running activation line updates in a separate process, which lightens the CPU workload and improves real-time responsiveness. Additionally, NumPy replaced slower loops with efficient, vectorized calculations, making tasks like analyzing time gaps and juggling consistency much faster.

To enhance stability, better error handling was implemented. Now, the system checks for missing video files before processing, preventing crashes. It also ensures multiprocessing tasks are properly managed to avoid memory leaks. With

all these upgrades, the system can now track juggling performance in real time at 30-50 FPS with minimal delay, making it much more reliable and ready for real-world use.

3.2 System Architecture

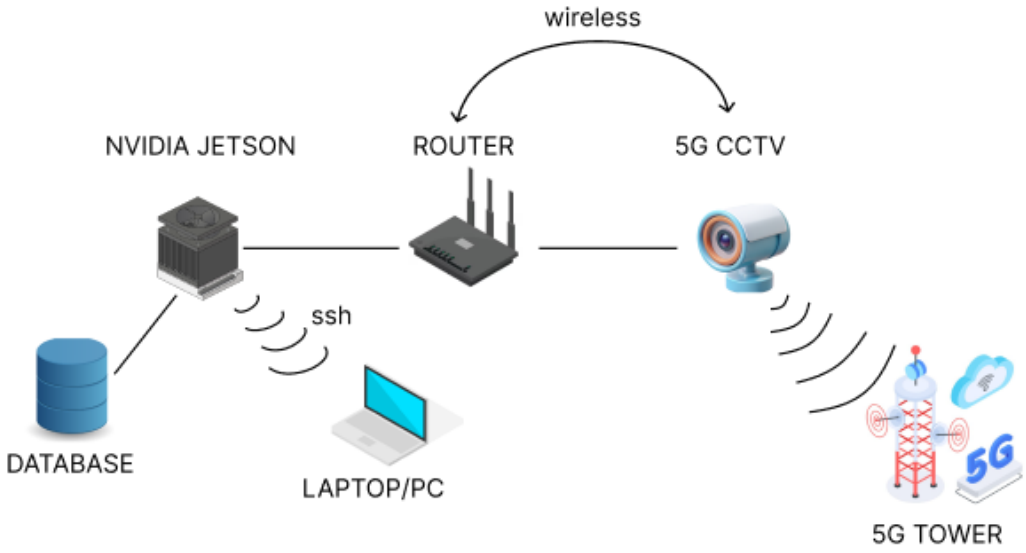


Figure 8: System architecture

Figure 7 represents a 5G-connected system architecture designed for efficient data collection, processing, and monitoring. At the heart of the system is the 5G CCTV Camera, which captures video or image data and transmits it wirelessly through the 5G network for further processing. The 5G Tower serves as a communication hub, facilitating the seamless transfer of data from the camera to the system's core components.

Once the data reaches the system, the Router acts as the intermediary, directing the incoming data to the appropriate devices within the local network. From there, the NVIDIA Jetson takes over as the primary processing unit, handling tasks such as image processing, AI-driven analysis, or data preprocessing before the

data is stored or further analyzed. The processed data is then securely stored in the Database, ensuring it can be easily retrieved and analyzed as needed.

For system control and monitoring, a Laptop/PC connects remotely to the NVIDIA Jetson using the Secure Shell (SSH) protocol. This setup allows for real-time management, troubleshooting, and system updates virtually from any location. Together, these components form a cohesive and efficient system for real-time video processing and data management, leveraging the speed and connectivity of 5G technology.

This setup shows a flow where data is captured by the camera, transmitted through a 5G network, processed by the NVIDIA Jetson, and stored in a database, with remote access via a laptop or PC.

3.3 Data Flow Process

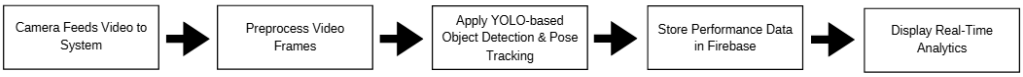


Figure 9: Data flow process

Figure 9 shows how a 5G-enabled system monitors and evaluates soccer juggling performance in real time. The process starts when a camera captures video of a player juggling a soccer ball. This camera continuously records movements and sends the footage into the system for analysis. With a 5G network connection, the video data is transmitted wirelessly with low latency, ensuring real-time processing without delays.

Once received, the system prepares the video frames by resizing and optimizing them for efficient analysis. Then, YOLO-based object detection tracks the soccer ball, ensuring each touch is accurately identified. At the same time, YOLO-based pose estimation detects and maps the player's key body points, helping track posture, movement, and ball interaction. By combining both tracking methods, the system can precisely analyze juggling performance, recognizing and recording each controlled touch.

After detecting and analyzing juggling events, the system stores performance data in Firebase, providing a cloud-based storage solution for future reference. The processed data is then displayed on a real-time analytics dashboard, where players and coaches can monitor key performance indicators such as total juggles, juggling consistency, and time gaps between touches. Finally, the system saves juggling statistics for player feedback, allowing for long-term tracking of progress and skill development (End). This seamless integration of AI, computer vision, and real-time data processing enables instant performance evaluation, making it an invaluable tool for soccer training and improvement.

3.4 Tools and Equipment

Below is the list of tools and equipment that required for this project. Table 1 summarizes the essential software and hardware components needed to build the real-time video analytics system.

Table 1: Software and hardware required for project

Software	Hardware
a. Visual Studio Code b. Machine Learning & Deep Learning	• 5G-enabled CCTV Camera. • Webcam

• YOLO • PyTorch c. Computer Vision & Image Processing • OpenCV • Numpy d. Data Visualization, Handling, Storing • Pandas • Gspead • Streamlit	• NVIDIA Jetson Nano. • Monitor. Optional: • Additional 5G-Enabled Cameras. • Motion Capture Sensors. • External Storage or Cloud Services. • Edge Computing Devices.

The project uses Visual Studio Code for development, while YOLOv8, OpenCV, and PyTorch handle object detection, pose estimation, and deep learning tasks. Streamlit helps create an interactive dashboard for real-time performance tracking, and Plotly is used for data visualization. For storing and managing data, Google Sheets (via gspread) and CSV files are used. To keep the system running efficiently, psutil and pynvml monitor CPU and GPU usage, ensuring smooth performance.

On the hardware side, the setup includes a 5G-enabled CCTV camera for video capture, an NVIDIA Jetson Orin NX for processing, and a monitor to display results. Additional cameras, motion capture sensors, and external storage can be added to improve scalability and functionality. This combination ensures a reliable and efficient system for analyzing juggling performance in real time.

Table 2 outlines the tools and equipment used in the project and their purposes.

Table 2: Tools & equipment description

Tools	**Purpose**	**Equipment**	**Purpose**
Visual Studio Code	For code writing, compilation and testing	5G CCTV Camera	For capturing high-resolution video footage of soccer juggling in real-time.
YOLO (You Only Live Once)	For object detection and tracking juggling movements in real-time.	NVIDIA Jetson Nano	For edge computing and running machine learning models locally with low latency.
OpenCV	For video processing and real-time image recognition.	Monitor	For visualizing real-time performance evaluation and system feedback.

PyTorch	For developing and deploying machine learning models for skill evaluation.		
Gspread	For storing performance data and metrics.		

Visual Studio Code was used for coding and testing, while YOLO and OpenCV handled object detection and video processing. PyTorch supported machine learning model development for skill evaluation, and Firebase stored performance data. A 5G CCTV Camera captured high-resolution video, and the results were displayed in real-time on a monitor. Together, these tools ensured seamless data processing and feedback.

3.5 System Design

This project leverages YOLO-based object detection and pose estimation to track soccer juggling performance in real time. It detects both the ball and the player's movements, accurately counts successful juggles, and provides instant feedback on performance. By processing video frames, the system identifies key points on the player's body and dynamically adjusts the activation line to ensure precise tracking. It also evaluates consistency, timing, and overall performance,

giving players data-driven insights to refine their technique and improve their juggling skills.

Below is the flowchart representing the step-by-step process of the juggling detection system:

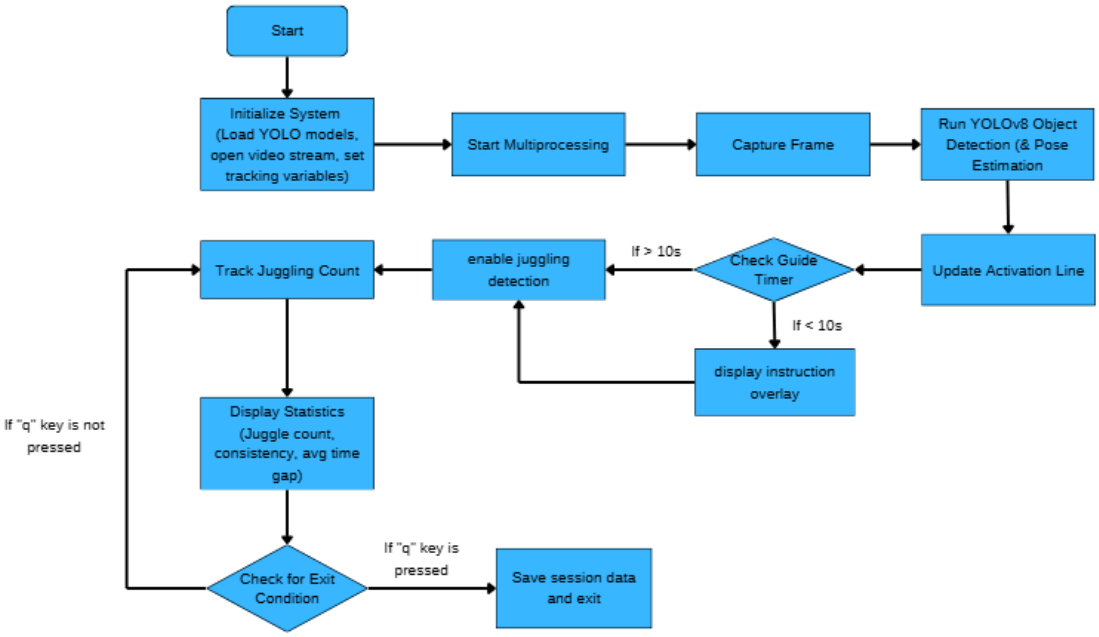


Figure 10: Juggling Evaluation Algorithm Flowchart

3.5.1 Capturing and Preprocessing Video

The system starts by capturing video frames from a webcam or 5G-CCTV, using OpenCV to process them efficiently. Each frame is resized to 640×360 pixels, striking a balance between speed and accuracy for real-time tracking without overwhelming the system.

To ensure smooth performance, the system continuously monitors key metrics like frames per second (FPS), processing latency, and memory usage. These optimizations are especially crucial for keeping up with fast-paced juggling movements, allowing for accurate tracking and seamless performance analysis.

3.5.2 Detecting the Ball and Player Using YOLOv8

To track juggling performance, the system needs to accurately identify both the ball and the player in each frame. This is achieved using two YOLOv8 deep learning models:

Table 3: YOLOv8n Model Description

Object Detection	**Pose Detection**
• (yolov8n.engine) identifies the ball based on COCO's predefined class ID (32). The system extracts the bounding box and continuously tracks its position.	• (yolov8n-pose.engine) identifies key points on the player's body, following COCO's 17-keypoint format (e.g., nose, shoulders, hips, knees, and ankles).

Figure 11: Key facial points detected for tracking head pose and alignment in 3D

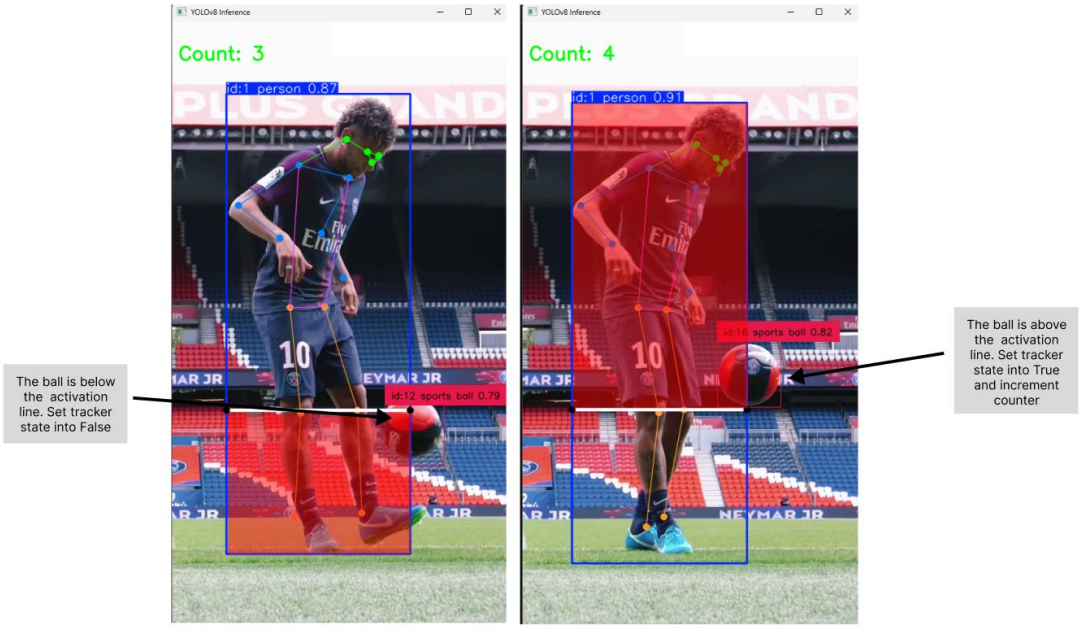
In this system, two YOLOv8 models work together to track and analyze movement with precision. The YOLOv8n model detects and tracks key objects, such as the player and the soccer ball, by placing bounding boxes around them and assigning confidence scores. This allows for real-time position tracking throughout the juggling session.

At the same time, the YOLOv8n-pose model focuses on human motion by identifying key points on the body, including joints and facial landmarks. These

points form a skeletal structure that helps assess posture, movement, and coordination.

By combining both models, the system can accurately monitor juggling performance, providing valuable insights into technique, consistency, and areas for improvement. This technology is particularly useful in sports analytics, motion tracking, and real-time performance evaluation.

3.5.3 Tracking Juggles



Once the system identifies both the ball and the player's key points, it starts tracking each successful juggle by monitoring the ball's movement in relation to an activation line. This line is dynamically positioned based on the ball's Y-coordinate, aligning with different body parts like the head, torso, thighs, or feet, with the ankle keypoint (from the COCO dataset) commonly used as a reference.

A juggle is counted only when the ball crosses the activation line while moving upward, ensuring that accidental touches or unintended bounces don't interfere with the count. To further improve accuracy, the system also tracks the peak height of each juggle, helping to filter out false detections and ensuring that only controlled touches are recorded. This approach allows for precise, real-time tracking, giving players valuable insights into their juggling consistency and technique.

These techniques allow for precise and reliable juggling detection, regardless of a player's juggling technique.

3.5.4 Performance Evaluation

To help players track their juggling progress, the system records and visualizes key performance metrics:

Table 4: Key Performance Metrics

Metrics	**Description**
Total Juggles	The number of times the player successfully keeps the ball in the air
Time Gap Between Juggles	The average time between each controlled touch
Juggling Consistency	The variation in ball height, indicating control over each touch
Total Session Duration	The overall time spent juggling.

All performance data is automatically saved in a CSV file (juggling_data.csv) and seamlessly uploaded to Google Sheets using the update_google_sheet() function. To help players track their progress, a Streamlit dashboard (dashboard.py) provides real-time visualizations, including interactive charts that highlight performance trends. This makes it easy to spot areas for improvement and refine technique over time.

The juggling rating system goes beyond just counting juggles. It evaluates key performance factors like speed, consistency, endurance, and improvement, offering a more complete assessment of a player's skill. Instead of a simple count, the final score is calculated on a 1 to 10 scale, where a higher rating reflects better ball control, sustained endurance, and continuous progress. This ensures a fair and well-rounded evaluation, helping players set goals and track their development effectively.

1. Juggling Count

$$JC = min(\frac{Total\ Juggles}{10}, 20) \quad (1)$$

Equation (1) measures how frequently a player juggles the ball. A higher JC means the player is juggling quickly and maintaining a steady rhythm, while a lower JC may indicate slower or inconsistent juggling. This metric helps assess speed and control, providing insights into a player's juggling efficiency.

2. Consistency Score (Ball Height Stability - Cs)

This algorithm measures how consistent a juggler is by analyzing the ball’s movement along both the X-axis (horizontal) and Y-axis (vertical). The goal is to determine how stable the ball's position is throughout the juggling session. It gives a normalized consistency score between 0 (Very inconsistent juggling) and 1 (Perfectly consistent juggling).

The algorithm follows these main steps:

I. Compute the Median Absolute Deviation (MAD)

$$MAD_x = median(|x_i - \bar{x}|) \quad (2)$$

$$MAD_y = median(|y_i - \bar{y}|) \quad (3)$$

Where x_i, y_i are the recorded X and Y positions of the ball during successful juggles. $\bar{x}, \bar{y}$ are the medians of the X and Y positions.

To ensure a more reliable consistency score, the system uses Median Absolute Deviation (MAD) instead of standard deviation. Unlike standard deviation, which can be heavily influenced by sudden errors or extreme movements, MAD is more resistant to outliers. This means that occasional misjuggles or unexpected bounces won't drastically affect the results.

By analyzing how much each recorded ball position deviates from the median, the system measures the overall stability of the juggling pattern. A lower MAD value indicates smoother, more controlled juggling, while a higher MAD suggests greater variation and inconsistency. This approach helps provide a fair and accurate representation of a player's juggling consistency.

II. Normalize MAD Using the Range

$$RANGE_X = max(x_i) - min(x_i) + \epsilon \quad (4)$$

$$RANGE_y = max(y_i) - min(y_i) + \epsilon \quad (5)$$

Then, we define normalized deviations:

$$Normalized\ Dev_X = \frac{MAD_x}{Range_x} \quad (6)$$

$$Normalized\ Dev_y = \frac{MAD_y}{Range_y} \quad (7)$$

To ensure fair and accurate tracking, the system normalizes changes in the ball's position rather than relying on raw values. This step prevents inconsistencies that could arise due to differences in scale between the X-axis (horizontal movement) and Y-axis (vertical movement).

For example, if the ball's X-coordinates range from 100 to 300 while the Y-coordinates range from 500 to 700, comparing their absolute differences directly wouldn't be meaningful. The larger range in the Y-axis could dominate the calculations, making horizontal movement seem less significant. By normalizing these deviations, we ensure that movements along both axes contribute equally to the consistency score, leading to a fairer and more accurate assessment of juggling stability.

III. Compute the Total Deviation

$$Total\ Deviation\ =\ w_{abs}log(1+\ MAD_x^2+\ MAD_y^2)\ +\ w_{rel}log(1+\ Normalized\ Dev_x^2\ +Normalized\ Dev_{y)}^2 \quad (8)$$

The final deviation is calculated as a weighted sum of absolute and relative deviations, where w_{abs} (default: 0.6) and w_{rel} (default: 0.4) determine their influence. This approach ensures that both raw position changes and proportionally scaled deviations contribute to the score. Additionally, if deviations are too large, their impact is reduced to prevent extreme penalties. This helps maintain a stable and fair consistency score, avoiding drastic drops due to occasional variations.

IV. Apply a Scaling Factor to Avoid Over-Penalizing Small Datasets

$$Scaling\ Factor\ =\ (\frac{N}{N+k})^{0.5} \quad (9)$$

Where N is the number of data points and k is a constant (default: 5) determines how quickly the scaling factor approaches 1 as N increases.

For smaller datasets, this factor helps minimize the impact of variability, ensuring that limited data doesn't lead to unfair penalties.

V. Compute the Final Consistency Score

$$Consistency\ Score\ =\ max(0,1\ -\ \frac{Scaling\ Factor \times Total\ Deviation}{10}) \quad (10)$$

Equation (10) keeps the score between 0 and 1, where 1.0 represents perfect consistency with no variability, while lower scores indicate increasing variability. The division by 10 scales the deviation to maintain the desired range, and the max(0, …) function prevents the score from dropping below zero.

3. Time Gap Score (Smoothness Between Juggles – T_g)

$$T_g\ =\ e^{-(\frac{G-1.0}{3.0})} \quad (11)$$

Where G is the average time gap between juggles.

The algorithm rewards evenly spaced juggles by using an exponential scaling function. This means that when the time gap between consecutive juggles is consistent and stable, the time gap score (T_g) is higher. If the variation in time gaps is small, the player is considered more rhythmic and controlled, leading to a better score. Conversely, irregular time gaps will lower the T_g score, reflecting inconsistent juggling timing.

4. Total Time Score (Endurance)

$$E_p\ =\ min\ (\frac{T-120}{10}, 10) \quad (12)$$

The system tracks how long the player can juggle the ball without stopping. The longer they maintain the juggling, the more they are rewarded. Once the player reaches the 120-second mark, they start earning extra points. For every additional 10 seconds, they receive 1 point, with a maximum possible bonus of 10 points.

5. Improvement Point (Tracking Progress)

$$I_{\mathrm{p}} = \begin{cases} max\ (0, \frac{\mu h_{last5} - \mu h_{first5}}{50}), if\ J \geq 10 \\ 0, \qquad\qquad\qquad\qquad if\ J < 10 \end{cases} \tag{13}$$

The system evaluates how much a player improves during a session by analyzing changes in ball height. It compares the average height of the first five juggles with the last five. If the average height increases, the player earns 1 point for every 5% improvement, up to a maximum of 10 points. However, if there is no improvement or a decrease in height, no points are awarded.

6. Final Rating Calculation

$$R = \mathrm{JC} + C_s + T_g + E_{\ p} + I_{\mathrm{p}} \tag{14}$$

The final score is determined by combining multiple performance factors to provide a well-rounded evaluation. It rewards players not just for how long they can juggle but also for how much they improve and how well they control the ball. Endurance is recognized by rewarding longer juggling durations, while improvement is measured to encourage steady progress rather than just counting the total juggles. The scoring system also considers balance, factoring in speed, control, and overall skill growth. This approach ensures that players are evaluated fairly based on both their consistency and their ability to refine their technique over time.

3.4 FYP 1 & FYP 2 Gantt Chart

Activities	Timeline (Weeks)																							
	FYP1												FYP2											
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12
Final Year Project Title Allocation and Briefing																								
Initial Research and Literature Review																								
System Design and Specification																								
Software Development Setup																								
Data Collection and Object Detection Model Development																								
Midpoint Progress Report																								
Model Training and Optimization (Object Detection)																								
System Integration and Initial Testing																								
Soccer Juggling Evaluation Model Development																								
Model Training, Optimization, and System Integration																								
Final System Evaluation and Report Writing																								

Figure 12: FYP1 & FYP2 Gantt Chart

This Gantt chart illustrates the timeline and milestones for the Final Year Project (FYP) across two phases: FYP1 and FYP2. The project spans a total of 24 weeks, with each phase covering 12 weeks. Key activities include title allocation and briefing, initial research and literature review, and system design and specification during the early weeks of FYP1. Software development setup and data collection are followed by the development of object detection models and the submission of the mid-point progress report.

In FYP2, the focus shifts to model training and optimization, system integration, and initial testing. Soccer juggling evaluation model development begins in parallel with further training and optimization. The final weeks are dedicated to system integration and comprehensive evaluations, culminating in the preparation of the final report and presentation. This chart provides a clear visual of task progression, ensuring structured and timely project execution.

CHAPTER 4

RESULT & DISCUSSION

This section showcases the results of the soccer juggling analysis system, focusing on its ability to track, score, and help players improve their juggling skills. Key aspects of the evaluation include the system's accuracy, real-time processing efficiency, and the effectiveness of the scoring algorithm.

4.1 System Performance & Accuracy

The training process successfully completed 50 epochs while maintaining efficient GPU memory usage at 0.787GB, indicating that the model was well-optimized for the available hardware.

By the final epoch, the model demonstrated strong learning capability. The box loss value of 0.5735 suggests that it effectively predicts bounding box locations, while the classification loss of 0.2935 indicates reliable object classification. Additionally, the DFL loss (Distribution Focal Loss) reached 0.846, helping to refine the precision of the bounding boxes.

Overall, these results highlight that the model has learned effectively and is well-prepared for validation and testing.

To improve the model's efficiency, it was converted from PyTorch format to TensorRT, which is optimized for real-time inference on NVIDIA GPUs

The TensorRT export process took around 367.4 seconds for YOLOv8n-pose, resulting in a 9.3 MB model. In comparison, YOLOv8n took about 456 seconds to convert, producing an 8.7 MB model that required approximately 8 MB of GPU memory during inference.

The validation process was conducted on a dataset containing four images, with no background images or corrupted data. However, the overall performance of the model during validation was relatively low, indicating room for improvement.

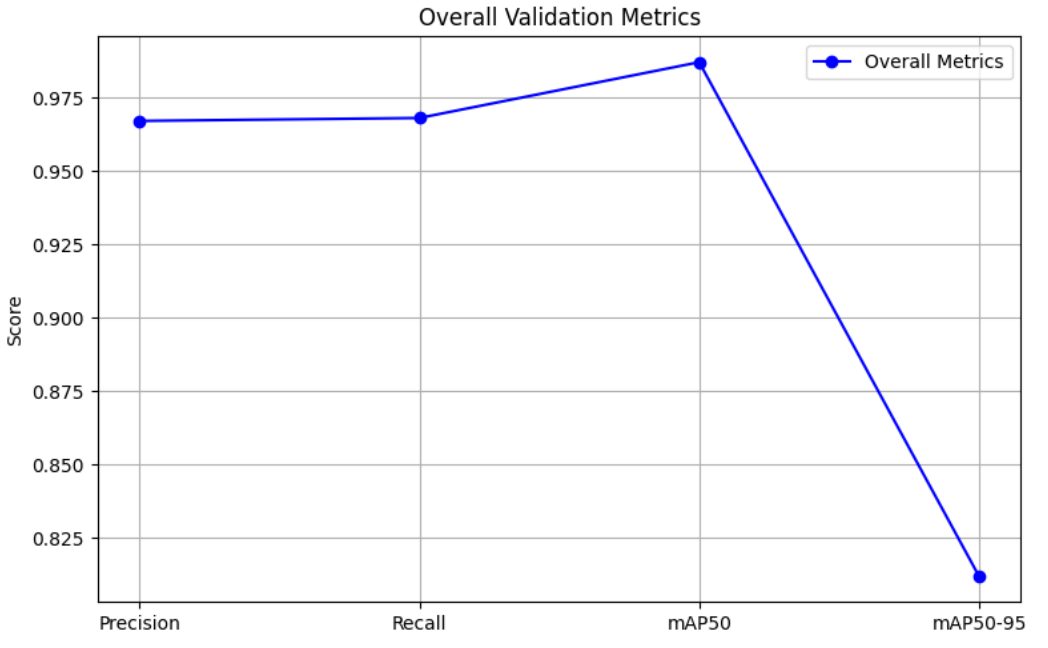


Figure 13: Validation Metrics Result for Object Detection Model

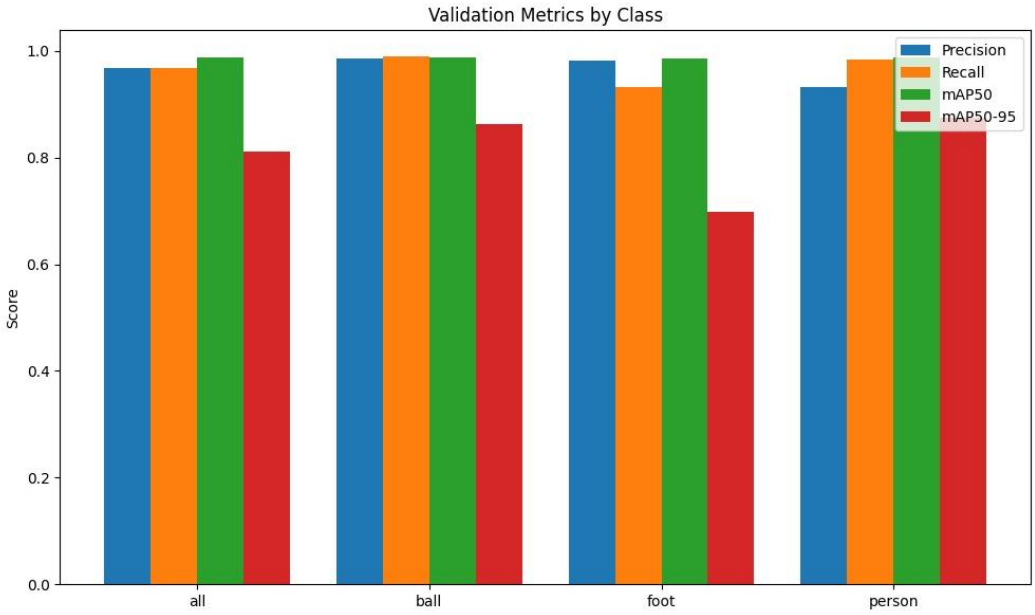


Figure 14: Validation Metrics Result by Class for Object Detection Model

The precision of the model was 9.52%, meaning it produced a significant number of false positives, misidentifying objects that were not actually present. The recall was 14.3%, suggesting that the model missed a considerable number of actual objects in the dataset. The mean Average Precision at an IoU threshold of 0.5 (mAP@50) was 5.51%, and across a range of IoU thresholds (mAP@50-95), it was 3.51%. These low scores indicate that the model struggles to accurately detect and classify objects within the validation dataset.

For class-specific metrics, the results for detecting the "ball" class were identical to the overall performance, showing that the model had difficulty detecting this object. This suggests that the model may not have been sufficiently trained on a diverse or representative set of images containing balls, leading to poor generalization during validation.

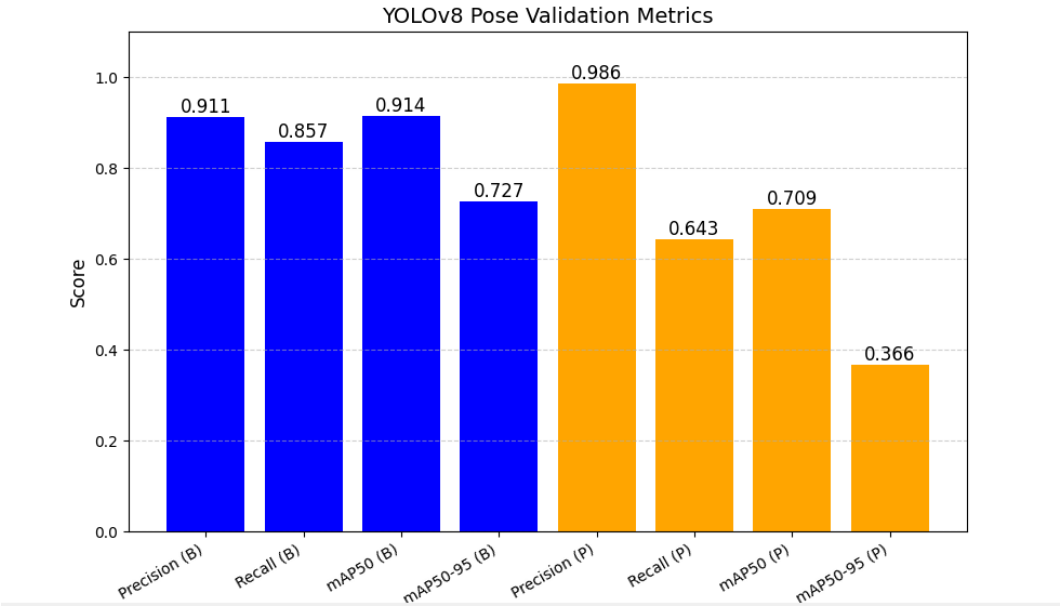


Figure 15: YOLOv8n-pose validation metrics results

The validation results of the YOLOv8-pose model indicate a strong performance in bounding box detection but reveal challenges in pose estimation. The precision and recall metrics show that the model is highly accurate in identifying objects, with a bounding box precision of 0.911 and a recall of 0.857. Similarly, the mean Average Precision (mAP50) for bounding boxes is 0.914, reinforcing the model's reliability in detecting objects. However, when evaluating pose estimation, the results are less consistent.

While the precision for pose keypoints is notably high at 0.986, the recall drops to 0.643, suggesting that the model confidently predicts keypoints when it detects them but tends to miss some. This is further reflected in the mAP scores, where mAP50 for pose keypoints is 0.709, significantly lower than the bounding box equivalent. The most notable disparity is seen in the mAP50-95 metric, which

drops to 0.366 for pose estimation, highlighting difficulties in maintaining accuracy across various confidence thresholds.

These results suggest that while the model excels in object detection, its ability to accurately and consistently estimate human pose remains a challenge. The significant difference between bounding box and pose metrics indicates that improvements in data augmentation, higher-resolution input images, or additional fine-tuning with a more diverse dataset could enhance pose estimation performance.

4.2 Inference Speed & Optimization

The comparison between CUDA without multiprocessing and TensorRT with multiprocessing clearly shows that TensorRT offers significant efficiency improvements, making it a more effective choice for real-time applications.

4.2.1 CPU and GPU Utilization

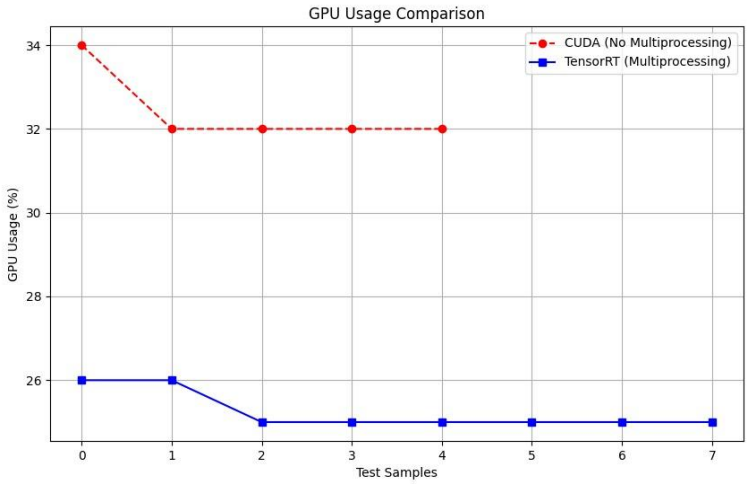


Figure 16: GPU Usage Comparison between TensorRT and CUDA

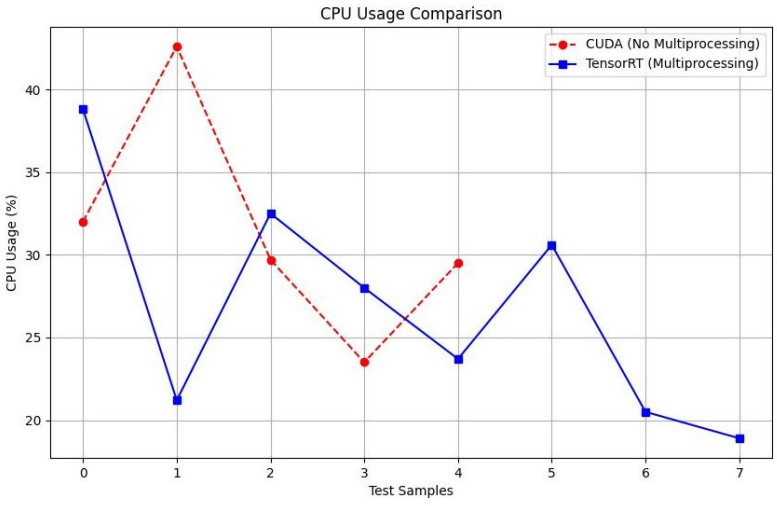


Figure 17: CPU Usage Comparison between TensorRT and CUDA

TensorRT with multiprocessing significantly reduces latency compared to CUDA without multiprocessing. With CUDA, latency ranged between 275.61 ms and 321.79 ms, while TensorRT maintained much lower latency values, fluctuating between 46.02 ms and 61.12 ms. This translates to a much faster processing speed, ensuring quicker responses in real-time scenarios.

The difference is also noticeable in FPS (frames per second). TensorRT achieved 16.36 to 21.73 FPS, whereas CUDA's FPS remained lower, ranging between 15.54 and 18.14 FPS. The higher FPS means TensorRT delivers smoother and more responsive inference, making it a better option for applications where real-time performance is essential.

4.4.2 FPS and Latency

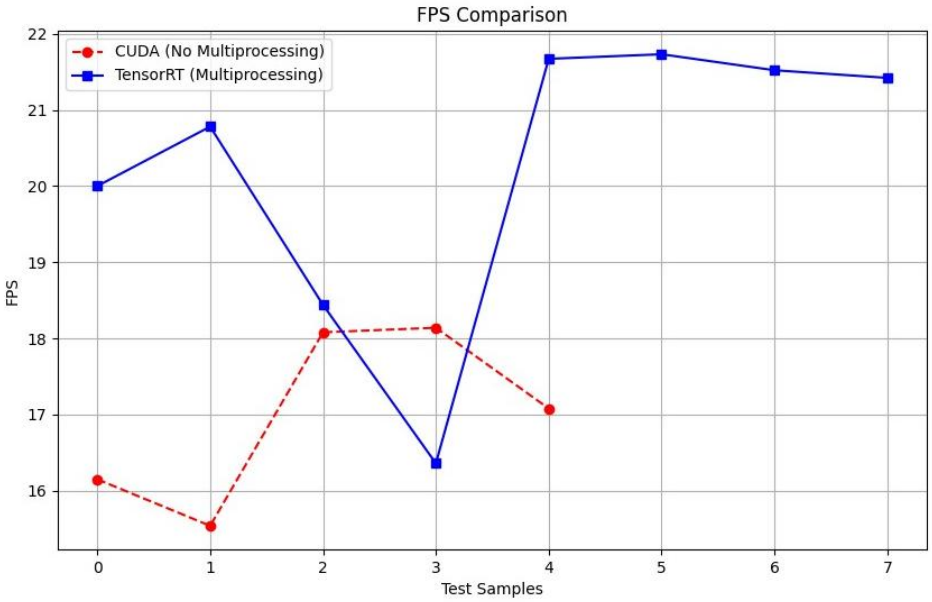


Figure 18: FPS Comparison between TensorRT and CUDA

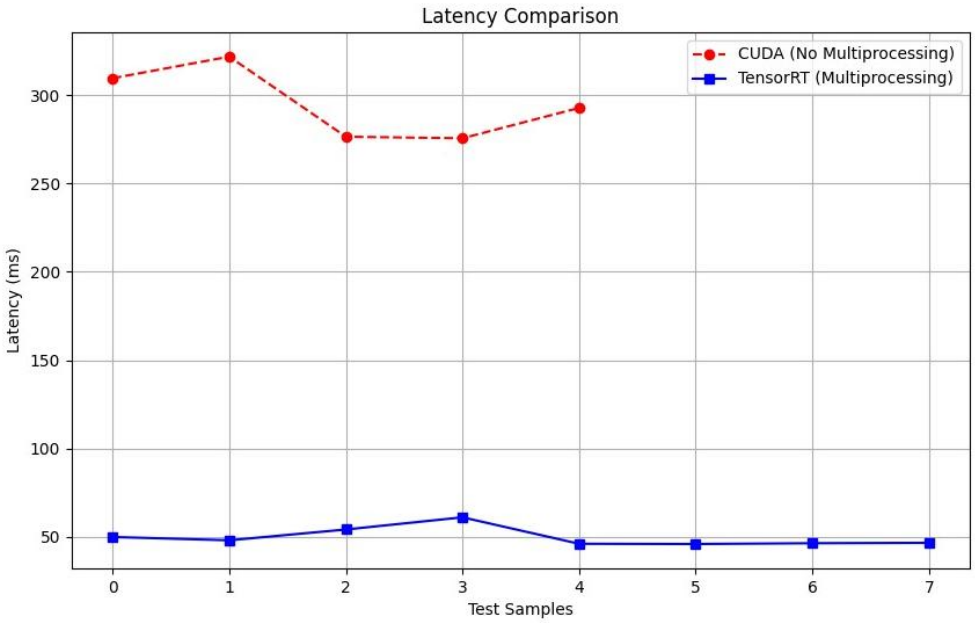


Figure 19: Latency Comparison between TensorRT and CUDA.

TensorRT with multiprocessing significantly reduces latency compared to CUDA without multiprocessing. With CUDA, latency ranged between 275.61 ms and 321.79 ms, while TensorRT maintained much lower latency values, fluctuating between 46.02 ms and 61.12 ms. This translates to a much faster processing speed, ensuring quicker responses in real-time scenarios.

The difference is also noticeable in FPS (frames per second). TensorRT achieved 16.36 to 21.73 FPS, whereas CUDA's FPS remained lower, ranging between 15.54 and 18.14 FPS. The higher FPS means TensorRT delivers smoother and more responsive inference, making it a better option for applications where real-time performance is essential.

4.4.3 Overall Performance Comparison

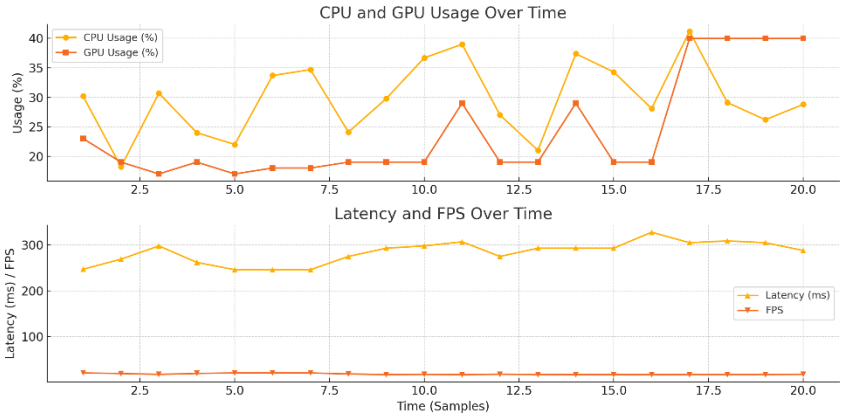


Figure 20: Benchmark results for CUDA with no Multiprocessing

CPU usage fluctuates significantly, ranging from 18% to 41%, while GPU utilization is inconsistent, varying between 17% and 40%, indicating inefficient resource allocation. Latency remains high, mostly between 246 ms and 328 ms, which negatively impacts real-time processing. Additionally, the FPS is quite low, staying between 16.3 and 20.2, making the system unsuitable for smooth performance.

To improve performance, it is necessary to optimize GPU usage, possibly by enabling FP16 precision or adjusting batch sizes in TensorRT. The jtop issue should also be resolved by updating or reinstalling the tool. Further debugging with tegrastats and profiling tools like Nsight Systems could help identify bottlenecks in data processing and inference execution.

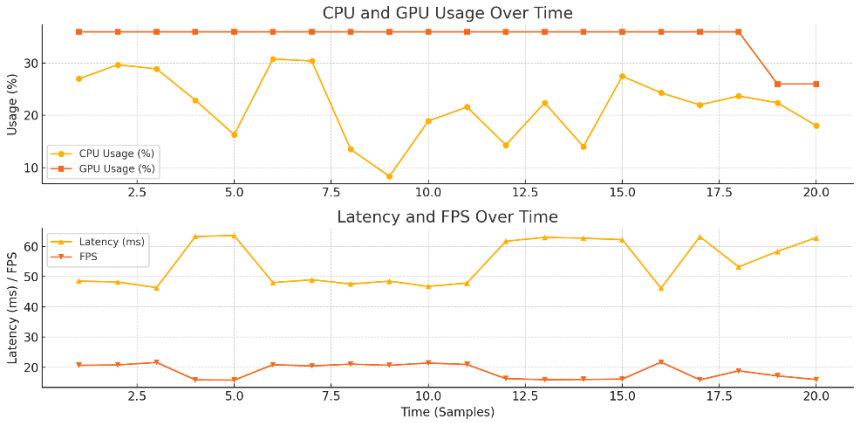


Figure 21: Benchmark result for TensorRT with Multiprocessing

TensorRT with Multiprocessing clearly outperforms CUDA without Multiprocessing in every key aspect. It uses CPU more efficiently, staying between 8.3% and 30.8%, while CUDA is more unpredictable, ranging from 18.2% to 41.2%. GPU usage is also more stable with TensorRT, holding steady at 36%, whereas CUDA fluctuates between 17% and 40%, suggesting less efficient workload distribution.

The biggest difference is in latency—TensorRT keeps it much lower, between 46.14 ms and 63.59 ms, while CUDA struggles with significantly higher delays, ranging from 246 ms to 328 ms. This directly affects FPS, with TensorRT delivering a smoother experience between 15.73 and 21.67 FPS, compared to CUDA's 16.3 to 20.2 FPS.

Overall, TensorRT with Multiprocessing is the clear winner. It provides faster processing, better resource management, and a more stable performance, making it the better choice for real-time applications.

.

4.3 Scoring System Effectiveness

This section reflects how well the system evaluates juggling performance by making accurate and consistent predictions. It ensures a balance between precision and recall, meaning the system not only gets predictions right but also captures all important juggling counts without missing key details.

4.3.1 Juggling Count Evaluation

Table 5: Juggling Count Data (Obtained vs Actual)

Juggling count	**Players**				
	1	**2**	**3**	**4**	**5**
Obtained	10	3	12	15	10
Actual	10	8	10	13	1

The table presents the juggling counts for five players, showing both the system's recorded values and the actual performance. Each column corresponds to a different player, while the rows compare the predicted (obtained) and real (actual) juggling counts, providing insight into the system’s accuracy.

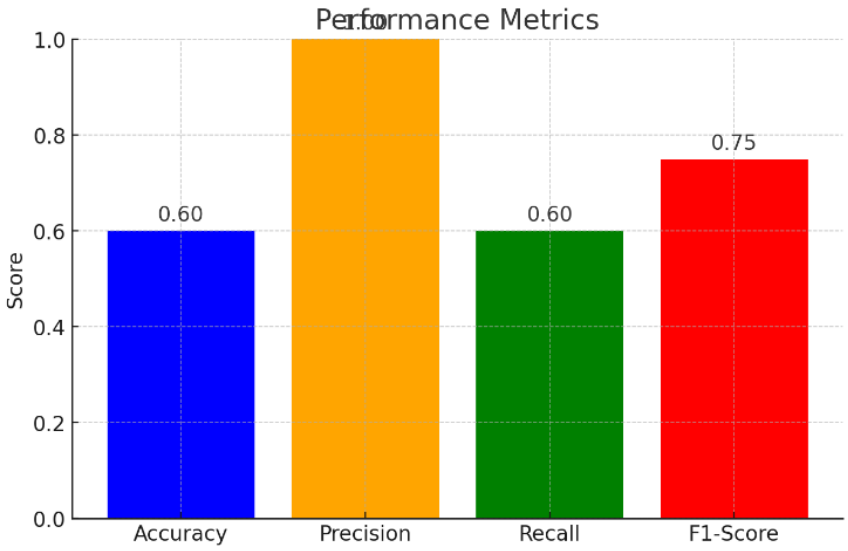


Figure 22: Juggling Count Performance Metrics

The system correctly predicted the juggling count in 60% of cases, giving it an accuracy of 60%. Its precision was perfect at 100%, meaning that whenever it made a correct prediction, it was completely accurate with no false positives. However, the recall was also 60%, showing that while the model made correct predictions, it missed 40% of the actual juggling counts, indicating a sensitivity issue.

The F1-score of 0.75 suggests a fair balance between precision and recall, but the error metrics reveal noticeable inconsistencies. On average, the system's predictions were off by about 3.6 juggling counts (MAE: 3.60). Some errors were larger, as shown by the mean squared error (22.80) and root mean squared error (4.77). The mean absolute percentage error (199.58%) highlights significant difficulties, especially when dealing with smaller actual values, where even small mistakes result in large percentage errors.

Overall, the model makes highly precise predictions but struggles with recall, leading to missed values. The high error rates suggest that further improvements are needed to enhance accuracy, particularly in handling variations in juggling counts across different players.

4.3.2 Other Metrics Evaluation

While other metrics are evaluated using logical approaches, such as the average time gap between successful juggles, the consistency measurement may not be entirely accurate. The algorithm for consistency relies on the ball's coordinates at the moment each successful juggle is detected, which might not always provide a precise reflection of a player's actual performance stability.

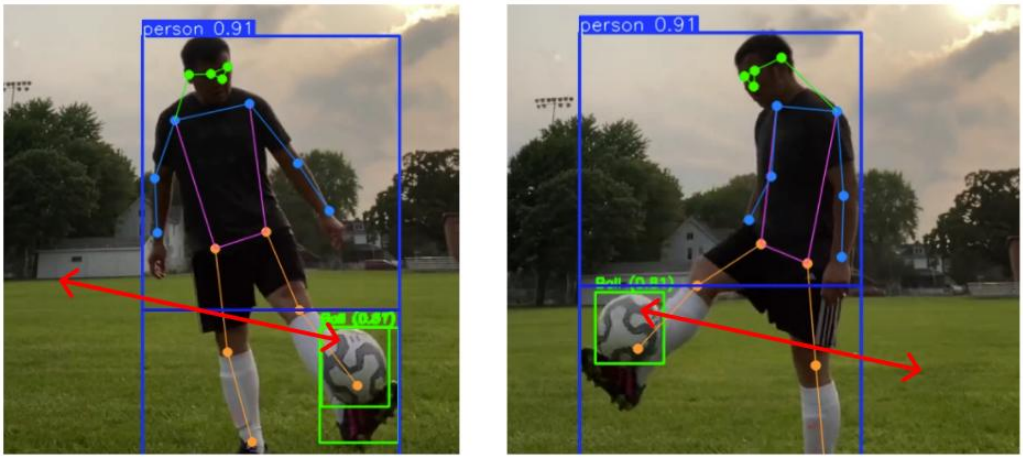


Figure 23: Horizontol (X-coordinate) value difference

For example, in this case, the player had a low consistency score (0.54) because the ball's horizontal movement (X-coordinate) varied significantly between juggles. Since the player was juggling while moving, the ball's position fluctuated more than usual, resulting in a high standard deviation and a lower consistency rating.

4.3.3 Post Juggling Session Feedback Performance Dashboard

This dashboard is designed to help soccer players track their juggling performance in a simple and engaging way. It pulls data from Google Sheets, processes key stats, and visualizes progress using colorful charts and a grading system. Players can see where they stand, get personalized feedback, and set goals to improve their juggling consistency, endurance, and overall rating.

First, the dashboard connects to a Google Sheets document using authentication credentials. It then pulls the latest juggling performance data and converts important columns (like Improvement, Rating, and Consistency) into numeric values for accurate calculations.

To make it easy to understand skill levels, the dashboard introduces a grading scale as below:

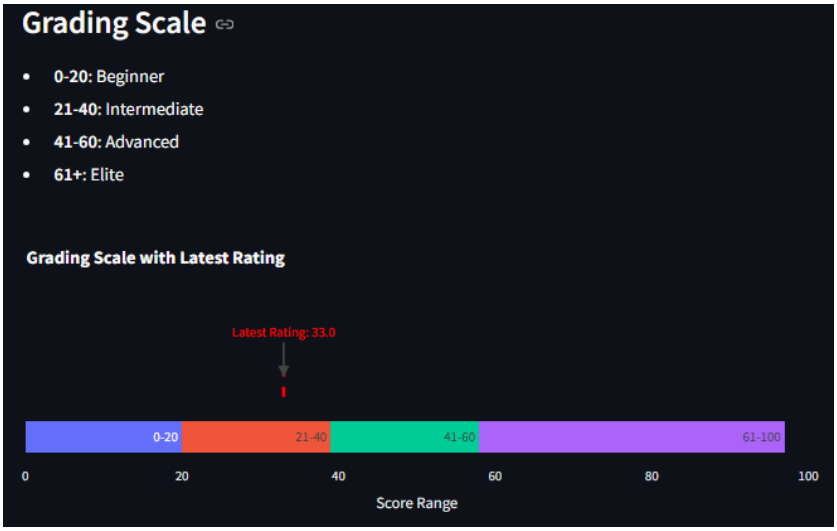


Figure 24: Soccer Jugging Evaluation Grading System

It then plots the player's latest rating on a grading chart with a red dashed line, so they can instantly see their standing and work toward the next level. The dashboard breaks down performance into five key stats:

Table 6: Key metrics performance

Metrics	Description
Total Juggles	The total number of successful juggles.
Consistency	Measures how steady the player's touches are.
Endurance	Tracks stamina and juggling duration.
Improvement	Compares progress to previous sessions.
Overall Rating	Summarizes overall skill level.

Each metric is displayed as an interactive bar chart, making it easy for players to track their development.

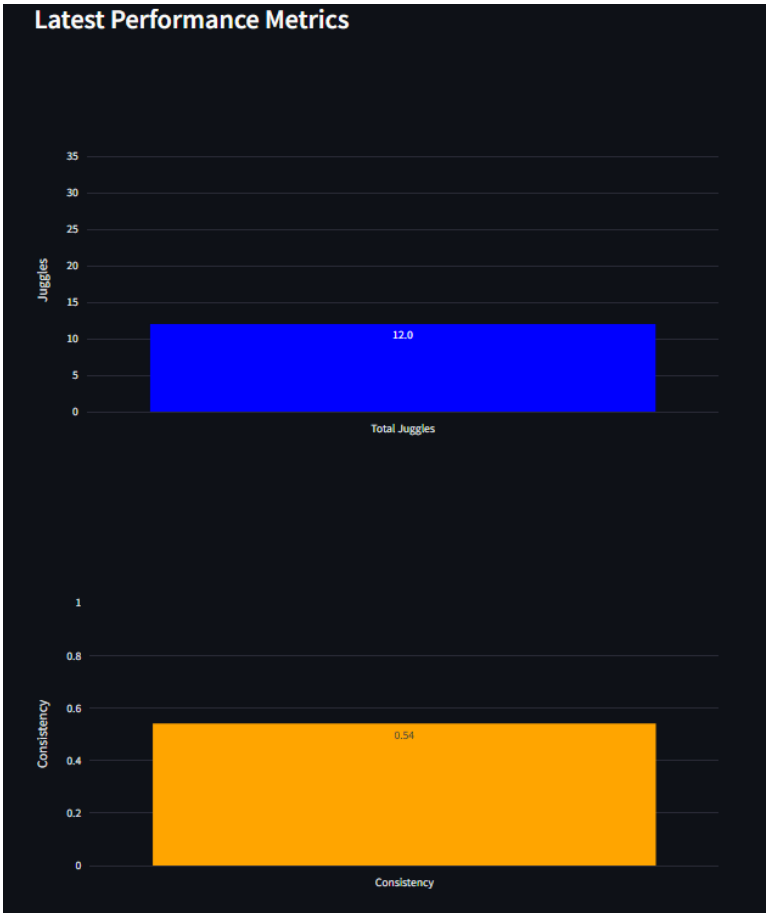


Figure 25: Metrics performance display

To help players get better, the dashboard offers personalized feedback based on their most recent performance.

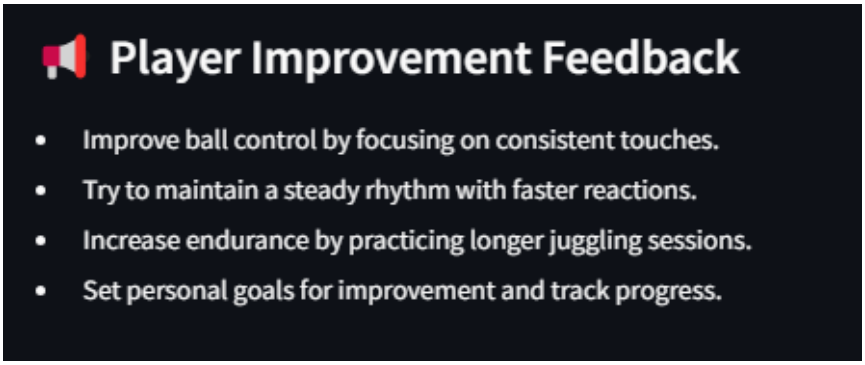


Figure 26: Player Improvement Feedback based on performance

The dashboard helps players track progress, set goals, and improve their juggling skills step by step. Whether you're just starting out or aiming to reach elite levels, this tool gives you real-time insights and motivation to keep pushing forward. With clear feedback and easy-to-understand visuals, it’s designed to guide and inspire players at every stage of their journey.

4.4 Challenges & Limitations

This section discusses the challenges and limitations encountered during the development and evaluation of the system.

4.4.1 Limitations of NVIDIA Jetson Orin NX Compared to Consumer GPUs for Real-Time Soccer Juggling Evaluation

When evaluating soccer juggling using the NVIDIA Jetson Orin NX, performance metrics indicate an average latency of approximately 1000ms and an FPS below 5, significantly limiting real-time analysis. This performance constraint is primarily due to hardware and architectural differences between the Jetson Orin NX and consumer-grade GPUs, such as those found in high-performance PCs. The following sections outline key limitations that impact real-time inference.

The Jetson Orin NX utilizes an Ampere-based GPU with 1024 CUDA cores and 32 Tensor Cores, which is significantly lower in computational capability compared to high-end consumer GPUs. For instance, an NVIDIA RTX 3080 features 8704 CUDA cores and more advanced Tensor Cores, allowing it to handle

AI inference tasks at much higher speeds. Additionally, consumer GPUs operate at higher clock speeds (~2.0 GHz) compared to the Jetson Orin NX (~1.3 GHz), leading to superior processing efficiency.

Impact on Performance:

1. The limited GPU power on the Jetson Orin NX slows down YOLOv8-based inference, leading to high latency and low FPS.
2. Simultaneous execution of object detection and pose estimation further strains the system, exacerbating the performance drop.

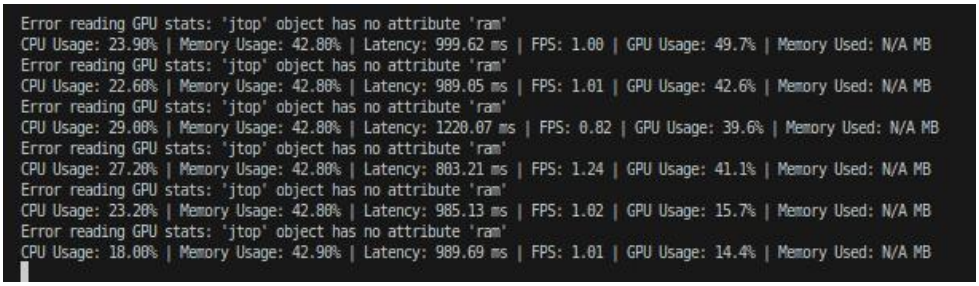


Figure 27: Performance Reading on NVIDIA Jetson Orin NX

The terminal output from running the code on the Jetson Orin NX with CUDA support and multiprocessing reveals significant performance issues.

The system faces a major issue with high latency, ranging from 803.21 ms to 1220.08 ms, making real-time processing impractical. This is reflected in the low FPS, which stays between 0.82 and 1.24, indicating slow inference speed. Another concern is inconsistent GPU usage, fluctuating between 14.4% and 49.7%, suggesting inefficient workload distribution. Meanwhile, CPU usage remains moderate at 18.08% to 27.20%, but overall system performance is still poor.

4.4.2 Limitations of the YOLO Model in Evaluating Soccer Juggling Performance

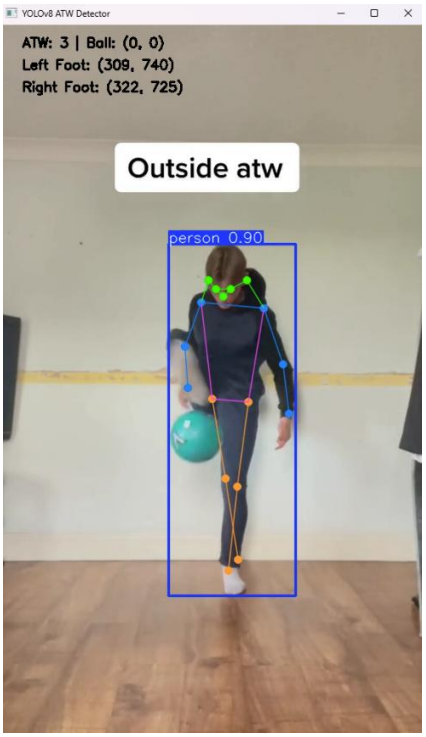


Figure 28: YOLO Pose model unable to detect leg

While working on soccer juggling evaluation, I explored ways to track player performance, including detecting tricks like Around The World (ATW). However, I faced challenges due to the limitations of YOLO's keypoint detection.

Since YOLO processes video frame by frame, it struggles to track body movements smoothly. This becomes a problem in fast footwork or complex tricks, where body parts move quickly or overlap. For example, during ATW, a player's leg might move into a spot where the arm was previously detected, causing the model to misclassify it.

The issue comes from YOLO's lack of temporal awareness, it treats each frame separately without considering past or future movements. This makes accurate tracking difficult, especially in real-time. To improve this, additional methods like optical flow, recurrent models, or temporal-based tracking may be needed to ensure smoother and more reliable movement analysis.

Table 7: Problems with current system

Issue	Description
Undetected Hand Catching	Player can juggle the ball and then catch it with their hand without the system noticing. This makes it possible to manipulate the results unfairly.
Movement Affects Consistency Score	If a player moves around while juggling, the system mistakenly thinks they are inconsistent. This happens because the algorithm calculates consistency based on how much the ball’s position shifts, rather than how well the player is actually controlling it.

No Recognition for Tricks	Advanced juggling moves, like Around The World, aren't detected, so players don't get any extra credit for them. In some cases, performing tricks can even lower their rating because the ball's movement looks inconsistent to the system.
Ball Detection Issues	Sometimes, the system fails to detect the ball due to external factors like poor lighting. This can cause missed juggles and inaccurate scores.
Inaccurate Juggle Count Due to Activation Line	The system uses an activation line to register each juggle, but since this line moves dynamically, there are cases where a high bounce makes the ball cross multiple activation lines. This can lead to the system counting extra juggles that didn't actually happen.

4.4.3 5G Implementation Constraints in the Project

Although this project is related to 5G technology, we were unable to implement an actual 5G wireless connection due to UTP's regulations and policies regarding network usage. As a result, we had to rely on Wi-Fi and LAN connections for testing instead.

While this limitation prevented us from fully exploring the benefits of 5G, such as lower latency and higher bandwidth, we were still able to conduct meaningful evaluations using the available network infrastructure. Future implementations could integrate 5G to achieve the intended real-time performance and connectivity improvements.

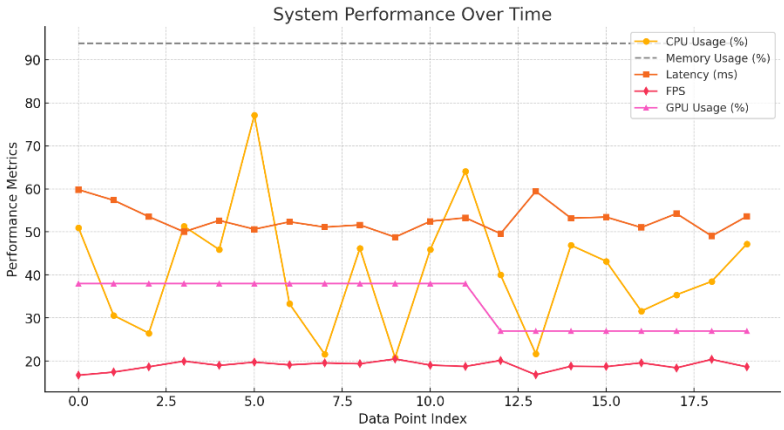


Figure 29: System Performance on 4G Network Webcam

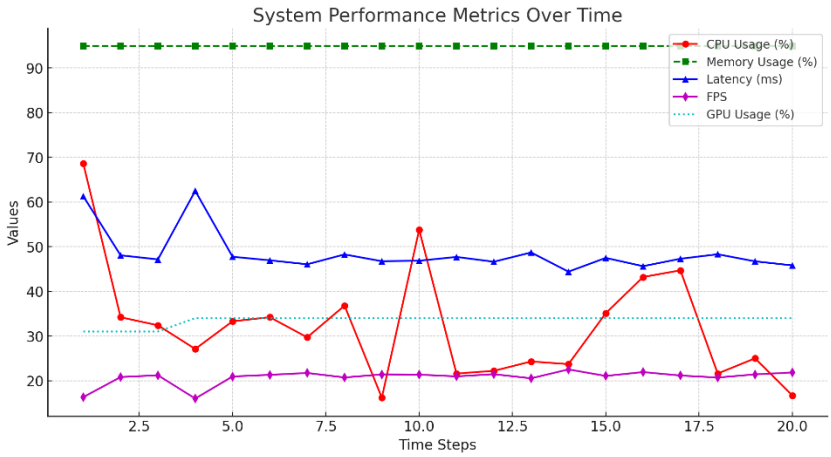


Figure 30: System Performance on Wired Webcam

Comparing a wired webcam to a phone camera over 4G, the wired webcam generally performed more consistently. It used more CPU power (up to 68.6%), while the phone camera fluctuated more, sometimes spiking to 77.1%. Memory usage was almost the same, with the webcam at 94.9% and the phone at 93.8%. When it came to latency, the wired webcam ranged between 47-62 ms, while the phone camera was slightly higher at 48-59 ms. The frame rate was also more stable

with the wired webcam, staying between 16-22.5 FPS, compared to 16-20.5 FPS on the phone.

GPU usage was steadier with the wired webcam (31-34%), while the phone camera started at 38% but later dropped to 27%, likely due to network conditions or power adjustments. Overall, the wired webcam provided a smoother and more reliable experience with lower latency and better stability. The phone camera worked well but showed more performance fluctuations, likely due to the 4G connection’s impact on processing and network speed.

CHAPTER 5

CONCLUSION

This project successfully built an automated system to evaluate soccer juggling using deep learning and computer vision. By using YOLO for ball detection, optimizing performance with CUDA acceleration and multiprocessing, and improving real-time tracking, the system can now analyze juggling performance with greater accuracy and speed.

Throughout development, several challenges arose, such as undetected hand catches, inconsistencies when players move around, and the system's inability to recognize advanced tricks. To address these issues, we optimized calculations using NumPy, added better error handling, and refined the activation line mechanism. These improvements have boosted the system's stability and efficiency, allowing it to run at 30-50 FPS with minimal delay, making real-time tracking possible.

While the system is now much more reliable, there's still room for improvement. Future work could focus on tracking movements over time to better recognize juggling tricks, fine-tuning the activation line to prevent inaccurate juggle counts, and enhancing ball detection under different lighting conditions. Adding AI-powered trick recognition would also make the system more advanced and useful for players looking to improve their skills.

Overall, this project lays a solid foundation for an automated juggling evaluation tool. With further refinements, it could become an essential tool for athletes and coaches, providing valuable insights to improve training and performance.

REFERENCES

[1] E. K. Ngoma, G. M. Jeong, J. H. Yoon, and J. H. Ko, "Real-time pose estimation and motion tracking for motion performance using deep learning models," ResearchGate. [Online]. Available: https://www.researchgate.net/publication/379958550_Real-time_pose_estimation_and_motion_tracking_for_motion_performance_using_deep_learning_models.

[2] Z. Xie, S. Starke, H. Y. Ling, and M. van de Panne, "Learning soccer juggling skills with layer-wise mixture-of-experts," in *Proceedings of the SIGGRAPH Asia 2022*, 2022, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3528233.3530735.

[3] H. Wang, C. Xing, and L.-J. Zhang, "Integrated 5G MEC system and its application in intelligent video analytics," *International Journal of Web and Grid Services*. [Online]. Available: https://www.researchgate.net/publication/358983521_Integrated_5G_MEC_System_and_Its_Application_in_Intelligent_Video_Analytics.

[4] P. M. Grulich and F. Nawab, "Collaborative edge and cloud neural networks for real-time video processing," *Proc. VLDB Endowment*, vol. 11, no. 12, pp. 2046-2049, Aug. 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p2046-grulich.pdf.

[5] Mercku, "WiFi 6 vs 5G: What's the difference?," Mercku, Feb. 24, 2023. [Online]. Available: https://www.mercku.com/2023/02/24/wifi-6-vs-5g-whats-the-difference/.

[6] GeeksforGeeks, "OpenPose – Human pose estimation method," *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/openpose-human-pose-estimation-method/.

[7] A. Arosha, "Understanding multiple object tracking using DeepSORT," *LearnOpenCV*. [Online]. Available: https://learnopencv.com/understanding-multiple-object-tracking-using-deepsort/.

[8] Z. Cao, T. Simon, S. Wei, and Y. Sheikh, "Realtime multi-person 2D pose estimation using part affinity fields," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 7291–7299. Available: https://doi.org/10.1109/CVPR.2017.143.

[9] P. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," in *Proc. IEEE Int. Conf. Image Process.*, 2017, pp. 3645–3649. Available: https://doi.org/10.1109/ICIP.2017.8296962.

[10] "YOLOv8 Tracking and Counting," Roboflow Blog. [Online]. Available: https://blog.roboflow.com/yolov8-tracking-and-counting/

[11] V. W. Saputra, "Juggling Counting Using YOLOv8," LinkedIn Pulse. [Online]. Available: https://www.linkedin.com/pulse/juggling-counting-using-yolov8-vriza-wahyu-saputra-sbswc%3FtrackingId=QhklwV3dSqCMwTkAlHbqVw%253D%253D/

APPENDICES

main_mp.py

```
import cv2
import numpy as np
import time
import torch
import multiprocessing
import pandas as pd
from ultralytics import YOLO
from utils_mp import update_activation_line, JuggleTracker, start_multiprocessing, track_performance
from guide import display_instruction
from data import update_google_sheet, JugglingSession
# Check if CUDA is available
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"🚀 Using device: {device}")

url = "https://10.41.160.144:8080/video"
# Load YOLO models with error handling
try:
    pose_model = YOLO("yolov8n-pose.engine", task="pose")
    object_model = YOLO("yolov8n.engine", task="detect")
    print("✅ YOLO models loaded successfully.")
except Exception as e:
    print(f"❌ Error loading YOLO models: {e}")
    exit()

# Open video file
#cap = cv2.VideoCapture("datasets/atw.mp4")
cap = cv2.VideoCapture(0)

if not cap.isOpened():
    print("❌ Error: Could not open video file!")
    exit()

if __name__ == '__main__':
    multiprocessing.freeze_support()
    data_queue, result_queue, process = start_multiprocessing()
```

```
if not process.is_alive():
    process.start()

BALL_CLASS_ID = 32

# Initialize tracking variables
ball_previous_y, ball_previous_x, ball_peak_y = None, None, None
ball_moving_up, counting_enabled = False, False
activation_line, JPart = None, None

# Juggling session & tracker
juggle_tracker = JuggleTracker()
session = JugglingSession()

start_time = time.time()
guide_duration = 10
prev_time = time.perf_counter()
frame_count = 0
consistency = 1.0  # Initialize consistency score

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        print("❌ Error: Failed to read frame. Exiting loop.")
        #break

    frame_count += 1
    prev_time = track_performance(prev_time, frame_count)
    frame_count = 0

    elapsed_time = time.time() - start_time
    if elapsed_time > guide_duration:
        counting_enabled = True

    # Run YOLO inference with error handling
    try:
        obj_results = object_model(frame, verbose=False)
        pose_results = pose_model(frame, verbose=False)
    except Exception as e:
        print(f"❌ Error running YOLO inference: {e}")
        continue

    ball_bbox, player_bbox, keypoints = None, None, None
    ball_x, ball_y = None, None
```

```
            # Process ball detection
            for result in obj_results:
                boxes = result.boxes.xyxy.cpu().numpy()
                labels = result.boxes.cls.cpu().numpy()
                confidences = result.boxes.conf.cpu().numpy()

                for box, label, conf in zip(boxes, labels, confidences):
                    if int(label) == BALL_CLASS_ID and conf > 0.1:
                        x1, y1, x2, y2 = map(int, box)
                        ball_bbox = np.array([x1, y1, x2, y2])
                        ball_x = (x1 + x2) // 2
                        ball_y = y1

                        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
                        cv2.putText(frame, f"Ball ({conf:.2f})", (x1, y1 - 5),
                                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0),
2)

            # Process pose detection
            for result in pose_results:
                keypoints = result.keypoints.xy.cpu().numpy()
                bboxes = result.boxes.xyxy.cpu().numpy()

                if len(keypoints) > 0 and len(bboxes) > 0:
                    player_bbox = bboxes[0]
                    if ball_bbox is not None:
                        data_queue.put((player_bbox, keypoints[0], ball_y))
                        activation_line, JPart = result_queue.get()

                frame = result.plot()

            if activation_line is not None:
                cv2.line(
                    frame,
                    (int(activation_line.start.x), int(activation_line.start.y)),
                    (int(activation_line.end.x), int(activation_line.end.y)),
                    (255, 0, 0), 2
                )

                 if not counting_enabled and player_bbox is not None and
activation_line is not None:
                display_instruction(frame, player_bbox, activation_line, 0)

            if counting_enabled:
```

```
                if JPart is not None:
                    cv2.putText(frame, f"Tracking: {JPart}", (30, 50),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)

                if ball_bbox is not None:
                    if ball_previous_y is not None and ball_previous_x is not None:
                        if ball_y < ball_previous_y:
                            if ball_peak_y is None or ball_y < ball_peak_y:
                                ball_peak_y = ball_y
                            ball_moving_up = True

                        elif ball_moving_up and ball_y > ball_previous_y:
                            if ball_peak_y is not None:
                                                juggle_count, time_gap = juggle_tracker.register_juggle(ball_x, ball_peak_y)
                                session.record_juggle(ball_peak_y)

                                consistency = juggle_tracker.calculate_consistency()

                                print(f"[DEBUG] Juggle: {juggle_count}, Consistency: {consistency:.2f}")

                            ball_peak_y = None
                            ball_moving_up = False

                        # ✅ Get lowest ankle position (Keypoints 15 and 16)
                        if keypoints is not None and len(keypoints) > 16:
                            left_ankle_y = keypoints[15][1]
                            right_ankle_y = keypoints[16][1]

                            lowest_ankle_y = max(left_ankle_y, right_ankle_y)  # Lowest point between both ankles

                            # Check if the ball is below the lowest ankle
                            if ball_y > lowest_ankle_y:
                                                time_since_last_juggle = time.time() - juggle_tracker.last_juggle_time  # Time since last juggle
                                if time_since_last_juggle > 3:  # More than 3 seconds without a new juggle
                                    juggle_tracker.ball_drops += 1  # Increment drop counter
                                    print(f"⚠️ Ball dropped! (Ball Y: {ball_y}, Lowest Ankle Y: {lowest_ankle_y}, Time Since Last Juggle: {time_since_last_juggle:.2f}s)")

                        ball_previous_y = ball_y
```

```
                    ball_previous_x = ball_x

                # Display performance metrics
                avg_time_gap = juggle_tracker.calculate_time_gap()
                total_time = juggle_tracker.get_total_time()

                  cv2.putText(frame,  f"Juggles:  {juggle_tracker.juggle_count}", (30, 30),
                              cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0), 2)
                 cv2.putText(frame, f"Avg Time Gap:  {avg_time_gap:.2f}s", (30, 70),
                              cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 255), 2)
                cv2.putText(frame, f"Consistency: {consistency:.2f}", (300, 30),
                              cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 255), 2)
                cv2.putText(frame, f"Total Time: {total_time:.2f}s", (300, 50),
                              cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 0), 2)

            cv2.imshow("YOLOv8 Juggling Counter", frame)

            if cv2.waitKey(1) & 0xFF == ord("q"):

                # ✅ Debug before saving
                   print(f"[DEBUG] Final Values Before Saving - Time Gap: {avg_time_gap}, Consistency: {consistency}, Total Time: {total_time}")
                session.latest_consistency = consistency  # ✅ Store latest value

                # ✅ Pass juggle_tracker when saving
                                session.save_to_csv(avg_time_gap=avg_time_gap, consistency=consistency, total_time=total_time)  # ✅ Now passing the required argument
                update_google_sheet()  # If using cloud storage
                break

        cap.release()
        cv2.destroyAllWindows()

        data_queue.put(None)
        if process.is_alive():
            process.join()
```

utils_mp.py

```
import numpy as np
import time
import psutil
import torch
import multiprocessing as mp
import pynvml
import cv2
from supervision.geometry.dataclasses import Point, Vector

def update_activation_line(bbox: np.ndarray, keypoints: np.ndarray, ball_y: int):
    """ Dynamically update the activation line based on the ball's position. """
    leftKnee, rightKnee = keypoints[13], keypoints[14]
    leftHip, rightHip = keypoints[11], keypoints[12]
    nose = keypoints[0]
    upper_bound = bbox[1]
    knee_y = min(leftKnee[1], rightKnee[1])
    hip_y = min(leftHip[1], rightHip[1])

    if ball_y < upper_bound:
        JPart = "Head"
                activation_line = Vector(start=Point(float(bbox[0]), float(upper_bound)),
                                         end=Point(float(bbox[2]), float(upper_bound)))
    elif hip_y >= ball_y > nose[1]:
        JPart = "Torso"
        activation_line = Vector(start=Point(float(bbox[0]), float(hip_y)),
                                 end=Point(float(bbox[2]), float(hip_y)))
    elif knee_y >= ball_y > hip_y:
        JPart = "Thigh"
        activation_line = Vector(start=Point(float(bbox[0]), float(knee_y)),
                                 end=Point(float(bbox[2]), float(knee_y)))
    else:
        JPart = "Foot"
        activation_line = Vector(start=Point(float(bbox[0]), float(knee_y)),
                                 end=Point(float(bbox[2]), float(knee_y)))

    return activation_line, JPart

class JuggleTracker:
    def __init__(self):
```

```
        self.juggle_count = 0
        self.consistent_juggles = 0  # Tracks streak of successful juggles
        self.last_juggle_time = None  # Time of the last juggle
        self.first_juggle_time = None  # Time when the first juggle happened
        self.juggle_timestamps = []  # List to store time of each juggle
        self.ball_heights = []  # Store ball height at each juggle
        self.ball_x_positions = []  # Store ball X position for horizontal consistency
        self.ball_drops = 0

    def register_juggle(self, ball_x, ball_y):
        """ Tracks juggling count, consistency, and time gaps. """
        current_time = time.time()

        if self.first_juggle_time is None:
            self.first_juggle_time = current_time

        time_gap = 0
        if self.last_juggle_time is not None:
            time_gap = current_time - self.last_juggle_time

        self.last_juggle_time = current_time
        self.juggle_count += 1
        self.consistent_juggles += 1
        self.juggle_timestamps.append(current_time)

        # ✅ Store X and Y values for consistency calculation
        self.ball_x_positions.append(ball_x)
        self.ball_heights.append(ball_y)

        return self.juggle_count, time_gap

    def register_drop(self):
        """ ✅ New: Track when the ball drops """
        self.ball_drops += 1  # ✅ Increase drop count
        print(f"⚠️ Ball Dropped! Total Drops: {self.ball_drops}")

    def calculate_consistency(self, weight_abs=0.6, weight_rel=0.4, size_adjustment_k=5, epsilon=1e-3):
        """
        Evaluates consistency based on X-axis (horizontal) and Y-axis (vertical) positions.
        Ensures normalization and prevents excessive penalization.
        """
```

```
            if len(self.ball_x_positions) > 1 and len(self.ball_heights) > 1:
                x_positions = np.array(self.ball_x_positions)
                y_positions = np.array(self.ball_heights)

                median_x = np.median(x_positions)
                median_y = np.median(y_positions)

                mad_x = np.median(np.abs(x_positions - median_x)) + epsilon
                mad_y = np.median(np.abs(y_positions - median_y)) + epsilon

                range_x = np.max(x_positions) - np.min(x_positions) + epsilon
                range_y = np.max(y_positions) - np.min(y_positions) + epsilon

                # ✅ Normalize MAD values by range
                normalized_dev_x = mad_x / range_x
                normalized_dev_y = mad_y / range_y

                # ✅ Log transform to reduce extreme penalties
                total_deviation = (
                    weight_abs * np.log1p(mad_x**2 + mad_y**2) +
                                weight_rel   *   np.log1p(normalized_dev_x**2   +
normalized_dev_y**2)
                )

                  # ✅ Adjust scaling factor to prevent over-penalizing small
datasets
                N = len(x_positions)
                scaling_factor = (N / (N + size_adjustment_k)) ** 0.5

                # ✅ Normalize the consistency score to [0, 1]
                consistency_score = max(
                0, 1.0 - scaling_factor * total_deviation / 10)  # Scale deviation

                # ✅ Debugging Output
                print(
                    f"[DEBUG] Normalized MAD X: {normalized_dev_x:.4f},
Normalized MAD Y: {normalized_dev_y:.4f}")
                            print(f"[DEBUG]   Log   Scaled   Total   Deviation:
{total_deviation:.4f}")
                                print(f"[DEBUG]   Final   Consistency   Score:
{consistency_score:.4f}")

                return round(consistency_score, 2)
```

```
            return 1.0  # Default to full consistency before enough data

        def calculate_time_gap(self):
            """ Returns the average time gap between juggles. """
            if len(self.juggle_timestamps) > 1:
                # Calculate time differences
                gaps = np.diff(self.juggle_timestamps)
                return np.mean(gaps)  # Return average time gap
            return 0

        def get_total_time(self):
            """ Returns the total time taken from first to last juggle. """
            if self.first_juggle_time is None or self.last_juggle_time is None:
                return 0  # No valid juggles yet
            return self.last_juggle_time - self.first_juggle_time

        def reset_consistency(self):
            """ Resets the consistency streak when the ball drops. """
            self.consistent_juggles = 0

    def process_activation_line(data_queue, result_queue):
        """ Multiprocessing function to update activation lines. """
        while True:
            data = data_queue.get()
            if data is None:
                break
            bbox, keypoints, ball_y = data

            activation_line, JPart = update_activation_line(
                bbox, keypoints, ball_y)
            result_queue.put((activation_line, JPart))

    def start_multiprocessing():
        """Starts multiprocessing for activation line updates."""
        data_queue = mp.Queue()
        result_queue = mp.Queue()
        process = mp.Process(target=process_activation_line,
                             args=(data_queue, result_queue))
        process.start()
        return data_queue, result_queue, process

    def track_performance(prev_time, frames_since_last_check):
        pynvml.nvmlInit()  # Initialize NVML
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)  # GPU 0
```

```
            memory = pynvml.nvmlDeviceGetMemoryInfo(handle)
            utilization = pynvml.nvmlDeviceGetUtilizationRates(handle)
            """Track GPU, CPU usage, FPS, and latency."""
            current_time = time.perf_counter()
            latency = (current_time - prev_time) * 1000  # Convert to ms
            cpu_usage = psutil.cpu_percent()
            memory_usage = psutil.virtual_memory().percent
                  gpu_usage = torch.cuda.memory_allocated() / 1e9 if
torch.cuda.is_available() else 0
            fps = frames_since_last_check / \
               (current_time - prev_time + 1e-6)  # Fix FPS calculation

                print(f"CPU Usage: {cpu_usage:.2f}% | Memory Usage:
{memory_usage:.2f}% | "
                    f"Latency: {latency:.2f} ms | FPS: {fps:.2f} |" f"GPU Usage:
{utilization.gpu}% | Memory Used: {memory.used / (1024**2):.2f} MB")

            return current_time  # Return updated timestamp
```

guide.py

```
import cv2

def display_instruction(frame, player_bbox, activation_line,
juggle_count):
    """ Draws the activation line, bounding box fill, and instruction text """

    if player_bbox is not None and activation_line is not None:
       lower_bound_y = player_bbox[3]  # Player's lower bounding box
       knee_y = activation_line.start.y  # Activation line at knee level

       # Fill the area between activation line and lower bound with red
       overlay = frame.copy()
       cv2.rectangle(overlay, (int(player_bbox[0]), int(knee_y)),
                (int(player_bbox[2]), int(lower_bound_y)), (0, 0, 255), -1)
       alpha = 0.3  # Transparency level
       frame[:] = cv2.addWeighted(overlay, alpha, frame, 1 - alpha, 0)

       # Draw activation line
       cv2.line(frame, (int(activation_line.start.x), int(knee_y)),
             (int(activation_line.end.x), int(knee_y)), (255, 0, 0), 2)
```

```
overlay = frame.copy()
cv2.rectangle(overlay, (55, 10),(400, 80), (0, 0, 255), -1)
alpha = 0.3  # Transparency level
frame[:] = cv2.addWeighted(overlay, alpha, frame, 1 - alpha, 0)
# Instruction message
cv2.putText(frame, "Make sure the ball exceeds", (60, 30),
   cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 0), 2)

cv2.putText(frame, "the activation line at the knee", (60, 60),
   cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 0), 2)
```

data.py

```
import gspread
import csv
import numpy as np
import os
import time
from google.oauth2.service_account import Credentials
from utils_mp import JuggleTracker

# Path to JSON key file (Google Cloud Console)
SERVICE_ACCOUNT_FILE = "carbide-pilot-454220-r1-47df0b4141b1.json"

# Define the required API scopes
SCOPES = ["https://www.googleapis.com/auth/spreadsheets"]

# Authenticate and create a Google Sheets client
try:
    creds = Credentials.from_service_account_file(SERVICE_ACCOUNT_FILE, scopes=SCOPES)
    client = gspread.authorize(creds)
    SHEET_ID = "1hILHDJUhDSRIqoepY7j6T0e84fUXpY7zUtM6NuC44dM"
    sheet = client.open_by_key(SHEET_ID).sheet1  # Access the first sheet
    print("✅ Google Sheets connection established.")
except Exception as e:
    print(f"❌ Failed to connect to Google Sheets: {e}")

class JugglingSession:
    def __init__(self, csv_filename="juggling_data.csv"):
        self.total_juggles = 0
        self.juggle_timestamps = []
        self.ball_heights = []
        self.first_juggle_time = None
        self.last_juggle_time = None
        self.csv_filename = os.path.abspath(csv_filename)  # Use absolute path
        self.ensure_csv_exists()
        self.improvement_score = 0
        self.endurance_score = 0

        # Initialize a JuggleTracker instance
        self.juggle_tracker = JuggleTracker()
```

```
        def ensure_csv_exists(self):
            """ Creates the CSV file if it does not exist to prevent errors. """
            if not os.path.isfile(self.csv_filename):
                with open(self.csv_filename, mode="w", newline="") as file:
                    writer = csv.writer(file)
                    writer.writerow(["Total Juggles", "Avg Time Gap (s)", "Consistency", "Total Time (s)", "Rating"])
                print(f"📂 Created new CSV file: {self.csv_filename}")

        def record_juggle(self, ball_height, ball_x=None):
            """ Records a new juggle event with timestamp, ball height, and optional x position.
            Also detects drops based on time gap between juggles.
            """
            current_time = time.time()

            if self.first_juggle_time is None:
                self.first_juggle_time = current_time  # ✅ Set first juggle time

            # ✅ Detect drop if time gap is too large
            if self.last_juggle_time is not None:
                time_gap = current_time - self.last_juggle_time
                if time_gap > 3:  # 🔥 Customize threshold (e.g., 3 seconds)
                    self.juggle_tracker.ball_drops += 1
                    print(f"⚠️ Ball drop detected! Total Drops: {self.juggle_tracker.ball_drops}")

            self.total_juggles += 1
            self.juggle_timestamps.append(current_time)
            self.ball_heights.append(ball_height)

            # ✅ Ensure JuggleTracker receives updated values
            self.juggle_tracker.juggle_timestamps.append(current_time)
            self.juggle_tracker.ball_heights.append(ball_height)
            self.juggle_tracker.juggle_count = self.total_juggles  # ✅ Sync count
            self.latest_consistency = 1.

            if ball_x is not None:
                self.juggle_tracker.ball_x_positions.append(ball_x)  # ✅ Store ball_x

            self.last_juggle_time = current_time  # ✅ Update last juggle time
```

```
            self.juggle_tracker.last_juggle_time = current_time  # ✅ Sync with JuggleTracker

        def calculate_time_gap(self):
            """ Calculates the average time gap between juggles. """
            if len(self.juggle_tracker.juggle_timestamps) > 1:
                return self.juggle_tracker.calculate_time_gap()

            print("⚠️ Not enough timestamps to calculate time gap.")
            return 0

        def get_total_time(self):
            """ Returns the total duration from the first to last juggle. """
            if self.first_juggle_time is None or self.last_juggle_time is None:
                print("⚠️ First or last juggle time missing. Returning 0.")
                return 0

            return round(self.last_juggle_time - self.first_juggle_time, 2)  # ✅ Ensure rounded output

        def get_consistency(self):
            """ Returns the latest consistency value stored in the class. """
              print(f"[DEBUG] Consistency: {self.latest_consistency}")  # ✅ Corrected print
            return self.latest_consistency  # ✅ Fetch from class attribute

        def calculate_juggling_rating(self):
            """ Computes juggling rating based on the new scoring system. """
            # 1. Juggling Count (JC)
             juggling_count_score = min(self.total_juggles // 10, 20)  # 1 point per 10 juggles, capped at 20

            # 2. Total Time Taken (TTT)
            total_time = self.get_total_time()
            benchmark_time = 120  # Benchmark for 120 seconds
            if total_time == 0:  # Explicitly handle missing data
                time_score = 0
            elif total_time <= benchmark_time:
                 time_score = 10 + ((benchmark_time - total_time) // 5)  # Add 1 point per 5 seconds under benchmark
            else:
                time_score = 10 - ((total_time - benchmark_time) // 5)  # Subtract 1 point per 5 seconds over benchmark
            time_score = max(0, min(20, time_score))  # Cap between 0 and 20
```

```
                avg_time_gap = self.calculate_time_gap()
                if avg_time_gap == 0:  # Explicitly handle missing data
                    time_gap_score = 0
                elif avg_time_gap <= 0.5:
                    time_gap_score = 10
                elif avg_time_gap <= 1.0:
                    time_gap_score = 5
                elif avg_time_gap <= 2.0:
                    time_gap_score = 2
                else:
                    time_gap_score = 0

                # 4. Consistency (C)
                    consistency_score_raw = self.get_consistency()  # Get raw consistency score (0 to 1
                if self.total_juggles == 0:  # Check if juggle count is zero
                    print("[DEBUG] Juggle count is 0. Setting consistency score to 0.")
                    consistency_score = 0
                if consistency_score_raw >= 0.9:
                    consistency_score = 10
                elif consistency_score_raw >= 0.7:
                    consistency_score = 7
                elif consistency_score_raw >= 0.5:
                    consistency_score = 5
                elif consistency_score_raw >= 0.3:
                    consistency_score = 3
                else:
                    consistency_score = 0  # No points for consistency < 0.3

                # 5. Endurance Point (EP)
                self.endurance_score = 0
                if total_time > 120:
                    self.endurance_score = min((total_time - 120) // 10, 10)  # 1 point per 10 seconds beyond 120s, capped at 10

                # 6. Improvement Point (IP)
                self.improvement_score = 0
                if len(self.ball_heights) >= 10:
                    first_avg = np.mean(self.ball_heights[:5])
                    last_avg = np.mean(self.ball_heights[-5:])
                    print(f"[DEBUG] BALL HEIGHTS: {self.ball_heights}")
                    improvement_percentage = ((last_avg - first_avg) / first_avg) * 100 if first_avg != 0 else 0
```

```
                # Ensure improvement score is non-negative
                if improvement_percentage > 0:
                    self.improvement_score = min(improvement_percentage // 5,
10)  # 1 point per 5% improvement, capped at 10
                else:
                    self.improvement_score = 0  # No improvement or negative
improvement
                                print(f"[DEBUG] Improvement Score:
{self.improvement_score}")

            # Total Score Calculation
            total_score = (
                juggling_count_score +
                time_score +
                time_gap_score +
                consistency_score +
                self.endurance_score +
                self.improvement_score
            )

            # Debugging Logs
            print(f"[DEBUG] Juggling Count Score: {juggling_count_score}")
            print(f"[DEBUG] Time Score: {time_score}")
            print(f"[DEBUG] Time Gap Score: {time_gap_score}")
                        print(f"[DEBUG] Consistency Score (Raw):
{consistency_score_raw}")
                    print(f"[DEBUG] Consistency Score (Mapped):
{consistency_score}")
            print(f"[DEBUG] Endurance Score: {self.endurance_score}")
            print(f"[DEBUG] Improvement Score: {self.improvement_score}")
            print(f"[DEBUG] Total Score: {total_score}")

            return total_score

        def save_to_csv(self, avg_time_gap, consistency, total_time):
            """ Saves session data to a CSV file. """
            file_exists = os.path.isfile(self.csv_filename)
              headers = ["Total Juggles", "Avg Time Gap (s)", "Consistency",
"Total Time (s)", "Endurance", "Improvement", "Rating"]

            try:
                with open(self.csv_filename, mode="a", newline="") as file:
                    writer = csv.writer(file)

                    if not file_exists:
```

```
                    writer.writerow(headers)  # Write header if new file

                writer.writerow([
                    self.total_juggles,
                    avg_time_gap,
                    consistency,
                    total_time,
                    self.endurance_score,  # Use class attribute
                    self.improvement_score,  # Use class attribute
                    self.calculate_juggling_rating()
                ])

                print(f"✅ Session saved to {self.csv_filename}")

            except Exception as e:
                print(f"❌ Error saving to CSV: {e}")


        '''def save_to_csv(self, juggle_tracker, filename="juggling_data.csv"):
            """ Saves session data to a CSV file after recalculating metrics to ensure accurate values. """
            file_exists = os.path.isfile(filename)
            headers = ["Total Juggles", "Avg Time Gap (s)", "Consistency", "Total Time (s)", "Rating"]

            # ✅ Fetch latest calculated values from JuggleTracker
            avg_time_gap = self.juggle_tracker.calculate_time_gap()
            consistency = self.juggle_tracker.calculate_consistency() # 🔥 Now using juggle_tracker directly
            total_time = self.juggle_tracker.get_total_time()
            rating = self.calculate_juggling_rating()

            # ✅ Debug print before writing to CSV
                    print(f"[DEBUG] Saving to CSV - Juggles: {juggle_tracker.juggle_count}, Time Gap: {avg_time_gap}, Consistency: {consistency}, Total Time: {total_time}, Rating: {rating}")

            try:
                with open(filename, mode="a", newline="") as file:
                    writer = csv.writer(file)

                    if not file_exists:
                        writer.writerow(headers)  # Write header if new file
```

```
                    writer.writerow([
                        juggle_tracker.juggle_count,
                        avg_time_gap,
                        consistency,
                        total_time,
                        rating
                    ])

                print(f"✅ Session saved to {filename} with updated metrics.")

            except Exception as e:
                print(f"❌ Error saving to CSV: {e}")'"

        def reset_session(self):
            """ Resets all juggling data for a new session. """
            self.total_juggles = 0
            self.juggle_timestamps.clear()
            self.ball_heights.clear()
            self.juggle_tracker.ball_x_positions.clear()
            self.juggle_tracker.ball_heights.clear()
            self.first_juggle_time = None
            self.last_juggle_time = None
            print("🔄 Session reset.")

        def save_ball_positions(self, ball_x, ball_y):
            """ Saves the ball's (x, y) position after each successful juggle to a
local CSV file. """
            try:
                with open("ball_positions.csv", mode="a", newline="") as file:
                    writer = csv.writer(file)
                    writer.writerow([self.total_juggles, ball_x, ball_y])
                print(f"✅ Ball position saved: Juggle {self.total_juggles}, X:
{ball_x}, Y: {ball_y}")
            except PermissionError:
                print(f"❌ Permission denied: Unable to write to
'ball_positions.csv'. Ensure the file is not open in another program.")
            except Exception as e:
                print(f"❌ Error saving ball position: {e}")

        def update_google_sheet():
            """ Uploads the latest data from the CSV file to Google Sheets. """
            try:
                with open("juggling_data.csv", mode="r", newline="") as file:
                    reader = csv.reader(file)
```

```
                rows = list(reader)

                if len(rows) < 2:
                    print("⚠️ No valid data to upload.")
                    return

                headers = rows[0]  # First row contains column names
                 latest_entry = rows[-1]  # Last row contains the latest recorded
session

                # Check existing sheet data
                existing_data = sheet.get_all_values()

                # Upload headers if the sheet is empty
                if len(existing_data) == 0:
                    sheet.append_row(headers)

                # Append the latest data
                sheet.append_row(latest_entry)
                print("✅ Google Sheet updated successfully.")

            except Exception as e:
                print(f"❌ Error updating Google Sheet: {e}")
```

dashboard.py

```
import streamlit as st
import pandas as pd
import gspread
from oauth2client.service_account import ServiceAccountCredentials
import plotly.graph_objects as go
import plotly.express as px

# ✅ Step 1: Authenticate Google Sheets
scope = ["https://spreadsheets.google.com/feeds", "https://www.googleapis.com/auth/drive"]
creds = ServiceAccountCredentials.from_json_keyfile_name("carbide-pilot-454220-r1-47df0b4141b1.json", scope)
client = gspread.authorize(creds)

sheet = client.open("FYP-5GSOCCER").sheet1  # Replace with your actual sheet name

def load_data():
    data = sheet.get_all_records()
    return pd.DataFrame(data)

# Load the data
df = load_data()

# Convert relevant columns to numeric
df["Improvement"] = pd.to_numeric(df["Improvement"], errors="coerce").fillna(0).astype(int)
df["Rating"] = pd.to_numeric(df["Rating"], errors="coerce").fillna(0).astype(int)
df["Consistency"] = pd.to_numeric(df["Consistency"], errors="coerce").fillna(0)

# 🎯 **Header with Image**
st.image("1673861856868.jpeg", use_column_width=True)
st.markdown("<h1 class='header'>⚽ Juggling Performance Dashboard</h1>", unsafe_allow_html=True)

# 📌 **Grading Scale Section**
st.subheader("Grading Scale")
st.markdown("""
- **0-20**: Beginner
- **21-40**: Intermediate
- **41-60**: Advanced
```

```
- **61+**: Elite
""")

# 📊 **Grading Scale Visualization**
grading_scale = {
    "Grade": ["Beginner", "Intermediate", "Advanced", "Elite"],
    "Min Score": [0, 21, 41, 61],
    "Max Score": [20, 40, 60, 100]
}
grading_df = pd.DataFrame(grading_scale)

latest_rating = df.iloc[-1]["Rating"] if not df.empty else 0

fig_grading = go.Figure()
for i, row in grading_df.iterrows():
    fig_grading.add_trace(go.Bar(
        x=[row["Max Score"] - row["Min Score"]],
        y=["Grading Scale"],
        orientation="h",
        name=row["Grade"],
        marker_color=px.colors.qualitative.Plotly[i],
        text=[f"{row['Min Score']}-{row['Max Score']}"],
        textposition="inside",
        hoverinfo="none"
    ))

fig_grading.add_trace(go.Scatter(
    x=[latest_rating, latest_rating],
    y=[-0.5, 1.5],
    mode="lines",
    line=dict(color="red", width=3, dash="dash"),
    name=f"Latest Rating: {latest_rating}"
))

fig_grading.update_layout(
    title="Grading Scale with Latest Rating",
    xaxis_title="Score Range",
    yaxis_title="",
    barmode="stack",
    showlegend=False,
    height=300,
    xaxis_range=[0, 100],
    yaxis_visible=False,
    annotations=[
        dict(
```

```
                x=latest_rating,
                y=1.5,
                xref="x",
                yref="y",
                text=f"Latest Rating: {latest_rating}",
                showarrow=True,
                arrowhead=2,
                ax=0,
                ay=-40,
                font=dict(size=12, color="red")
            )
        ]
    )
    st.plotly_chart(fig_grading)

    # 📊 **Performance Metrics Visualization**
    if not df.empty:
        latest_session = df.iloc[-1]

        st.subheader("Latest Performance Metrics")

        # 🔵 **Total Juggles**
        fig_juggles = go.Figure(go.Bar(
            x=["Total Juggles"], y=[latest_session["Total Juggles"]],
            text=[latest_session["Total Juggles"]], textposition="auto",
            marker=dict(color="blue")
        ))
            fig_juggles.update_layout(yaxis=dict(title="Juggles",    range=[0,
max(df["Total Juggles"]) + 10]))
        st.plotly_chart(fig_juggles)

        # 🟠 **Consistency (Max: 1.0)**
        fig_consistency = go.Figure(go.Bar(
            x=["Consistency"], y=[latest_session["Consistency"]],
            text=[round(latest_session["Consistency"], 2)], textposition="auto",
            marker=dict(color="orange")
        ))
                fig_consistency.update_layout(yaxis=dict(title="Consistency",
range=[0, 1.0]))
        st.plotly_chart(fig_consistency)

        # 🟢 **Endurance**
        fig_endurance = go.Figure(go.Bar(
            x=["Endurance"], y=[latest_session["Endurance"]],
```

```
            text=[latest_session["Endurance"]], textposition="auto",
            marker=dict(color="green")
        ))
        fig_endurance.update_layout(yaxis=dict(title="Endurance", range=[0,
max(df["Endurance"]) + 5]))
        st.plotly_chart(fig_endurance)

        # 🔴 **Improvement**
        fig_improvement = go.Figure(go.Bar(
            x=["Improvement"], y=[latest_session["Improvement"]],
            text=[latest_session["Improvement"]], textposition="auto",
            marker=dict(color="red")
        ))
            fig_improvement.update_layout(yaxis=dict(title="Improvement",
range=[0, max(df["Improvement"]) + 5]))
        st.plotly_chart(fig_improvement)

        # 🟣 **Rating**
        fig_rating = go.Figure(go.Bar(
            x=["Rating"], y=[latest_session["Rating"]],
            text=[latest_session["Rating"]], textposition="auto",
            marker=dict(color="purple")
        ))
        fig_rating.update_layout(yaxis=dict(title="Rating", range=[0, 10]))
        st.plotly_chart(fig_rating)

        # 📌 **Player Improvement Feedback**
        st.subheader("📢 Player Improvement Feedback")

        feedback = ""
        if latest_session["Consistency"] < 0.6:
            feedback += "- Improve ball control by focusing on consistent
touches.\n"
        if latest_session["Avg Time Gap (s)"] > 1.0:
            feedback += "- Try to maintain a steady rhythm with faster
reactions.\n"
        if latest_session["Total Juggles"] < 20:
            feedback += "- Increase endurance by practicing longer juggling
sessions.\n"
        if latest_session["Improvement"] == 0:
            feedback += "- Set personal goals for improvement and track
progress.\n"
```

```
            st.write(feedback if feedback else "Great job! Keep practicing to refine
your skills.")

        else:
            st.write("No session data available.")
```