h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

A
Project Report on

"Multi-Camera Object Detection on Nvidia Jetson Orin Nano"

Submitted in fulfillment of the requirements for award of the degree of ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY- INTERNATIONAL
For the academic year
2023-2024

Team Members	Matriculation Number
Shriya Chavan	1119292
Pooja Chavan	1119293
Supriya Chilukuri	1119235

Submitted in fulfillment of the requirements for award of the degree of ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY- INTERNATIONAL
For the academic year
2023-24

Under the guidance of
Dr. Prof. Schumann, Thomas
Faculty of Electrical Engineering and Information Technology

DEPARTMENT OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Abstract

This document outlines the implementation of an object detection system utilizing two ZED cameras on the Nvidia Jetson Orin Nano platform. The primary objective is to deploy the YOLO algorithm for object detection and subsequently compare the performance of various YOLO models. The focus is on providing comprehensive guidance for integrating this functionality into existing projects, aiming to empower readers to incorporate similar capabilities into their own projects. By leveraging this groundwork, future endeavors can explore the development of more sophisticated and efficient object detection algorithms tailored to specific applications.

Table of Contents

Abstract 2

List of abbreviations 5

1 Introduction 6

1.1 Goal of this project 6

1.2 Used Hardware 6

1.2.1 Jetson Orin Nano 8GB Module 6

1.2.2 ZED Camera 8

1.3 Introduction to Neural Network 8

1.3.1 Neural Network 8

1.3.2 Convolutional Neural Network 9

1.4 Object Detection 10

1.4.1 Object Detection 10

1.4.2 You Only Look Once (YOLO) 10

1.5 Operating Principle of ZED Camera 11

1.6 Distance estimation 12

1.6.1 Depth Sensing 12

1.6.2 How is depth map generated? 12

1.7 System and Software Specification 13

1.7.1 System Specification 13

1.7.2 Software Specification 14

2 Implementation 16

2.1 Implementation at glance 16

2.2 Implementation in Detail 16

2.3 Process Flow Diagram 21

3 Performance Comparison 22

3.1 GPU & CPU Performance 22

3.2 Frames per Second 24

4 Application 27

4.1 Retail Stores 27

4.2 Transportation 27

4.3 Healthcare 27

4.4 Agriculture 28

4.5 Entertainment 28

5 Future Scope 29

6 Conclusion 29

References 30

List of figures

Figure 1: Performance and efficiency comparison between Orin Nano & Nano.....4
Figure 2: Artificial Neural Network.....6
Figure 3: Convolutional Neural Network.....7
Figure 4: Operating principle of ZED Camera.....9
Figure 5: Camera triangulation for depth sensing.....13
Figure 6: Code snippet for importing libraries.....17
Figure 7: Code snippet of detections to custom box object data.....17
Figure 8: Code snippet of processing detections and calculating inference.....18
Figure 9: Code snippet of threaded model inference initialization.....18
Figure 10: Code snippet of acquiring the frame.....18
Figure 11: Code snippet of ingest detection from captured image.....19
Figure 12: Code snippet of depth measurement.....19
Figure 13: Code snippet of displaying the measured distance.....19
Figure 14: Code snippet of closing camera.....20
Figure 15: Result image.....20
Figure 16: Process Flow Diagram.....21
Figure 17: GPU Graph for YOLOv8n model.....22
Figure 18: GPU Graph for YOLOv8l model.....22
Figure 19: CPU Graph for YOLOv8l model.....23
Figure 20: CPU Graph for YOLOv8l model.....24
Figure 21: Terminal output when using two different models.....25

List of Tables

Table 1: Software Specifications.....14

List of abbreviations

YOLO Model	You Only Look Once Model
AI	Artificial Intelligence
FPS	Frames Per Second
GPU	Graphics Processing Unit
CPU	Central Processing Unit
ANN	Artificial Neural Network
SSN	Simulated Neural Networks
SLAM	Simultaneous localization and mapping
SD	Storage Device
RAM	Random Access Memory
MP	Megapixel
FPS	Frames per Second
CNN	Convolutional Neural Networks
DNN	Deep Neural Network
OpenCV	Open-Source Computer Vision
ISP	Image Signal Processor

1 Introduction

Artificial Intelligence (AI) operations are gaining in interest in the last couple of years. Since hardware is rapidly becoming cheaper and more accessible and also software is getting more advanced, AI operations can nowadays be found everywhere. Starting with Virtual Assistant or Chatbots, Security and Surveillance all the way to self-driving cars or autonomous vehicles. According to Statista1 the revenue from the artificial intelligence software market is increasing exponentially. In 2018, the market revenue was about 10 billion USD. In 2021, the revenue increased to approximately 34 billion USD. It is estimated that by the year 2025 the revenue of this market will reach an astonishing revenue of about 126 billion USD. This shows the interest and need in this technology.

1.1 Goal of this project

In this project, AI will also be used to implement different functionalities. The primary objective of this collaborative project is to develop a real-time object detection system using the YOLOv8 algorithm, integrated with two ZED2i cameras on the Jetson Orin Nano platform, powered by artificial intelligence (AI). YOLOv8 is celebrated for its exceptional computational speed and accuracy, making it an ideal choice for real-time object detection tasks employing neural networks. This project focuses on comparing the performance of different YOLO models by analyzing the frames per second (FPS) and GPU performance. By conducting comprehensive performance evaluations, the team aims to assess the efficiency and effectiveness of various YOLO models, enabling informed decision-making and optimization of the object detection system. To evaluate the performance of the features, a benchmark test will be executed. This can then be used to analyze the performance and to further increase it later on.

1.2 Used Hardware

1.2.1 *Jetson Orin Nano 8GB Module*

The developer kit consists of a Jetson Orin Nano 8 GB module and a reference carrier board that can accommodate all NVIDIA Jetson Orin Nano and NVIDIA Jetson Orin NX modules, providing an ideal platform for prototyping next-generation edge AI products.

The Jetson Orin Nano 8 GB module features an NVIDIA Ampere architecture GPU with 1024 CUDA cores, 32 third-generation Tensor Cores, and a 6-core Arm CPU, enabling multiple concurrent AI application pipelines and high-performance inference. The developer kit carrier board boasts a wide array of connectors, including two MIPI CSI connectors supporting camera modules with up to four lanes, enabling higher resolution and frame rates than before.

The prior-generation Jetson Nano Developer Kit made AI accessible to everyone. The new Jetson Orin Nano Developer Kit raises the bar for entry-level AI development with 80x the performance, enabling developers to run any kind of modern AI models, including transformer and advanced robotics models. Not only does it provide a huge boost in AI performance over the prior-generation Jetson Nano, Jetson Orin Nano also provides 5.4x the CUDA compute, 6.6x the CPU performance, and 50x the performance per watt.

- GPU: NVIDIA Ampere architecture with 1024 NVIDIA® CUDA® cores and 32 Tensor cores
- CPU: 6-core Arm Cortex-A78AE v8.2 64-bit CPU 1.5MB L2 + 4MB L3
- Memory: 8GB 128-bit LPDDR5 68 GB/s

Fig 1 : Performance and efficiency comparison between NVIDIA Jetson Orin Nano and NVIDIA Jetson Nano

1.2.2 *ZED Camera*

We are using two ZED 2i camera for object detection. The ZED can be used both indoors and outdoors at long range (up to 20m). The ZED Mini can also be used indoors and outdoors up to 15m. The configuration that offers the best depth accuracy is to place the camera at a working distance of 1 meter of the scene for the ZED. To capture fast movements, use the camera high-speed modes (1080p @30fps or VGA @ 100 FPS).

- Neural Depth Sensing
- Spatial Object Detection
- Built-in Next-Gen IMU, Gyroscope, Barometer & Magnetometer

1.3 Introduction to Neural Network

1.3.1 *Neural Network*

A neural network is a machine learning program, or model, that makes decisions in a manner similar to the human brain, by using processes that mimic the way biological neurons work together to identify phenomena, weigh options and arrive at conclusions. Neural networks are sometimes called artificial neural networks (ANNs) or simulated neural networks (SNNs). They are a subset of machine learning, and at the heart of deep learning models.

A neural network is a machine learning program, or model, that makes decisions in a manner similar to the human brain, by using processes that mimic the way biological neurons work together to identify phenomena, weigh options and arrive at conclusions. Neural networks are sometimes called artificial neural networks (ANNs) or simulated neural networks (SNNs). They are a subset of machine learning, and at the heart of deep learning models. An artificial neural network consists of innumerable neurons assigned to different node layers within this network. These layers are divided into three categories: an input layer, several so-called hidden layers and an output layer. Each node connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network. A simple artificial neural network is shown in Figure below.

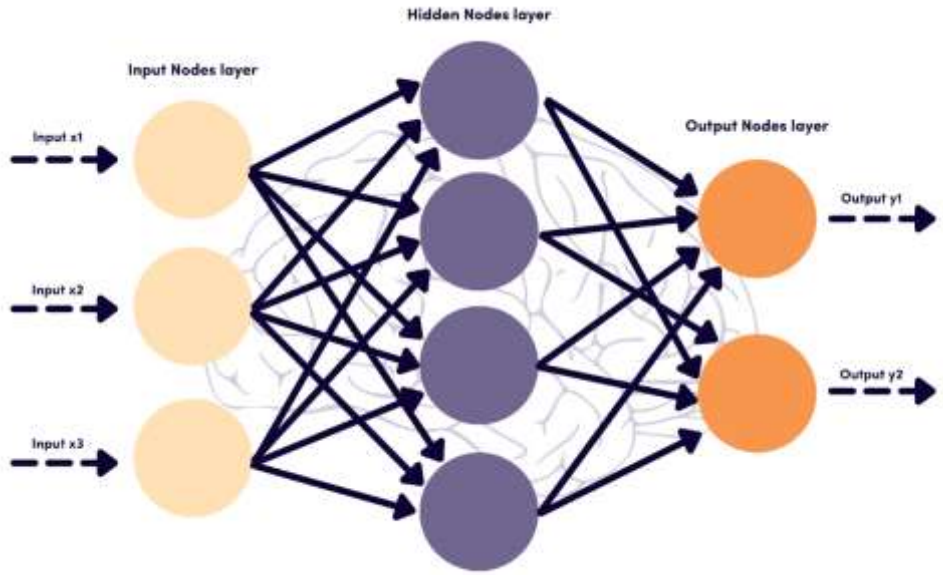


Fig 2: Artificial Neural Network

1.3.2 *Convolutional Neural Network*

Convolution Neural Networks are mainly credited for their role in image and video recognition, recommendation systems, and image analysis and classification. Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

- Convolutional layer
- Pooling layer
- Fully-connected (FC) layer

The convolutional layer is the first layer of a convolutional network. While convolutional layers can be followed by additional convolutional layers or pooling layers, the fully-connected layer is the final layer. With each layer, the CNN increases in its complexity, identifying greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object. With each layer, the CNN increases in its complexity, identifying greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object. R-CNN, Fast R-CNN, Faster R-CNN and YOLO models use Convolutional

Neural Network as their backbone architecture.

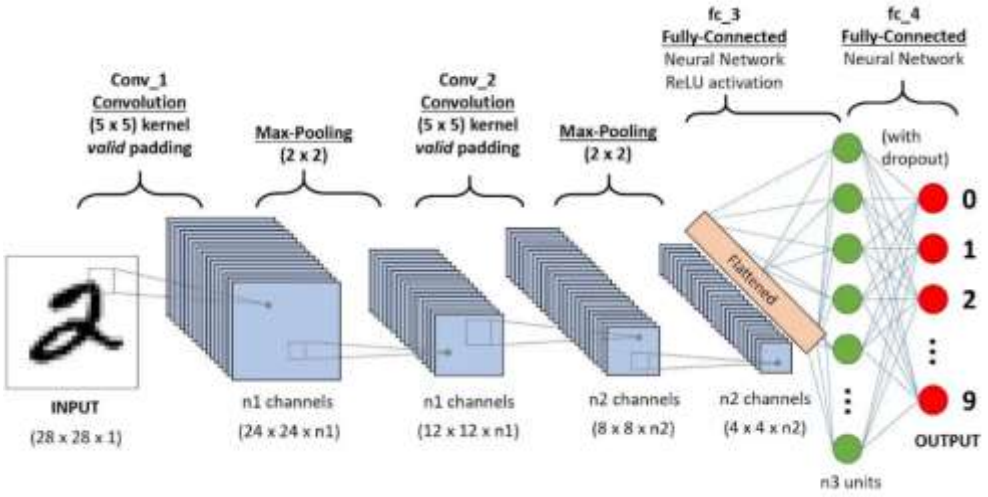


Fig 3: Convolutional Neural Network

1.4 Object Detection

1.4.1 *Object Detection*

Object detection is a computer vision task that involves identifying and locating objects in images or videos. It is an important part of many applications, such as surveillance, self-driving cars, or robotics. Object detection algorithms can be divided into two main categories: single-shot detectors and two-stage detectors. Object detection algorithms are broadly classified into two categories based on how many times the same input image is passed through a network, namely, two stage or one stage.

1.4.2 *You Only Look Once (YOLO)*

You Only Look Once (YOLO) proposes using an end-to-end neural network that makes predictions of bounding boxes and class probabilities all at once. It differs from the approach taken by previous object detection algorithms, which repurposed classifiers to perform detection. YOLO itself is a Convolutional Neural Network (CNN), a type of neural network, which is very good at detecting patterns (and by extension objects and the like) in images.

While algorithms like Faster RCNN work by detecting possible regions of interest using the Region Proposal Network and then performing recognition on those regions separately, YOLO performs all of its predictions with the help of a single fully connected layer.

YOLO has been developed in several versions, such as YOLOv1, YOLOv2, YOLOv3, YOLOv4, YOLOv5, YOLOv6, and YOLOv7, YOLOv8. Each version has been built on top of the previous version with enhanced features such as improved accuracy, faster processing, and better handling of small objects.

The version of YOLO that we are using in this project is YOLOv8.

YOLOv8 is the latest family of YOLO based Object Detection models from Ultralytics providing state-of-the-art performance.

Leveraging the previous YOLO versions, the YOLOv8 model is faster and more accurate while providing a unified framework for training models for performing

- Object Detection,
- Instance Segmentation, and
- Image Classification.

There are five models in each category of YOLOv8 models for detection, segmentation, and classification namely YOLOv8n, YOLOv8s, YOLOv8m, YOLOv8l, YOLOv8x. YOLOv8 Nano is the fastest and smallest, while YOLOv8 Extra Large (YOLOv8x) is the most accurate yet the slowest among them.

1.5 Operating Principle of ZED Camera

The ZED is a passive stereovision-based camera that reproduces the way human vision works. Using its two “eyes” and through triangulation, the ZED understands its surroundings and creates a three-dimensional model of the scene it observes. The Stereo labs ZED 2i is a stereo 3D camera designed for spatial intelligence and augmented reality applications. Its maximum depth perception goes up to 20 meters making it suitable for both indoor and outdoor scenarios. The camera includes different sensors like IMU, barometer, and magnetometer, that support motion tracking. The camera also integrates visual-inertial odometry (VIO) and SLAM technology to enable precise spatial awareness. Its characteristics make it a suitable option for robotics, drones, and AR projects. Additionally, if combined with the provided software development kit (SDK), it allows developers to harness its capabilities for custom projects, making it ideal for 3D mapping, computer vision, and environmental perception applications.

The ZED uses triangulation to estimate depth from the disparity image, with the following formula describing how depth resolution changes over the range of the camera:

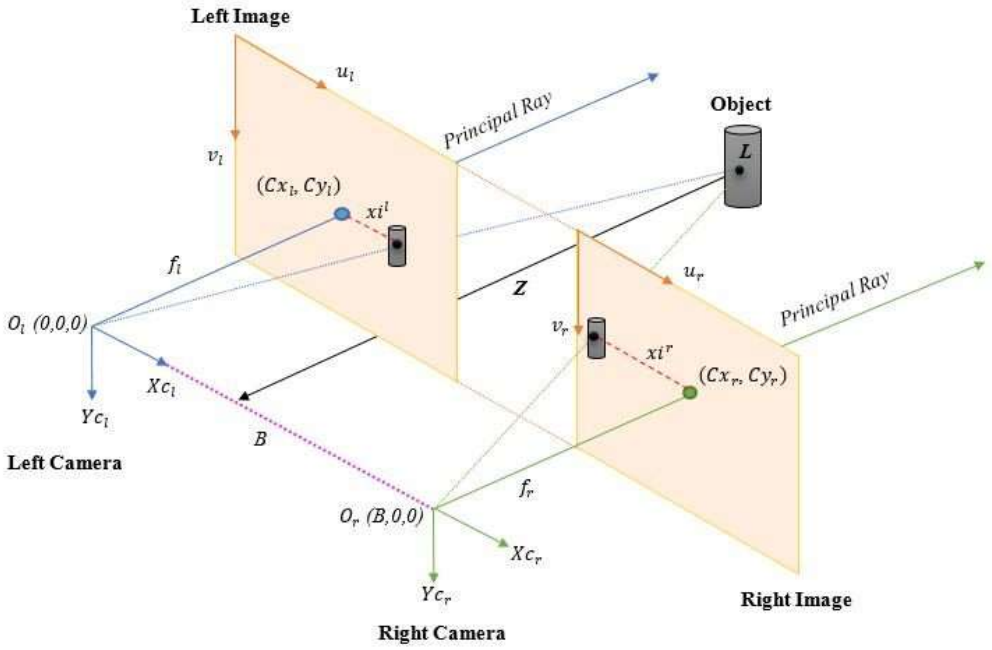


Fig 4: Operating principle of ZED Camera

`Dr=Z^2*alpha`, where `Z` is the distance and `alpha` a constant.

1.6 Distance estimation

1.6.1 *Depth Sensing*

Depth perception is the ability to determine distances between objects and see the world in three dimensions. Up until now, depth sensors have been limited to perceiving depth at short range and indoors, restricting their application to gesture control and body tracking. Using stereo vision, the ZED is the first universal depth sensor:

- Depth can be captured at longer ranges, up to 20m.
- Frame rate of depth capture can be as high as 100 FPS.
- Field of view is much larger, up to 110° (H) x 70° (V).
- The camera works indoors and outdoors, contrary to active sensors such as structured-light or time of flight.

1.6.2 *How is depth map generated?*

Depth maps captured by the ZED store a distance value (Z) for each pixel (X, Y) in the image. The distance is expressed in metric units (meters for example) and calculated from the back of the left eye of the camera to the scene object.

Camera triangulation is a powerful technique for determining the distance and depth of objects in a scene. By using multiple cameras and triangulating the position of the same feature in each image, it is possible to accurately determine the 3D position of the feature

and therefore its distance from the cameras.

To perform distance and depth analysis using camera triangulation, the first step is to calibrate the cameras. Once the cameras are calibrated, the next step is to capture multiple images of the scene from different positions. The next step is to extract and match features in the images. Features can be any distinctive points or patterns that can be easily identified in multiple images, such as corners or edges.

Once the features are matched, the triangulation process can begin. The position of a feature in 3D space can be computed by intersecting the lines of sight from each camera that pass through the feature.

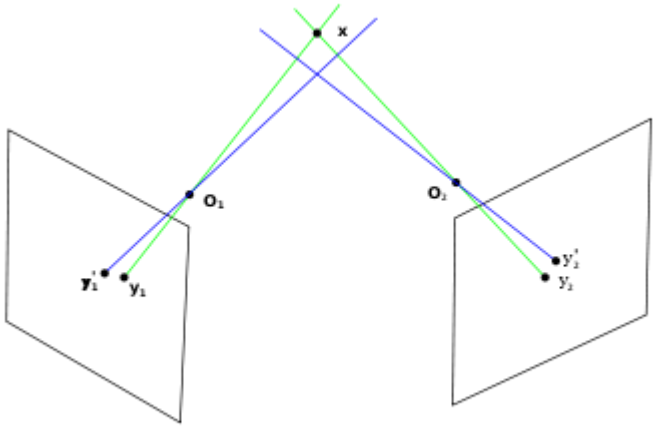


Fig 5 : Camera triangulation for depth sensing

A depth map is a representation of the scene where each pixel in an image corresponds to a depth value, indicating the distance from the camera to the corresponding point in the scene.

Depth maps are generated using depth-sensing technologies and provide a grayscale or color-coded visualization of the scene's depth information.

1.7 System and Software Specification

1.7.1 *System Specification*

- Nvidia Jetson TX2: 8GB Module
- Object detection shall be based on Yolo v8 model.
- Distance estimation shall be based on depth map of ZED camera.
- Two ZED Camera: 2i version

1.7.2 *Software Specification*

Software	Versions
Operating System	Ubuntu 20.04
Programming language	Python 3.8
NVIDIA SDK	Jetpack 5.1.2
ZED SDK	4.0.
CUDA	11.4.19
YOLO (Neural Network)	Version 8
PyTorch	2.1.2

Table 1: Software Specifications

1.7.2.1 Jetpack

NVIDIA Jetpack is a Software Development Kit (SDK) provided by NVIDIA as a complete solution suite to get software developers started with NVIDIA Jetson development kits, the Jetson Orin Nano included.

We followed the installation of Jetpack using SD Card image method which is explained on the NVIDIA website. Jetson devices can be booted from a mass storage class USB device with bulk-only protocol, such as a flash drive. Hot plugging is not supported; the flash drive must be attached before the Jetson device is booted. You can manually set up a flash drive for booting. All Jetson devices can boot from internal storage using a boot partition and can mount an external USB drive as the root file system.

Jetpack 5.1.2 is installed with the required software components as mentioned in the software specifications.

1.7.2.2 CUDA

CUDA is a parallel processing and application programming interface platform developed by NVIDIA. Like Direct3D or OpenGL, CUDA allows software developers to tap into the hardware-accelerated computing power of GPUs, given that the target GPU is CUDA enabled. Instead of exclusive graphics programming, CUDA is designed to work with general purpose programming in languages like C, C++ and Fortran by providing access to the GPU parallel instruction sets, thus increases its accessibility and usability.

1.7.2.3 Open CV

OpenCV (Open-Source Computer Vision) is a software library aimed at real-time computer vision. This cross-platform library is now one of the most widely used for image processing. Since OpenCV supports Python interface, it could be used to provide easy access to video capturing devices (ZED Camera, onboard camera and ordinary USB webcams) along with a collection of useful image pre-processing functions.

1.7.2.4 ZED SDK

The ZED device is composed of stereo 2K cameras with dual 4MP RGB sensors. It is an UVC-compliant USB 3.0 camera, backward compatible with USB 2.0. Left and right video frames are synchronized and streamed as a single uncompressed video frame in the side-by-side format. Several configuration parameters of on-board ISP (Image Signal Processor) as resolution, brightness, contrast, saturation can be adjusted through the SDK that is provided by ZED development team. This camera has a compact structure and reduced size, compared to other stereo cameras such as the Bumblebee XB3 for example. These characteristics make it relatively simple to incorporate into robotic systems or drones. In this project, we have used ZED SDK 2.8.5 for distance estimation.

2 Implementation

2.1 Implementation at glance

1. The script starts with importing necessary libraries
2. The ‘`xywh2abcd`’ function converts bounding box coordinates from ‘`xywh`’ format to ‘`abcd`’ format.
3. The ‘`detections_to_custom_box`’ function converts YOLO detections to custom box object data for the ZED SDK.
4. The main function orchestrates the execution of the entire script. It starts threads for YOLO model inference for each camera, initializes ZED cameras, and sets up parameters for object detection and tracking.
5. The script initializes ZED cameras with specified parameters using the ZED SDK.
6. Object Detection and Rendering.
7. The script counts the number of detections for each camera and displays this count on the frame.
8. For each detected object, the script calculates the distance from the camera using depth information obtained from the depth map.
9. User Interaction and Exiting

2.2 Implementation in Detail

2.2.1 *Importing Libraries*

In this project, NumPy is likely used for efficient handling and manipulation of image data and numerical computations related to object detection. Argparse is used to parse command-line arguments such as model weights, SVO file path, image size, and confidence threshold.

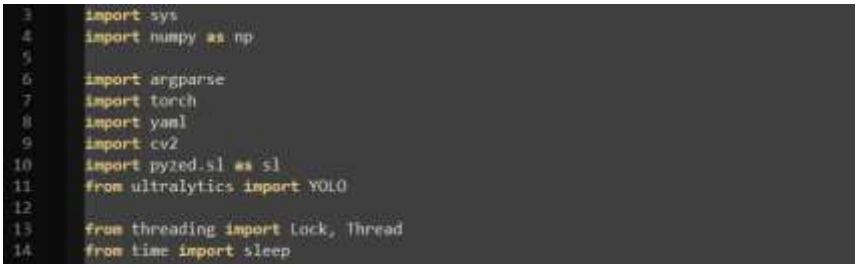
Torch is used to implement the YOLO (You Only Look Once) object detection algorithm, specifically through the PyTorch library.

OpenCV is used for tasks such as reading and displaying images, as well as rendering object detections.

The Ultralytics YOLO implementation is used for real-time object detection.

Threading is used to create and manage multiple threads for parallel execution of tasks such as model inference.

Time is used for tasks such as measuring inference time and introducing delays between operations.

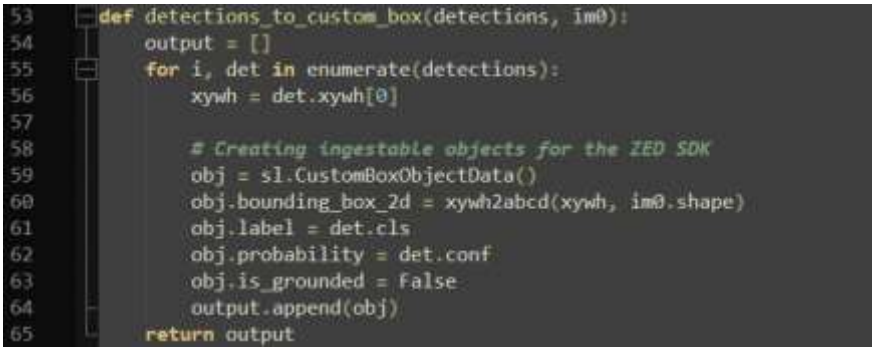


```
import sys
import numpy as np

import argparse
import torch
import yaml
import cv2
import pyzed.sl as sl
from ultralytics import YOLO

from threading import Lock, Thread
from time import sleep
```

Fig 6: Code snippet for importing libraries

2.2.2 *Convert detections to custom box object data for the ZED SDK*

The ‘`detections_to_custom_box`’ function is responsible for converting detections obtained from the YOLO model into custom box object data that can be ingested by the ZED SDK for further processing. The function takes two parameters: detections, which represents the list of detections obtained from the YOLO model, and im0, which is the original image frame where the detections were made.



```
def detections_to_custom_box(detections, im0):
    output = []
    for i, det in enumerate(detections):
        xywh = det.xywh[0]

        # Creating ingestable objects for the ZED SDK
        obj = sl.CustomBoxObjectData()
        obj.bounding_box_2d = xywh2abcd(xywh, im0.shape)
        obj.label = det.cls
        obj.probability = det.conf
        obj.is_grounded = False
        output.append(obj)
    return output
```

Fig 7 : Code snippet of detections to custom box object data

2.2.3 *Processing detections and calculating inference*

Inference time refers to the time taken by the model to process an input image and generate predictions. It includes the time taken for pre-processing the image, forwarding it through the neural network, and post-processing the output to obtain meaningful detections. In this context, inference time is crucial for assessing the efficiency and real-time performance of the object detection system, as lower inference times enable faster processing of frames, resulting in higher frame rates and smoother operation of the system.

```
        if run_signals[camera_id - 1]:
            lock[camera_id - 1].acquire()
            img = cv2.cvtColor(image_nets[camera_id - 1], cv2.COLOR_BGRA2RGB)
            det = model.predict(img, save=False, imgsz=img_size, conf=conf_thres, iou=iou_thres)[0].cpu().numpy().boxes
            detections[camera_id - 1] = detections_to_custom_box(det, image_nets[camera_id - 1])
            lock[camera_id - 1].release()
            end_time = time.time()  # End time for measuring inference time
            inference_time = end_time - start_time  # Calculate inference time for the current frame
            fps = 1.0 / inference_time  # Calculate frames per second
```

Fig 8 : Code snippet of processing detections and calculating inference

2.2.4 *Threaded model inference initialization*

We need threading in this scenario to enable concurrent execution of object detection for multiple cameras. By running each camera's object detection process in a separate thread, we can leverage the multi-core capabilities of modern CPUs or parallel processing units like GPUs to improve overall system performance and efficiency. Threading allows the system to handle multiple tasks simultaneously, thereby reducing the time required to process frames from each camera and enhancing the real-time performance of the object detection system. Additionally, threading can help prevent one slow or blocking task from delaying the processing of frames from other cameras, leading to smoother operation and better responsiveness of the system.

```
    capture_threads = [
        Thread(target=torch_thread, args=(opt.weights[i], opt.img_size, i + 1, opt.conf_thres)) for i in range(2)
    ]
    for thread in capture_threads:
        thread.start()
```

Fig 9 : Code snippet of threaded model inference initialization

2.2.5 *Acquiring the frame*

This process effectively captures an image from the left view of the ZED camera specified by the index i and makes it available for subsequent operations such as object detection or display. By retrieving images from each camera in the system, this code segment enables the real-time processing of multiple camera feeds, facilitating tasks such as object detection, tracking, and visualization in applications such as surveillance, robotics, or augmented reality.

```
            if zed[i].grab(runtime_params) == sl.ERROR_CODE.SUCCESS:

                lock[i].acquire()
                zed[i].retrieve_image(image_left_tmp[i], sl.VIEW.LEFT, sl.MEM.CPU, display_resolutions[i])
                image_nets[i] = image_left_tmp[i].get_data()

                lock[i].release()
```

Fig 10 : Code snippet of acquiring the frame

2.2.6 *Ingest detections from captured image*

These lines of code are responsible for ingesting object detections from the captured image obtained from a ZED camera. First, a lock is acquired using '`lock[i].acquire()`' to ensure thread safety and prevent data corruption during access to shared resources. Next, the '`ingest_custom_box_objects`' method of the ZED camera object (zed[i]) is called to ingest the detected objects represented by the detections[i] list.

```
213                     zed[i].ingest_custom_box_objects(detections[i])
214                     det_list = detections[i]
```

Fig 11 : Code snippet of ingest detections from captured image

2.2.7 *Depth measurement*

In the context of 3D vision and depth sensing, a depth map refers to a representation of the scene where each pixel in the image is associated with a corresponding depth value, indicating the distance from the camera to the objects in the scene. This line of code initializes a list named '`depth_map`' containing two '`sl.Mat()`' objects. Each '`sl.Mat()`' object represents a depth map associated with a specific camera. These depth maps are typically obtained from depth-sensing cameras or stereo vision systems, such as the ZED camera used in this project.

```
depth_map = [sl.Mat(), sl.Mat()]
```

Fig 12 : Code snippet of depth measurement

2.2.8 *Display of measured distance*

This line retrieves the depth value at a specific pixel location (x, y) from the depth map associated with the '`i`'-th camera. The '`get_value()`' method takes the coordinates of the pixel as input and returns the depth value along with an error code if the operation encounters any issues. This information is crucial for tasks such as object localization, scene understanding, and depth-based analysis in computer vision applications.

```
254                         err, depth_value = depth_map[i].get_value(x,y)
```

Fig 13: Code snippet of displaying the measured distance

2.2.9 *Close camera*

These lines close the connections to the ZED cameras indexed at 0 and 1 respectively. The '`close()`' method is used to release the resources associated with the cameras and terminate their operation.

```
272     zed[0].close()
273     zed[1].close()
```

Fig 14 : Code snippet of closing camera

2.2.10 *Results*

This result shows terminal window, both the camera windows and GPU performance. Ideally, we are running two different YOLO models.

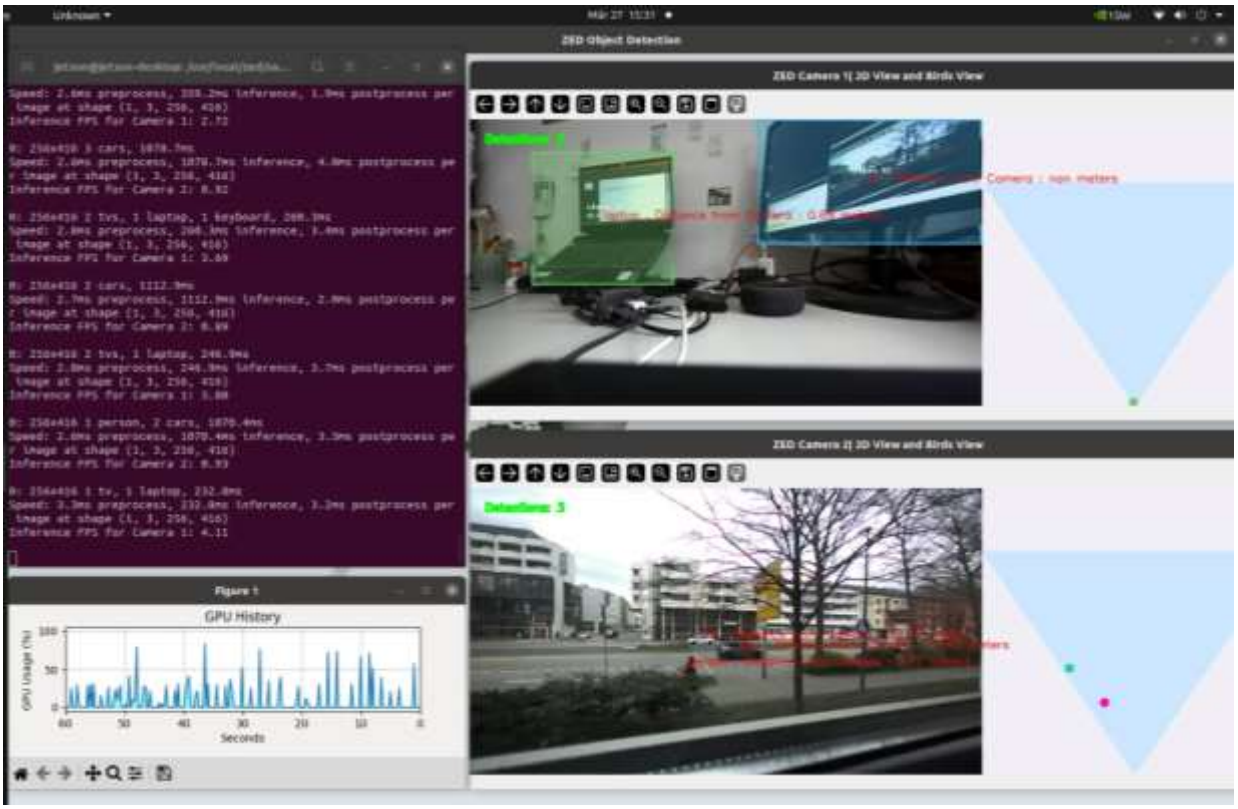


Fig 15: Result image

2.3 Process Flow Diagram

First, we obtain the image data from the ZED camera. This data is processed by a Yolo model to detect the object and get their bounding boxes. Further Yolo Model also classifies whether the person has a mask on or not. After the detection, the distance of the people from the camera is estimated using the point cloud data from ZED SDK. This flow diagram explains how the object detection and distance estimation is done.

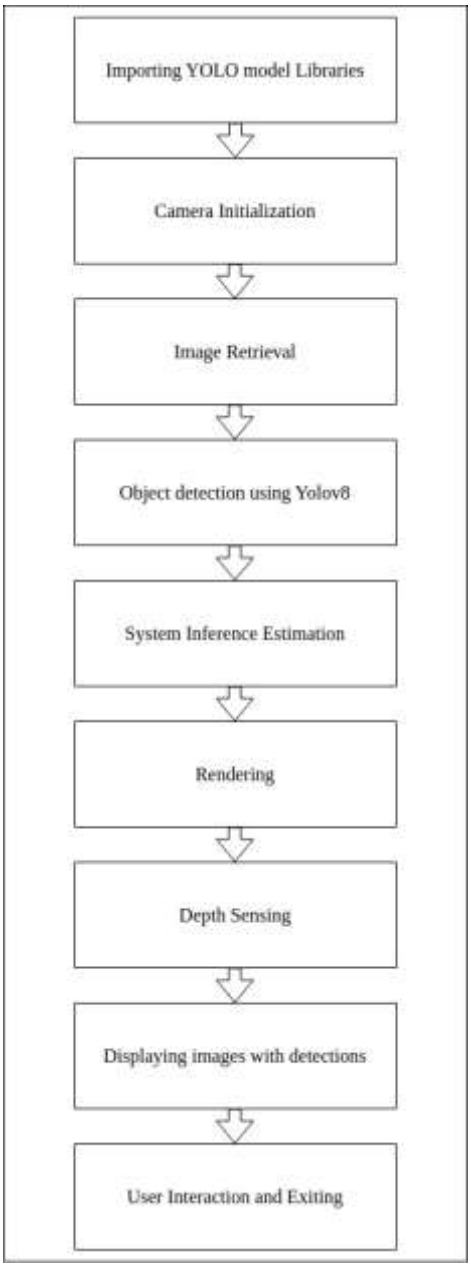


Fig 16: Process Flow Diagram

3 Performance Comparison

3.1 GPU & CPU Performance

GPU Performance:

- YOLOv8 models heavily rely on GPU acceleration for efficient inference due to their complex neural network architectures.
- A powerful GPU on the Jetson Orin Nano can significantly accelerate the inference process by parallelizing computations across multiple cores.
- Higher GPU performance allows for faster processing of convolutional layers, which are computationally intensive in YOLO models.
- Improved GPU performance results in higher frames per second (FPS) during inference, enabling real-time object detection in video streams or high-resolution images.
- Different YOLOv8 models may have varying levels of computational complexity, and a better GPU can handle more complex models with larger input sizes and higher resolution images efficiently.

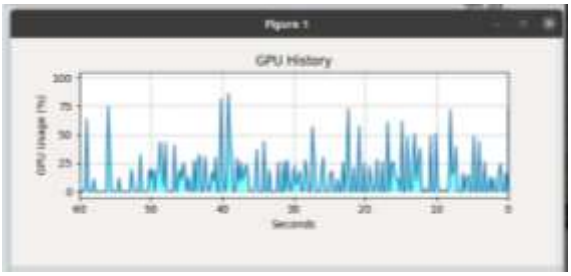


Fig 17 : GPU Graph for YOLOv8n model

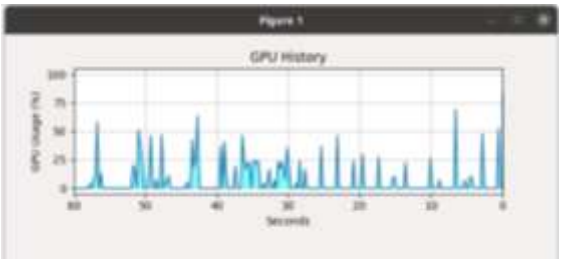


Fig 18 : GPU Graph for YOLOv8l model

A GPU usage of 75% for the YOLOv8n model suggests that the model is making effective use of a significant portion of the available GPU resources. This high GPU

utilization typically indicates that the model is efficiently leveraging the GPU's parallel processing capabilities to perform computations, resulting in faster inference speeds and more effective object detection.

On the other hand, a GPU usage of 60% for the YOLOv8l model indicates that it is utilizing a slightly lower proportion of the GPU's computational power compared to the YOLOv8n model. While still a substantial utilization, it may suggest that the YOLOv8l model either requires fewer computational resources or is less optimized for parallel processing on the GPU compared to the YOLOv8n model.

Overall, higher GPU usage percentages generally correlate with faster inference speeds and more efficient utilization of hardware resources, leading to better performance in real-time object detection applications. Hence, we can see that YOLOv8n has better performance

CPU Performance:

- While GPU acceleration is crucial for deep learning inference, the CPU also plays a supporting role, especially in preprocessing tasks, data loading, and post-processing steps.
- CPU performance impacts tasks such as reading images from disk, resizing images to the input size required by the YOLO model, and performing non GPU-accelerated operations.
- In multi-threaded applications like object detection using YOLOv8 on the Jetson Orin Nano, efficient CPU utilization can help in parallelizing tasks across CPU cores, improving overall system throughput.
- Although the primary workload of YOLOv8 inference is handled by the GPU, a well-optimized CPU can contribute to reducing overall latency and improving system responsiveness.
- In the context of object detection, CPU performance is crucial for real-time or near-real-time applications, especially in scenarios where GPU resources are limited or unavailable.

Fig 19: CPU usage for YOLOv8l model

Fig 20: CPU usage for YOLOv8n model

In terms of CPU measurements, we discovered that every version of our project makes use of all the 6 CPU cores.

YOLOv8L model utilizes approximately 41.4% of the CPU's computational capacity during inference. A lower percentage suggests that the model is less computationally demanding and requires fewer resources to perform inference tasks. This could be due to a simpler architecture, fewer layers, or smaller input/output dimensions compared to other models.

In contrast, YOLOv8n model utilizes a higher percentage of the CPU's computational capacity, approximately 85.4%. A higher percentage indicates that the model is more computationally intensive, likely due to a more complex architecture, deeper layers, larger input/output dimensions, or additional computational requirements.

The preference of model ultimately depends on the specific requirements and constraints of the application. The constrains being Real time requirements, Resource constraints, Accuracy and Efficiency Trade off.

The choice between low and high CPU performance models should be based on a thorough understanding of the application requirements, available hardware resources, performance targets, and cost considerations.

3.2 Frames per Second

Frames per Second (FPS) in object detection refers to the number of frames or images processed by the detection system in one second. It measures the speed at which the system can analyze and detect objects within the input video stream or sequence of images.

In object detection applications, FPS is a crucial metric as it directly impacts the system's real-time performance and responsiveness. Higher FPS values indicate that the system can process more frames per unit of time, resulting in smoother and more responsive object detection in real-world scenarios. Faster FPS rates enable the system

to keep up with dynamic environments, where objects may move quickly or appear/disappear rapidly.

In the context of object detection using neural networks like YOLO (You Only Look Once), "inference" refers to the process of applying a trained model to input data (such as images or video frames) to make predictions or detections. During inference, the model analyzes the input data and produces output, such as bounding boxes around objects, their classifications, and confidence scores.

Frames per second (FPS) during inference refers to the number of input frames processed by the model in one second. It measures the speed at which the object detection algorithm can analyze successive frames of a video stream or images.

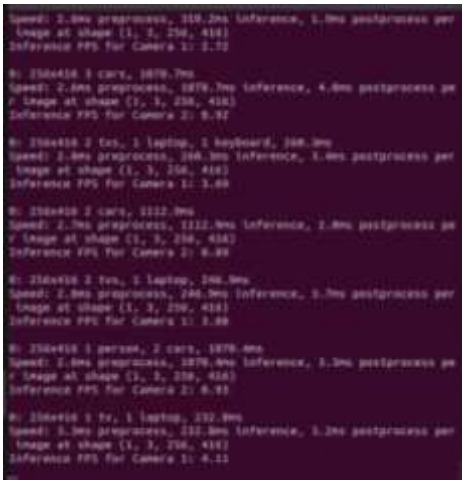


Fig 21: Terminal output when using two different models

1. For two different YOLO models

For YOLOv8n model (Camera 1), the inference time is 359.2 milliseconds and inference FPS is 2.72 and for YOLOv8l model (Camera 2), the inference time is 1078.7 milliseconds and inference FPS is 0.92. Depending upon these observations, we reach to a conclusion. Higher inference time requires more computational resources and time for inference due to its larger size and complexity. Camera 1 has a higher inference FPS (2.72) compared to Camera 2 (0.92), indicating that the YOLOv8n model performs faster on Camera 1. This could be due to the smaller size and complexity of the YOLOv8n model compared to YOLOv8l.

2. When reduced the image size from 416 to 256

For YOLOv8n model (Camera 1), the inference time is 1024.4 milliseconds and inference FPS is 0.95 and for YOLOv8l model (Camera 2), the inference time is 1665.4 milliseconds and inference FPS is 0.59. We reached to a conclusion that Changing the image size from 416 to 256 pixels has significantly reduced the time taken for both inference and post-processing. This reduction in image size has led to faster processing times for each image. Despite the reduction in image size, the inference frames per second (FPS) have decreased for both cameras. This decrease in FPS indicates that although the processing time per image has decreased, it has not decreased enough to compensate for the reduced image size.

3. When increased the image size from 256 to 1088

For YOLOv8n model (Camera 1), the inference time is 906.8 milliseconds and inference FPS is 1.07 and for YOLOv8l model (Camera 2), the inference time is 8573.1 milliseconds and inference FPS is 0.12. Based on the provided outputs with an image size of 640x1088 pixels, here are some observations:

Increasing the image size from 256 to 1088 pixels has resulted in longer processing times for both cameras. This increase in image size has led to higher inference and post-processing times per image.

With the larger image size, the inference frames per second (FPS) have decreased significantly for both cameras. The decrease in FPS indicates that the processing time per image has increased substantially, leading to slower overall performance.

Overall, the decision to use larger image sizes should consider the specific requirements of the application, balancing the need for detailed information with the available computational resources and desired processing speed.

4 Application

4.1 Retail Stores

Shopping malls, convenience stores, restaurants, and brick-and-mortar stores can leverage object detection applications to detect people and store items. This helps them better understand shopper behavior and improve operations. Edge CV gives business managers access to real-time information that allows them to make improvements instantly, rather than days or weeks later.

Object detection helps retail operators intelligently track things like product interactions by analyzing how often customers stop at displays and what products they pick up or put back. Quick service restaurants can also benefit from CV by monitoring food preparation and order taking to ensure optimal speed of service.

4.2 Transportation

Object detection is the fundamental technology used in developing self-driving systems. In autonomous driving, object detection is used to detect and localize other vehicles, pedestrians, and obstacles on the road. This allows self-driving vehicles to navigate safely on the roads and avoid collisions. Object detection can also be used to monitor traffic and road conditions in smart cities. CV systems provide real-time data to transportation agencies about current traffic levels, potential hazards, and accidents.

4.3 Healthcare

In healthcare, specifically in the radiology sector, object detection can be used to identify and localize tumors and other abnormalities in medical images such as MRIs, CT scans, x-rays, etc. This can help doctors and radiologists make more accurate diagnoses and develop more effective treatment plans. Object detection can help automate and augment the analysis and interpretation of medical images, by providing accurate and consistent detection and identification of anatomical structures, lesions, tumors, or other abnormalities. Object detection can also help reduce the workload and improve the efficiency and productivity of human experts, by highlighting the areas of interest or concern, and providing suggestions or recommendations based on the detected objects.

4.4 Agriculture

Object detection can be used in areas such as Crop monitoring, Yield estimation, Pest detection. This can help farmers take corrective measures before the issue ruins the whole crop. One of the techniques that is popular is Few-Shot Learning (FSL). FSL is a type of meta-learning in which a learner is given practice on several related tasks during the meta-training phase to be able to generalize successfully to new but related activities with a limited number of instances during the meta-testing phase. Here, the application of FSL in smart agriculture, with particular in the detection and classification is reported.

4.5 Entertainment

Among the earliest sports events to use object detection were football and rugby games; in fact, the NFL uses real-time object recognition to track the football during a game. This simplifies the analysis and makes it simpler to track the ball when numerous players encircle it, and no one camera viewpoint can fully capture it.

Additionally, the sports betting sector extensively uses object detection to gather data effectively, accelerating and improving analysis.

5 Future Scope

The proposed enhancement involves extending the project to develop a Real-time Multi-Object Tracking System using YOLOv8 for object detection and OpenCV trackers like KCF, MOSSE, and CSRT for object tracking. Integration of the SORT algorithm will enhance tracking across multiple camera feeds, associating object detections over time and views for simultaneous tracking with high accuracy. Additionally, exploring Kalman filtering will improve tracking accuracy, especially in noisy environments, by mitigating uncertainties in object position estimation. This enhanced system will provide real-time, robust, and accurate multi-object tracking capabilities across diverse camera feeds, serving applications like surveillance, autonomous driving, and smart city infrastructure. Furthermore, this can implement using the latest YOLOv9 algorithm.

6 Conclusion

The project presents a robust multi-camera object detection and tracking system that prioritizes both accuracy and speed by utilizing YOLOv8 for inference. With the flexibility to employ different YOLOv8 models tailored to specific object detection applications, the system ensures optimal performance across various scenarios. Additionally, the code is designed to be scalable, facilitating the seamless integration of additional cameras as needed. This versatility allows the system to find utility in numerous applications, including but not limited to smoke detection, autonomous driving, and surveillance, where real-time object analysis and monitoring are crucial. Moreover, the system holds potential for further enhancement through potential advancements such as transitioning to YOLOv9 for improved accuracy and incorporating advanced tracking algorithms. By continually evolving and adapting to emerging technologies, the system can continue to meet the evolving needs of object detection and tracking in diverse environments.

References

[1] https://www.stereolabs.com/docs/depth-sensing/depth-settings
[2] https://medium.com/@rohinfablabz/camera-triangulation-for-depth-and-distance-analysis-6e9da94cc9d7
[3] https://github.com/rbonghi/jetson_stats?tab=readme-ov-file
[4] https://pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/
[5] https://github.com/stereolabs/zed-sdk/tree/master/object%20detection/custom%20detector/python/pytorch_yolov8
[6] https://github.com/ultralytics/ultralytics
[7] https://forums.developer.nvidia.com/t/monitor-gpu-usage/72250
[8] https://www.stereolabs.com/docs/depth-sensing/using-depth
[9] https://developer.nvidia.com/embedded/jetpack-sdk-512
[10] https://docs.ultralytics.com/quickstart/#conda-docker-image
[11] https://medium.com/@rohinfablabz/camera-triangulation-for-depth-and-distance-analysis-6e9da94cc9d7
[12] https://docs.nvidia.com/jetson/jetpack/index.html/
[13] https://www.v7labs.com/blog/yolo-object-detection
[14] https://developer.nvidia.com/embedded/jetpack-sdk-512
[15] https://learnopencv.com/ultralytics-yolov8/
[16] https://levity.ai/blog/neural-networks-cnn-ann-rnn
[17] https://research.aimultiple.com/object-detection/
[18] https://alwaysai.co/blog/object-detection-for-business