

A

*Project Report on*

**“Multicamera Object Detection and Distance Measurement on Nvidia Jetson TX2”**

*Submitted in partial fulfillment of the requirements for award of the degree of  
ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY – INTERNATIONAL*

*For the academic year 2023-2024*

<b>Team Members</b>	<b>Matriculation Number</b>
Girish Bheemaraddi Tabaraddi	1119236
Drashti Rajendrakumar Shah	1119295
Madeshwar Ganapathy Selvamani	1120093
Trusha Sanjiv Tagade	1120564
Yogita Netaram Labde	1119264

*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 discusses the deployment of an algorithm for face mask recognition, object detection, and distance estimation. The algorithm is executed on the Nvidia Jetson TX2 board with multiple ZED-Cameras. Additionally, it provides an explanation of how to implement this capability using the Cuda DNN module. The primary objective is to offer readers insight into incorporating this functionality into their own projects. Building upon this foundation, upcoming projects can create more intricate and effective algorithms.

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>6</b>
1.1	Goals of the Project .....	7
1.2	Hardware Used .....	7
1.3	Introduction to Artificial Neural and Convolutional Neural Network.....	8
1.3.1	Artificial Neural Network (ANN) .....	8
1.3.2	Convolutional Neural Network (CNN).....	9
<b>2</b>	<b>Jetpack Installation .....</b>	<b>10</b>
2.1	Introduction .....	10
2.2	Hardware Requirements.....	10
2.3	Software Requirements.....	11
2.4	Jetson TX2 NX Jetpack.....	11
2.4.1	Nvidia Jetson SDK Manager Installation Guide.....	12
2.4.2	Zed SDK Manager Installation Guide .....	15
2.4.3	OpenCV CUDA, cuDNN module Installation Guide .....	16
2.4.4	TensforFlow Installation Guide .....	19
<b>3</b>	<b>Implementation .....</b>	<b>21</b>
3.1	Object Detection .....	21
3.1.1	Implementation at glance .....	21
3.1.2	Implementation in Detail.....	22
3.1.2.1	Import Libraries .....	22
3.1.2.2	Load Pre-Trained Algorithm .....	22
3.1.2.3	Create Network and Detection Model: .....	23
3.1.2.4	Multi-Camera Integration and ZED SDK Initialization:.....	23
3.1.2.5	Detection of Objects using YOLO Model: .....	24
3.2	Distance Estimation.....	25
3.2.1	Introduction.....	25
3.2.2	Depth sensing overview .....	25
3.2.2.1	Depth Perception.....	26
3.2.2.2	Depth Map .....	26
3.2.2.3	3D Point Cloud .....	27
3.2.3	The Operation Principle of Zed Camera .....	27
3.2.4	Implementation in Detail.....	28

3.2.4.1	Capture Data .....	28
3.2.4.2	Measure Distance in point cloud.....	29
3.3	Face Mask Detection .....	29
3.3.1	Introduction.....	29
3.3.2	Implementation in detail .....	30
3.3.3	Methodology .....	30
3.3.3.1	Model Initialization.....	30
3.3.3.2	Detection .....	30
3.3.3.3	Non-Maximum Suppression (NMS) .....	30
3.3.3.4	Classification Statistics with NMS.....	31
3.3.3.5	Displaying Results on the Frame .....	31
3.4	Implementation Output .....	32
4	Performance Comparison .....	35
5	Future Scope - Fusion of Object detection .....	37
5.1	Introduction - Sensor Fusion .....	37
5.2	The Challenge of Synchronization .....	37
5.3	Implementing Sensor Fusion .....	37
5.3.1	Data Acquisition and Preparation: .....	37
5.3.2	Object Detection Using YOLO and R-CNN:.....	38
5.3.3	Output Synchronization: .....	38
5.3.4	Kalman Filter Integration for Fusion: .....	38
5.3.5	Fusion and Enhanced Object Detection:.....	38
5.3.6	Accurate Object Tracking and Future Predictions: .....	39
6	References.....	40

## List of Figures

Figure 1 YOLO Detection System. ....	7
Figure 2 Three Layer Artificial Neural Network.....	8
Figure 3 Convolutional Neural Network .....	9
Figure 4 Virtual Machine Settings .....	12
Figure 5 Login using Nvidia Credentials .....	13
Figure 6 Target Hardware Selection.....	13
Figure 7 Installing Target Components.....	14
Figure 8 Setup Process.....	14
Figure 9 Summary Finalization.....	15
Figure 10 Import Python Libraries.....	22
Figure 11 Load YOLOv3-Tiny Algorithm from configuration and weights file.....	23
Figure 12 Store Class names in a list.....	23
Figure 13 Network and Detection Model.....	23
Figure 14 Retrieve the list of connected ZED Camera and Initialize Camera parameters .....	24
Figure 15 Opening the Camera based on serial number and setting the runtime parameters.....	24
Figure 16 Object Detection function using YOLO model .....	24
Figure 17 Object Detection using YOLO model .....	25
Figure 18 Depth Map.....	26
Figure 19 3D Point Cloud.....	27
Figure 20 The Operation Principle of Zed Camera.....	28
Figure 21 Capture Data.....	28
Figure 22 Distance Calculation.....	29
Figure 23 Load Face mask Algorithm from configuration and weights file.....	30
Figure 24 Indexing of the Face mask .....	30
Figure 25 Detection and Increment.....	31
Figure 26 Displaying the Final Count .....	31
Figure 27 Distance Estimation and Object Detection image.....	32
Figure 28 Distance estimation and object detection using multiple cameras .....	32
Figure 29 Multi camera object detection and face mask detection separately .....	33
Figure 30 Multi camera object detection and face mask detection Separately .....	33
Figure 31 Face mask detection image .....	34
Figure 32 face mask detection image.....	34
Figure 33 Performance Compare .....	35
Figure 34 Error Analysis: Fast R-CNN vs. YOLO .....	36
Figure 35 Accuracy vs Time.....	36

# 1 Introduction

The human ability to glance at an image and instantly comprehend the objects within it, along with their positions and interactions, is remarkable. Our visual system's speed and accuracy enable us to undertake intricate tasks like driving with minimal conscious effort. Developing rapid and precise algorithms for object detection holds the potential for computers to drive vehicles without specialized sensors, empower assistive devices to offer real-time scene details to users, and open doors to versatile and responsive robotic systems.

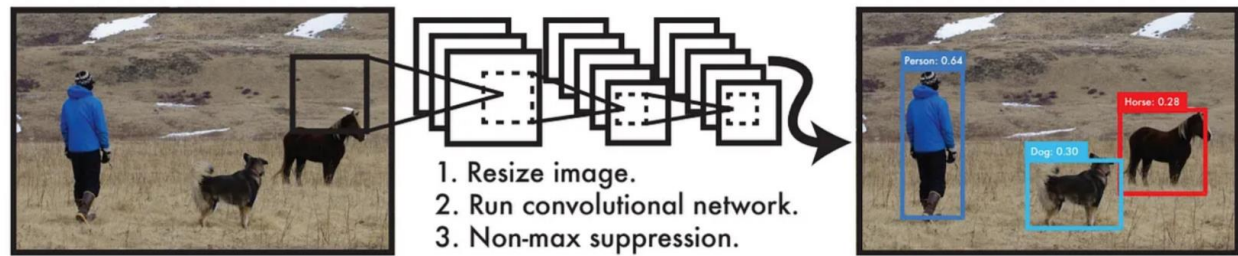
Presently, existing detection systems adapt classifiers to execute detection tasks. To identify an object, these systems employ a classifier designed for that specific object and assess it at different locations and scales across a test image. Techniques such as deformable parts models (DPM) utilize a sliding window method, wherein the classifier is executed at evenly spaced points throughout the entire image. Newer techniques, such as R-CNN, employ region proposal methods as a step to initially generate potential bounding boxes within an image. Subsequently, a classifier is applied to these proposed boxes. Following classification, post-processing is utilized to enhance the accuracy of bounding boxes, remove redundant detections, and reevaluate the boxes considering other objects present in the scene. However, these intricate pipelines are challenging to optimize due to the necessity of training each component separately.

As referenced in market research, the momentum behind the artificial intelligence software market remains a compelling narrative of growth and potential. Statistical data underscores this trend, highlighting the widespread recognition of AI's transformative power across industries. In examining the fiscal trajectory of the artificial intelligence software market, insights reveal a trajectory akin to a shooting star. The year 2018 marked the inception of this journey, with market revenue registering at a notable 10 billion USD. This foundational phase laid the groundwork for subsequent surges, as demonstrated by the revenue surge to approximately 34 billion USD in 2021. This remarkable three-fold expansion in just three years offers a tangible representation of AI's ascendancy.

Anticipating the path ahead, industry analysts project an even more remarkable vista for the year 2025. With estimations pointing towards an awe-inspiring market revenue of approximately 126 billion USD, the momentum becomes a beacon of technology's undeniable relevance. This robust projection reaffirms the collective belief in AI's transformative capacity and underscores its indispensability in shaping our technological landscape.

## 1.1 Goals of the Project

This current project uses detection models to implement different functionalities. The main goal however of this project is to use the YOLO model for object detection. YOLO is refreshingly simple: see Figure 1. A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance. This unified model has several benefits over traditional methods of object detection.



**FIGURE 1 YOLO DETECTION SYSTEM.**

YOLO reasons globally about the image when making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time, so it implicitly encodes contextual information about classes as well as their appearance. Fast R-CNN, a top detection method, mistakes background patches in an image for objects because it can't see the larger context. YOLO makes less than half the number of background errors compared to Fast R-CNN.

Another goal of this project is to implement a face recognition system and additionally a feature to recognize if the person in frame is wearing a mask or not. This can be especially useful in studies to show how many people are wearing a mask voluntarily. This project also extends the idea of object detection and distance estimation to multiple ZED cameras, where each of the ZED cameras has distinctive scope in the project. Object detection and distance estimation is obtained through one camera, while the other camera is responsible for the face-mask recognition, in this way both the cameras are connected to the board and are working with different functionalities.

YOLO still lags state-of-the-art detection systems in accuracy. While it can quickly identify objects in images it struggles to precisely localize some objects, especially small ones.

## 1.2 Hardware Used

This project requires power packed hardware and Nvidia TX2 board provides the right amount of varying performance metrics. The Jetson TX2 NX series module incorporates these same GPU architectural enhancements to further increase performance and reduce power consumption for computationally intensive applications. The below mentioned are some of the most important features of the Jetson TX2 board.

- AI Performance up to 1.33 TFLOPS.
- Pascal GPU – 256 NVIDIA CUDA Cores with maximum operating frequency of 1.3GHz

- Denver 2 CPU and Cortex A57 with ARMv8 64-bit heterogeneous multi-processing CPU architecture.
- It has a 4GB 128-bit LPDDR4 DRAM and 16GB eMMC 5.1 Flash Storage.

Additionally, to the Nvidia TX2 board, two ZED cameras will be used to capture the video feed. The ZED 2 is a stereo camera that provides high-definition 3D video and neural depth perception of the environment. It has been designed for the most challenging applications, from autonomous navigation and mapping to augmented reality and 3D analytics. It is the first stereo camera that uses neural networks to reproduce human vision and can detect and track objects with spatial context. By combining AI and 3D the ZED 2 localizes the objects in space and provides the tools to create the next-generation spatial awareness. The ZED 2 camera comes with a Depth range of 0.3m to 20m with accuracy of <1% up to 3m and <5% up to 15m. It has a dual 4MP camera integrated with an option to capture 2K 3D videos. Also, the high frame rate makes the ZED camera attractive for developers and engineers. With 30FPS (frames per second) on 1080p and 100FPS on WVGA (Wide Video Graphic Adapter) it is one of the most advanced depth cameras.

### 1.3 Introduction to Artificial Neural and Convolutional Neural Network

#### 1.3.1 Artificial Neural Network (ANN)

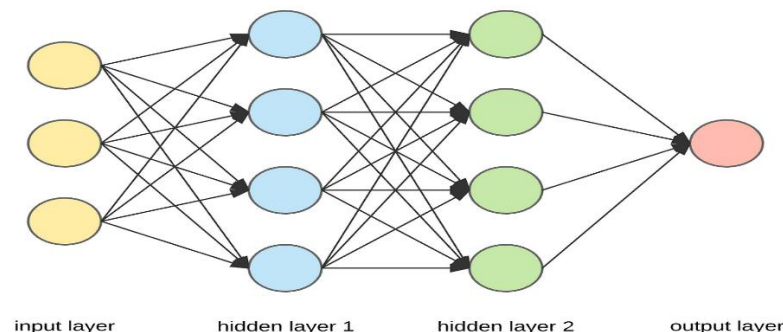
Artificial Neural Networks, inspired by the human brain's structure and functionality, form the basis of Deep Learning—a powerful subset of Machine Learning. Deep Learning exhibits human-like decision-making abilities and progressively enhances its performance over time. ANNs consist of interconnected nodes organized into layers, including an input layer, hidden layers, and an output layer. Each node, resembling a neuron, is associated with a weight and a threshold.

*Components of an ANN:*

**Input Layer:** Represents the features of the dataset and receives the initial data.

**Hidden Layers:** Process and transform the input data through interconnected nodes.

**Output Layer:** Produces the final result, often indicating recognized categories.



**FIGURE 2 THREE LAYER ARTIFICIAL NEURAL NETWORK**



ANNs adapt by adjusting the weights and thresholds in response to data inputs, enabling them to learn and improve their decision-making capabilities. The network activates nodes based on threshold values, facilitating the flow of data to subsequent layers. ANNs exhibit a structure akin to Figure 2, with the input layer, hidden layers, and output layer interconnected.

### 1.3.2 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) revolutionized the analysis of visual and sequential data, particularly images, audio, and speech signals. Unlike traditional ANNs, CNNs incorporate unique layers that enhance their performance in processing and recognizing intricate patterns.

*Convolutional Layers:* Detect patterns within the input data using convolutional filters.

*Pooling Layers:* Condense and abstract the detected features, reducing computational load.

*Fully Connected Layers:* Analyze and process the extracted features to produce results.

The structure of a CNN shares similarities with ANNs, consisting of input and output layers, as well as hidden layers featuring convolutional and pooling layers. As data traverses through the network, the complexity of the patterns detected increases. Initial layers focus on simple features like edges and colors, while subsequent layers discern larger elements or shapes until the desired object is recognized.

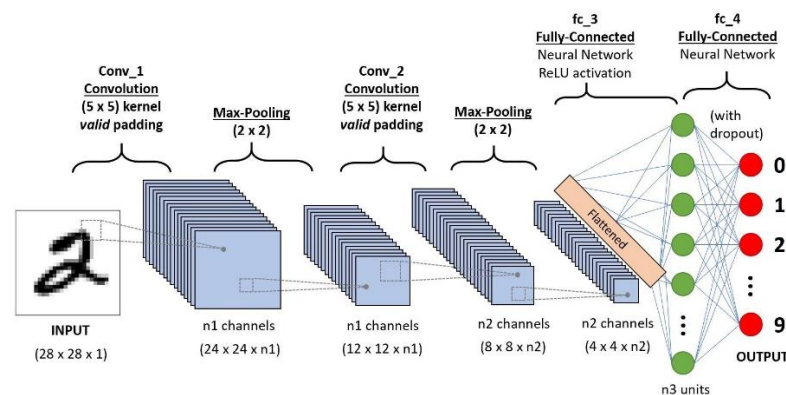


FIGURE 3 CONVOLUTIONAL NEURAL NETWORK

CNNs have become the backbone architecture for various advanced models, such as R-CNN, Fast R-CNN, Faster R-CNN, and YOLO. These models employ the prowess of CNNs to achieve accurate and efficient object detection in images.

In conclusion, Artificial Neural Networks and Convolutional Neural Networks are pivotal elements in the realm of artificial intelligence, enabling machines to replicate human-like decision-making processes and excel in tasks involving pattern recognition and data analysis.

## 2 Jetpack Installation

### 2.1 Introduction

Object detection serves as a foundational task within computer vision, involving the identification and localization of objects within images or videos. In the contemporary landscape of real-time video applications, the imperative to process video streams promptly and across multiple cameras is steadily growing. The NVIDIA Jetson TX2 emerges as a high-performance embedded system-on-module (SoM), meticulously crafted for applications in artificial intelligence (AI). Its robust graphics processing unit and multicore CPU render it exceptionally well-suited for real-time processing of video streams from multiple camera sources.

Leveraging the multicamera capability of the NVIDIA TX2 for object detection necessitates the configuration of multiple cameras, with subsequent video stream processing facilitated by robust deep learning frameworks like TensorFlow or Pytorch. This strategy is widely applicable, finding utility across surveillance systems, robotics, and autonomous vehicles—environments where real-time object detection and tracking are prerequisites.

In this segment, we shall embark on an exploration of the steps entailed in configuring multiple ZED cameras on the NVIDIA TX2. Our journey will encompass the requisites, encompassing both hardware and software, and delve into the practical implementation of object detection across multiple camera streams using TensorFlow. The discourse will encompass not only the technical essentials but also extend to the nuances of code implementation for seamless realization of object detection within the realm of multiple camera streams.

### 2.2 Hardware Requirements

Implementing object detection with multiple ZED cameras on the NVIDIA Jetson TX2 is a task that demands potent hardware resources. Below are the detailed hardware prerequisites:

- *NVIDIA Jetson TX2 Module:* The Jetson TX2 serves as a dedicated platform, tailored for robust high-performance computation. Boasting a 64-bit ARM A57 CPU, a 256-core Pascal graphics processor, and an 8 GB LPDDR4 memory, it stands as the cornerstone of this setup.
- *Multiple ZED 2 Cameras:* The ZED 2 camera emerges as a stereo camera, adept at capturing depth particulars alongside delivering high-resolution images. By integrating multiple ZED 2 cameras, the capability to capture diverse viewing angles is magnified, thereby amplifying the coverage extent for seamless object detection.
- *USB 3.0 Hubs:* The necessity for multiple USB 3.0 hubs arises to facilitate the connection of the array of ZED cameras to the Jetson TX2 module. It is of paramount importance to ensure that the selected USB hub offers ample bandwidth and is fully compatible with the Jetson TX2.

- *Supplementary Memory (if needed):* Certain object recognition algorithms mandate substantial data quantities, be it for training or generating output. Hence, it might prove prudent to incorporate external storage options, such as SSD or HDD devices, for the storage of extensive data sets.

This comprehensive hardware configuration forms the bedrock for undertaking efficient and effective object detection employing multiple ZED cameras in conjunction with the NVIDIA Jetson TX2.

## 2.3 Software Requirements

To implement object detection using multiple ZED Cameras on NVIDIA TX2, several software requirements need to be fulfilled. These requirements include the Jetpack, Tensor flow, and Zed SDK.

## 2.4 Jetson TX2 NX Jetpack

The Jetson TX2 NX, powered by NVIDIA, presents an exceptional platform for a wide range of AI and robotics projects. This compact and energy-efficient system-on-module is coupled with the versatile Jetpack software stack, offering a comprehensive toolkit for AI development and deployment.

- *Deep Learning Frameworks:* Jetpack includes popular deep learning frameworks like TensorFlow, PyTorch, and MXNet, simplifying AI model development and deployment.
- *CUDA Toolkit:* Benefit from the power of parallel processing with the CUDA toolkit, optimizing GPU-accelerated computing for enhanced performance.
- *TensorRT:* JetPack integrates TensorRT, NVIDIA's deep learning inference optimizer and runtime, to optimize and deploy AI models for low-latency inferencing.
- *VisionWorks:* Develop sophisticated computer vision applications using NVIDIA's VisionWorks library, enabling efficient image and video processing.
- *Developer Tools:* JetPack includes developer tools like NVIDIA Nsight Systems and NVIDIA Nsight Graphics, aiding in performance optimization and debugging.
- *Containerization:* Jetson Container Runtime allows seamless deployment of containerized applications, streamlining development and deployment workflows.

Jetson Jetpack serves as a comprehensive software development kit (SDK) designed to equip developers with the essential tools and libraries for crafting artificial intelligence (AI) and machine learning applications on NVIDIA Jetson platforms. Within Jetpack, a range of crucial components converges, encompassing the Operating System, CUDA Toolkit, cuDNN, TensorRT, VisionWorks, Multimedia API, and Deep Stream SDK. Collectively, these components empower developers with a holistic ecosystem to seamlessly engineer potent AI and machine learning applications across NVIDIA Jetson platforms.

## 2.4.1 Nvidia Jetson SDK Manager Installation Guide

Flashing the NVIDIA TX2 with JetPack involves two essential components: *the host system* and *the target system*.

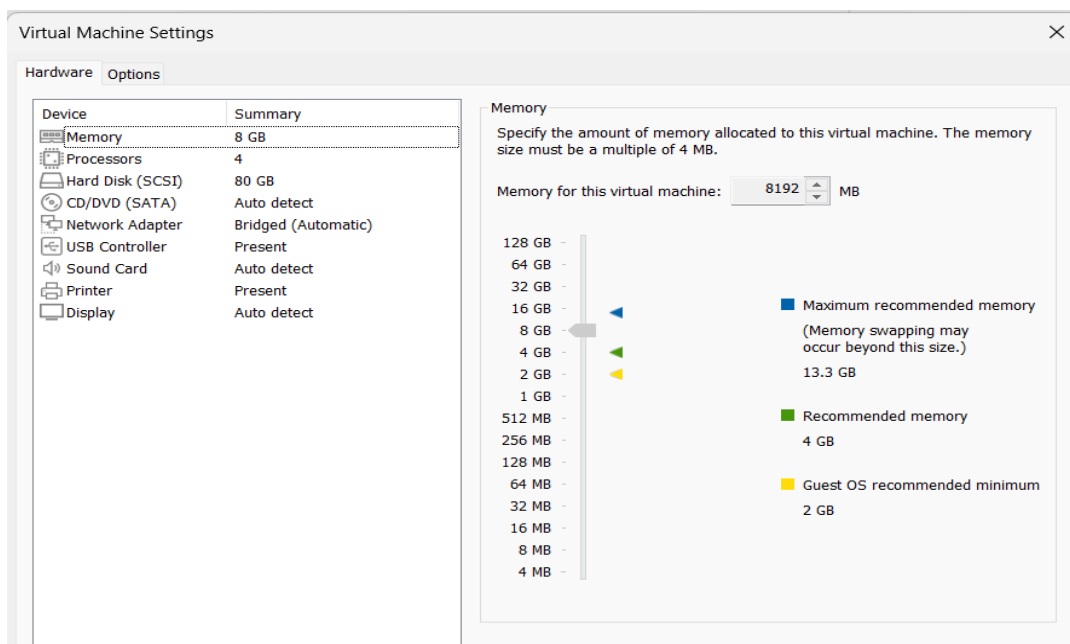
The host system serves as the foundation where the Jetson TX2 flashing software is installed. This software plays a pivotal role in deploying the operating system, drivers, and additional software onto the Jetson TX2. Typically, the host system is a Linux-based desktop or laptop computer.

On the other hand, the target system embodies the Jetson TX2 device itself. It functions as the recipient of the operating system and software installation. The interaction between the target system and the host system occurs via a USB connection. Through this connection, the flashing software establishes communication with the target system, orchestrating the installation process seamlessly.

Virtualization is the process of creating a virtual (software-based) representation of a physical computing resource, such as a server, storage device, network, or operating system. It enables multiple virtual instances, known as virtual machines (VMs), to run on a single physical machine.

On the Virtual Machine installed in the host system, we need to change the network setting, to find the IP address of our Jetson TX2 board after flashing the OS. Go to VM Setting > Network Adapter > Bridged. Check the *replicate physical network connection state*. This step must be performed even before the SDK manager installation.

Before installing the Nvidia SDK manager, the host system needs a virtual machine with ubuntu installed with the following settings,

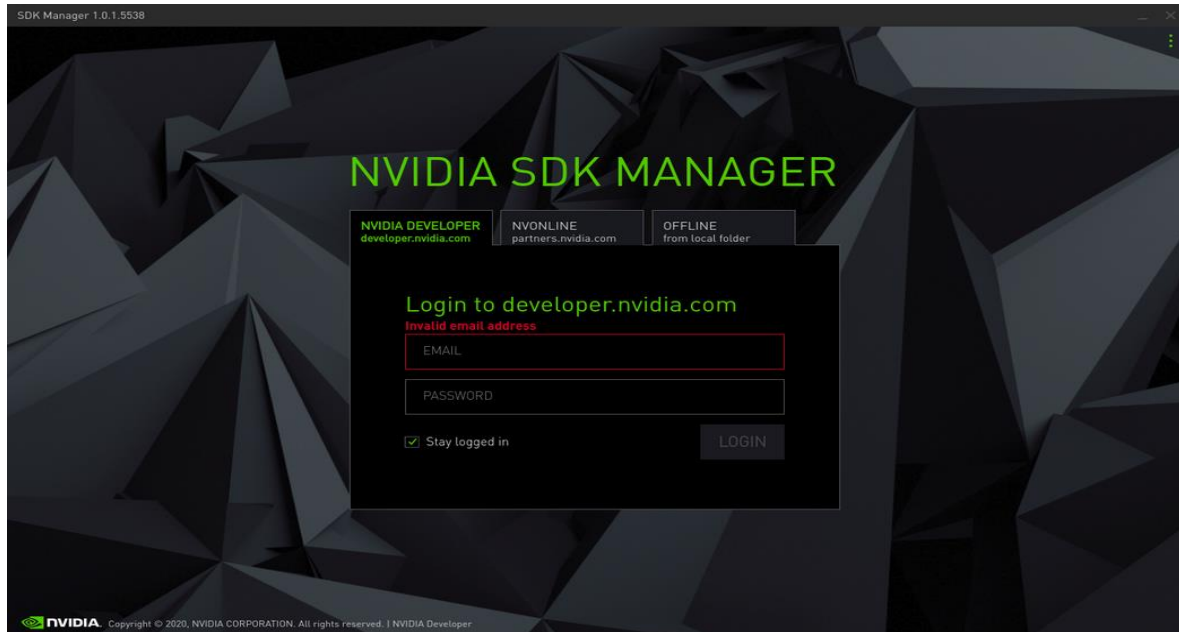


**FIGURE 4 VIRTUAL MACHINE SETTINGS**

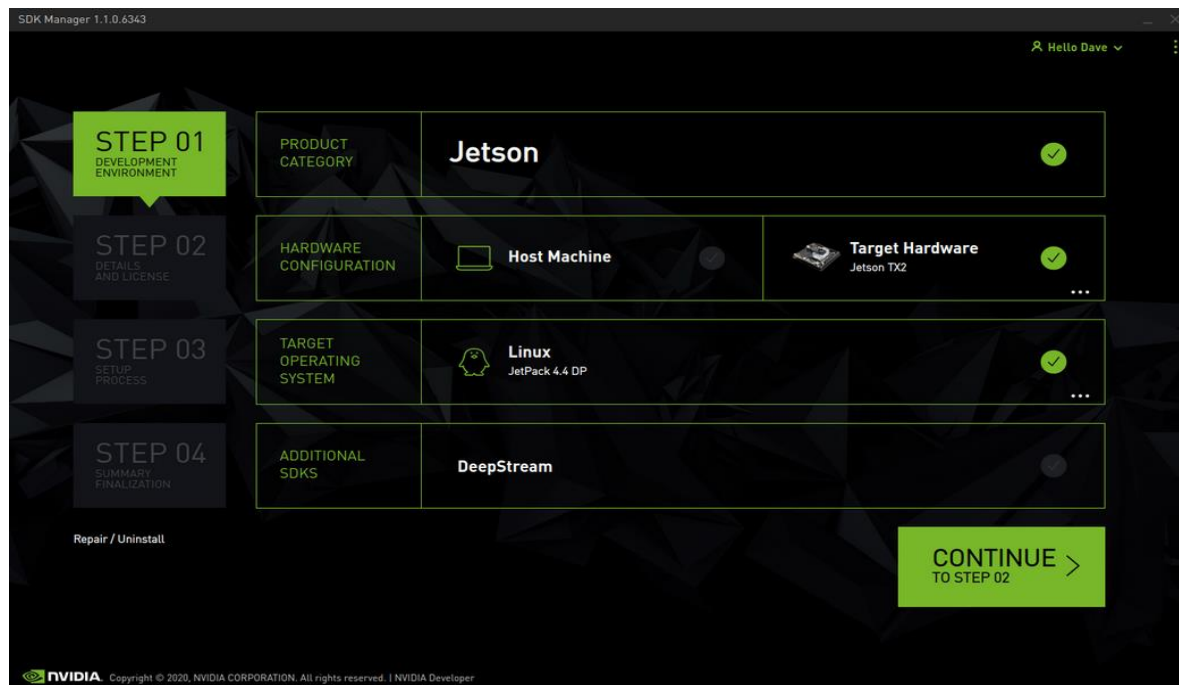
Here are the general steps for installing Jetpack 4.6.2 on Nvidia Jetson TX2:

1. Download Jetpack SDK Manager from the official Nvidia website on the host system.

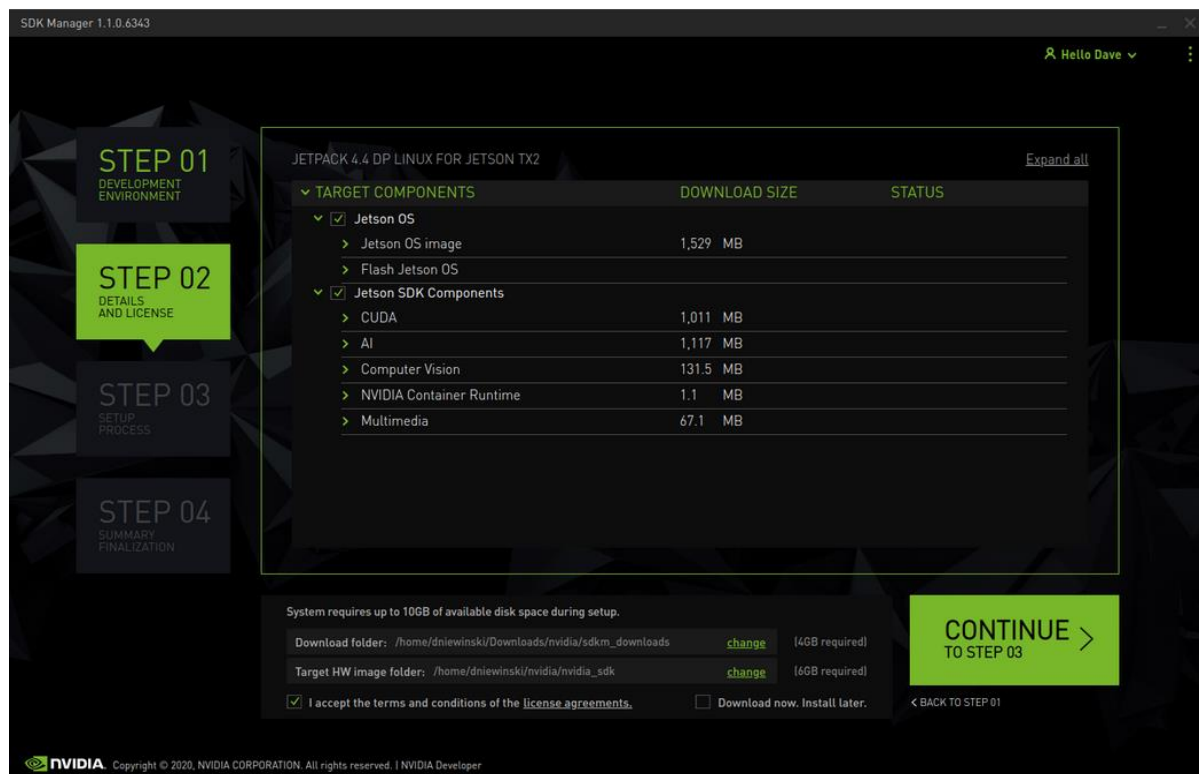
- (VMware Workstation Ubuntu 18.04).
2. Connect the Jetson TX2 to the host system via a USB cable.
  3. Boot the Jetson TX2 into recovery mode by following the instructions from NVIDIA.
  4. Follow the below pics for detailed steps to install the manager.



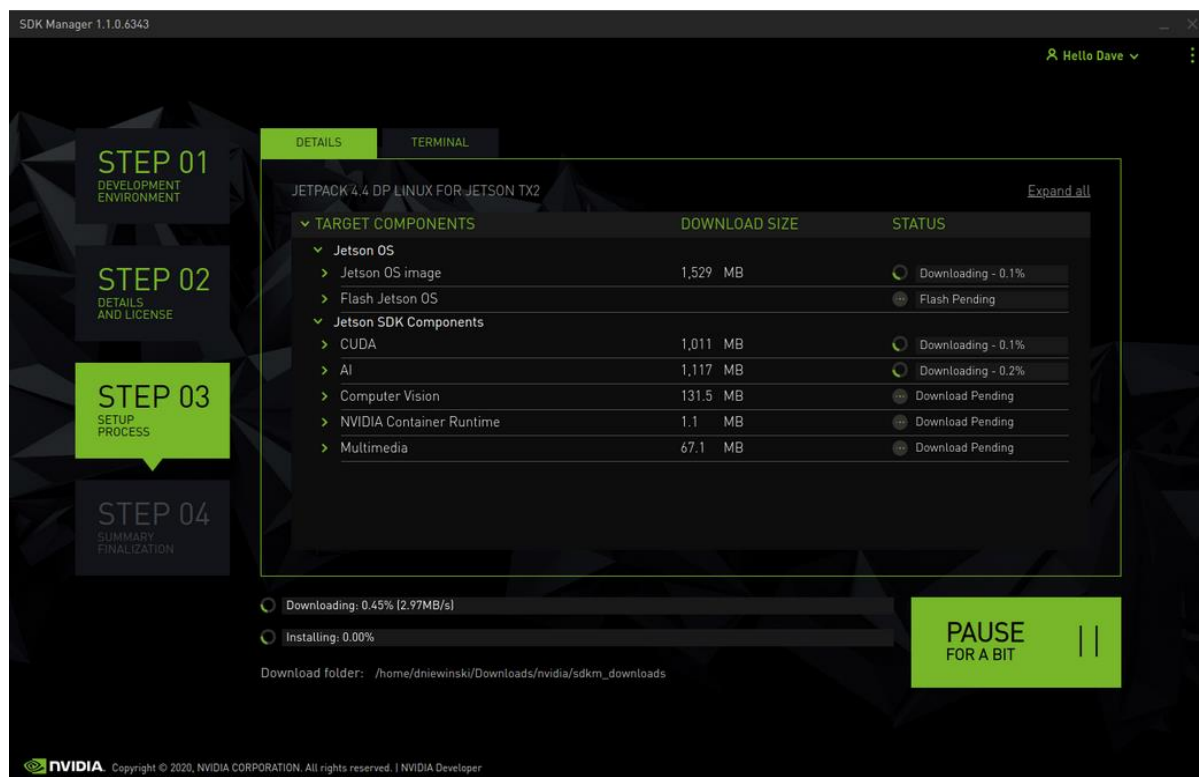
**FIGURE 5 LOGIN USING NVIDIA CREDENTIALS**



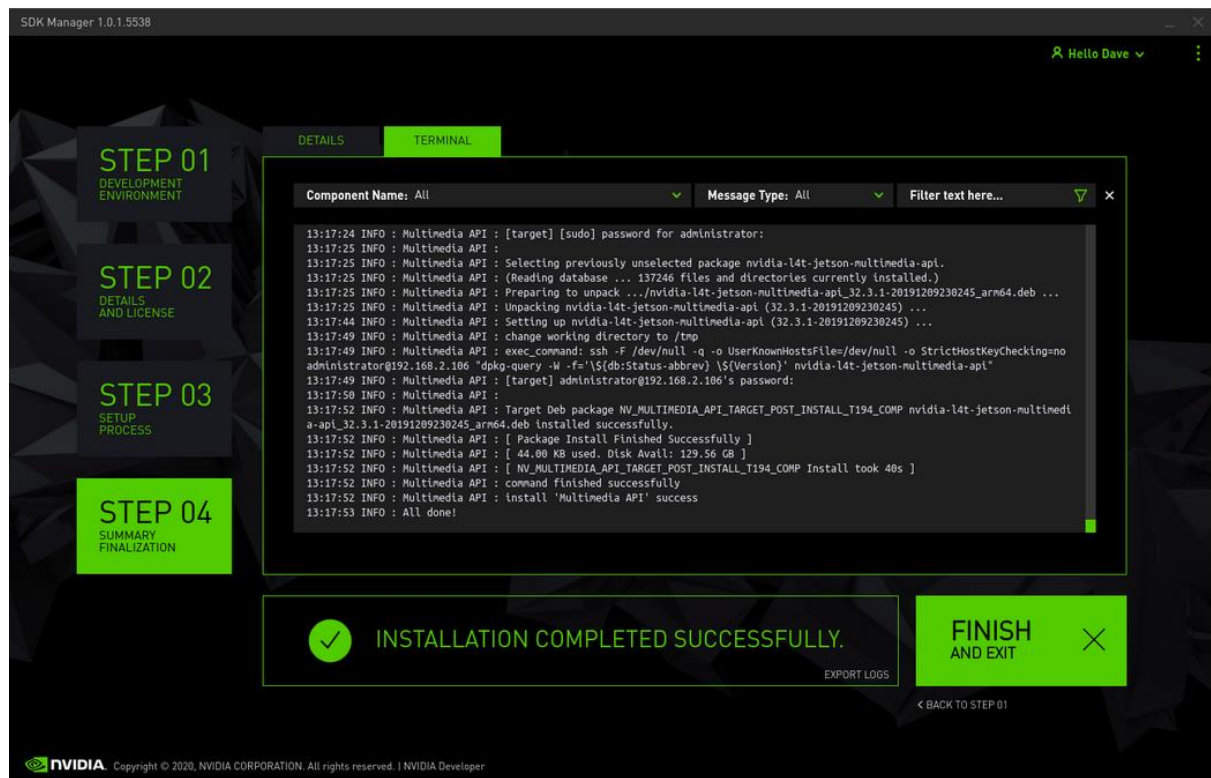
**FIGURE 6 TARGET HARDWARE SELECTION**



**FIGURE 7 INSTALLING TARGET COMPONENTS**



**FIGURE 8 SETUP PROCESS**



**FIGURE 9 SUMMARY FINALIZATION**

- Once the installation is complete, disconnect the Jetson TX2 from the host system and power it on.

For detailed information and steps, the installation steps can be referred to from Jetson Hacks YouTube video titled *Nvidia Jetson SDK Manager – JetPack 4.2* and playlist *Nvidia Jetson TX2 development Kit*.

## 2.4.2 Zed SDK Manager Installation Guide

The Zed SDK serves as an essential software development kit (SDK) meticulously designed for the ZED stereo camera; an innovative depth-sensing camera developed by Stereo Labs. This camera harnesses the power of stereo vision technology, enabling the capture of intricate 3D images and depth maps. Facilitating the creation of diverse applications that leverage the capabilities of the ZED camera, the Zed SDK encompasses a comprehensive collection of tools and libraries. These applications span a spectrum, including object detection, tracking, 3D mapping, and augmented reality.

Furthermore, the Zed Python API emerges as a valuable addition, presenting a Python wrapper tailored to the Zed SDK. This API streamlines developer access to the ZED camera and its functionalities directly from Python. By utilizing the Zed Python API, developers are granted a seamless and user-friendly interface to undertake tasks such as image capture, depth map processing, and camera parameter manipulation. This integration of technology and ease of access enhances the potential for innovation and creativity in leveraging the capabilities of the ZED stereo camera.

Here are the general steps for installing ZED SDK on NVIDIA TX2:

- Download ZED SDK and Install the ZED SDK on NVIDIA TX2



- Install ZED Python API
- If the installation in step 2 does not detect pyzed.sl then, the ZED Python API should be built from source.

#### *How it works?*

- Video capture for each camera is done in a separate thread for optional performance. You can specify the number of ZED used by changing the NUM\_CAMERAS parameter.
- Each camera has its own timestamp (uncomment a line to display it). These time stamps can be used for device synchronization,
- OpenCV is used to display images and depth maps. To stop the application, simply press 'q'.

#### *Limitations:*

- USB Bandwidth: The ZED in 1080p30 mode generates around 250MB/s of image data. USB 3.0 maximum bandwidth is around 400MB/s, so the number of cameras, resolutions and framerates you can use on a single machine will be limited by the USB 3.0 controller on the motherboard. When bandwidth limit is exceeded, corrupted frames (green or purple frames, tearing) can appear.
- Using single USB 3.0 controller, here are configurations that we tested:
  - 2 ZEDs in HD1080 @ 15fps and HD720 @ 30fps
  - 3 ZEDs in HD720 @ 15fps
  - 4 ZEDs in VGA @ 30fps
- To use multiple ZED at full speed on a single computer, we recommend adding USB3.0 PCIe expansion cards.
- You can also use multiple GPUs to load-balance computations (use param.device to select a GPU for a ZED) and improve performance.

### **2.4.3 OpenCV CUDA, cuDNN module Installation Guide**

- The OpenCV-DNN module only supports inference so although you will get much faster inference out of it, the training however will be the same as was for the OpenCV we set up without CUDA backend support.
- Install CUDA & cuDNN  
The first step in configuring OpenCV's "dnn" module for NVIDIA GPU inference is to install the proper dependencies:
  - sudo apt-get update
  - sudo apt-get upgrade
  - sudo apt-get install build-essential cmake unzip pkg-config
  - sudo apt-get install libjpeg-dev libpng-dev libtiff-dev
  - sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev
  - sudo apt-get install libv4l-dev libxvidcore-dev libx264-dev
  - sudo apt-get install libgtk-3-dev
  - sudo apt-get install libatlas-base-dev gfortran
  - sudo apt-get install python3-dev
- Download and unzip opencv and opencv\_contrib  
There is no "pip-installable" version of OpenCV that comes with NVIDIA GPU support – instead, we'll need to compile OpenCV from scratch with the proper NVIDIA GPU configurations set. The first step in doing so is to download the source code for OpenCV v4.2:



- o `cd ~`
- o `wget -O opencv-4.5.5.zip https://github.com/opencv/opencv/archive/4.5.5.zip`
- o `unzip -q opencv-4.5.5.zip`
- o `mv opencv-4.5.5 opencv`
- o `rm -f opencv-4.5.5.zip`

Next, download the `opencv_contrib` archive, unzip it and rename the folder to *opencv\_contrib*.

**Note:** Make sure to download the exact same version of the `opencv_contrib` as the `opencv` archive you downloaded before that.

- o `wget -O opencv_contrib-4.5.5.zip https://github.com/opencv/opencv_contrib/archive/4.5.5.zip`
- o `unzip -q opencv_contrib-4.5.5.zip`
- o `mv opencv_contrib-4.5.5 opencv_contrib`
- o `rm -f opencv_contrib-4.5.5.zip`

- Setup the python environment

Install `virtualenv` and `virtualenvwrapper` if you don't have them on your system already.

- o `Sudo pip install virtual virtualenvwrapper`

Open the `bashrc` script file using `nano`.

- o `nano ~/.bashrc`

Enter the following export paths to set the path for the `virtualenvwrapper` at the bottom and save the file.

- o `export WORKON_HOME=$HOME/.virtualenvs`
- o `export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3`
- o `source /usr/local/bin/virtualenvwrapper.sh`

Run `source` to assert the changes in the system for the `bashrc` script.

- o `source ~/.bashrc`

Create a virtual environment in a terminal. We will be installing the `OpenCV-DNN-CUDA` module in a virtual environment and not our base environment.

- o `mkvirtualenv opencv_dnn_cuda -p python3`

Install `NumPy` package.

- o `pip install numpy`

Activate the virtual environment by entering the following command:

- o `workon opencv_dnn_cuda`

- Configure `OpenCV` using the `cmake` command to install some libraries

Navigate to the `opencv` folder we created in step 3. Create another folder inside it named `build` and navigate inside it.

- o `cd ~/opencv`
- o `mkdir build`
- o `cd build`

Run the `CMake` command to configure settings for the `OpenCV` you want to build from source with your custom libraries and functions selected.

Change the `CUDA_ARCH_BIN` parameter in the command below to your GPU's compute capability. Also, change the virtual environment name in the `PYTHON_EXECUTABLE` parameter if you are using a different name else leave it as it is.

```
cmake -D CMAKE_BUILD_TYPE=RELEASE \
-D CMAKE_INSTALL_PREFIX=/usr/local \
-D INSTALL_PYTHON_EXAMPLES=ON \
-D INSTALL_C_EXAMPLES=OFF \
-D OPENCV_ENABLE_NONFREE=ON \
-D WITH_CUDA=ON \
-D WITH_CUDNN=ON \
-D OPENCV_DNN_CUDA=ON \
-D ENABLE_FAST_MATH=1 \
-D CUDA_FAST_MATH=1 \
-D CUDA_ARCH_BIN=6.1 \
-D WITH_CUBLAS=1 \
-D OPENCV_EXTRA_MODULES_PATH=~/.opencv_contrib/modules \
-D HAVE_opencv_python3=ON \
-D PYTHON_EXECUTABLE=~/.virtualenvs/opencv_dnn_cuda/bin/python \
-D BUILD_EXAMPLES=ON ..
```

In the above CMake command, we are compiling OpenCV with both CUDA and cuDNN support enabled (WITH\_CUDA and WITH\_CUDNN, respectively). We are also enabling the OPENCV\_DNN\_CUDA parameter to build the DNN module with CUDA backend support. We are also setting ENABLE\_FAST\_MATH, CUDA\_FAST\_MATH, and WITH\_CUBLAS for optimization purposes.

- Lastly, build the OpenCV-DNN module from source with CUDA backend support for Jetson TX2 Nvidia GPU.

Run the make command to build the OpenCV with the above-configured settings.

```
o make -j4
```

This will build the OpenCV-DNN module from source with the CUDA backend. This process can take up to an hour or so.

Next, install the OpenCV on your system once it has finished building the OpenCV in the make command above. Run the following commands:

```
o sudo make install
o sudo ldconfig
```

Next, the final step is to create a symbolic link between the newly installed OpenCV library to our python virtual environment. For that, you need the path where the OpenCV bindings were installed. You can determine that path via the install path configuration in the CMake command step output.

```
o ls -l /usr/local/lib/python3.6/site-packages/cv2/python-3.6
o total 10104
o -rw-r--r-- 1 root staff 10345488 Mar 23 07:06 cv2.cpython-36m-x86_64-linux-gnu.so
```

We will see an output like above confirming the cv2 binding is here. I have Python 3.6 and a 64-bit architecture which explains the file name.

Next, navigate to the virtual environment “site-packages” folder, and finally create a link for the cv2 bindings we found above using the ln -s command. This creates a cv2.so sym-link inside the current working directory which is the site-packages folder in our virtual environment.

- o cd ~/.virtualenvs/opencv\_dnn\_cuda/lib/python3.6/site-packages/
- o ln -s /usr/local/lib/python3.6/site-packages/cv2/python-3.6/cv2.cpython-36m-x86\_64-linux-gnu.so cv2.so

Make sure to double-check everything as this command silently fails even if the path is incorrect meaning, you will get no warning or error message and the output will not work. So be mindful of this step and check if the cv2.so file is created or not.

That's it! We have successfully built the OpenCV-DNN module with CUDA backend support.

***Check if cv2 is installed correctly.***

```
workon opencv_dnn_cuda
python
```

```
Python 3.6.9 (default, Jul 02 2023, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
>>> cv2.__version__
'4.5.5'
```

***Check if the OpenCV is installed correctly with CUDA backend support using the test\_DNN\_CV.py script below.***

```
import numpy as np
import cv2 as cv
import time

npTmp = np.random.random((1024, 1024)).astype(np.float32)

npMat1 = np.stack([npTmp,npTmp],axis=2)
npMat2 = npMat1

cuMat1 = cv.cuda_GpuMat()
cuMat2 = cv.cuda_GpuMat()
cuMat1.upload(npMat1)
cuMat2.upload(npMat2)

start_time = time.time()
cv.cuda.gemm(cuMat1, cuMat2,1,None,0,None,1)

print("CUDA using GPU --- %s seconds ---" % (time.time() - start_time))

start_time = time.time()
cv.gemm(npMat1,npMat2,1,None,0,None,1)

print("CPU --- %s seconds ---" % (time.time() - start_time))
```

## **2.4.4 TensforFlow Installation Guide**

TensorFlow™ is an open-source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the

multidimensional data arrays (tensors) that flow between them. This flexible architecture lets you deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code.

### ***Prerequisites and Dependencies:***

Before you install TensorFlow for Jetson, ensure that:

- Jetpack is installed on your device.
- Make sure the system packages are installed which are required by TensorFlow
  - o `sudo apt-get update`
  - o `sudo apt-get install libhdf5-serial-dev hdf5-tools libhdf5-dev zlib1g-dev zip libjpeg8-dev liblapack-dev libblas-dev gfortran`
- Install and upgrade pip3
  - o `sudo apt-get install python3-pip`
  - o `sudo python3 -m pip install --upgrade pip`
  - o `sudo pip3 install -U testresources setuptools==65.5.0`
- Install the Python package dependencies
  - o `sudo pip3 install -U numpy==1.22 future==0.18.2 mock==3.0.5 keras_preprocessing==1.1.2 keras_applications==1.0.8 gast==0.4.0 protobuf pybind11 cython pkgconfig packaging h5py==3.6.0`

### ***Installing TensorFlow:***

Install TensorFlow using pip3. This command will install the version of TensorFlow compatible with Jetpack 4.6

- o `sudo pip3 install --extra-index-url https://developer.download.nvidia.com/compute/redist/jp/v46 tensorflow==1.15.5+nv23.04`

Next, activate the virtual environment

- o `source <chosen_venv_name>/bin/activate`

For more information and detailed explanation, the nvidia docs hub for tensorflow can be referred.

## 3 Implementation

### 3.1 Object Detection

Object detection uses a neural network architecture to accurately identify and localize objects of interest within an image or video frame. The goal of object detection is to not only determine what objects are present in a scene but also precisely locate them by drawing bounding boxes around them. Deep learning models, particularly Convolutional Neural Networks (CNNs), have revolutionized object detection by automating the process of feature extraction and classification.

The steps involved in object detection are:

1. *Data preparation:* Each image in the dataset is labeled with bounding boxes around objects and their corresponding class labels.
2. *Neural Network Architecture:* Deep learning models designed for object detection often use specialized architectures. Some popular architectures include:
  - ✓ *YOLO (You Only Look Once):* YOLO divides the image into a grid and predicts bounding boxes and class probabilities within each grid cell.
  - ✓ *Faster R-CNN (Region Convolutional Neural Network):* Faster R-CNN uses a Region Proposal Network (RPN) to generate potential bounding box proposals, which are then refined by the network.
  - ✓ *SSD (Single Shot MultiBox Detector):* SSD generates multiple bounding box predictions of varying sizes and aspect ratios for each location in the image.
3. *Bounding Box and Class Prediction:* The neural network is trained on the labeled dataset using optimization techniques where it learns to identify features that distinguish different object classes and predicts bounding box coordinates.
4. *Non-Maximum Suppression (NMS):* After predictions, NMS is applied to remove redundant and overlapping bounding boxes, retaining only the most confident ones.
5. *Visualization:* The final output includes annotated images or video frames with bounding boxes drawn around detected objects and labeled with their corresponding classes.

Object detection has a wide range of practical applications, including autonomous driving for detecting pedestrians and vehicles, surveillance systems for monitoring security, retail for inventory management, medical imaging for identifying anomalies, and more.

#### 3.1.1 Implementation at glance

- Load the configuration of YOLO models for object and face mask detection.
- Load the weight files of pre-trained YOLO models for object and face mask detection.
- Store all the classes names in the NAMES file in a list
- Read a model stored in Darknet model files and create a model from Deep learning network
- Create a ZED Camera object and retrieve the number of ZED Cameras that are connected. Retrieve the images (frame by frame of live feed) using ZED APIs or an OpenCV object to read frames using OpenCV APIs. Obtain the Serial ID of each ZED Camera, open it and set the runtime parameters.
- Create the Initialize parameters object and set the configuration parameters of the model

- Retrieve the class IDs, confidence scores, and bounding box coordinates (x, y, height, width) for each frame to facilitate the drawing of bounding boxes.
- Apply non-maximum suppression to eliminate overlapping and redundant bounding boxes.
- Iterate through the indices obtained after non-maximum suppression, draw the bounding boxes, extract class names based on class IDs, and display class names in the output frame.
- Compare the ZED Camera ID and implement object detection and distance estimation on one ZED camera and face mask detection on another ZED camera.

## 3.1.2 Implementation in Detail

### 3.1.2.1 Import Libraries

```
import time
import threading
import pyzed.sl as sl
import cv2
import numpy as np
import math
import signal
```

**FIGURE 10 IMPORT PYTHON LIBRARIES**

*Pyzed.sl* library is used to create a zed camera object and retrieve video frame by frame.

*NumPy* stands for Numerical Python and is the core library for numeric and scientific computing. We use the NumPy library to generate random numbers.

The Deep Neural Network module of cv2 (OpenCV) is used to run YOLO model. Cv2 module is also used to take images, pre-stored video, webcam video as an input (alternate to using ZED Camera APIs).

The *Threading module* is used to implement the thread concept to improve the speed of the algorithm by reading the next frame in parallel to the processing of the previous frame.

*Time module* is used to calculate Frames Per Second. Math module is used to calculate the distance between object and the camera.

*Signal module* is used to retrieve images, measure depth, and get the current timestamp.

### 3.1.2.2 Load Pre-Trained Algorithm

The aim of the project is to implement object detection and face mask detection using YOLO architecture.

*YOLO Versions (v3/v4 and Tiny v3/Tiny v4):* YOLO is a popular real-time object detection system known for its speed and accuracy. YOLO comes in different versions, including v3, v4, and smaller versions like Tinyv3 and Tinyv4.

*Weight File:* The weight file contains the learned parameters of the neural network after training on a specific dataset. These weights are crucial for making accurate predictions. In our project, we have used separate weight files for the regular YOLO model (for object detection) and the modified YOLO model (for face mask detection).

*Configuration File:* The configuration file holds the architecture and hyperparameters of the YOLO model. It defines the structure of the neural network, including the number of layers, layer types (like convolutional,

pooling, etc.), filter sizes, strides, etc. The configuration file is essential for recreating the YOLO architecture for both object detection and face mask detection.

**NAMES File:** The NAMES file contains a list of class names that the YOLO model is trained to detect. Each class corresponds to a specific object or category.

```
#Load YOLO Configurations and weight files for object and face mask detection
modelConfiguration = '/home/nvidia/Project_codes/yolov3/cfg/yolov3-tiny.cfg'
modelWeights = '/home/nvidia/Project_codes/yolov3/weights/yolov3-tiny.weights'

faceMaskConfiguration = '/home/nvidia/Project_codes/yolov3/cfg/yolov3-tiny-face_mask.cfg'
faceMaskModelWeights = '/home/nvidia/Project_codes/yolov3/weights/yolov3-tiny-face_mask.weights'
```

**FIGURE 11 LOAD YOLOV3-TINY ALGORITHM FROM CONFIGURATION AND WEIGHTS FILE**

```
#Store class name in a list
LABELS = []
LABELS_MASK = []

with open('/home/nvidia/Project_codes/yolov3/data/coco.names', 'r') as f:
    LABELS = [line.strip() for line in f.readlines()]
with open('/home/nvidia/Project_codes/yolov3/data/mask_classes.names', 'r') as f:
    LABELS_MASK = [line.strip() for line in f.readlines()]
```

**FIGURE 12 STORE CLASS NAMES IN A LIST**

### 3.1.2.3 Create Network and Detection Model:

Using OpenCV's Deep Neural Network API, a network is constructed by providing the YOLO configurations and weights files as parameters. Subsequently, a model is established by utilizing the pre-existing network configuration.

```
#Read a network model stored in Darknet model files
#Parameters: Configuration file, Weights file
#Return: Net
net = cv2.dnn.readNetFromDarknet(modelConfiguration,modelWeights)
facenet = cv2.dnn.readNetFromDarknet(faceMaskConfiguration, faceMaskModelWeights)

#Set Backend and Target
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA_FP16)

#Create model from Deep learning network
model = cv2.dnn_DetectionModel(net)
modelFace = cv2.dnn_DetectionModel(facenet)
```

**FIGURE 13 NETWORK AND DETECTION MODEL**

### 3.1.2.4 Multi-Camera Integration and ZED SDK Initialization:

The camera object is created which retrieves the list of connected ZED Cameras using `sl.camera.get_device_list()` function. The initialization parameters, encapsulated in the `init_params` object, are set to define the camera's operating parameters. These include camera resolution, depth mode, coordinate units, and camera frame rate.

```

#List and open cameras
name_list = []
last_ts_list = []
cameras = sl.Camera.get_device_list()
index = 0

# Create a InitParameters object and set configuration parameters
init_params = sl.InitParameters()
init_params.camera_resolution = sl.RESOLUTION.HD720 # Use HD720 video mode
init_params.depth_mode = sl.DEPTH_MODE.PERFORMANCE
init_params.coordinate_units = sl.UNIT.METER
init_params.camera_fps = 30 # Set fps at 30

```

**FIGURE 14 RETRIEVE THE LIST OF CONNECTED ZED CAMERA AND INITIALIZE CAMERA PARAMETERS**

For each camera, the initialization parameter is set based on the serial number obtained from the camera list. A variety of data structures and variables are prepared to manage the camera data, including **zed\_list** for camera objects, **left\_list** for left images and **depth\_list** for depth images. The **name\_list** stores the serial number of each ZED camera. The ZED camera is then opened using the specified initialization parameters and serial number. After the camera is opened, runtime parameters are defined using `sl.RuntimeParameters()`. These runtime parameters are used to configure the behavior of the ZED SDK during camera operation.

```

for cam in cameras:
    init_params.set_from_serial_number(cam.serial_number)
    name_list.append("ZED {}".format(cam.serial_number))
    print("Opening {}".format(name_list[index]))
    zed_list.append(sl.Camera())
    left_list.append(sl.Mat())
    depth_list.append(sl.Mat())
    point_cloud_list.append(sl.Mat())
    timestamp_list.append(0)
    last_ts_list.append(0)
    status = zed_list[index].open(init_params)
    if status != sl.ERROR_CODE.SUCCESS:
        print(repr(status))
        zed_list[index].close()
        index = index + 1

#Start camera threads
for index in range(0, len(zed_list)):
    if zed_list[index].is_opened():
        print("Opening ZED Camera...")
        thread_list.append(threading.Thread(target=grab_run, args=(index,)))
        thread_list[index].start

#Set runtime parameters after opening the camera
runtime_parameters = sl.RuntimeParameters()

```

**FIGURE 15 OPENING THE CAMERA BASED ON SERIAL NUMBER AND SETTING THE RUNTIME PARAMETERS**

### 3.1.2.5 Detection of Objects using YOLO Model:

The function `detectObject_YOLO` streamlines object detection using a pre-configured YOLO model. It accepts the model and an image created in section 3.1.2.3 and 3.1.2.4 as inputs. The function sets up model parameters, converts the image format, and conducts object detection. It then returns class labels, confidence scores, and bounding box coordinates by using the detect API of DNN (Deep Neural Network) module of OpenCV.

```

def detectObject_YOLO(modelName, img):
    modelName.setInputParams(size=(320, 320), scale=1/255, swapRB=True)
    image_test = cv2.cvtColor(img, cv2.COLOR_RGBA2RGB)
    image = image_test.copy()

    # print('image', image.shape)
    classes, confidences, boxes = modelName.detect(image, confThreshold, nmsThreshold)

    return classes, confidences, boxes

```

**FIGURE 16 OBJECT DETECTION FUNCTION USING YOLO MODEL**



The obtained data is then used to draw the bounding boxes using cv2.rectangle API and add the label to the image using cv2.putText API. For each detected object, the distance is computed using point cloud data and displayed using cv2.putText API.

```

if(name_list[index] == 'ZED 27860387'):
    if zed_list[index].is_opened():
        err = zed_list[index].grab(runtime_parameters)
        if (err == sl.ERROR_CODE.SUCCESS):
            # Retrieve the left image, depth image in the half-resolution
            zed_list[index].retrieve_image(left_list[index], sl.VIEW.LEFT)

            # Retrieve the RGBA point cloud in half resolution
            zed_list[index].retrieve_measure(point_cloud_list[index], sl.MEASURE.XYZRGBA)

            # To recover data from sl.Mat to use it with opencv, use the get_data() method
            # It returns a numpy array that can be used as a matrix with opencv
            image_ocv = left_list[index].get_data()

            classes,confidences,boxes = detectObject_YOLO(model,image_ocv)

            for cl,score,(left,top,width,height) in zip(classes,confidences,boxes):
                start_point = (int(left),int(top))
                end_point = (int(left+width),int(top+height))

                x = int(left + width/2)
                y = int(top + height/2)

                color = COLORS[0]

                img =cv2.rectangle(image_ocv,start_point,end_point,color,3)
                text = f'!LABELS[{cl}]'
                cv2.putText(img, text, (int(left), int(top+7)), cv2.FONT_ITALIC, 1, COLORS[0], 2)

                x = round(x)
                y = round(y)

                err, point_cloud_value = point_cloud_list[index].get_value(round(x), round(y))
                distance = math.sqrt(point_cloud_value[0] * point_cloud_value[0] + point_cloud_value[1] * point_cloud_value[1] + point_cloud_value[2] * point_cloud_value[2])
                distance_text = "Distance: {:.2f}m".format(distance)
                cv2.putText(img, distance_text, (int(left), int(top + 25)), cv2.FONT_HERSHEY_COMPLEX, 1, COLORS[1], 2)
                cv2.imshow(name_list[index],image_ocv)

            frame_count = frame_count + 1

```

**FIGURE 17 OBJECT DETECTION USING YOLO MODEL**

## 3.2 Distance Estimation

### 3.2.1 Introduction

Distance estimation serves as a fundamental aspect of the ZED camera's functionality, relying on its advanced depth sensing capabilities.

### 3.2.2 Depth sensing overview

Depth sensing is the cornerstone of the ZED camera's ability to gauge distances within a captured scene. This technique capitalizes on the concept of stereo vision, emulating the human visual system's depth perception mechanism. Human eyes are horizontally separated by about 65 mm on average. Thus, each eye has a slightly different view of the world around. By comparing these two views, our brain can infer not only depth but also 3D motion in space.

The ZED camera leverages a binocular configuration, comprising two synchronized cameras, to capture images from slightly disparate vantage points. By discerning disparities between corresponding points in these images, the camera extrapolates depth information. Likewise, StereoLabs stereo cameras have two eyes separated by 6 to 12 cm, which allow capturing high-resolution 3D video of the scene and estimate depth and motion by comparing the displacement of pixels between the left and right images.

### 3.2.2.1 Depth Perception

Depth perception, the ability to discern distances between objects and perceive the world in three dimensions, is a fundamental aspect of human and computer vision. Traditionally, depth sensors have been confined to short-range capabilities and indoor environments, predominantly finding utility in applications like gesture control and body tracking. However, the emergence of stereo vision technology has led to the development of the ZED camera, a pioneering universal depth sensor that revolutionizes the field.

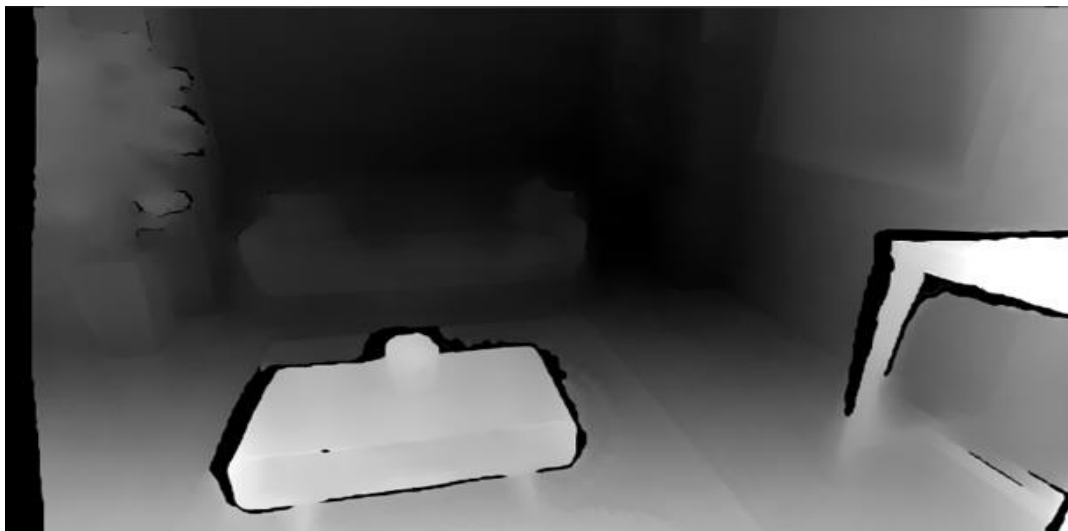
#### Key Advancements of the ZED Camera in Depth Perception:

- Depth can be captured at longer ranges, up to 20m.
- Frame rate of depth capture can be as high as 100 FPS.
- The 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

### 3.2.2.2 Depth Map

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.

Depth maps cannot be displayed directly as they are encoded on 32 bits. To display the depth map, a monochrome (grayscale) 8-bit representation is necessary with values between [0, 255], where 255 represents the closest possible depth value and 0 is the most distant possible depth value.



**FIGURE 18 DEPTH MAP**

### 3.2.2.3 3D Point Cloud

Another common way of representing depth information is by a 3D point cloud. A point cloud can be seen as a depth map in three dimensions. While a depth map only contains the distance or Z information for each pixel, a point cloud is a collection of 3D points (X, Y, Z) that represent the external surface of the scene and can contain color information.

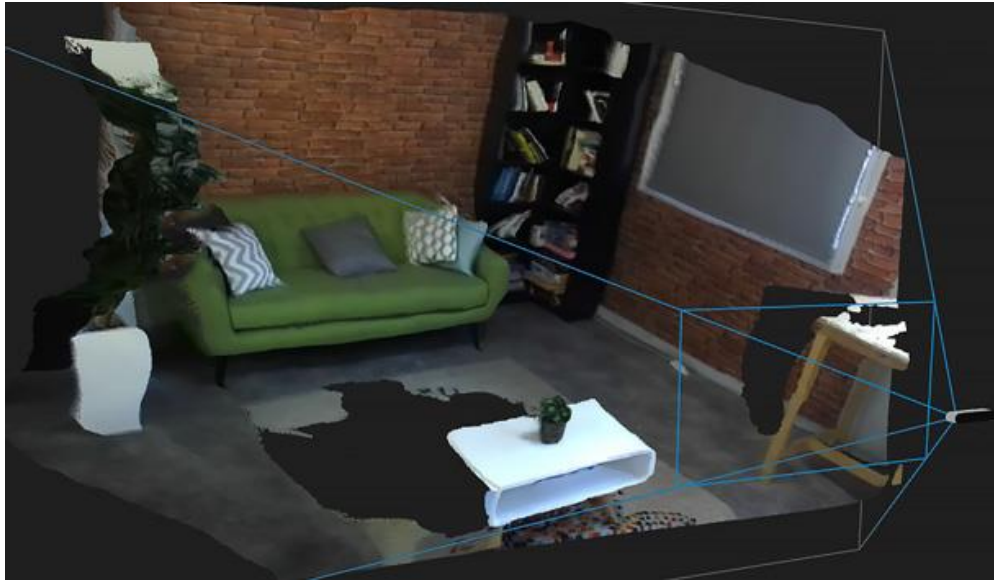


FIGURE 19 3D POINT CLOUD

### 3.2.3 The Operation Principle of Zed Camera

The functionality of the ZED camera revolves around the principle of triangulation, employing a geometric model of non-distorted rectified cameras. This method computes in-depth information by leveraging the relative positions of the camera system's components. This section elucidates the operational mechanics of the ZED camera and its utilization of triangulation for depth estimation.

#### Triangulation for Depth Computation:

Triangulation is the fundamental mechanism utilized by the ZED camera to calculate depth information. This process involves the precise determination of depth by intersecting lines of sight from different viewpoints. In the context of the ZED camera's geometric model, which consists of non-distorted and rectified cameras, the principle of triangulation is deployed to estimate depth.

#### Depth Calculation Equation:

The depth (Z) of each point (L) within the scene is calculated using the following Equation 1:

$$Z = B * f_l / (x_{il} - x_{ir})$$

In this equation:

- **B**: Baseline distance between the cameras.
- **f<sub>l</sub>**: Focal length of the cameras (which is equal for both).
- **x<sub>il</sub>**: Horizontal pixel coordinate of the point in the left image.
- **x<sub>ir</sub>**: Horizontal pixel coordinate of the same point in the right image.

### Depth and Disparity Relationship:

The disparity, represented as  $x_{il} - x_{ir}$ , is a crucial parameter in the depth computation process. Disparity signifies the horizontal shift of a point between the left and right images. It serves as an inverse indicator of depth: greater disparity corresponds to closer objects, and vice versa.

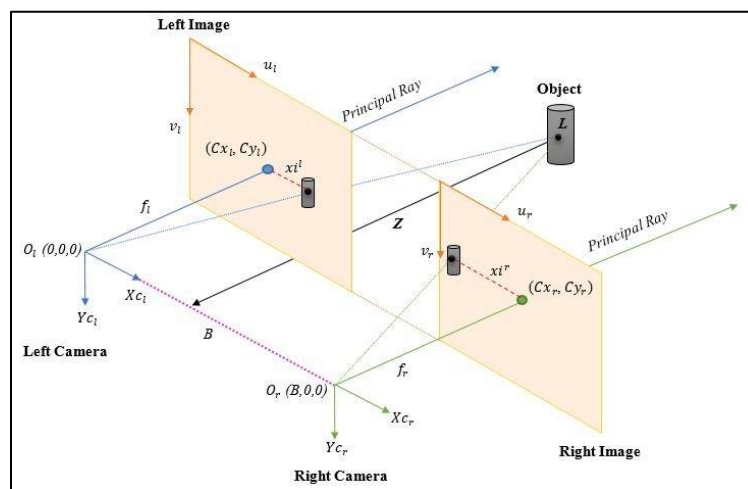


FIGURE 20 THE OPERATION PRINCIPLE OF ZED CAMERA

## 3.2.4 Implementation in Detail

### 3.2.4.1 Capture Data

To extract the depth map of a scene, use 'grab ( )' to grab a new image and 'retrieveMeasure( )' can be used to retrieve a depth map, a confidence map, or normal map, or a point cloud.

A 3D point cloud with (X, Y, Z) coordinates and RGBA colour can be retrieved using 'retrieveMeasure( )'

```

if zed_list[index].is_opened():
    err = zed_list[index].grab(runtime_parameters)
    if (err == sl.ERROR_CODE.SUCCESS):
        # Retrieve the left image, depth image in the half-resolution
        zed_list[index].retrieve_image(left_list[index], sl.VIEW.LEFT)

        # Retrieve the RGBA point cloud in half resolution
        zed_list[index].retrieve_measure(point_cloud_list[index], sl.MEASURE.XYZRGBA)

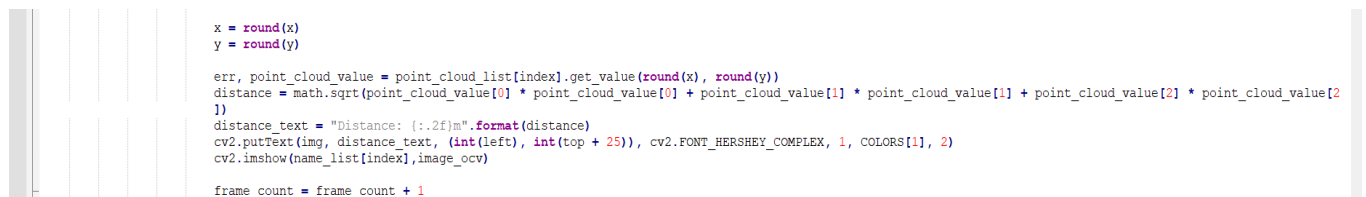
```

FIGURE 21 CAPTURE DATA

### 3.2.4.2 Measure Distance in point cloud

To access a specific pixel value, use 'getValue( )'. Now that we have retrieved the depth map, we may want to get the depth at a specific pixel. In our project, we extract the distance of the point at the centre of each object (x, y).

The Euclidean distance formula is used to calculate the distance of an object relative to the left eye of the camera. We measure the distance of a point in the scene represented by pixel (x, y) (i.e., at the centre of the image).



```
x = round(x)
y = round(y)

err, point_cloud_value = point_cloud_list[index].get_value(round(x), round(y))
distance = math.sqrt(point_cloud_value[0] * point_cloud_value[0] + point_cloud_value[1] * point_cloud_value[1] + point_cloud_value[2] * point_cloud_value[2])
distance_text = "Distance: {:.2f}m".format(distance)
cv2.putText(img, distance_text, (int(left), int(top + 25)), cv2.FONT_HERSHEY_COMPLEX, 1, COLORS[1], 2)
cv2.imshow(name_list[index], image_ocv)

frame_count = frame_count + 1
```

**FIGURE 22 DISTANCE CALCULATION**

## 3.3 Face Mask Detection

In the era of the COVID-19 pandemic, ensuring public safety is of utmost importance. Face mask detection has become a critical application to enforce safety protocols. This report details the implementation of face mask detection using the NVIDIA Jetson TX2 platform and OpenCV's Deep Neural Network (DNN) module. The provided code demonstrates the process of detecting face masks in real-time video streams and presents the results in an informative graphical format.

### 3.3.1 Introduction

Amidst the challenges posed by the global COVID-19 pandemic, technological advancements have emerged as vital tools in ensuring public health. An essential innovation in this realm is real-time face mask detection, facilitated by the powerful NVIDIA Jetson TX2 edge computing platform and OpenCV's DNN module. This collaborative synergy equips us to create an intelligent system that not only identifies face masks but also categorizes instances where masks are worn and detects cases of non-compliance.

The NVIDIA Jetson TX2, renowned for its high-performance computing, serves as the system's backbone, processing visual data in real-time. Combined with OpenCV's DNN module, the solution employs deep learning models for accurate image recognition. Notably, the system is enhanced by Non-Maximum Suppression (NMS), a crucial step that refines results by eliminating redundant detections. This innovative blend of hardware and software exemplifies technology's potential to safeguard public health, offering an intelligent face mask detection system that contributes to a safer environment for all.

One of the key features that elevate our system's accuracy and efficiency is the incorporation of Non-Maximum Suppression (NMS). In the process of face mask detection, NMS plays a crucial role in refining the results by eliminating duplicate or redundant detections. This ensures that each instance of a detected face mask is precisely represented by a single bounding box, enhancing the precision and clarity of the system's outputs.

### 3.3.2 Implementation in detail

Face mask detection stands as a crucial cornerstone of the comprehensive system, with the primary objective of distinguishing individuals who are adhering to mask-wearing protocols from those who are not. This integral process employs a specialized model that has been meticulously trained to accurately identify instances of face masks

### 3.3.3 Methodology

The implementation follows a systematic approach, effectively integrating machine learning and computer vision techniques. The key stages of the process are as follows:

#### 3.3.3.1 Model Initialization

Like the object detection process, the face mask detection process begins by initializing the model using the `detectObject_Yolo`` function. The function is provided with the pre-trained model for face mask detection, `modelFace``, and the current video frame `img``.

```
#Load YOLO Configurations and weight files for object and face mask detection
modelConfiguration = '/home/nvidia/Project_codes/yolov3/cfg/yolov3-tiny.cfg'
modelWeights = '/home/nvidia/Project_codes/yolov3/weights/yolov3-tiny.weights'

faceMaskConfiguration = '/home/nvidia/Project_codes/yolov3/cfg/yolov3-tiny-face_mask.cfg'
faceMaskModelWeights = '/home/nvidia/Project_codes/yolov3/weights/yolov3-tiny-face_mask.weights'
```

**FIGURE 23 LOAD FACE MASK ALGORITHM FROM CONFIGURATION AND WEIGHTS FILE**

#### 3.3.3.2 Detection

The face mask detection model analyzes the input frame and identifies regions where face masks might be present. The model outputs several pieces of information for each detected instance:

*classes\_face*: This array contains the class IDs corresponding to the detected objects. In this context, class IDs associated with face masks and no masks are assigned.

*boxes\_face*: Bounding box coordinates are provided for each detection. These coordinates define the region where the face mask is detected within the frame.

#### 3.3.3.3 Non-Maximum Suppression (NMS)

Like the object detection process, Non-Maximum Suppression (NMS) is performed to eliminate redundant or overlapping detections. This step ensures that each detected face mask instance is represented by a single bounding box.

```
indexes = cv2.dnn.NMSBoxes(boxes, confidences, confThreshold, nmsThreshold)
indexes_face = cv2.dnn.NMSBoxes(boxes_face, confidences_face, confThreshold, nmsThreshold)
```

**FIGURE 24 INDEXING OF THE FACE MASK**

### 3.3.3.4 Classification Statistics with NMS

After filtering out redundant detections, the code proceeds to count the number of people with masks and those without masks. This is achieved by iterating through the remaining detections and analyzing their associated class labels.

```
font = cv2.FONT_HERSHEY_SIMPLEX
person_count = 0
peopleWithMasks = 0
peopleWithNoMasks = 0

indexes = cv2.dnn.NMSBoxes(bboxes, confidences, confThreshold, nmsThreshold)
indexes_face = cv2.dnn.NMSBoxes(bboxes_face, confidences_face, confThreshold, nmsThreshold)

for i in range(len(bboxes)):
    if i in indexes:
        if str(LABELS[classes[i]]) == "person":
            person_count = person_count + 1

for i in range(len(bboxes_face)):
    if i in indexes_face:
        if str(LABELS_MASK[classes_face[i]]) == "Mask":
            peopleWithMasks = peopleWithMasks + 1
        elif str(LABELS_MASK[classes_face[i]]) == "No Mask":
            peopleWithNoMasks = peopleWithNoMasks + 1
```

FIGURE 25 DETECTION AND INCREMENT

### 3.3.3.5 Displaying Results on the Frame

The counts of people wearing masks and those without masks are crucial pieces of information for safety analysis. To communicate this information effectively, the code employs the `cv2.putText` function to overlay the frame with text displaying the counts.

```
img = cv2.rectangle(image_ocv, start_point, end_point, color, 3)
cv2.putText(img, "No of People: " + str(person_count), (10, 25), font, 1, (160, 4, 183), 2)
cv2.putText(img, "People with masks: " + str(peopleWithMasks) + ", People with no masks: " + str(peopleWithNoMasks), (10, 50), font, 1, (160, 4, 183), 2)
# img = cv2.circle(img, (x,y), 5, [0,0,255], 5)
text = f'{LABELS[c1]}'
cv2.putText(img, text, (int(left), int(top-7)), cv2.FONT_ITALIC, 1, COLORS[0], 2)
```

FIGURE 26 DISPLAYING THE FINAL COUNT



### 3.4 Implementation Output

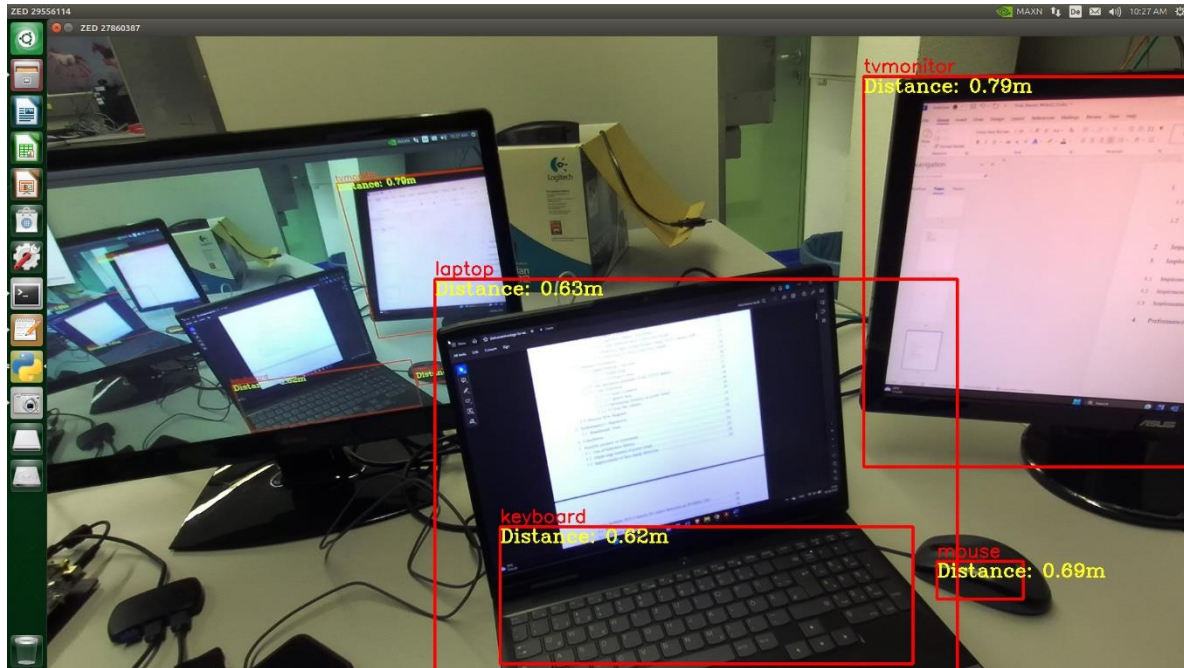


FIGURE 27 DISTANCE ESTIMATION AND OBJECT DETECTION IMAGE

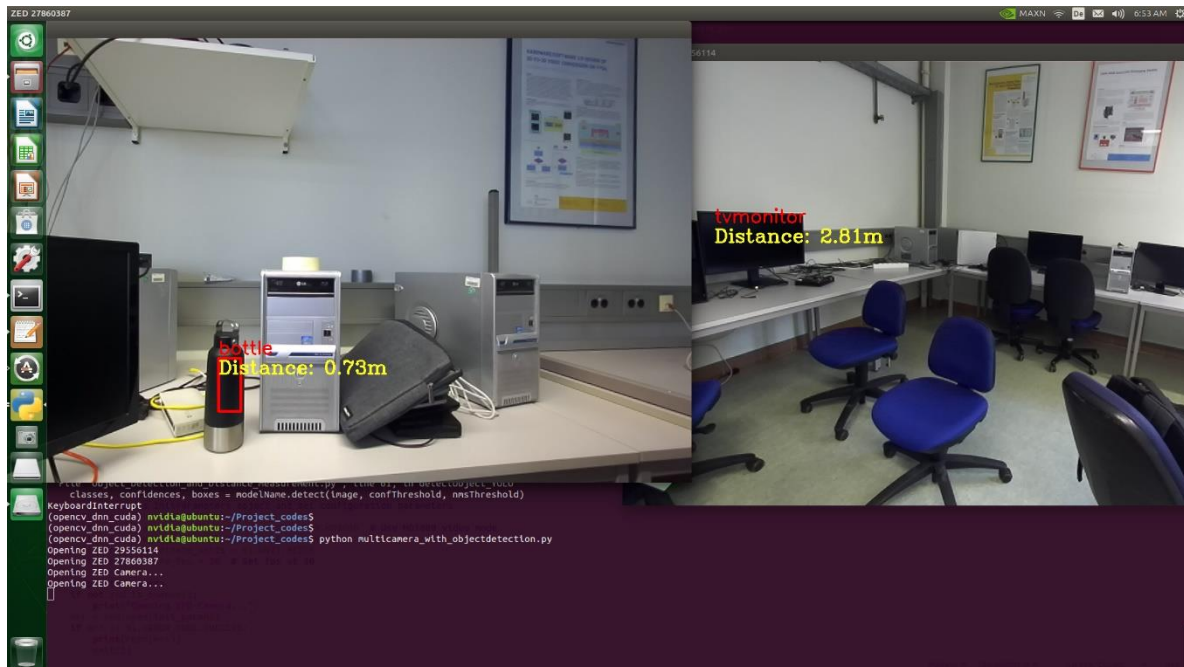


FIGURE 28 DISTANCE ESTIMATION AND OBJECT DETECTION USING MULTIPLE CAMERAS



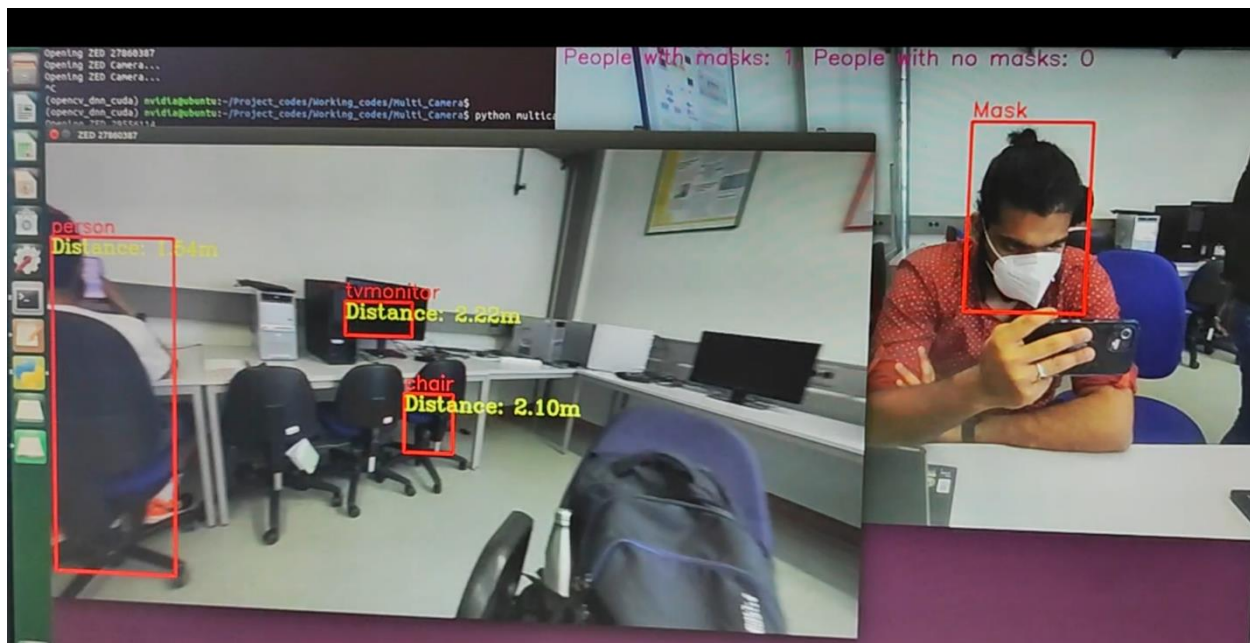


FIGURE 29 MULTI CAMERA OBJECT DETECTION AND FACE MASK DETECTION SEPARATELY

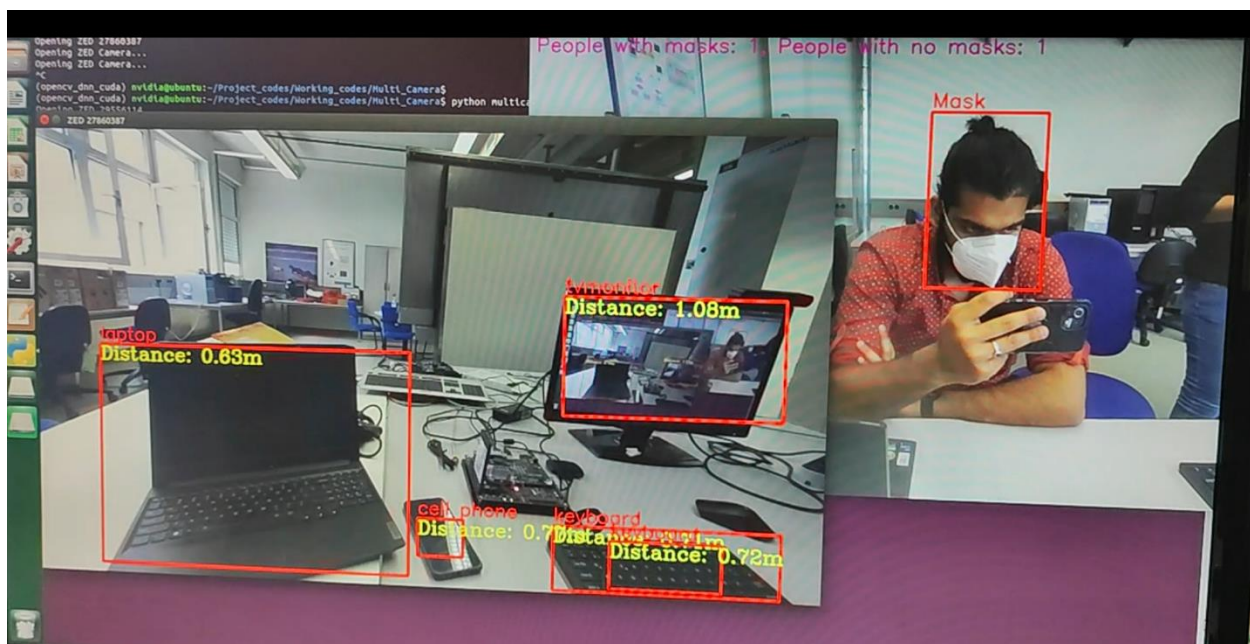
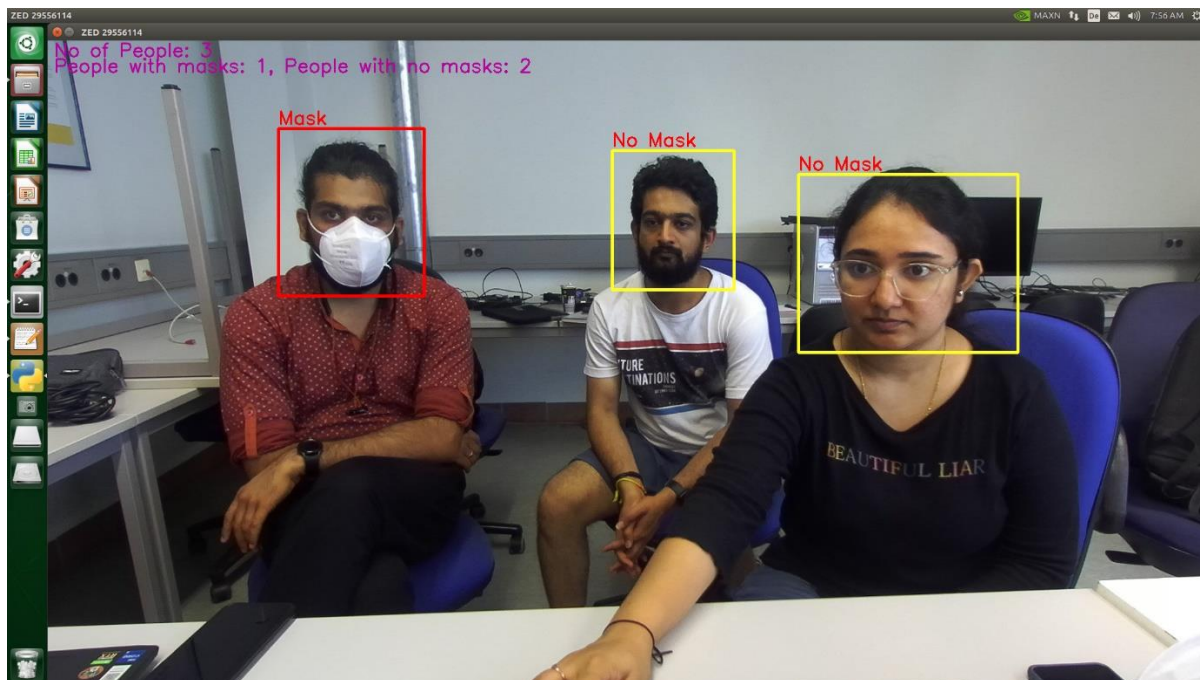
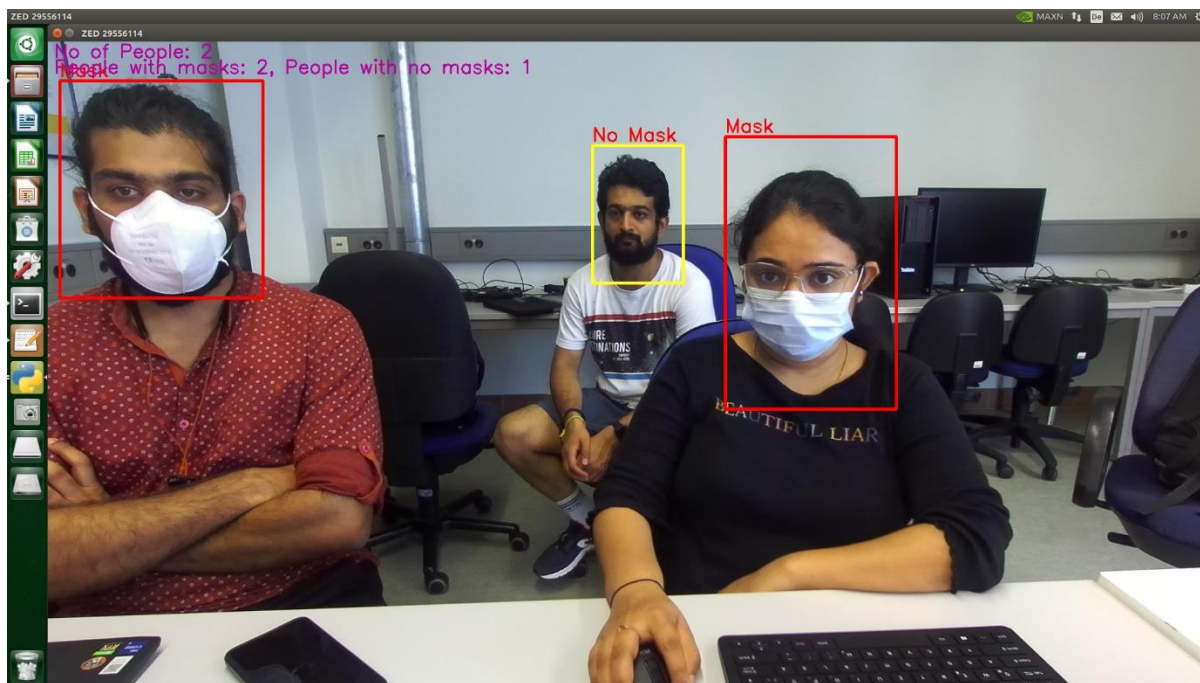


FIGURE 30 MULTI CAMERA OBJECT DETECTION AND FACE MASK DETECTION SEPARATELY



**FIGURE 31 FACE MASK DETECTION IMAGE**



**FIGURE 32 FACE MASK DETECTION IMAGE**

## 4 Performance Comparison

The field of computer vision has witnessed rapid evolution in object detection techniques. The fusion of object classification and localization renders this realm one of the most intricate domains within computer vision. Put simply, object detection aims to ascertain the spatial coordinates of objects within an image, referred to as object localization, alongside classifying each object into relevant categories, termed object classification. This analysis delves into the comparison and examination of Faster R-CNN and YOLO (You Only Look Once), two representative algorithms that have significantly advanced performance through the utilization of Convolutional Neural Networks (CNNs).

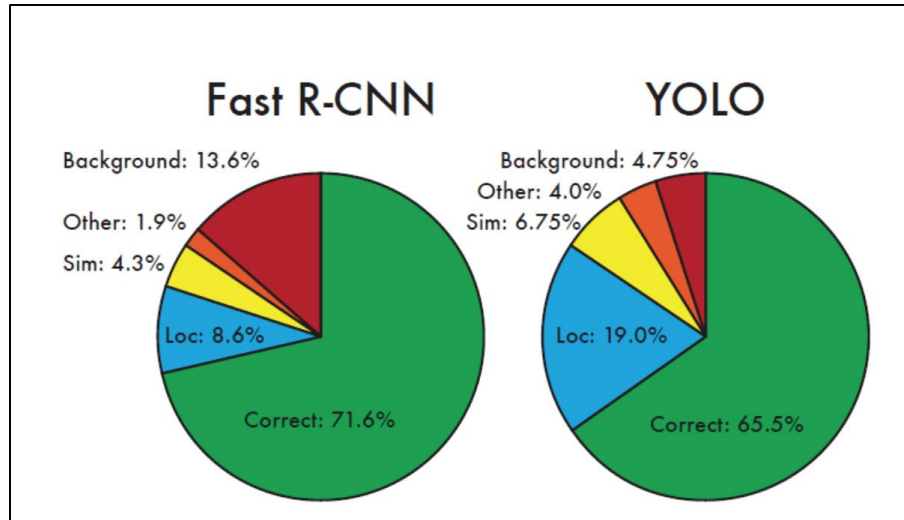
The matrix of performance enhancement remains consistent across both algorithms, yet their frameworks exhibit variances. While R-CNN retains the entire image and subsequently divides candidate regions, YOLO employs a distinct approach by segmenting images prior to CNN processing. YOLO embraces a sliding window technique, eschewing the utilization of local candidate regions. Notably, Faster R-CNN employs 9 anchor boxes for its framework, whereas YOLO v2 employs 5 anchor boxes, with YOLO v1 dispensing with anchor boxes altogether. YOLO v2 introduces multidimensional clustering to enhance precision, a feature absent in Faster R-CNN.

A comparative evaluation in Table below juxtaposes speed and performance among high-speed detectors using the commonly employed PASCAL VOC dataset. In this context, Fast YOLO exhibits the swiftest processing velocity, while YOLO demonstrates superiority in terms of accuracy when compared to alternative versions. A trade-off surfaces between relative mean Average Precision (mAP) and speed of detectors, with Fast YOLO boasting a 52.7% mAP, outperforming other detectors. YOLO sustains real-time performance with a mAP of 66.4%, doubling accuracy.

	Detection Frameworks	Train	mAP ↑	FPS	PS (mAP×FPS)	PS Order
<b>Less Than Real-Time Detectors</b>	1 Fastest DPM [26]	2007	30.4	15	456	11
	2 R-CNN Minus R [27]	2007	53.5	6	321	13
	3 Faster R-CNN ZF[20]	2007+2012	<b>62.1</b>	<b>18</b>	1118	9
	4 YOLO VGG-16[24]	2007+2012	<b>66.4</b>	<b>21</b>	1394	8
	5 Fast R-CNN[22]	2007+2012	70.0	0.5	35	14
	6 Faster R-CNN VGG-16[20]	2007+2012	73.2	7	512	10
	7 Faster R-CNN ResNet[20]	2007+2012	76.4	5	382	12
<b>Real-Time Detectors</b>	1 Fast YOLO [24]	2007+2012	52.7	155	8169	1
	2 YOLO(YOLOv1)[24]	2007+2012	63.4	45	2853	7
	3 YOLOv2 288×288[24]	2007+2012	69.0	91	6279	2
	4 YOLOv2 352×352[24]	2007+2012	73.7	81	5970	3
	5 YOLOv2 416×416[24]	2007+2012	<b>76.8</b>	<b>67</b>	5146	4
	6 YOLOv2 480×480[24]	2007+2012	<b>77.8</b>	<b>59</b>	4590	5
	7 YOLOv2 544×544[24]	2007+2012	<b>78.6</b>	<b>40</b>	3144	6

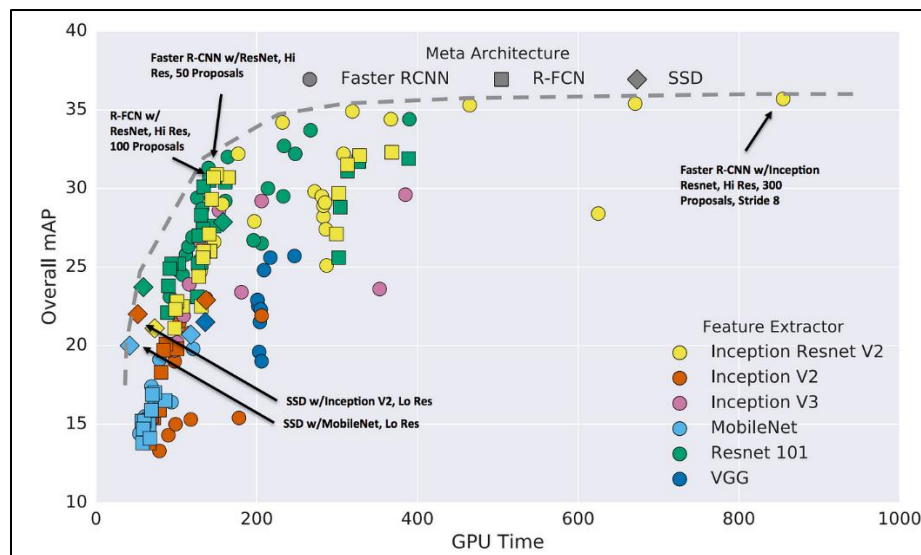
**FIGURE 33 PERFORMANCE COMPARE**





**FIGURE 34 ERROR ANALYSIS: FAST R-CNN VS. YOLO**

As a model, compared to CNN, the construction is simple and the whole image can be directly learned and used in real applications. seems to be suitable for the Access rooms based on other classifiers Unlike the law, YOLO is a hand-me-down that directly responds to detection performance. Train on real functions and improve real-time objects in terms of processing time. body detection is possible.



**FIGURE 35 ACCURACY VS TIME**

In addition, YOLO compared to other detectors Fast and robust detection is possible by more generalizing the object representation. capable Through these several advantages, it is more effective in object detection. It is judged to be highly likely to be neutralized. Algorithm for multiple detectors Besides the structure of the data set, an important issue in machine learning is the scope of the dataset. it can be said Availability of suitable training data depends on application It is an important process in the program, and it determines the appropriateness of this It will have to be studied together with a separate algorithm.

## **5 Future Scope - Fusion of Object detection**

As YOLO is comparatively less accurate compared to two layered architecture such as R-CNN but much faster, yolo model can be used on frames grabbed by one camera and R-CNN can be used on frames grabbed by another camera. The output can be fused to improve the overall accuracy of object detection. (Challenge is to synchronize this process as both run with different speeds)

### **5.1 Introduction - Sensor Fusion**

In the realm of computer vision and object detection, striking a balance between accuracy and speed is a continuous challenge. The two-layered architectures, YOLO (You Only Look Once) and R-CNN (Region-based Convolutional Neural Networks), exemplify this trade-off. YOLO, although fast, might not achieve the same level of accuracy as R-CNN, which is computationally more intensive. However, by leveraging sensor fusion and combining the outputs of these models, it is possible to enhance the overall accuracy of object detection while capitalizing on the strengths of each architecture.

### **5.2 The Challenge of Synchronization**

Central to this approach is the challenge of synchronizing the outputs of YOLO and R-CNN, which inherently operate at different speeds. YOLO's single-pass nature allows it to quickly detect objects but may lead to potential inaccuracies. In contrast, R-CNN's multi-stage approach ensures higher accuracy but comes at the expense of speed. The fundamental challenge is to fuse these outputs effectively, aligning them temporally to ensure that the results correspond to the same point in time

### **5.3 Implementing Sensor Fusion**

Implementing sensor fusion involves combining data from different sources, in this case, the detection outputs of YOLO and R-CNN models, to obtain a more accurate and reliable estimation of object positions within a scene. The following steps outline the process of implementing sensor fusion for enhanced object detection:

#### **5.3.1 Data Acquisition and Preparation:**

##### **Multi-Camera Setup:**

Set up two cameras, each dedicated to running either the YOLO or R-CNN model. These cameras should capture frames simultaneously from different viewpoints.

##### **Preprocessing:**

Preprocess the images captured by the cameras to ensure they are compatible with both the YOLO and R-CNN models. This may involve resizing, normalization, and other required transformations.

### **5.3.2 Object Detection Using YOLO and R-CNN:**

#### **Parallel Processing:**

Run the YOLO model on the frames captured by one camera to quickly detect objects. Simultaneously, process the frames from the other camera using the R-CNN model for more accurate detection.

### **5.3.3 Output Synchronization:**

#### **Timestamp Alignment:**

Ensure that the outputs from both models are timestamped accurately to facilitate synchronization. This may involve using a common time reference for both cameras.

#### **Buffering and Interpolation:**

Implement a buffer to hold the output of the faster model until the corresponding output of the slower model becomes available. Interpolation techniques can be employed to estimate missing outputs if needed.

### **5.3.4 Kalman Filter Integration for Fusion:**

Integrate a Kalman filter into the fusion process to enhance accuracy and smooth object trajectories. The Kalman filter operates in two key steps: prediction and correction.

#### **Prediction Step:**

The filter predicts the next object positions based on the previous state and motion model. It considers velocity and acceleration estimates.

#### **Correction Step:**

The predicted positions are refined using the fused YOLO and R-CNN detections. The filter corrects deviations and inconsistencies, producing more accurate object positions.

### **5.3.5 Fusion and Enhanced Object Detection:**

#### **Combining Detections:**

Combine the bounding box detections obtained from YOLO and R-CNN for the same objects. Each detection should be associated with its confidence score.

#### **Weighted Fusion:**

Assign weights to the detections based on the confidence scores of each model's predictions. This allows the fusion process to account for the relative strengths of YOLO and R-CNN.

### **5.3.6 Accurate Object Tracking and Future Predictions:**

#### **Object Tracking:**

Utilize fused detections for object tracking. By leveraging the synchronized and fused outputs, more accurate trajectories can be estimated, leading to reduced false positives and negatives.

In conclusion, sensor fusion offers a compelling avenue for optimizing object detection accuracy by combining the outputs of YOLO and R-CNN models. The challenge of synchronization is at the core of this approach, demanding innovative strategies to harmonize the outputs of models with different processing speeds. Through accurate fusion and synchronization, the strengths of YOLO and R-CNN can be harnessed to create a unified and enhanced object detection solution.

## 6 References

- [1] <https://github.com/Alro10/YOLO-darknet-on-Jetson-TX2>
- [2] <https://blog.karatos.in/a?ID=01000-29a50a4e-6b6e-4421-a9b4-eb60b6fee9df>
- [3] <https://github.com/pjreddie/darknet/issues/723>
- [4] <https://learnopencv.com/deep-learning-based-object-detection-using-yolov3-withopencvpython-c/>
- [5] <https://github.com/dannylee1020/object-detection-video/tree/master/app>
- [6] <https://www.stereolabs.com/developers/release/>
- [7] <https://www.stereolabs.com/docs/app-development/python/install/>
- [8] <https://github.com/stereolabs/zed-python-api>
- [9] <https://techzizou.com/install-cuda-and-cudnn-on-windows-and-linux/>
- [10] <https://techzizou.com/setup-opencv-dnn-cuda-module-for-linux/>
- [11] [https://github.com/VictorLin000/YOLOv3\\_mask\\_detect](https://github.com/VictorLin000/YOLOv3_mask_detect)
- [12] [https://www.v7labs.com/blog/yolo-object-detection#:~:text=YOLO%20\(You%20Only%20Look%20Once,for%20its%20speed%20and%20accuracy.](https://www.v7labs.com/blog/yolo-object-detection#:~:text=YOLO%20(You%20Only%20Look%20Once,for%20its%20speed%20and%20accuracy.)
- [13] <https://github.com/szahid86/Object-Detection-with-Jetson-Tx2>
- [14] <https://www.youtube.com/@JetsonHacks>
- [15] <https://www.geeksforgeeks.org/r-cnn-vs-fast-r-cnn-vs-faster-r-cnn-ml/>
- [16] <https://analyticsindiamag.com/r-cnn-vs-fast-r-cnn-vs-faster-r-cnn-a-comparative-guide/>
- [17] <https://www.semanticscholar.org/paper/Comparison-of-CNN-and-YOLO-for-Object-Detection-Lee-Kim/22060a770c18be320e193da7a7806d83c8ea3336>
- [18] [https://www.researchgate.net/publication/339685696\\_A\\_survey\\_Comparison\\_between\\_Convolutional\\_Neural\\_Network\\_and\\_YOLO\\_in\\_image\\_identification](https://www.researchgate.net/publication/339685696_A_survey_Comparison_between_Convolutional_Neural_Network_and_YOLO_in_image_identification)
- [19] <https://www.stereolabs.com/docs/depth-sensing/depth-settings/>
- [20] [https://github.com/MINED30/Face\\_Mask\\_Detection\\_YOLO](https://github.com/MINED30/Face_Mask_Detection_YOLO)
- [21] [https://www.researchgate.net/publication/350577616\\_Real-Time\\_Face\\_Mask\\_Detection\\_Method\\_Based\\_on\\_YOLOv3](https://www.researchgate.net/publication/350577616_Real-Time_Face_Mask_Detection_Method_Based_on_YOLOv3)
- [22] <http://www.utias.utoronto.ca/wp-content/uploads/2019/07/71-Fusion-Object-Detection-with-convolutional-neural-network.pdf>
- [23] <https://ch.mathworks.com/help/vision/ug/using-kalman-filter-for-object-tracking.html>
- [24] <https://link.springer.com/article/10.1007/s11042-021-10711-8>