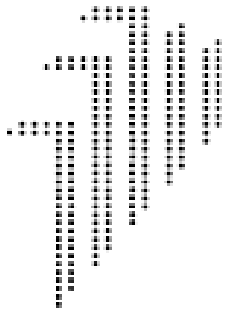

h_da

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

A
Project Report on

"Vision AI on Nvidia Jetson Nano: Sensor Fusion"

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

Team Members	Matriculation Number
Amrutha Venugopal	1119243
Nikita Patil	1119255
Riya Alias	1119262
Sreekuttan Sreedhar	1123158

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

DEPARTMENT OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Abstract

The project titled "Vision AI on Nvidia Jetson Nano: Sensor Fusion" focuses on leveraging the capabilities of the Nvidia Jetson Nano platform to enhance real-time object detection and recognition through sensor fusion. Sensor fusion involves integrating data from multiple sensors, such as cameras, LiDAR, and other environmental sensors, to create a more comprehensive and accurate understanding of the surroundings.

This project involves the integration of drowsiness detection and traffic sign recognition applications on the Nvidia Jetson Nano with the two Raspberry Pi Cameras, optimizing resource usage to ensure real-time performance. The system's ability to simultaneously run drowsiness detection and traffic sign recognition demonstrates the device's multitasking capabilities, crucial for comprehensive driver assistance systems. By combining these two applications, the project showcases the potential of the Nvidia Jetson Nano as a versatile and powerful edge computing solution for automotive safety.

Table of Contents

List of Figures

1. Introduction

In recent years, the rapid advancements in artificial intelligence (AI) and embedded computing have paved the way for innovative applications in various domains, including robotics, autonomous vehicles, and industrial automation. One of the critical aspects of enabling these applications is the integration of multiple sensors to provide a comprehensive understanding of the surrounding environment. This process, known as sensor fusion, plays a pivotal role in enhancing the accuracy, reliability, and contextual awareness of AI systems.

The NVIDIA Jetson Nano, a compact and energy-efficient embedded computing platform, has emerged as a popular choice for developing AI-powered solutions at the edge. With its impressive processing capabilities, GPU acceleration, and compatibility with various sensors, the Jetson Nano offers a solid foundation for implementing sensor fusion techniques. By combining data from diverse sensors such as cameras, lidar, radar, and IMUs (Inertial Measurement Units), developers can create intelligent systems that possess a more holistic perception of their surroundings.

1.1 Goals of this project

The main objective of this project is to develop a comprehensive and robust Vision AI system on the NVIDIA Jetson Nano that leverages sensor fusion techniques to enhance both drowsiness detection and traffic sign detection.

Specific Goals:

Drowsiness Detection :

Implement a drowsiness detection algorithm that analyzes facial cues from a camera feed to identify signs of driver drowsiness. Optimize the algorithm for real-time processing using the Jetson Nano's GPU acceleration.

Traffic Sign Detection:

Develop a traffic sign detection and recognition algorithm that utilizes camera inputs to identify and interpret traffic signs and signals.Implement a mechanism to recognize and react to different types of traffic signs, such as speed limits, stop signs, and yield signs.

By achieving these specific goals, the project aims to create a holistic and innovative solution that demonstrates the capabilities of sensor fusion in enhancing safety and awareness in driving scenarios. The integration of drowsiness detection and traffic sign detection through sensor fusion showcases the power of AI technologies and their potential to positively impact road safety.

1.1.1 Flow Diagram

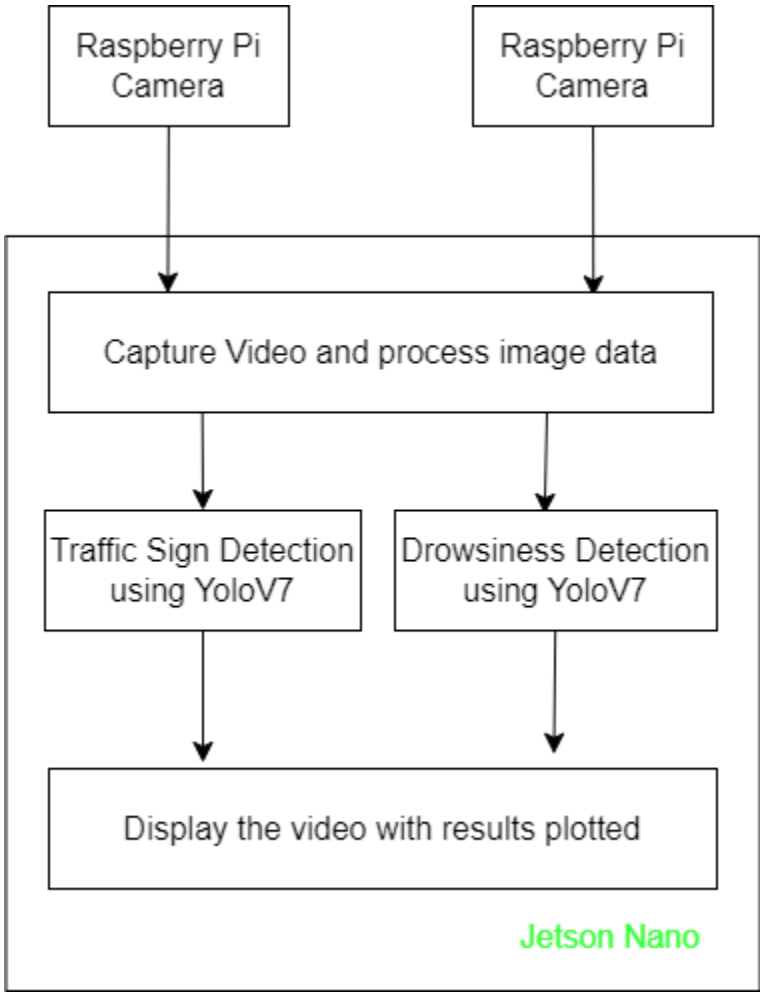

Figure 1: Flow Diagram

1.2 Used Hardware

For the successful implementation of this project, we have used Nvidia Jetson Nano Developer Kit and 2 Raspberry Pi Camera.

1.2.1 Nvidia Jetson Nano Developer Kit

NVIDIA Jetson Nano Developer Kit is a small, powerful computer that lets you run multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing. It serves as the central processing unit for our project and

provides the computational power necessary for real-time image processing and sensor fusion. The main specification of the board are :

- GPU : 128-core Maxwell
- CPU: Quad-core ARM A57 @ 1.43 GHz
- Memory: 4 GB 64-bit LPDDR4 25.6 GB/s
- Camera: 2x MIPI CSI-2 DPHY lanes

Additionally, it includes Gigabit Ethernet connectivity, HDMI display port , 4x USB 3.0, USB 2.0 Micro-B ports.

1.2.2 Camera

We utilized 2 Raspberry Pi cameras to capture visual data. The v2 Camera Module has a Sony IMX219 8-megapixel sensor. It can be used to take high-definition video, as well as stills photographs.It supports 1080p30, 720p60 and VGA90 video modes, as well as still capture. It attaches via a 15cm ribbon cable to the CSI port on the Jetson Nano.

1.3 Introduction to Artificial Neural and Convolutional Neural Network

1.3.1 Artificial Neural Network

Artificial Neural Networks (ANNs) are a class of machine learning models inspired by the structure and functioning of the human brain's neural networks. They are composed of interconnected nodes, known as artificial neurons or perceptrons, organized into layers. ANNs are used for a wide range of tasks, including pattern recognition, classification, regression, and more.

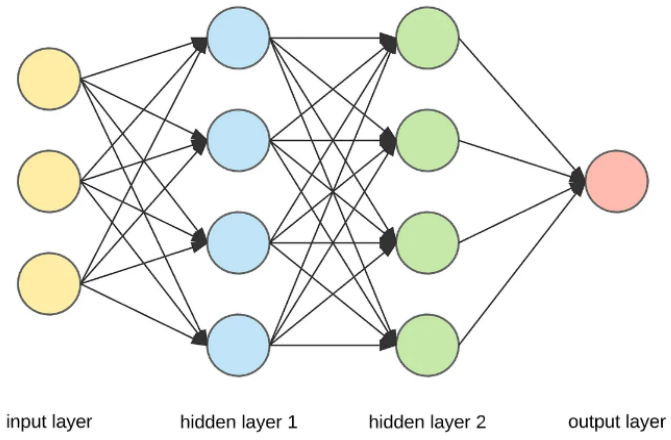

Figure 2: Three Layer Neural Network

Key Components of ANNs:

Neurons/Perceptrons: These are the basic processing units that receive inputs, apply weights, perform computations, and produce outputs.

Layers: Neurons are grouped into layers, including input, hidden, and output layers. Data flows through the network from input to output layers, with intermediate layers (hidden layers) enabling complex transformations.

Weights and Biases: Weights and biases are parameters associated with each connection between neurons. They determine the influence of inputs on a neuron's activation.

Activation Function:Activation functions introduce non-linearity to the network. They determine the output of a neuron based on its weighted sum of inputs and bias.

1.3.2 Convolutional Neural Networks (CNNs):

Convolutional Neural Networks (CNNs) are a specialized type of artificial neural network designed for processing grid-like data, such as images and videos. They excel in tasks involving feature extraction, pattern recognition, and image analysis. CNNs have revolutionized computer vision and have been crucial in achieving state-of-the-art performance in various image-related tasks.Key Components of CNNs:

Convolutional Layer: This layer uses convolutional operations to extract local features from input data. It applies learnable filters (kernels) to the input to detect specific patterns.

Pooling Layer: Pooling layers downsample the feature maps obtained from convolutional layers, reducing the spatial dimensions while retaining important information.

Activation Functions: Just like in ANNs, CNNs use activation functions to introduce non-linearity into the network, enhancing its ability to learn complex relationships.

Fully Connected Layers: These layers connect neurons from previous layers to the final output layer. They often follow convolutional and pooling layers to perform high-level reasoning and decision-making.

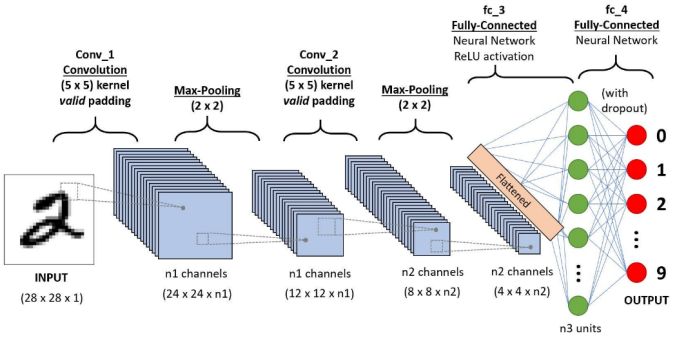

Figure 3: Convolutional Neural Networks

1.4 Object Detection

1.4.1 Deep Learning Model Used for Implementation - YOLO (You Only Look Once)

YOLO, short for "You Only Look Once," is a real-time object detection algorithm that revolutionized the field of computer vision by providing a highly efficient and accurate approach to detecting objects in images and videos. Unlike traditional object detection methods that involve multiple stages and complex computations, YOLO performs object detection in a single pass, making it incredibly fast and suitable for real-time applications.

The key innovation of YOLO is its ability to predict bounding boxes and class probabilities for multiple objects in a single forward pass of the neural network. This approach is in contrast to traditional region-based methods that require scanning an image multiple times using a sliding window approach or employing region proposal networks.

YOLO Architecture:

The YOLO algorithm takes an image as input and then uses a simple deep convolutional neural network to detect objects in the image. The architecture of the CNN model that forms the backbone of YOLO is shown below.

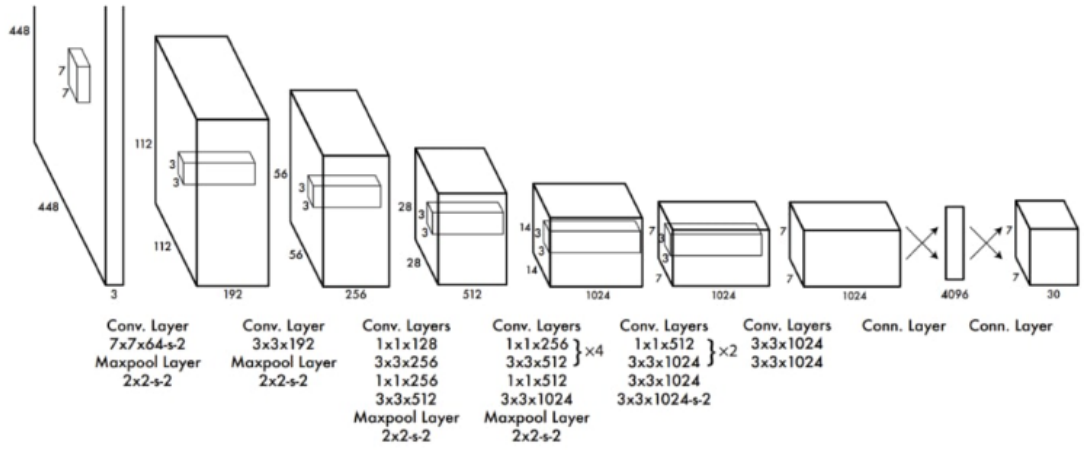

Figure 4: YOLO Architecture

YOLO Working Principle:

YOLO divides an input image into an S × S grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model

is that the box contains an object and how accurate it thinks the predicted box is.YOLO predicts multiple bounding boxes per grid cell. At training time, we only want one bounding box predictor to be responsible for each object. YOLO assigns one predictor to be “responsible” for predicting an object.

One key technique used in the YOLO models is non-maximum suppression (NMS). NMS is a post-processing step that is used to improve the accuracy and efficiency of object detection. In object detection, it is common for multiple bounding boxes to be generated for a single object in an image. These bounding boxes may overlap or be located at different positions, but they all represent the same object. NMS is used to identify and remove redundant or incorrect bounding boxes and to output a single bounding box for each object in the image.

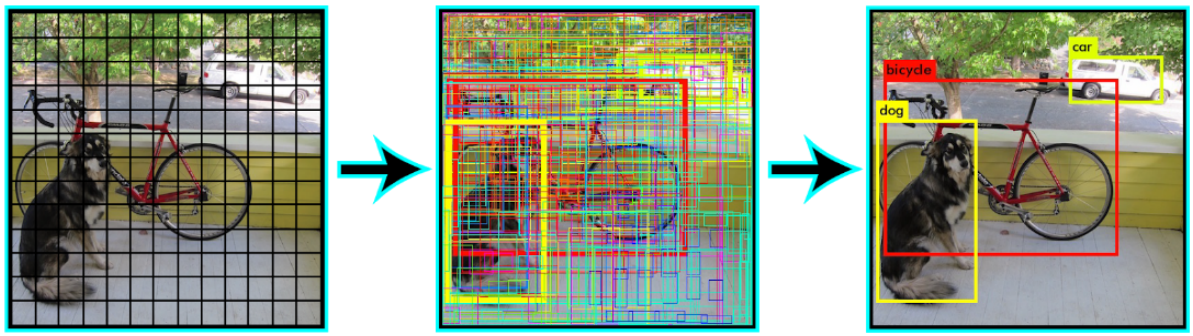

Figure 5: NMS-Post-Processing

To achieve this, YOLO first compares the probability scores associated with each decision, and takes the largest score. Following this, it removes the bounding boxes with the largest Intersection over Union with the chosen high probability bounding box. This step is then repeated until only the desired final bounding boxes remain.

1.4.2 YOLOv7

YOLOv7 is a state-of-the-art real-time object detector that surpasses all known object detectors in both speed and accuracy in the range from 5 FPS to 160 FPS. It has the highest accuracy (56.8% AP) among all known real-time object detectors with 30 FPS or higher on GPU V100.

In general, YOLOv7 provides a faster and stronger network architecture that provides a more effective feature integration method, more accurate object detection performance, a more robust loss function, and an increased label assignment and model training efficiency. As a result, YOLOv7 requires several times cheaper computing hardware than other deep learning models. It can be trained much faster on small datasets without any pre-trained weights.

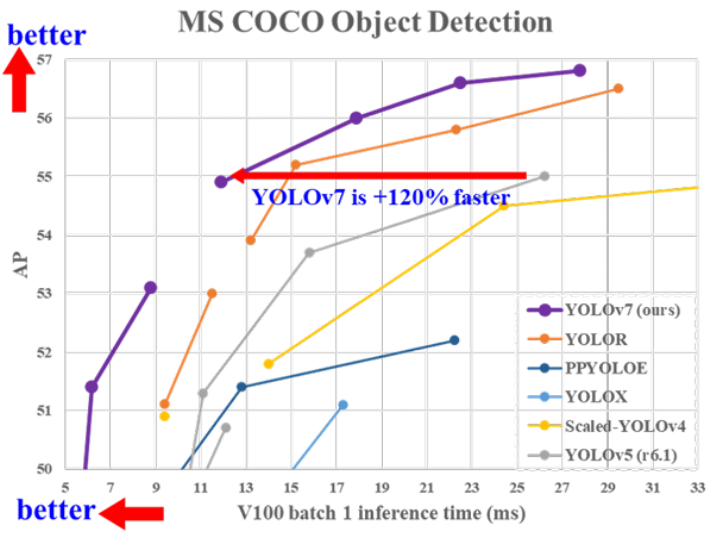

Figure 6: **Comparison With Other Real-time object detector**

YOLOv7 introduces several key features:

Model Re-parameterization: YOLOv7 proposes a planned re-parameterized model, which is a strategy applicable to layers in different networks with the concept of gradient propagation path. Re-parameterization techniques involve averaging a set of model weights to create a model that is more robust to general patterns that it is trying to model.The YOLOv7 authors use gradient flow propagation paths to see which modules in the network should use re-parameterization strategies and which should not.

Compound Model Scaling: The main purpose of model scaling is to adjust key attributes of the model to generate models that meet the needs of different application requirements. In traditional approaches with concatenation-based architectures (for example, ResNet or PlainNet), different scaling factors cannot be analyzed independently and must be considered together. YOLOv7 introduces compound model scaling for a concatenation-based model. The compound scaling method allows to maintain the properties that the model had at the initial design and thus maintain the optimal structure.

Auxiliary Head Coarse-to-Fine: A YOLO architecture contains a backbone, a neck, and a head. The head contains the predicted model outputs. YOLOv7 is not limited to one single head. The head responsible for the final output is called the lead head, and the head used to assist training in the middle layers is named auxiliary head. In addition, and to enhance the deep network training, a Label Assigner mechanism was introduced that considers network prediction results together with the ground truth and then assigns soft labels. Compared to traditional label assignment that directly refers to the ground truth to generate hard labels based

on given rules, reliable soft labels use calculation and optimization methods that also consider the quality and distribution of prediction output together with the ground truth.

Extended Efficient Layer Aggregation: YOLOv7 proposes "extend" and "compound scaling" methods for the real-time object detector that can effectively utilize parameters and computation. The computational block in the YOLOv7 backbone is named E-ELAN, standing for Extended Efficient Layer Aggregation Network. The E-ELAN architecture of YOLOv7 enables the model to learn better by using “expand, shuffle, merge cardinality” to achieve the ability to continuously improve the learning ability of the network without destroying the original gradient path.

Efficiency: The method proposed by YOLOv7 can effectively reduce about 40% parameters and 50% computation of state-of-the-art real-time object detectors, and has faster inference speed and higher detection accuracy.

Different YOLOv7 Models

YOLOv7-tiny:

YOLOv7-tiny is a basic model optimized for edge GPUs. The suffix “tiny” of computer vision models means that they are optimized for Edge AI and deep learning workloads, and more lightweight to run ML on mobile computing devices or distributed edge servers and devices. This model is important for distributed real-world computer vision applications. Compared to the other versions, the edge-optimized YOLOv7-tiny uses leaky ReLU as the activation function, while other models use SiLU as the activation function.

YOLOv7-W6:

YOLOv7-W6 is a basic model optimized for cloud GPU computing. Such Cloud Graphics Units (GPUs) are computer instances for running applications to handle massive AI and deep learning workloads in the cloud without requiring GPUs to be deployed on the local user device.

Other models include YOLOv7-X, YOLOv7-E6, and YOLOv7-D6, which were obtained by applying the proposed compound scaling method.

1.4.3 YOLOv5

YOLOv5, the fifth iteration of the revolutionary "You Only Look Once" object detection model, is designed to deliver high-speed, high-accuracy results in real-time. Built on PyTorch, this powerful deep learning framework has garnered immense popularity for its versatility, ease of use, and high performance.

YOLOv5 comes in four main versions: small (s), medium (m), large (l), and extra large (x), each offering progressively higher accuracy rates. Each variant also takes a different amount of time to train. The YOLO model was the first object detector to connect the procedure of predicting bounding boxes with class labels in an end to end differentiable network.

YOLOv5's architecture consists of three main parts:

Backbone: This is the main body of the network. For YOLOv5, the backbone is designed using the New CSP-Darknet53 structure, a modification of the Darknet architecture used in previous versions.

Neck: This part connects the backbone and the head. In YOLOv5, SPPF and New CSP-PAN structures are utilized.

Head: This part is responsible for generating the final output. YOLOv5 uses the YOLOv3 Head for this purpose.

The two main training procedures in YOLOv5 includes:

Data Augmentation: Data augmentation makes transformations to the base training data to expose the model to a wider range of semantic variation than the training set in isolation.

Loss Calculations: YOLO calculates a total loss function from the GIoU, obj, and class losses functions. These functions can be carefully constructed to maximize the objective of mean average precision.

YOLOv5 represents a significant step forward in the development of real-time object detection models. By incorporating various new features, enhancements, and training strategies, it surpasses previous versions of the YOLO family in performance and efficiency.

The primary enhancements in YOLOv5 include the use of a dynamic architecture, an extensive range of data augmentation techniques, innovative training strategies, as well as important adjustments in computing losses and the process of building targets. All these innovations significantly improve the accuracy and efficiency of object detection while retaining a high degree of speed, which is the trademark of YOLO models.

1.4.4 Edge Impulse

Edge Impulse is a platform that enables developers to create and deploy machine learning models directly onto edge devices, such as microcontrollers, without the need for extensive expertise in machine learning or deep technical knowledge. The platform focuses on the concept of "Tiny ML," which refers to running machine learning models on resource-constrained devices like microcontrollers, which have limited computational power and memory.

The platform provides a user-friendly interface for collecting, preprocessing, and labeling data, training machine learning models, and deploying these models onto edge devices. It supports a wide range of sensor data types commonly found in applications like Internet of Things (IoT), wearable devices, robotics, and more. Edge Impulse supports various stages of the machine learning workflow, including data collection, feature extraction, model training, model evaluation, and deployment.

Developers can train models using various algorithms and techniques, and the platform also allows for transfer learning, where a pre-trained model is fine-tuned for a specific task using a smaller dataset. Once a model is trained, it can be deployed directly onto edge devices, enabling real-time, on-device inference without relying on cloud-based processing.

The goal of Edge Impulse is to make it easier for developers to integrate machine learning capabilities into edge devices, enabling a wide range of applications that benefit from local processing, reduced latency, and enhanced privacy.

1.5 System and Software Specifications

1.5.1 System Specifications

- The system focuses on two main object detection models: the Driver Drowsiness Detection and Traffic Sign Detection.
- Driver Drowsiness Detection monitors the facial features like drowsy eyes and yawning for drowsiness detection and determines whether the driver is drowsy or not.
- Traffic Sign Detection monitors the various traffic signs on the road and classifies them as Mandatory, Prohibitory, Danger and Other.
- Object Detection is based on Yolov5 and Yolov7 models.
- Utilizes TensorRT framework to optimize model performance.
- Used Raspberry Pi camera for real time detection.
- Prototype analysis and performance evaluation is done on Nvidia Jetson Nano

1.5.2 Software Specifications

- Operating System : Ubuntu 18.04
- Nvidia SDK: JetPack 5.1.2
- Libraries: CUDA, TensorRT and cuDNN
- Programming Language : Python 3.7
- Image Processing Library: OpenCV, Gstreamer
- Python version : greater than or equal to 3.7

1.5.2.1 JetPack

NVIDIA JetPack SDK is the most comprehensive solution for building end-to-end accelerated AI applications. JetPack provides a full development environment for hardware-accelerated AI-at-the-edge development on Nvidia Jetson modules.

JetPack includes Jetson Linux with bootloader, Linux kernel, Ubuntu desktop environment, and a complete set of libraries for acceleration of GPU computing, multimedia, graphics, and computer vision.

1.5.2.2 CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

The toolkit includes a compiler for NVIDIA GPUs, math libraries, and tools for debugging and optimizing the performance of your applications. JetPack 5.1.2 includes CUDA 11.4.19

1.5.2.3 cuDNN

CUDA Deep Neural Network library provides high-performance primitives for deep learning frameworks. It provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. cuDNN accelerates widely used deep learning frameworks, including Caffe2, Chainer, Keras, MATLAB, MxNet, PaddlePaddle, PyTorch, and TensorFlow. JetPack 5.1.2 includes cuDNN 8.6.0

1.5.2.4 TensorRT

TensorRT is a high performance deep learning inference runtime for image classification, segmentation, and object detection neural networks. TensorRT is built on CUDA, NVIDIA's parallel programming model, and enables you to optimize inference for all deep learning frameworks. It includes a deep learning inference optimizer and runtime that delivers low latency and high-throughput for deep learning inference applications. JetPack 5.1.2 includes TensorRT 8.5.2

1.5.2.5 OpenCV

OpenCV is the leading open source library for computer vision, image processing and machine learning, and now features GPU acceleration for real-time operation. The library offers a wide range of features, including image manipulation, feature extraction, image stitching, object tracking, camera calibration, and machine learning integration. It supports multiple programming languages, including C++, Python, and Java, making it accessible to a diverse range of developers.

1.5.2.6 GStreamer

GStreamer is an extremely powerful and versatile framework for creating streaming media applications. It supports a wide variety of media-handling components, including simple audio playback, audio and video playback, recording, streaming and editing. The pipeline design serves as a base to create many types of multimedia applications such as video editors, transcoders, streaming media broadcasters and media players. It offers a powerful toolset for building multimedia applications and is often used in conjunction with other libraries like OpenCV for computer vision tasks or FFmpeg for multimedia encoding and decoding.

2. Implementation Methodology

2.1 Edge Impulse

First, we have tried implementation of our project using Edge Impulse. The various steps involved in the process are:

2.1.1 Building a Dataset

To implement our machine learning model it is required to capture a lot of sample images of the object under detection, in our case images for drowsiness detection and traffic signs. The data can be collected from the Studio- for the Raspberry Pi 4 and the Jetson Nano and or can be done with mobile phone, and then upload them into **Data acquisition**

Next we have to label all the collected dataset. This can also be done through Edge Impulse.The **labeling** queue shows you all the unlabeled data in your dataset. Labeling your objects is as easy as dragging a box around the object, and entering a label.

In case of Drowsiness detection, we have 2 classes: drowsy and awake. If the eyes are closed or the person is yawning, the image is classified as drowsy or else the person is awake.

For Traffic Sign Detection, we have focussed on 4 classes namely Danger, Prohibitory, Mandatory and Other. Prohibitory category consists of speed limit, no overtaking, no traffic both ways, no trucks signs. Danger category consists of priority at the next intersection, danger, bend left, bend right, bend, uneven road, slippery road, road narrows, construction, traffic signal, pedestrian crossing, school crossing, cycles crossing, snow, animals. Mandatory category consists of : go right, go left, go straight, go right or straight, go left or straight, keep right, keep left, roundabout signs. Restriction ends, priority road, give way, stop, no entry are grouped into Other categories.

The final step in dataset creation is **Rebalancing our dataset.** To validate whether a model works well, we keep some data (typically 20%) aside, and don't use it to build our model, but only to validate the model. This is called the 'test set'.

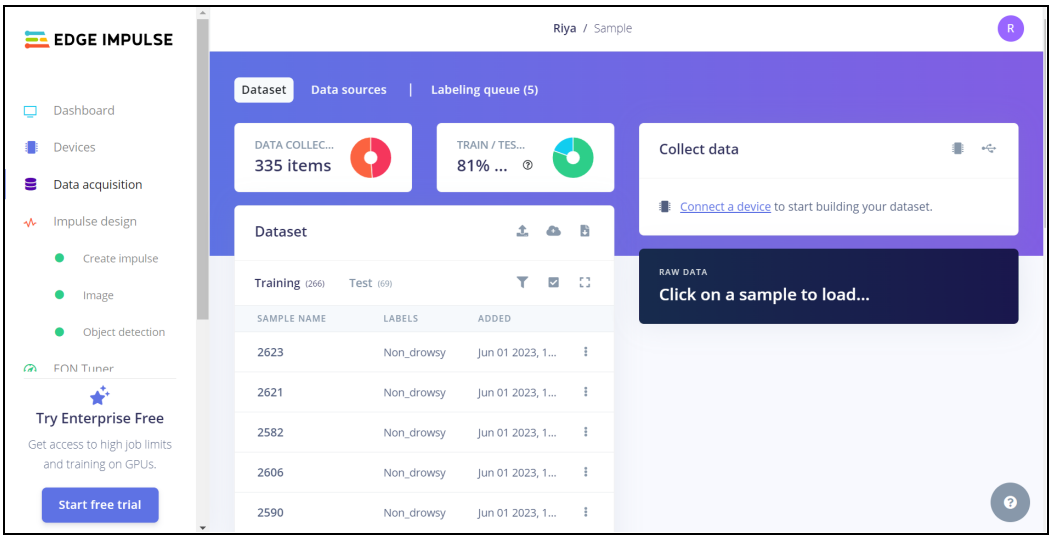

Figure 7: Building dataset using Edge Impulse

2.1.2 Designing an Impulse

An impulse takes the raw data, adjusts the image size, uses a preprocessing block to manipulate the image, and then uses a learning block to classify new data. Preprocessing blocks always return the same values for the same input (e.g. convert a color image into a grayscale one), while learning blocks learn from past experiences.

For our project we have used an 'Images' preprocessing block. This block takes in the color image, optionally makes the image grayscale, and then turns the data into a features array. Here we set the image width and image height to 96, the 'resize mode' to Fit shortest axis and add the 'Images' and 'Object Detection (Images)' blocks.

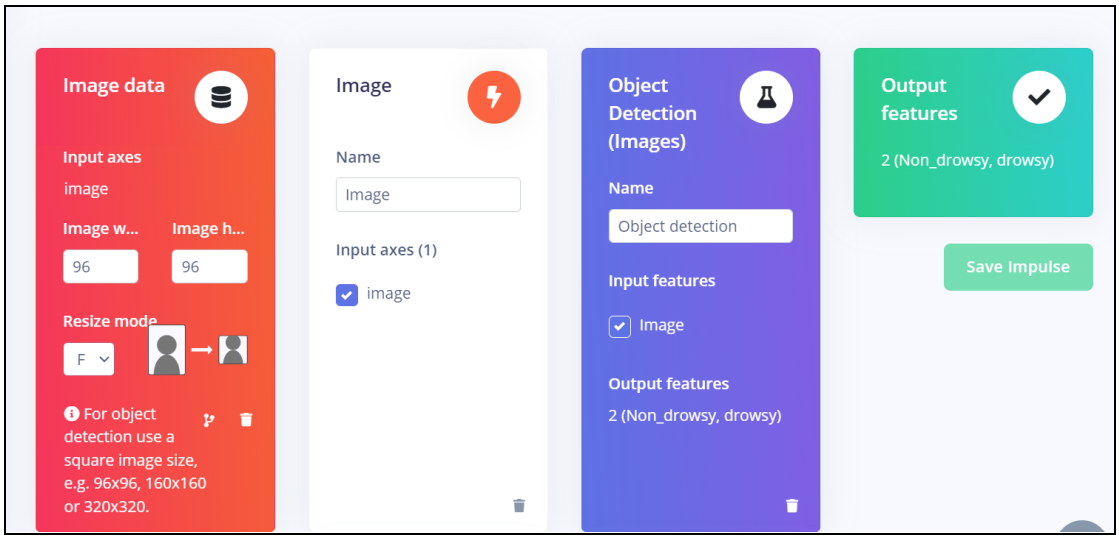

Figure 8: Designing impulse

Configuring the processing block

Processing block can be configured using the Images tab. This will show you the raw data on top of the screen and the results of the processing step on the right. You can use the options to switch between 'RGB' and 'Grayscale' mode. We have kept the color depth as RGB itself. With the generate features, we will be able to plot and clearly identify the main features of the object and make classification

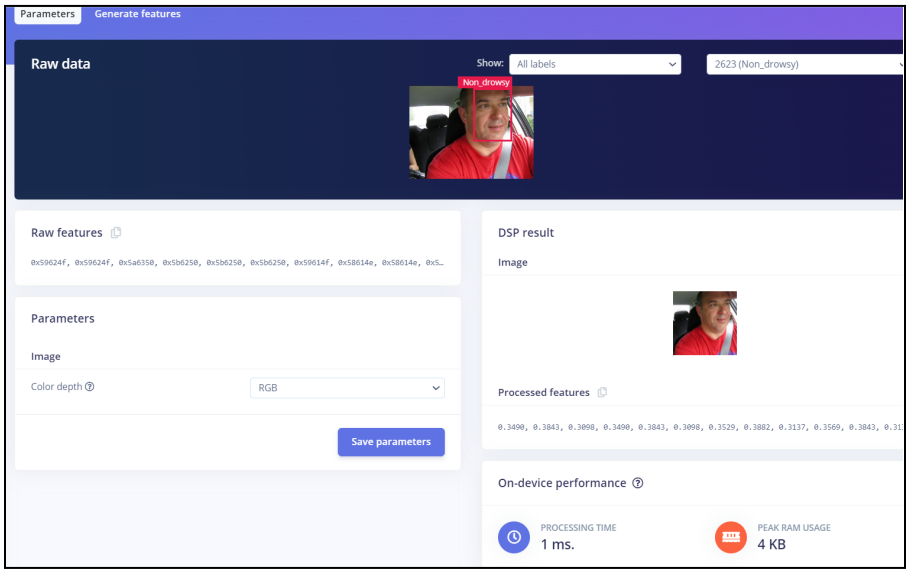

Figure 9: Processing block

2.1.3 Configuring the transfer learning model

With all data processed we can start training a neural network. We have used 100 the number of training cycles and 0.005 learning rate to train our model. Edge Impulse provides two different methods to perform object detection: Using MobileNetV2 SSD FPN and Using FOMO. We have used FOMO to train our model and it provides performance efficiency of the model

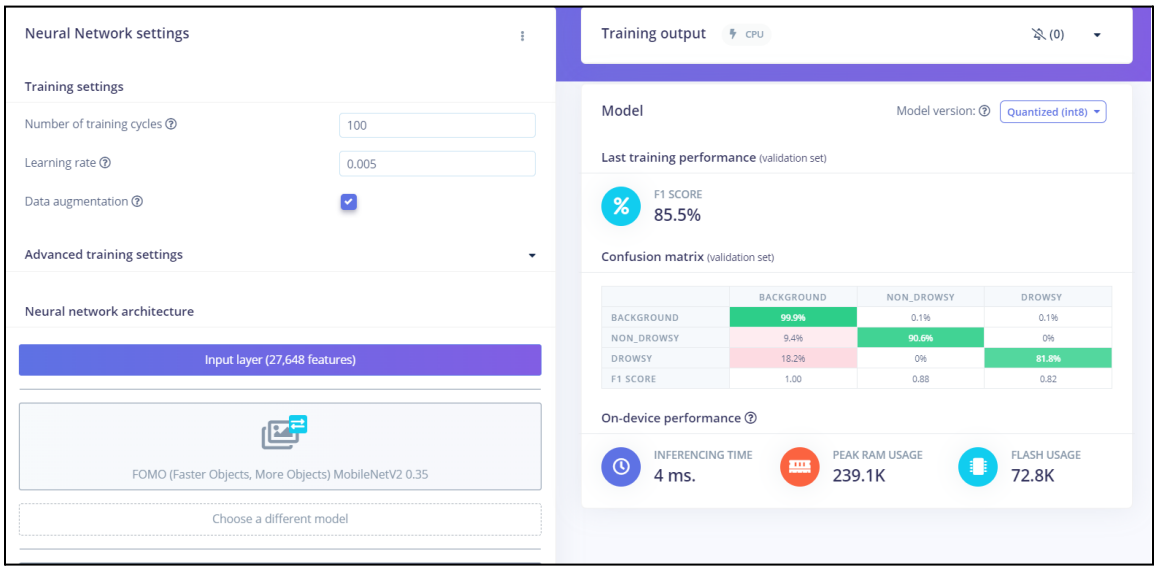

Figure 10: Training the model using FOMO

2.1.4. Validating your model

With the model trained, we can now test the data. The model was trained only on the training data, and thus we can use the data in the testing dataset to validate how well the model will work in the real world. Validation can also be performed using Model Testing in Edge Impulse. And we were able to achieve 94.02 % accuracy for our test data.

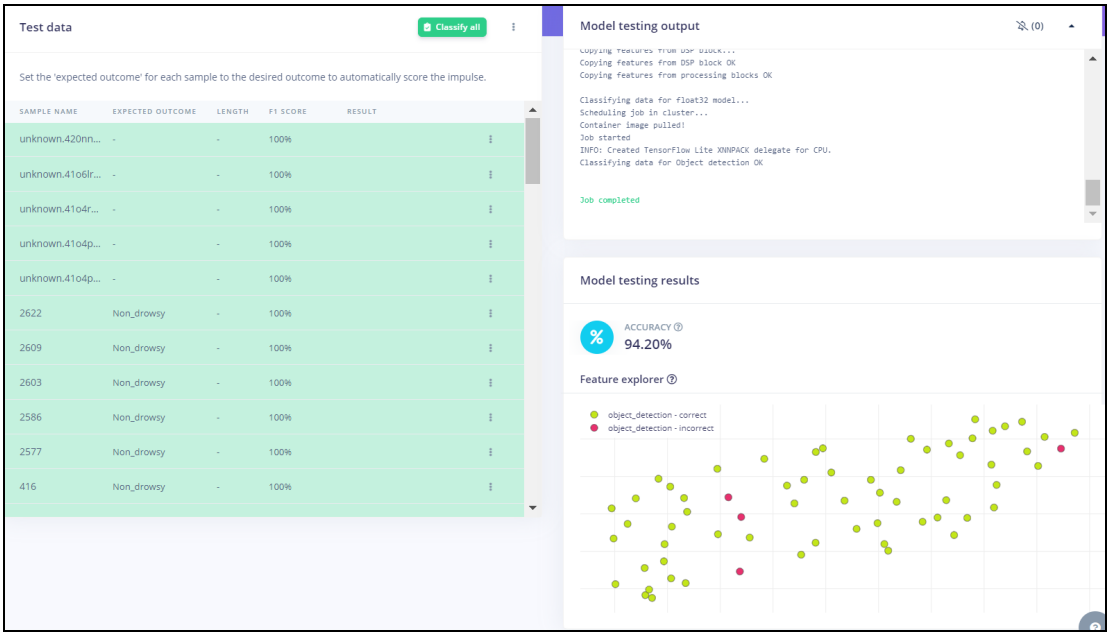

Figure 11: Validating the model

With the Edge Impulse model we were able to use only FOMO or MobileNetV2 for object detection. We were not able to include better object detection like YOLO into the training. Also the test results showed good results with existing test data, but while trying to give new real time data the results were not that good. Hence we moved into better detection models.

2.2 Yolov5 and Jupyter Notebook

2.2.1 Setting up the environment

Cloning the yolov5 repository and installing all the necessary packages like opencv, numpy, torch, torchvision etc required to train our model.

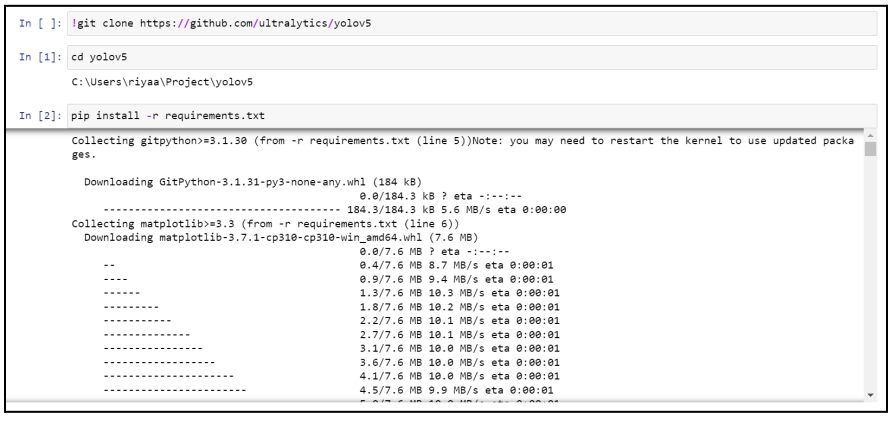

Figure 12: Installing yolov5 using jupyter Notebook

2.2.2 Import libraries

```
import torch
from matplotlib import pyplot as plt
import numpy as np
import cv2
```

Figure 13: Import Libraries

PyTorch provides the necessary tools for building and training neural networks. PyTorch's tensor operations are crucial for loading images, preprocessing them for the model input, performing forward passes through the neural network, and processing the model's output.

Matplotlib is a plotting library in Python. It's often used to visualize the results of object detection, such as drawing bounding boxes around detected objects on the input images.

Numpy is a fundamental library for numerical operations in Python. It's used extensively for data manipulation and array computations. During object detection, we need to process the raw output from the model, which is often in the form of tensors. numpy allows you to easily convert tensors to arrays, perform calculations, and manipulate data.

OpenCV is a computer vision library that provides tools for various image and video processing tasks. OpenCV can help with tasks like resizing images to the model's input size, normalizing pixel values, and other data augmentation techniques.

2.2.3 Data collection and preparation

2.2.3.1 For Drowsiness Detection

The images can be either captured directly through our camera or can be uploaded into the device using an already existing dataset. Here we capture some images through our camera and store it in the data directory in the cloned Yolov5 path.

```
cap = cv2.VideoCapture(0)
# Loop through labels
for label in labels:
    print('Collecting images for {}'.format(label))
    time.sleep(5)

    # Loop through image range
    for img_num in range(number_imgs):
        print('Collecting images for {}, image number {}'.format(label, img_num))

        # Webcam feed
        ret, frame = cap.read()

        # Naming out image path
        imgname = os.path.join(IMAGES_PATH, label+'.'+str(uuid.uuid1())+'.jpg')

        # Writes out image to file
        cv2.imwrite(imgname, frame)

        # Render to the screen
        cv2.imshow('Image Collection', frame)

        # 2 second delay between captures
        time.sleep(2)

        if cv2.waitKey(10) & 0xFF == ord('q'):
            break
cap.release()
cv2.destroyAllWindows()
```

```
Collecting images for awake
```

Figure 14: Collecting images for dataset

After collecting the images, we have to label each one of them as awake or drowsy. We have used LabelImg to label the images. And we split the total dataset to train(80 %) and val(20%). The train set is used to train the model and val is used to test the trained model.

2.2.3.2 For Traffic Sign Detection

The German Traffic Sign Detection Benchmark (GTSDB) by J. Stallkamp et al., 2013 represents a large multi-category classification benchmark. The dataset contains in total 43 classes which can be categorized in four categories called prohibitory, danger, mandatory and other. The classes are labeled numerically with the following mapping:

- 0 = speed limit 20 (prohibitory)
- 1 = speed limit 30 (prohibitory)
- 2 = speed limit 50 (prohibitory)
- 3 = speed limit 60 (prohibitory)
- 4 = speed limit 70 (prohibitory)
- 5 = speed limit 80 (prohibitory)
- 6 = restriction ends 80 (other)
- 7 = speed limit 100 (prohibitory)
- 8 = speed limit 120 (prohibitory)
- 9 = no overtaking (prohibitory)
- 10 = no overtaking (trucks) (prohibitory)
- 11 = priority at next intersection (danger)
- 12 = priory road (other)
- 13 = give way (other)
- 14 = stop (other)
- 15 = no traffic both ways (prohibitory)
- 16 = no trucks (prohibitory)
- 17 = no entry (other)

- 18 = danger
- 19 = bend left (danger)
- 20 = bend right (danger)
- 21 = bend (danger)
- 22 = uneven road (danger)
- 23 = slippery road (danger)
- 24 = road narrows (danger)
- 25 = construction (danger)
- 26 = traffic signal (danger)
- 27 = pedestrian crossing (danger)
- 28 = school crossing (danger)
- 29 = cycles crossing (danger)
- 30 = snow (danger)
- 31 = animals (danger)
- 32 = restriction ends (other)
- 33 = go right (mandatory)
- 34 = go left (mandatory)
- 35 = go straight (mandatory)
- 36 = go right or straight (mandatory)
- 37 = go left or straight (mandatory)
- 38 = keep right (mandatory)
- 39 = keep left (mandatory)
- 40 = roundabout (mandatory)
- 41 = restriction ends (overtaking) (other)
- 42 = restriction ends (overtaking (trucks)) (other)

We found a dataset which has already labeled in Yolo format from the Kaggle site which classed the above 43 classes into 4 main classes as follows:

- prohibitory = [0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 15, 16]
- danger = [11, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
- mandatory = [33, 34, 35, 36, 37, 38, 39, 40]
- other = [6, 12, 13, 14, 17, 32, 41, 42]

Each image of the GTSDB dataset now contains a txt file with the corresponding normalized bounding box coordinates and the corresponding CategoryID in YOLO format.We divided the dataset into train and val set for the training and validation of the model.

2.2.4 Train the model

```
!cd yolov5 && python train.py --img 320 --batch 16 --epochs 50 --data dataset.yml --weights yolov5s.pt --workers 2
8,  1.44s/it]
                 Class     Images  Instances          P          R      mAP50   mAP50-95:  77%|#######7  | 17/22 [00:24<00:0
7,  1.42s/it]
                 Class     Images  Instances          P          R      mAP50   mAP50-95:  82%|########1 | 18/22 [00:25<00:0
5,  1.42s/it]
                 Class     Images  Instances          P          R      mAP50   mAP50-95:  86%|########6 | 19/22 [00:27<00:0
4,  1.41s/it]
                 Class     Images  Instances          P          R      mAP50   mAP50-95:  91%|######### | 20/22 [00:28<00:0
2,  1.41s/it]
                 Class     Images  Instances          P          R      mAP50   mAP50-95:  95%|#########5| 21/22 [00:29<00:0
1,  1.41s/it]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|##########| 22/22 [00:30<00:0
0,  1.08s/it]
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|##########| 22/22 [00:30<00:0
0,  1.37s/it]
                   all        678        113      0.165       0.99      0.205       0.16
                drowsy        678         50      0.149       0.98      0.201      0.174
                 awake        678         63       0.18          1      0.209      0.147
Results saved to runs\train\exp8
```

Figure 15: Training the yolov5 model

We are training the model using train.py from the YoloV5 repository. The image size is configured as 320*320. The training process will use batches of 16 images at a time. The batch size determines how many images are processed in each forward and backward pass during each training iteration. An epoch is a complete pass through the entire training dataset. In our case, the training will run for 50 epochs, the model will be updated 50 times using the training data. Also we have to configure the dataset.yaml file. In the file we have to provide the paths to test and val images. Also we should mention the number of classes we have used for training. yolov5s.pt is the pre-trained weights file for the YOLOv5 small variant. These pre-trained weights are used as a starting point for training.

2.2.5 Validating the model

```
model = torch.hub.load('ultralytics/yolov5', 'custom', path='yolov5/runs/train/exp8/weights/best.pt', force_reload=True)
Downloading: "https://github.com/ultralytics/yolov5/zipball/master" to C:\Users\riyaa/.cache\torch\hub\master.zip
YOLOv5  2023-6-17 Python-3.10.11 torch-2.0.1+cu117 CPU

Fusing layers...
Model summary: 157 layers, 7055974 parameters, 0 gradients, 15.9 GFLOPs
Adding AutoShape...
```

Figure 16: Validating the model

After the training the weights file will be generated. We load the model using the best weights from the training and test our model.

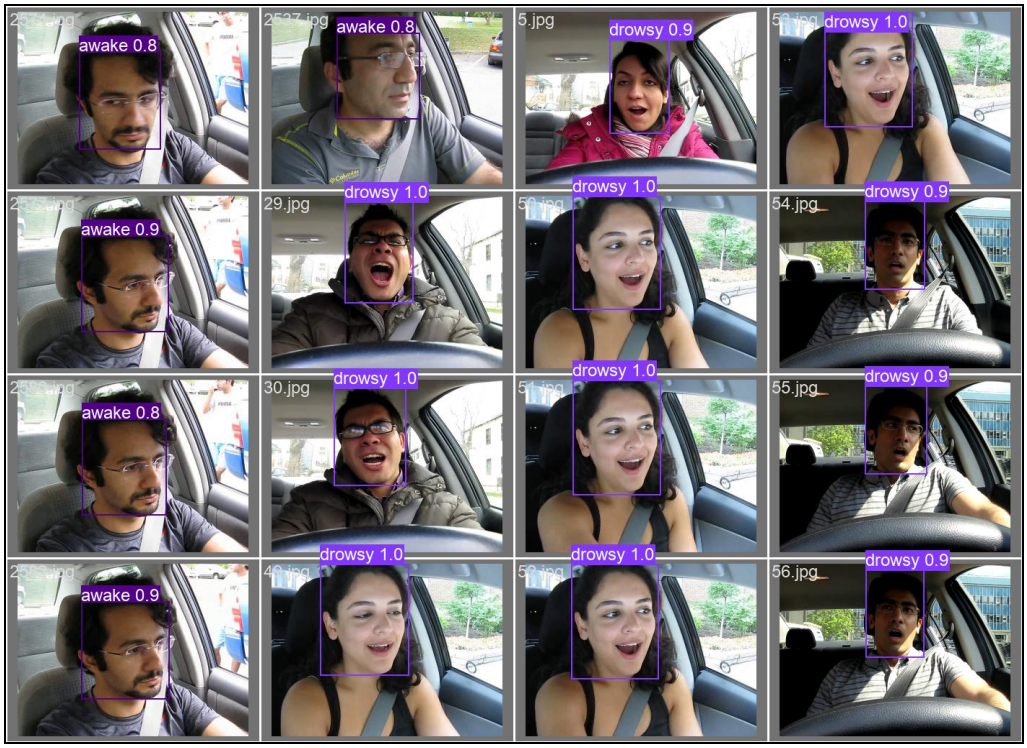

Figure 17: Training results

We were able to generate pretty good results with our training. We could generate images with precision in the range of 90 to 99%.

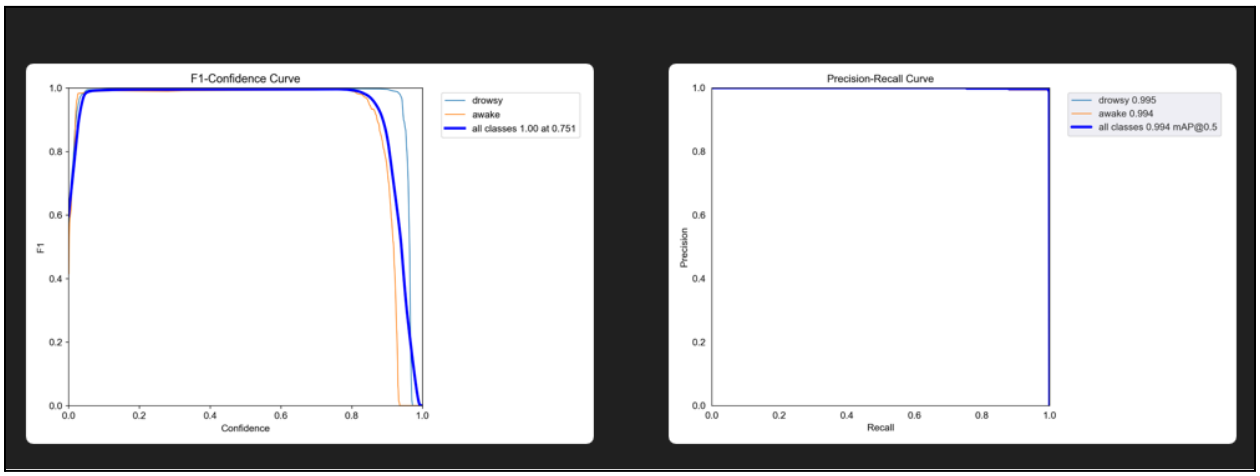

Figure 18: Training Confidence curve

2.3 Yolov7 and Google Colab

The training process conducted within the Jupyter Notebook environment was considerably time-consuming, spanning several hours to reach completion. As a result, moved our training procedure to Google Colab, harnessing the potential of GPU acceleration to significantly speed the training process. The availability of GPU resources in Colab proved instrumental in achieving significantly faster training times compared to our initial approach.

Furthermore, we observed that YOLOv7 tends to yield superior results in comparison to YOLOv5. Notably, YOLOv7 demonstrated improved object detection performance. Additionally, YOLOv7 exhibits optimized resource utilization, requiring fewer computational resources for training and the package dependencies for YOLOv7 were less demanding, simplifying the setup and configuration process.

2.3.1 Clone the yolov7 repository

```
[ ]  !git clone https://github.com/WongKinYiu/yolov7.git

     Cloning into 'yolov7'...
     remote: Enumerating objects: 1191, done.
     remote: Counting objects: 100% (6/6), done.
     remote: Compressing objects: 100% (4/4), done.
     remote: Total 1191 (delta 2), reused 5 (delta 2), pack-reused 1185
     Receiving objects: 100% (1191/1191), 74.23 MiB | 17.35 MiB/s, done.
     Resolving deltas: 100% (514/514), done.
     Updating files: 100% (108/108), done.
```

Figure 19: Cloning yolov7

We mount our google drive with the Colab and clone the official yolov7 repository. We have used the same dataset we used for yolov5 here also. Copy the dataset into the yolov7/data folder.

2.3.2 Train the model

```
!python train.py --device 0 --batch-size 16 --epochs 30 --img 640 640 --data data/custom_data.yaml --hyp data/hyp.scratch.custom.yaml --cfg cfg/training/yolov7x-custom.yaml --weights yolov7x.pt --name yolov7x-custom

2023-08-08 11:05:53.280887: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-08-08 11:05:54.599694: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
YOLOR 📌 v0.1-126-g84932d7 torch 2.0.1+cu118 CUDA:0 (Tesla T4, 15101.8125MB)

Namespace(weights='yolov7x.pt', cfg='cfg/training/yolov7x-custom.yaml', data='data/custom_data.yaml', hyp='data/hyp.scratch.custom.yaml', epochs=30, batch_size=16, img_size=[640, 640], rect=False, resume=False, nosave=Fa
tensorboard: Start with 'tensorboard --logdir runs/train', view at http://localhost:6006/
hyperparameters: lr0=0.01, lrf=0.1, momentum=0.937, weight_decay=0.0005, warmup_epochs=3.0, warmup_momentum=0.8, warmup_bias_lr=0.1, box=0.05, cls=0.3, cls_pw=1.0, obj=0.7, obj_pw=1.0, iou_t=0.2, anchor_t=4.0, fl_gamma=0
wandb: Install Weights & Biases for YOLOR logging with 'pip install wandb' (recommended)
```

Figure 20: Training yolov7 model

Train the model using train.py from the yolov7 repository. The device 0 indicates to use GPU for training. We have used batch size of 16, 30 epochs and image size 640 * 640. The

custom_data.yaml contains information about the test and validation image paths, the number of classes and their names. The hyperparameters file (hyp.scratch.custom.yaml) that contains various hyperparameters used during training. Hyperparameters control aspects like learning rate, momentum, weight decay, etc. The model configuration file (yolov7x-custom.yaml) defines the architecture and settings of the YOLOv7 model to be trained. This includes the backbone network, number of detection layers, anchor settings, and other model-specific configurations. We have used the yolov7-x model for training and hence we have added yolov7x.pt that serves as a starting point for training to leverage knowledge learned from pre-training. The results are stored in runs/train/yolo 7x-custom/weights folder.

2.3.3 Validating the model

```
!python detect.py --weights runs/train/yolov7x-custom2/weights/best.pt --img 640 --conf 0.25 --source data/val/images

Namespace(weights=['runs/train/yolov7x-custom2/weights/best.pt'], source='data/val/images', img_size=640, conf_thres=0.25, iou_thres=0.45,
YOLOR 🚀 v0.1-126-g84932d7 torch 2.0.1+cu118 CUDA:0 (Tesla T4, 15101.8125MB)

Fusing layers...
IDetect.fuse
Model Summary: 362 layers, 70890252 parameters, 0 gradients
 Convert model to Traced-model...
 traced_script_module saved!
 model is traced!
```

Figure 21 : Detection

The model can be tested using detect.py file. We have to provide the best.pt weights from the training and also the path to the folder which contains test images. The average inference time for detections was 30ms.

Figure 22 : Detection results: yolov7

3. Deployment into Jetson Nano

3.1 Setting up Jetson Nano

Download the latest Jetson Nano Developer Kit SD Card Image and write into an SD card using Balena Etcher. Insert the microSD card into the slot on the underside of the Jetson Nano module. Connect the monitor, keyboard, mouse and power supply to the jetson nano board. Configure the initial setup options like language, time zone etc for the device.

3.2 Setup Yolov7

We have deployed both yolov5 and yolov7 models in Jetson Nano. Inorder to isolate Python packages and dependencies for different projects, we have created separate virtual environments for each projects. We install all the dependencies and packages inside the environment and also can deactivate the env when done.

3.3 Camera Setup

The Jetson Nano Developer Kit comes equipped with two RPi camera compatible connectors. The necessary device drivers for the IMX 219 camera are pre-installed, and the camera is pre-configured, requiring only a simple plug-in.
When working with cameras on the Jetson Nano, the GStreamer framework is utilized for interfacing. The `nvarguscamerasrc` element in GStreamer is used, and the `sensor_mode` attribute is employed to specify the desired camera. The valid options for this attribute are either 0 or 1.

```
nvarguscamerasrc sensor_id=0
```

```
def gstreamer_pipeline(
    sensor_id=0,
    capture_width=1920,
    capture_height=1080,ss
    display_width=960,
    display_height=540,
    framerate=10,
    flip_method=2,
):
    return (
        "nvarguscamerasrc sensor-id=%d ! "
        "video/x-raw(memory:NVMM), width=(int)%d, height=(int)%d, framerate=(fraction)%d/1 ! "
        "nvvidconv flip-method=%d ! "
        "video/x-raw, width=(int)%d, height=(int)%d, format=(string)BGRx ! "
        "videoconvert ! "
        "video/x-raw, format=(string)BGR ! appsink"
        % (
            sensor_id,
            capture_width,
            capture_height,
            framerate,
            flip_method,
            display_width,
            display_height,
        )
    )
```

3.4 Testing using the images and videos

We tested the model using images and videos and observed that the performance speed was less. Hence, we decided to opt for optimized deep learning libraries like TensorRT and cuDNN. TensorRT can accelerate model inference by optimizing networks for the Jetson GPU architecture.

4. Performance analysis and Enhancements

While testing the images and videos we observed that the performance of the model was not sufficient to perform real-time detections as the frame rates were very low. Hence we decided to improvise the project by converting the model to tensorRT.

4.1 Setup TensorRT

We cloned the TensorRT repository to the system.

```
git clone https://github.com/wang-xinyu/tensorrtx
```

TensorRTx is used to convert your PyTorch model to TensorRT engine model. We firstly converted the .pt weights of the YoloV7 to the .wts file using the gen_wts.py. The generated .wts file is then converted to .engine file. This engine file is used for the testing and detections.

4.2 Testing using images and videos

We tested the images and videos using the tensortRT and observed that the FPS is increased to 2FPS.

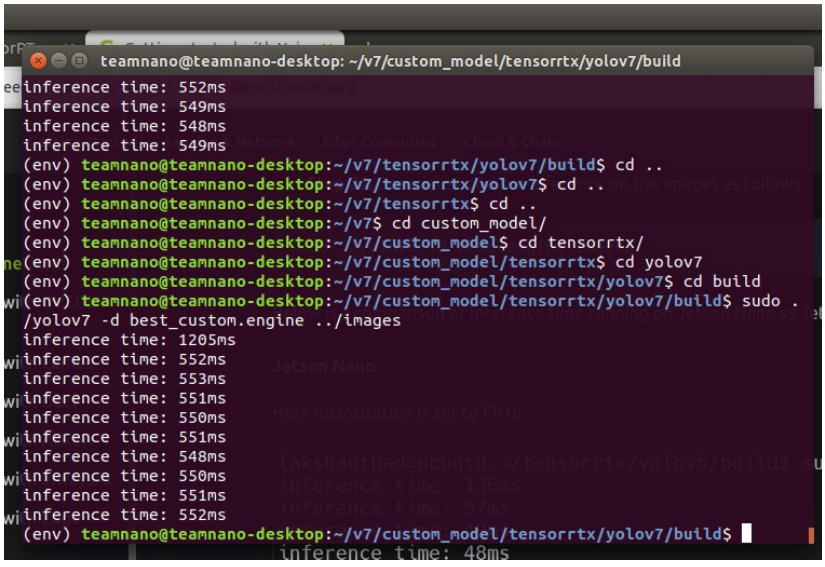

Figure 23: Inference time for images

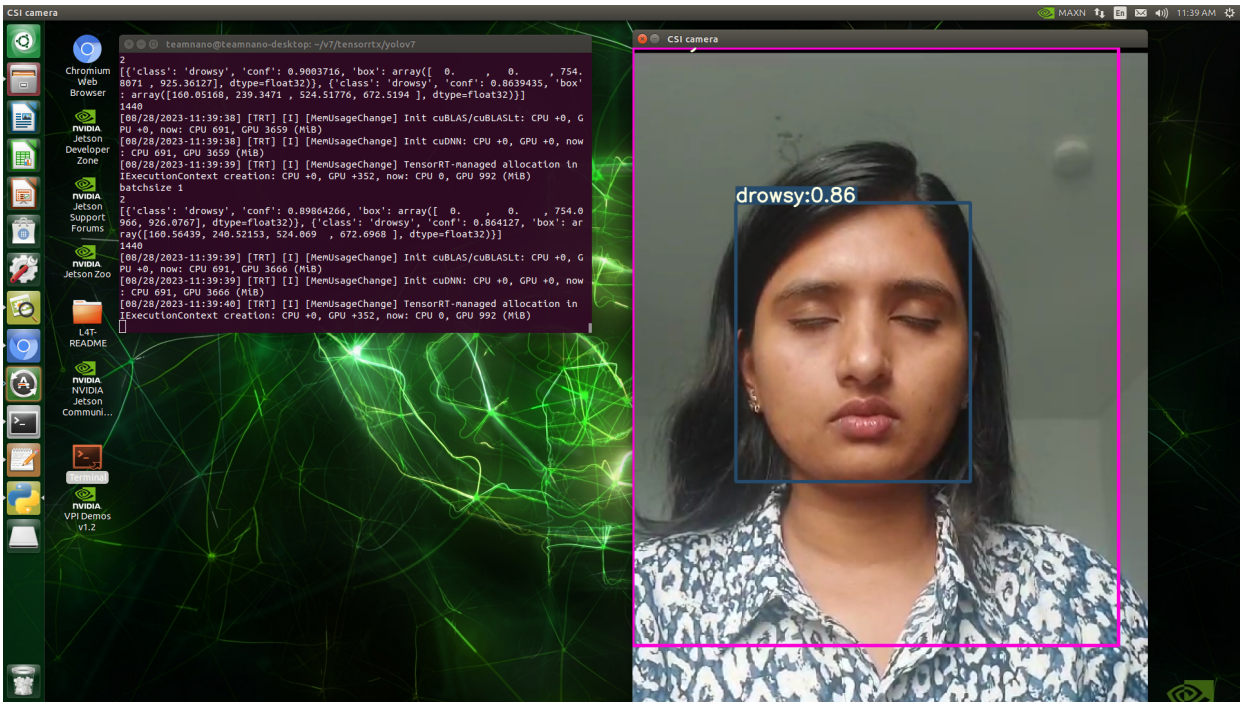

Figure 24: Video drowsiness detection

Figure 25: Traffic sign detection

4.3 Testing using live camera

We did the real-time processing for drowsiness detection.

- Starts the camera stream
- Continuously captures frames
- Sends the frames to the tensorRT model
- Doing the pre-processing of the captured images
- If the detection is observed drowsiness/ awake, it draws the corresponding bounding box of the original picture
- Returns [class, cropped picture / size of bounding box]

We observed that the real-time processing speed is low with a prediction accuracy between 80%-95%.

The source code:

```
def show_camera():
    window_title = "CSI Camera"
    model = YoloTRT(library="build/libmyplugins.so", engine="build/best.engine", conf=0.5, yolo_ver="v7")
    # To flip the image, modify the flip_method parameter (0 and 2 are the most common)
    print(gstreamer_pipeline(flip_method=2))
    video_capture = cv2.VideoCapture(gstreamer_pipeline(flip_method=2), cv2.CAP_GSTREAMER)
    if video_capture.isOpened():
        try:
            window_handle = cv2.namedWindow(window_title, cv2.WINDOW_AUTOSIZE)
            while True:
                ret_val, frame = video_capture.read()
                detections, t = model.Inference(frame)
                frame = imutils.resize(frame, width=600)
                # Check to see if the user closed the window
                # Under GTK+ (Jetson Default), WND_PROP_VISIBLE does not work correctly. Under Qt it does
                # GTK - Substitute WND_PROP_AUTOSIZE to detect if window has been closed by user
                if cv2.getWindowProperty(window_title, cv2.WND_PROP_AUTOSIZE) >= 0:
                    cv2.imshow(window_title, frame)
                else:
                    break
                keyCode = cv2.waitKey(1)
                # Stop the program on the ESC key or 'q'
                if keyCode == 27 or keyCode == ord('q'):
                    break
        finally:
            video_capture.release()
            cv2.destroyAllWindows()
    else:
        print("Error: Unable to open camera")


if __name__ == "__main__":
    show_camera()
```

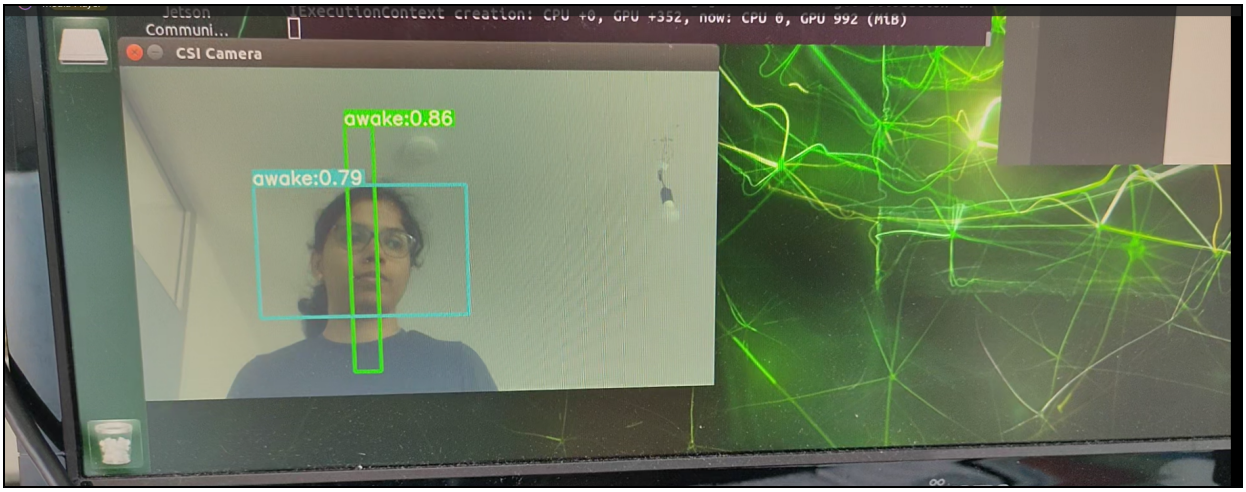

Figure 26: Real-time drowsiness detection(awake state)

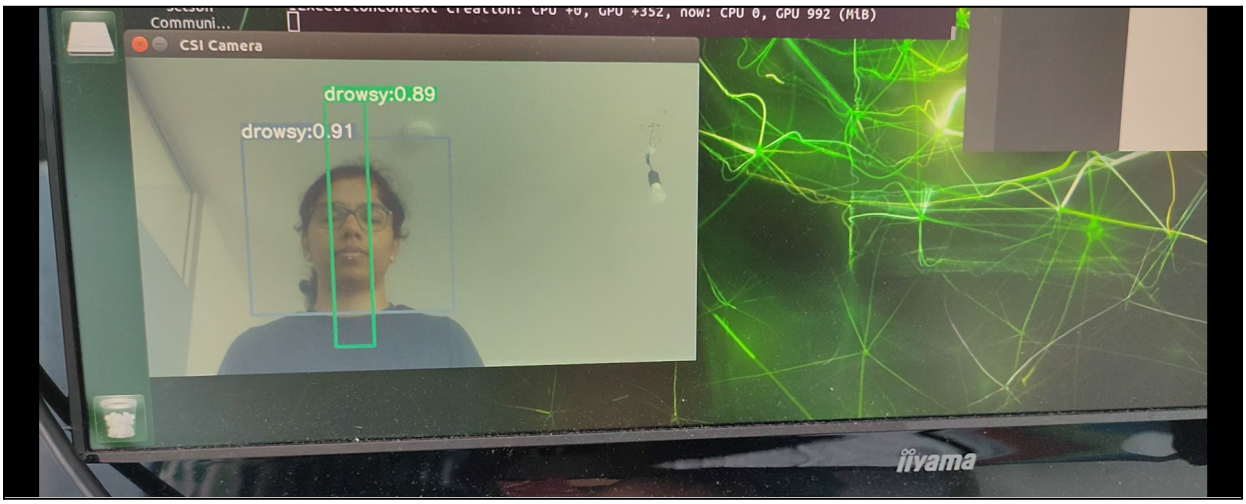

Figure 27: Real-time drowsiness detection(drowsy state)

4.4 Parallel processing of two cameras

For testing both the models(driver assistance system and driver monitoring system), live feed from both the cameras are taken and processed.Multithreading is used to run the inference on each camera feed independently.

During our testing phase, we observed that running resource-intensive tasks concurrently on the Jetson Nano led to power consumption levels that exceeded the device's capacity. As a result, the system experienced performance throttling to prevent overheating and maintain power efficiency. Given the limited power available, the execution of parallel tasks placed excessive demand on the device's resources, caused unexpected slowdowns, degraded inference performance, and an automatic rebooting.

This power-related limitation hindered our ability to assess the full potential of parallel processing.

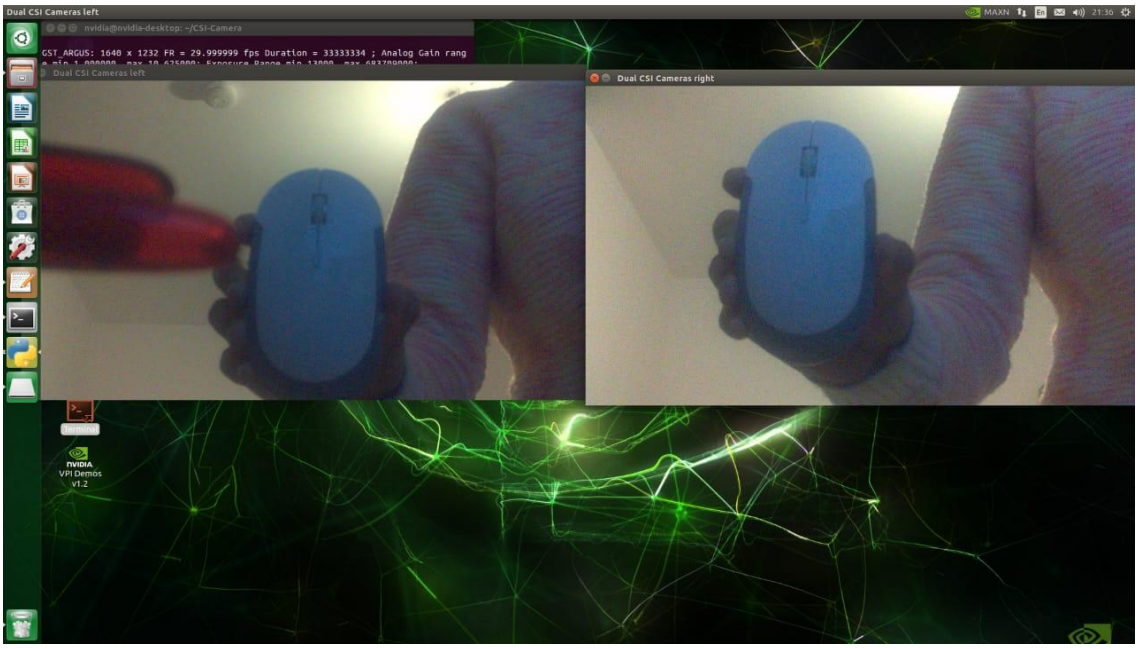

Figure 28: Dual camera working

5. Conclusion

In this project, we embarked on the development of a comprehensive driver assistance system designed to enhance road safety and prevent accidents by actively assisting the driver. Our approach involved the integration of advanced AI technologies, specifically YOLOv7 models, and the utilization of TensorRT to achieve high-performance processing. The primary objectives were to alert the driver about road signals and ensure real-time driver monitoring to mitigate risks associated with driver fatigue.

Through rigorous efforts, we successfully built and deployed two distinct models based on the YOLOv7 architecture. These models were meticulously fine-tuned and optimized to accurately detect road signals and monitor the driver's state. Leveraging the power of TensorRT, we were able to significantly enhance the inference speed of these models.

The integration of YOLOv7 and TensorRT proved to be pivotal in achieving real-time performance without compromising accuracy. YOLOv7's object detection capabilities were harnessed to identify road signs, and through the TensorRT conversion, we maximized the system's efficiency, making it suitable for real-world deployment.

In conclusion, our project successfully demonstrated the potential of advanced AI technologies, YOLOv7 and TensorRT, in creating an effective driver assistance system and monitoring system. By merging real-time object detection and driver monitoring, we have taken a significant step towards safer roads and accident prevention.

6. Possible project extensions

Drowsiness Detection: The Drowsiness Detection system can be further enhanced and expanded in the following ways:

1. Real-time Feedback and Alerts: Implement more intuitive and effective ways of alerting the driver, such as haptic feedback on the steering wheel or seat.
2. Integration with Autonomous Systems: Integrate drowsiness detection into autonomous vehicles to ensure that the system can take control when the driver becomes drowsy.

Traffic Sign Detection: Traffic Sign Detection system can be expanded and improved through the following avenues:

1. Multi-Sign Detection: Extend the system to detect multiple signs in a single frame, allowing for a more comprehensive understanding of the traffic environment.
2. Real-time Traffic Analysis: Integrate the detected traffic sign data with mapping and navigation systems to provide real-time information to the driver about upcoming road conditions and regulations.
3. Sign-State Prediction: Develop algorithms that can predict the state of dynamic traffic signs, such as variable speed limit signs, based on contextual information.

Parallel Processing (Sensor fusion) can be improved using **DeepStream SDKs**:

For better performance and efficiency, consider using NVIDIA's DeepStream SDK, which is designed for real-time AI inference on Jetson devices and includes features for working with multiple cameras and models.
Additionally, optimizing the models and algorithms for efficiency and resource utilization may help mitigate the effects of power throttling which hindered the parallel processing ability.

References

- https://wiki.seeedstudio.com/YOLOv5-Object-Detection-Jetson/
- https://medium.com/@jurespeh/yolov7-with-tensorrt-on-jetson-nano-with-python-script-example-63099fa7c8a5
- https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit
- https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6
- https://viso.ai/deep-learning/yolov7-guide/
- https://github.com/WongKinYiu/yolov7
- https://docs.edgeimpulse.com/docs/tutorials/end-to-end-tutorials/object-detection/object-detection
- https://github.com/ultralytics/yolov5
- https://github.com/JetsonHacksNano/CSI-Camera
- https://www.kaggle.com/datasets/valentynsichkar/traffic-signs-dataset-in-yolo-format