INTERNSHIP REPORT

*A report submitted to **MOTHERSON TECHNOLOGIES**,*
CHENNAI in partial fulfilment of the requirements for the Award of Degree
of

BACHELOR OF ENGINEERING
In

Electronics and Communication Engineering
Submitted by,

1. DHANUSH KUMAR V
2. RAMANAKRISHNAN A
3. SUVARNA LAKSHMI S
4. DEEPIKA R

MOTHERSON TECHNOLOGIES

Tower B, 3rd floor, RATTHA TEK MEADOWS,
No: 51, Rajiv Gandhi Salai, OMR, Sholinganallur,
Chennai, Tamil Nadu 600119

Sri Sai Ram Engineering College

(An Autonomous Institution)

1. Sam Kit

The SAM9X60-EK Evaluation Kit is excellently suited for the purpose of assessing and initial testing with the high-performance, extremely low-power SAM9X60 ARM926EJ-S based microprocessor (MPU), capable of running at speeds of up to 600 MHz. The SAM9X60 MPU boasts robust peripherals for facilitating connectivity and user interface applications, alongside offering advanced security functions.

Utilizing connectors and expansion headers, it enables straightforward personalization and swift access to cutting-edge embedded features such as Mikroelektronica click boards and Raspberry Pi expansion header. Comprehensive support is provided by the kit through both mainline Linux distribution and fundamental software frameworks, as well as real-time operating systems (RTOS), allowing for seamless initiation of your development endeavors. Moreover, the kit is inclusive of an integrated on-board Embedded Debugger, negating the necessity for external tools in programming or debugging. We have acquainted ourselves with the details of the SAM 9X60 Evaluation kit to enrich our prior understanding of the project.

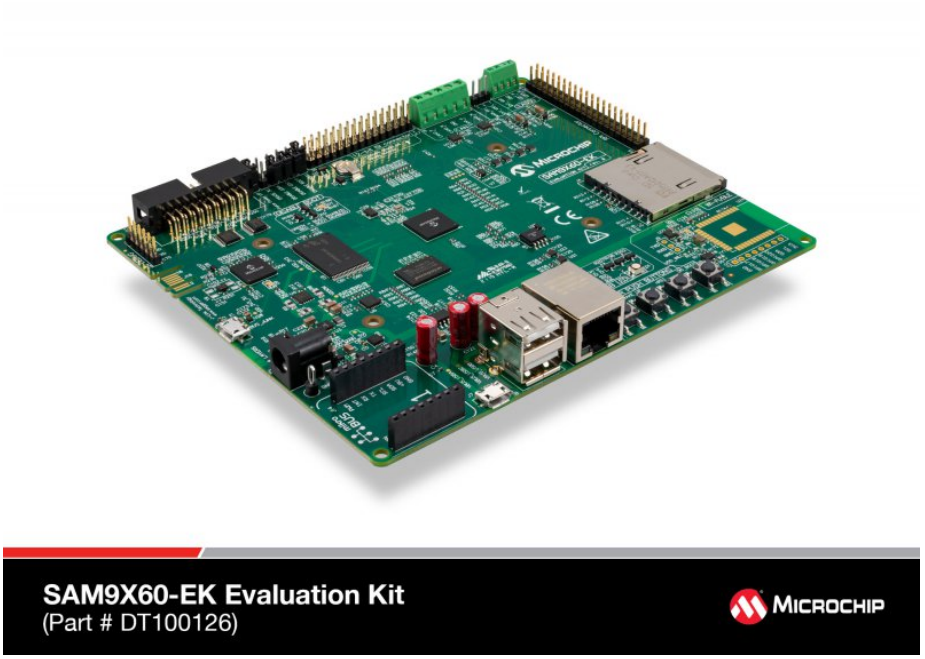


Figure 1.1

2. Linux:

Linux is a Unix-like, open source and community-developed operating system (OS) for computers, servers, mainframes, mobile devices and embedded devices. It is supported on almost every major computer platform, including x86, ARM and SPARC, making it one of the most widely supported operating systems. Every version of the Linux OS manages hardware resources, launches and handles applications, and provides some form of user interface. The enormous community for developers and wide range of distributions means that a Linux version is available for almost any task, and Linux has penetrated many areas of computing.

For example, Linux has emerged as a popular OS for web servers such as Apache, as well as for network operations, scientific computing tasks that require huge compute clusters, running databases, desktop and endpoint computing, and running mobile devices with OS versions like Android.

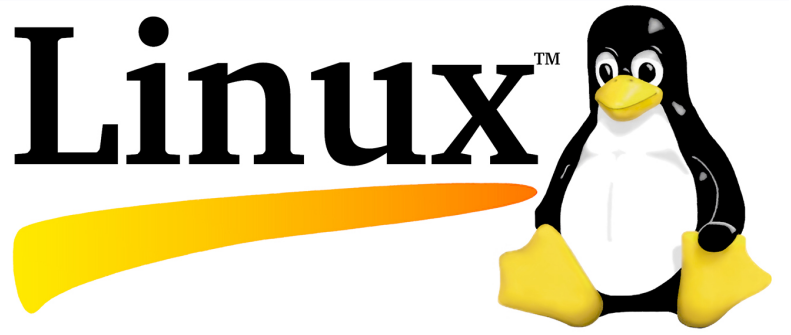


Figure 2.1

We used Ubuntu 16.04 to customize the operating system for Renesas. We had practiced some of the linux commands in the terminal. The list of commands are given below:

ls - The most frequently used command in Linux to list directories
pwd - Print working directory command in Linux
cd - Linux command to navigate through directories
mkdir - Command used to create directories in Linux
mv - Move or rename files in Linux
cp - Similar usage as mv but for copying files in Linux
rm - Delete files or directories
touch - Create blank/empty files
ln - Create symbolic links (shortcuts) to other files
cat - Display file contents on the terminal
clear - Clear the terminal display
echo - Print any text that follows the command
less - Linux command to display paged outputs in the terminal
man - Access manual pages for all Linux commands
uname - Linux command to get basic information about the OS
whoami - Get the active username

tar - Command to extract and compress files in Linux
grep - Search for a string within an output
head - Return the specified number of lines from the top
tail - Return the specified number of lines from the bottom
diff - Find the difference between two files
cmp - Allows you to check if two files are identical
comm - Combines the functionality of diff and cmp
sort - Linux command to sort the content of a file while outputting
export - Export environment variables in Linux
zip - Zip files in Linux
unzip - Unzip files in Linux
ssh - Secure Shell command in Linux
service - Linux command to start and stop services
ps - Display active processes
kill and killall - Kill active processes by process ID or name
df - Display disk filesystem information
mount - Mount file systems in Linux
chmod - Command to change file permissions
chown - Command for granting ownership of files or folders
ifconfig - Display network interfaces and IP addresses
traceroute - Trace all the network hops to reach the destination
wget - Direct download files from the internet
ufw - Firewall command
iptables - Base firewall for all other firewall utilities to interface with
apt, pacman, yum, rpm - Package managers depending on the distro
sudo - Command to escalate privileges in Linux
cal - View a command-line calendar
alias - Create custom shortcuts for your regularly used commands
dd - Majorly used for creating bootable USB sticks
whereis - Locate the binary, source, and manual pages for a command
whatis - Find what a command is used for
top - View active processes live with their system usage
useradd and usermod - Add new user or change existing users data
passwd - Create or update passwords for existing users

3. Buildroot:

Buildroot is an open-source project aimed at simplifying the process of creating embedded Linux systems. It provides a set of tools and configuration files that allow developers to generate custom Linux distributions tailored for specific hardware platforms. This flexibility is particularly valuable in embedded systems, where resource constraints and specialized requirements often demand optimized and minimalistic operating systems.

Customization: Buildroot allows developers to customize every aspect of their embedded Linux system, from the kernel configuration to the selection of software packages. This ensures that the resulting system is precisely tuned for the intended application and hardware.

Package Management: Buildroot employs a package-based approach to software management, making it easy to add, remove, or update packages in the embedded system. This enables developers to include only the necessary components, reducing the system's footprint and improving performance.

Cross-Compilation: Embedded systems usually target hardware architectures different from the development machine. Buildroot supports cross-compilation, allowing the build process to generate binaries that are suitable for the target platform.

Filesystem Generation: Buildroot generates a root filesystem image containing all the selected packages and libraries. This image can be loaded onto the target hardware, creating a functional embedded Linux environment.

Toolchain Management: Buildroot generates a toolchain that includes compilers, libraries, and other necessary tools for building software for the target architecture. This ensures consistency and compatibility across the system.

Kernel Configuration: Developers can configure the Linux kernel through Buildroot's menu-driven interface. This makes it easy to enable or disable specific kernel features, drivers, and modules.

Ease of Use: Buildroot simplifies the complex process of building an embedded Linux system by providing a user-friendly interface and automating many tasks that would otherwise be time-consuming and error-prone.
Configuration: Developers start by configuring the Buildroot system using its text-based configuration files. This involves specifying the target hardware architecture, selecting required software packages, and defining system settings.

Build Process: After configuring, running the Buildroot build process compiles the cross-toolchain, kernel, bootloader, root filesystem, and other selected components. The result is an integrated filesystem image.

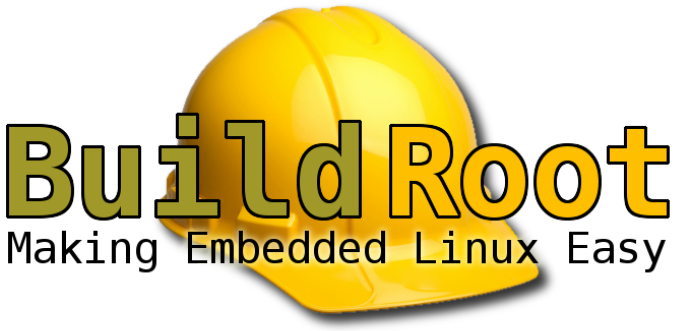


Figure 3.1

Deployment: The generated filesystem image can be deployed onto the target hardware, either through storage media (e.g., SD cards) or network booting methods.

But here we won't use the buildroot for the project given. Because of its limited configurations and some of the drawbacks. Setting up Buildroot for the first time can be more involved compared to using pre-existing distributions. The process involves configuring various aspects of the system, which might require trial and error for newcomers. Identifying and diagnosing issues in custom-built systems can be more challenging compared to using established, off-the-shelf distributions. Debugging might require deep knowledge of both Linux internals and the Buildroot build process.

4. RENESAS RZN1D :

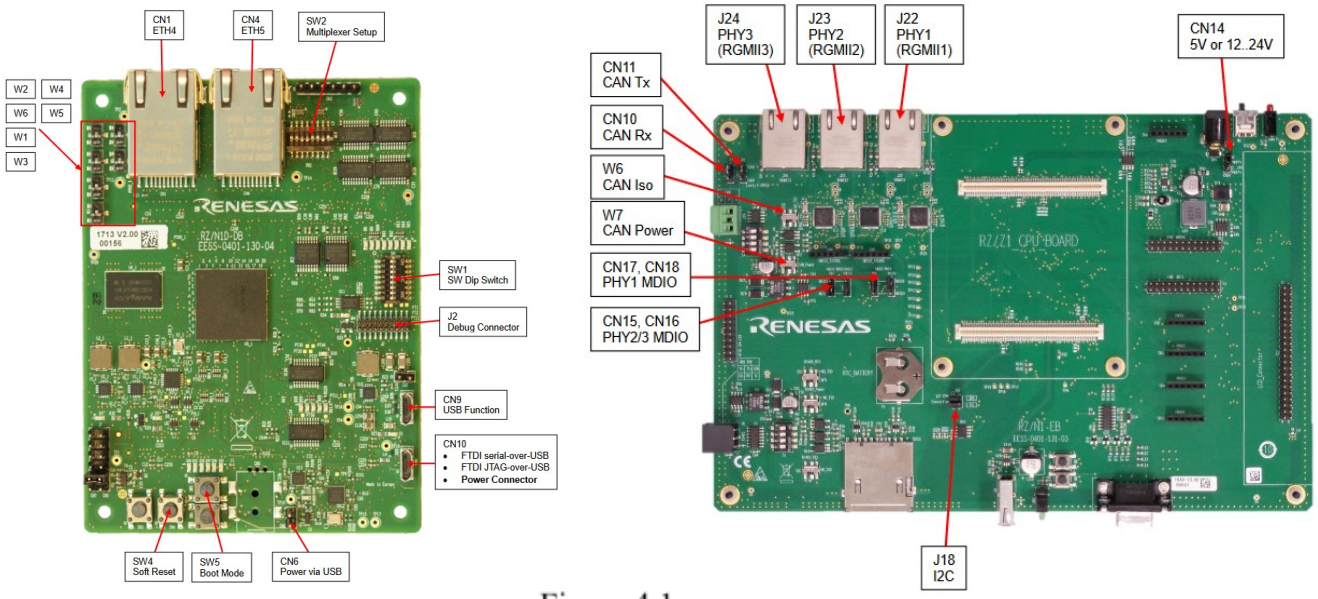


Figure 4.1

RZN1D Microcontroller: A Comprehensive Overview

Introduction:

The RZN1D microcontroller is an advanced system-on-chip (SoC) solution engineered to provide a robust platform for a diverse range of applications. Combining powerful processing cores, versatile memory interfaces, extensive connectivity options, and advanced real-time Ethernet capabilities, the RZN1D microcontroller is designed to excel in industrial automation, networking, and embedded systems scenarios.

Processor Cores:

At the heart of the RZN1D microcontroller lie two essential processor cores, each offering distinct advantages:

Arm Cortex-A7 MPCore: This 32-bit processor core delivers exceptional performance with clock speeds of up to 500 MHz. It offers the flexibility to be configured as a single core or a dual core, enabling optimized resource allocation for specific application requirements. The core's incorporation of a Floating Point Unit (FPU) ensures precision in numerical computations, while the VFPv4-D16 support enhances floating-point operations. The Memory Management Unit (MMU) empowers efficient memory usage, and the L1 cache of 16 KB for instruction and data, per core, accelerates data access. The microcontroller can further leverage an L2 cache of up to 256 KB for enhanced data handling.

Arm Cortex-M3 Processor: Operating at clock speeds of up to 125 MHz, this 32-bit processor core introduces a new level of security with its Memory Protection Unit (MPU). The MPU isolates different processes, safeguarding the system from unauthorized access and enhancing overall security.

Low Power Features:

Efficiency in power consumption is crucial for modern electronic devices. The RZN1D microcontroller offers the following features for optimized energy usage:

Clock Gating Management: This feature enables the microcontroller to selectively disable clock signals to specific components when they are not in use, effectively reducing power consumption during idle periods.

Clock Frequency Scaling: The microcontroller can dynamically adjust its clock frequency based on processing demands. This capability ensures that power is allocated efficiently, minimizing energy consumption while maintaining performance.

Memory and Data Handling:

Memory capacity and data integrity are paramount in various applications. The RZN1D microcontroller addresses these requirements effectively:

On-Chip Extended SRANM: With the potential to accommodate up to 6 MB of data, the Extended SRAM acts as a versatile memory buffer. The integrated Error Correction Code (ECC) mechanisms work tirelessly to detect and rectify errors, ensuring that data remains reliable and accurate.

Data Transfer and DMAC: The presence of two Direct Memory Access Controllers (DMAC), each equipped with eight channels, streamlines the process of transferring data within the microcontroller. This feature is especially valuable when dealing with significant volumes of data that need to be moved efficiently.

Memory Interfaces:

Efficient memory access and storage are pivotal for optimal system performance:

Quad SPI/XIP: Supporting up to two Quad Serial Peripheral Interface (SPI) configurations, the microcontroller facilitates high-speed data exchange and code execution directly from external memory sources.

NAND Flash with ECC: Advanced Error Correction Code (ECC) management is seamlessly integrated with NAND Flash memory, reinforcing data integrity and minimizing errors.

16-bit DDR Interface: The microcontroller caters to memory-intensive tasks with support for 16-bit Double Data Rate (DDR) interfaces, capable of achieving speeds of DDR2-500 and DDR3-1000. This feature ensures efficient and high-speed memory access.

SD/SDIO/eMMC Interfaces: Up to two Secure Digital (SD), SDIO, or eMMC interfaces provide versatile options for data storage and communication, catering to a wide range of requirements.

Connectivity and Peripherals:

Peripheral devices and connectivity options enhance the microcontroller's versatility and ability to interact with external systems:

CPU Resources: The microcontroller's CPU resources encompass a variety of components. The Mailbox facilitates efficient inter-core communication. Timer blocks, comprising both 16-bit × 6 channel and 32-bit × 2 channel configurations, enable precise timing tasks. The 16-channel PWMTimer supports waveform generation, vital for applications requiring analog-like signals. Dedicated Watchdog timers for each CPU core enhance system reliability, while Semaphore mechanisms facilitate synchronization between different processes.

General Connectivity: Connectivity options abound in the RZN1D microcontroller, ensuring seamless communication with external devices. The USB2.0 Host interface allows connections with USB devices for efficient data exchange. The USB2.0 Host & Function interface enhances versatility by enabling the microcontroller to function as both a host and a USB device. Eight UART interfaces facilitate asynchronous serial communication, while the six SPI interfaces (with configurations for four masters and two slaves) offer comprehensive serial communication capabilities. Two I2C interfaces allow efficient communication with compatible devices, while two CAN interfaces empower automotive and industrial applications. The microcontroller supports up to two 12-bit ADCs operating at speeds of up to 1 MSPS, enabling accurate analog data acquisition. The inclusion of a Parallel Bus Interface (MSEBI) further extends connectivity options, enabling efficient parallel data communication.

Conclusion:

The RZN1D microcontroller emerges as an exceptional technological achievement. Its amalgamation of potent processing cores, versatile memory interfaces, extensive connectivity options, and advanced real-time Ethernet capabilities positions it as a catalyst for innovation across various industries. Whether applied to industrial automation, networking solutions, or embedded systems, the RZN1D microcontroller offers a transformative solution that combines performance, adaptability, and connectivity in the ever-evolving landscape of technology.

5. YOCTO PROJECTS

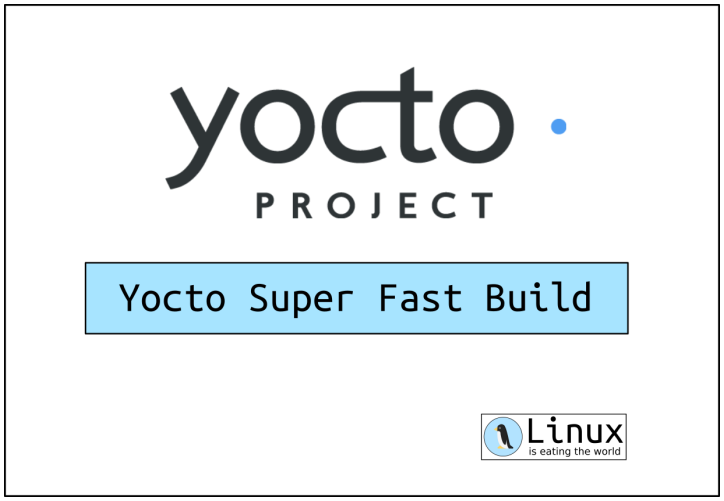


Figure 5.1

Introduction:

The Yocto Project is a groundbreaking open-source initiative that empowers developers to create tailored Linux-based systems for embedded devices. Designed to streamline the process of constructing and managing customized Linux distributions, the Yocto Project provides a comprehensive toolkit and framework that enables developers to focus on application development and features, rather than getting bogged down in the intricacies of configuring and building software stacks. This detailed overview delves into the key components, concepts, and benefits of the Yocto Project.

Key Components and Concepts:

1. OpenEmbedded Core:

At the heart of the Yocto Project lies the OpenEmbedded Core, which furnishes the essential metadata, build scripts, and tools required for crafting custom Linux distributions. This foundational component acts as the blueprint for defining layers, recipes, and configuration files that shape the architecture of the target system.

2. Layers:

A core organizational structure of the Yocto Project is the concept of layers. Layers encapsulate distinct functionalities and elements. They can contain recipes, configuration files, and other resources that define software components. The ecosystem includes both **core layers** and user-defined custom layers, enabling developers to build upon existing foundations or create new layers to suit specific requirements.

3. Recipes:

A pivotal element of the Yocto Project is the recipe. Recipes elucidate the intricate process of building and packaging software components. They encompass source locations, compilation steps, dependencies, and outputs. Crafted in the BitBake language, recipes provide the instructions needed to produce target artifacts effectively.

4. BitBake:

BitBake serves as the engine that processes recipes, orchestrating the construction of output artifacts like binaries, libraries, and images. This powerful tool masterfully handles cross-compilation complexities, dependency management, and parallel builds, eliminating the need for developers to tackle these intricacies manually.

5. Metadata:

Metadata entails an amalgamation of configuration files, recipe documents, and associated resources. Organized within layers, metadata plays a pivotal role in establishing how software components are constructed. This modular approach promotes configurability, reusability, and ease of maintenance.

6. Poky:

Poky stands as the Yocto Project's reference distribution, offering a foundational layer, recipes, and configurations that serve as a launchpad for designing personalized distributions. Many developers utilize Poky as a starting point, subsequently tailoring it to meet specific demands.

7. BitBake Recipes:

BitBake recipes are instrumental scripts that meticulously define the process of building individual software components. They encompass source locations, build commands, configuration settings, and more. Recipes are a fundamental and integral aspect of the Yocto Project's build ecosystem.

8. Cross-Compilation:

The Yocto Project facilitates cross-compilation, an essential capability for constructing software destined for target architectures differing from the developer's machine. This feature is of paramount importance for resource-constrained embedded systems.

9. Layers Index:

Fostering a collaborative environment, the Yocto Project maintains a Layers Index repository. This repository hosts a plethora of publicly accessible layers contributed by the community. Developers can seamlessly search for and integrate layers that furnish specific functionalities or cater to specific hardware platforms.

10. BitBake Tasks:

In the BitBake-driven environment, tasks are individual steps that contribute to the comprehensive process of building a software component. These tasks can encompass fetching source code, applying patches, compiling binaries, and packaging artifacts.

11. SDK (Software Development Kit):

The Yocto Project generates SDKs that furnish developers with a dedicated environment for crafting applications that seamlessly integrate with the target system. SDKs encapsulate libraries, headers, and tools necessary for effective cross-compilation.

12. Images:

The ultimate outcome of the Yocto Project's endeavors is the creation of images. These images constitute complete filesystems that execute on the target hardware. The Yocto Project facilitates the crafting of diverse image types, encompassing root filesystems, system images, and customized images tailored to particular use cases.

Conclusion:

In the realm of embedded systems, the Yocto Project emerges as a groundbreaking force, democratizing the creation of customized Linux distributions. By blending powerful processor cores, versatile memory interfaces, robust connectivity, and the magic of the BitBake engine, the Yocto Project has revolutionized the landscape of embedded software development. Offering both developers and industries an unparalleled degree of flexibility, the Yocto Project empowers the creation of tailored systems that are optimized for performance, security, and functionality, while also promoting collaboration and knowledge sharing within the vibrant open-source community.

Build custom OS for RENESAS RZN1D using YOCTO project:

Yocto Environment Setup and Configuration for RZ/N1 Platform

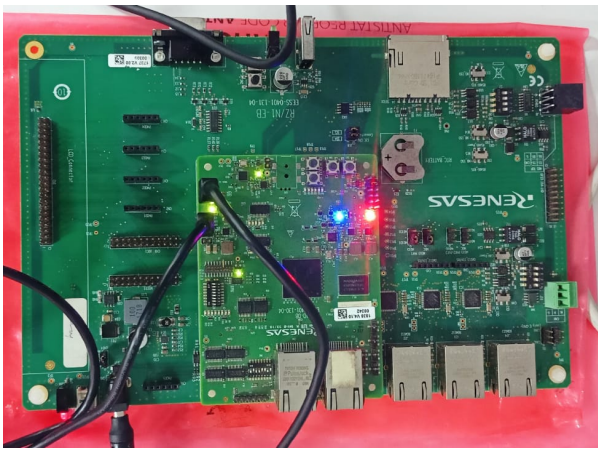


Figure 5.2

1. Initial Setup:

To initiate the Yocto environment setup, the Yocto Project repository was cloned using the following commands:

git clone http://git.yoctoproject.org/git/poky

cd poky
git checkout -b rocko-rzn1 yocto-2.4.3

These steps ensured the correct release version and branch were selected.

2. Applying Patches:
The RZ/N1 platform necessitates patches to the Rocko release to enable VPFv4d16 floating-point coprocessor support. The patches were applied as follows:

git am ${RELEASE_DIR}/yocto/rocko/0001-ARM-Add-Cortex-A7-vfpv4-d16*.patch
cd ..

This step ensured the platform's specific requirements were integrated seamlessly.

3. OpenEmbedded Recipes:
For software package integration, the OpenEmbedded recipes for the Rocko release were cloned:

git clone -b rocko git://git.openembedded.org/meta-openembedded

This allowed for a comprehensive range of software components.

4. Read-Only and Overlay File Systems:
To support a read-only root file system with an overlay read-write file system, the `meta-readonly-rootfs-overlay` layer was employed. The following steps achieved this:

git clone https://github.com/cmhe/meta-readonly-rootfs-overlay.git
cd meta-readonly-rootfs-overlay
git checkout -b rzn1 2a426495fe77330058bc0d6ef98e914649e7b415
cd ..

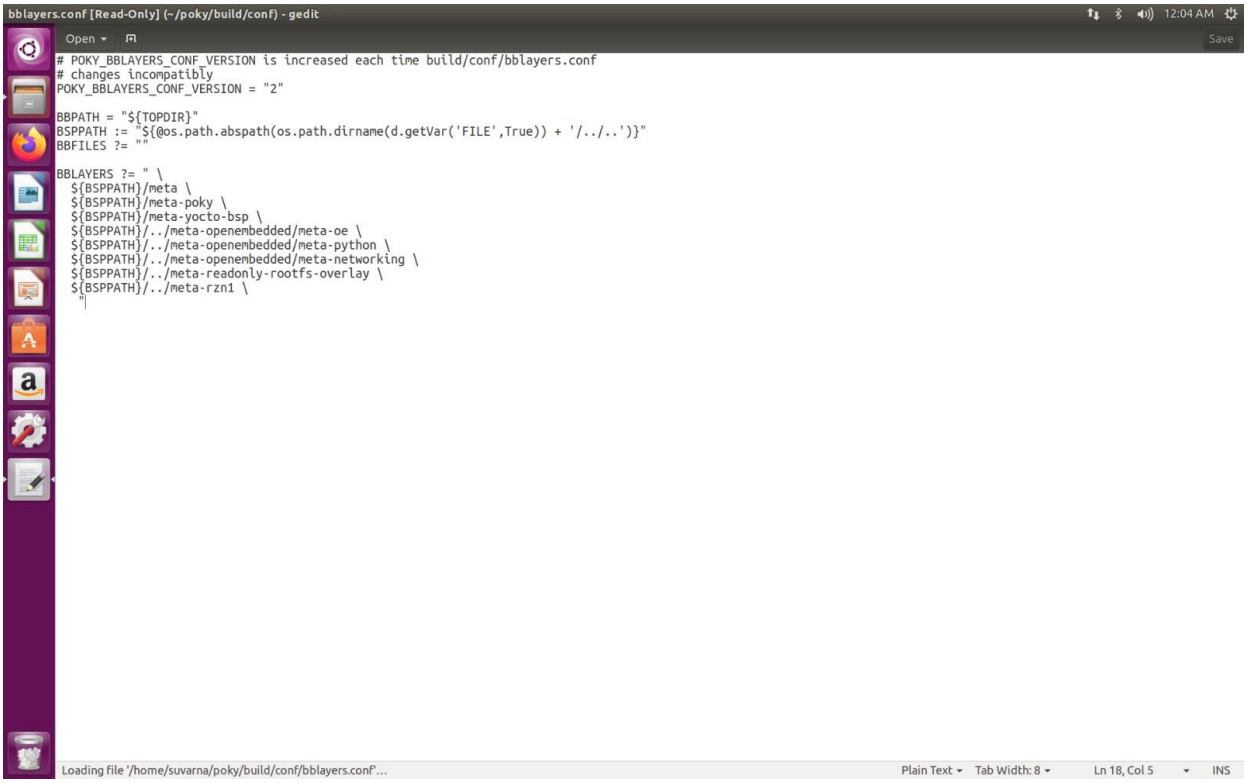


Figure 5.3

This ensured the desired file system configurations were attainable.

5. RZ/N1 Platform Support:
The `meta-rzn1` layer provided support for the RZ/N1 platform. The layer was cloned as follows:

git clone -b rocko-v4.19 https://github.com/renesas-rz/meta-rzn1.git

This step laid the foundation for tailored support to the platform.

6. Build Environment Configuration:
To establish the OpenEmbedded build environment, the following steps were taken:

cd poky
source oe-init-build-env

This ensured the necessary settings were initialized for the subsequent build steps.

7. Configuration Files:
The `bblayers.conf` and `local.conf` files were copied from `${RELEASE_DIR}/yocto` to `build/conf`. These files respectively defined the layers used in the build and build-specific settings:

cp ${RELEASE_DIR}/yocto/bblayers.conf build/conf/bblayers.conf
cp ${RELEASE_DIR}/yocto/local.conf build/conf/local.conf

Figure 5.4

This step facilitated the customization of the build environment.

NOTE: For adding layers in local.conf

Step 1: Go to local.conf and in IMAGE_INSTALL_append += “\
Add : zbar,python,opencv(required files) and save the file.

Step 2: In terminal first check if it is in build directory if not type:
cd poky
source oe-init-build-env

Step 3: Once it has entered the build directory type:
bitbake core-image-minimal

This will build the root file system and kernel

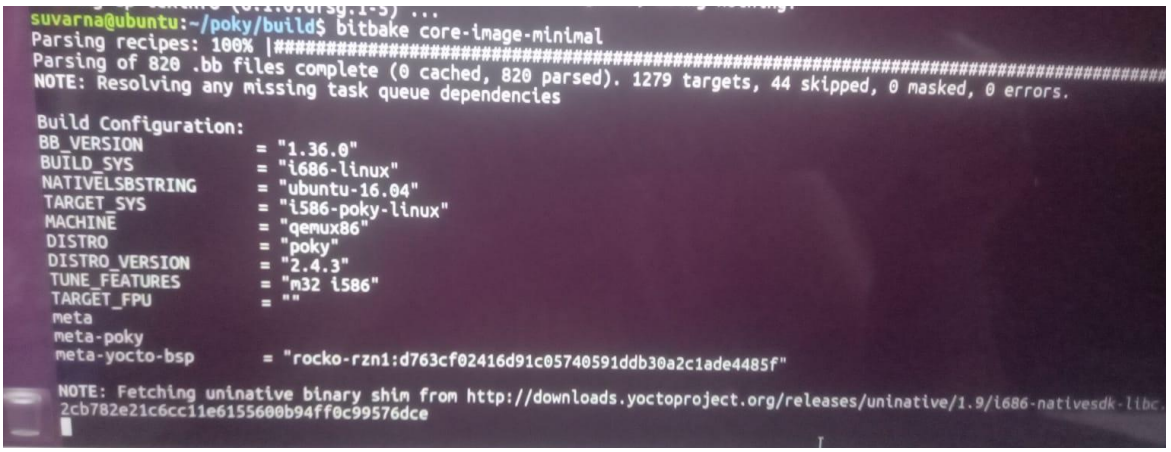


Figure 5.5

These comprehensive steps enabled the creation of a Yocto-based environment tailored to the RZ/N1 platform's requirements.

In conclusion, this report has outlined the systematic process of setting up and configuring the Yocto environment for the RZ/N1 platform using the Rocko release. Each step was meticulously executed to ensure optimal integration and customization, resulting in a well-tailored Yocto environment for the RZ/N1 platform's needs.

Flashing OS into RZN1D:

Report: Programming U-Boot, Writing Linux, and Configuring U-Boot Environment on RZ/N1 Board

This report presents a comprehensive guide for programming U-Boot onto the QSPI flash, writing Linux components to QSPI, and configuring U-Boot environment variables to facilitate automatic booting of the Linux kernel on the RZ/N1 board. The report is organized into three sections, each outlining the step-by-step procedures for the respective tasks.

Programming U-Boot to QSPI

This section details the process of programming U-Boot onto the QSPI flash using BootROM's DFU mode and U-Boot's dfu command.

1. Prerequisites:
Install the 'dfu-util' package on a Linux PC using the package manager. For Windows PC users, refer to the U-Boot User Manual for dfu-util installation instructions.

2. Switch to DFU Boot Mode:
On the RZ/N1 board, press switch SW5 and then switch SW4 (soft reset). The serial port output will confirm the DFU boot mode.

3. Download U-Boot to SRAM:
Use the host PC to download U-Boot in SPKG format to SRAM using the following command:

sudo dfu-util -D u-boot-rzn1d400-db.bin.spkg

4. U-Boot Console and Output:
Upon successful execution, U-Boot will run, and the serial port will display a console interface. Interrupt U-Boot's attempt to execute commands specified by bootcmd env variable by pressing any key.

5. Update Environment for Compatibility:

For compatibility with older U-Boot versions, address dfu_ext_info environment variable issues using:

env default -f dfu_ext_info
saveenv

6. Erase U-Boot/SPL Region:

Ensure the U-Boot/SPL region of the QSPI Flash is erased using the following commands:

sf probe
sf erase 0 10000

7. Initiate DFU in U-Boot:

Run ***`dfu`*** command on the U-Boot console.

8. Write U-Boot to QSPI:

Write U-Boot to QSPI using the following command on the host PC:

sudo dfu-util -a "sf_uboot" -D u-boot-rzn1d400-db.bin.spkg

Wait for completion; the U-Boot console will prompt you to press Ctrl-C when done.

9. Reset and Confirm U-Boot Execution:

Press switch SW4 to reset the board. BootROM will load and execute U-Boot, and the terminal will display the U-Boot console interface.

Writing Linux to QSPI

1. Initiate U-Boot and Run dfu:

Press switch SW4 on the board to trigger U-Boot execution, leading to a console interface.

2. Run dfu in U-Boot:

Execute the ***`dfu`*** command on the U-Boot console.

3. Write DTB to QSPI:

On the host PC, use the following command:

sudo dfu-util -a "sf_dtb" -D uImage-rzn1d400-db.dtb

4. Write Kernel to QSPI:

Run the following command on the host PC:

dfu

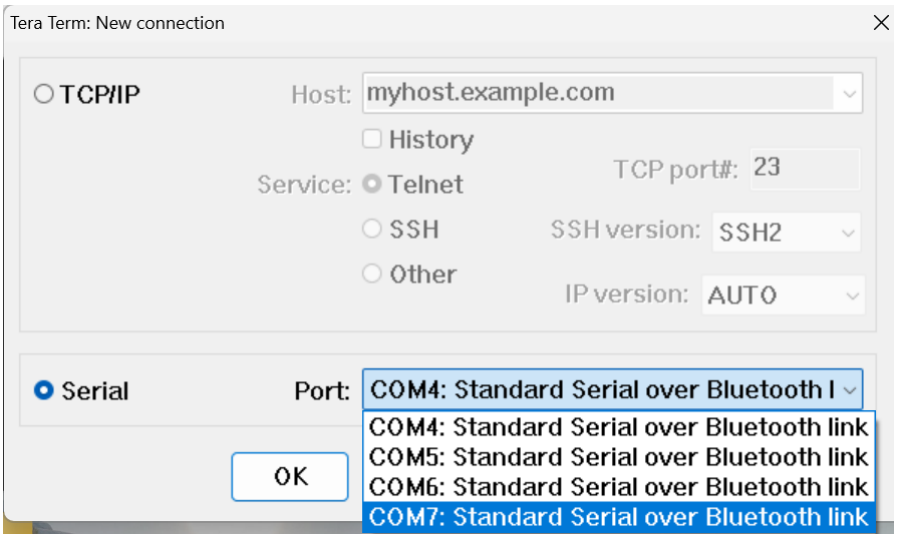


Figure 5.6

sudo dfu-util -a "sf_kernel" -D uImage

5. Write Rootfs to QSPI:

Execute the command below on the host PC:

sudo dfu-util -a "sf_data" -D core-image-minimal-rzn1.squashfs

6. Completion and U-Boot Prompt:

Wait for completion; the U-Boot console will prompt you to press Ctrl-C when done.

Setting Up U-Boot Environment Variables

This section provides instructions for configuring U-Boot environment variables to enable automated Linux kernel booting.

1. Set MAC Addresses:

From U-Boot, set MAC addresses corresponding to the board's MAC address sticker:

setenv -f ethaddr 74:90:50:02:00:FD
setenv -f eth1addr 74:90:50:02:00:FE

2. Define Linux Bootargs:

Set Linux boot arguments to include read-only rootfs, read-write JFFS2 file system, and a static IP address for GMAC1 in Linux:

setenv bootargs "console=ttyS0,115200 root=/dev/mtdblock7 init=/init rootwait ip=192.168.1.50:::::eth0 earlyprintk clk_ignore_unused"

3. Configure Boot Command:

Set the bootcmd to load the Cortex M3 image, DTB, and kernel from QSPI and start Linux:

setenv bootcmd "sf probe && sf read 0x4000000 d0000 80000 && rzn1_start_cm3 && sleep 4 && sf read 0x8ffe0000 b0000 20000 && sf read 0x80008000 1d

4. Save the environment variables:

Saveenv

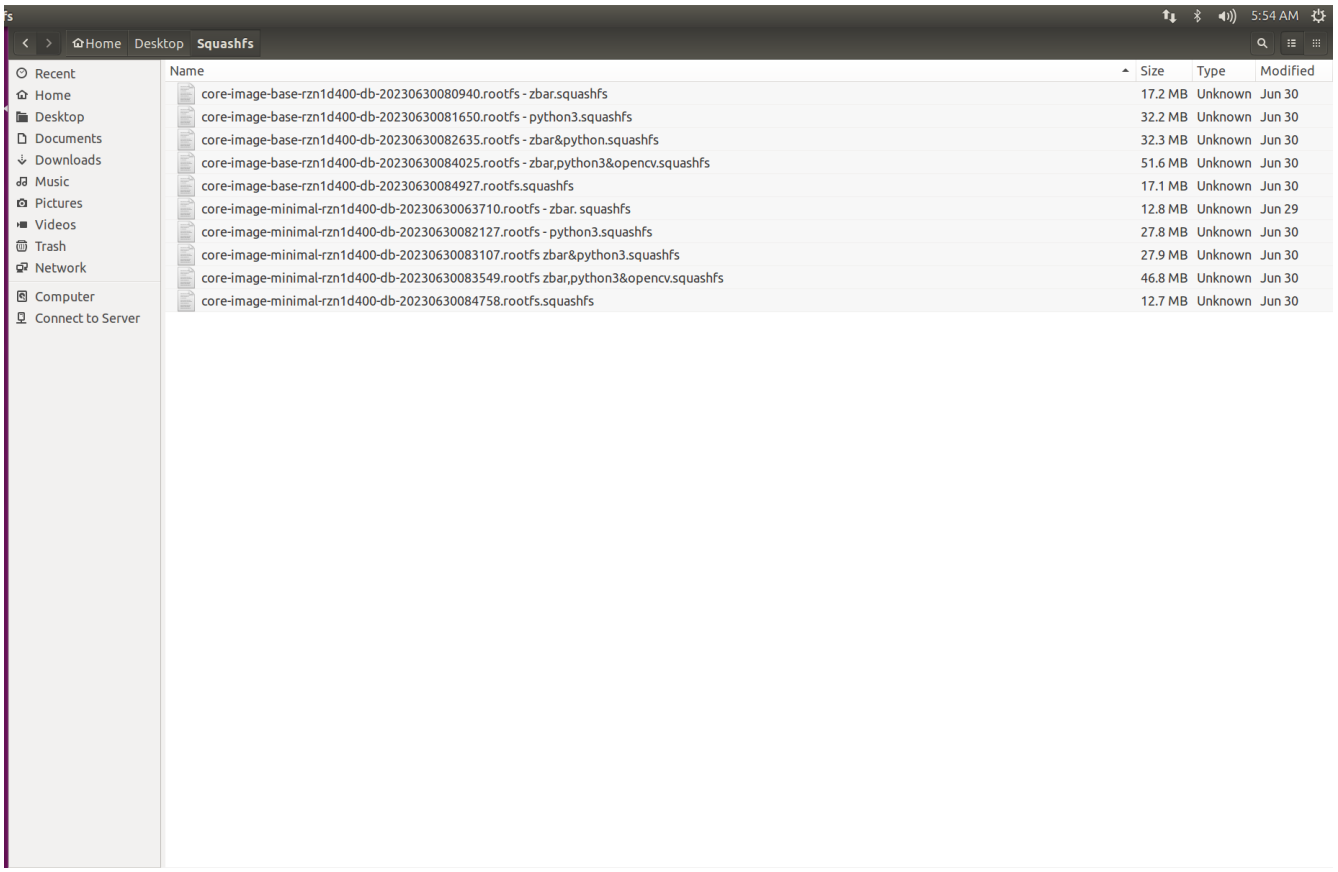


Figure 5.7

While we were trying to upload the required layers in local.conf we were unable to boot it properly. While analyzing the problem, we came to realize that the Renesas board has an internal memory of only 20 to 25MB.

While we tried uploading rootfs and zbar(min size file) files it booted properly since it has a memory size of only 17MB and 12MB, and while we tried uploading a python file we are unable to boot since it consumes a memory space of 32MB.

Hence we came to a conclusion that the internal memory space of Renesas board is only upto 25MB and hence it can’t serve our purpose.

6. MOBILE NET SSD – PRE-TRAINED

Source Code Link: https://github.com/nicknochnack/TFODCourse

Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos. It is used to detect objects in the real world. For example, dogs, cars, humans, birds etc. In this process we can detect the presence of any still object with much ease. Another great thing that can be done with it is that detection of multiple objects in a single frame can be done easily.

MobileNet is a class of well-organized models called for mobile and embedded vision applications. Mobilenet SSD is an object detection model that computes the output bounding box and object class from the input image. This Single Shot Detector (SSD) object detection model uses Mobilenet as a backbone and can achieve fast object detection optimized for mobile devices. This model uses COCO dataset for training the images. The COCO dataset consists of 80 classes of images, so the objects detected would belong to one of these classes only. Classes like person, traffic signs, animals like cats, dogs and vehicles like trucks, cars, motorcycles can be detected easily. Unlike many other object detection methods, it will be able to run this in real-time with a good amount of accuracy.

Code for Object Detection : It's essential to have a reasonably large and diverse dataset for effective training. The quality of annotations and the balance of classes in the dataset also play a crucial role in the model's performance.

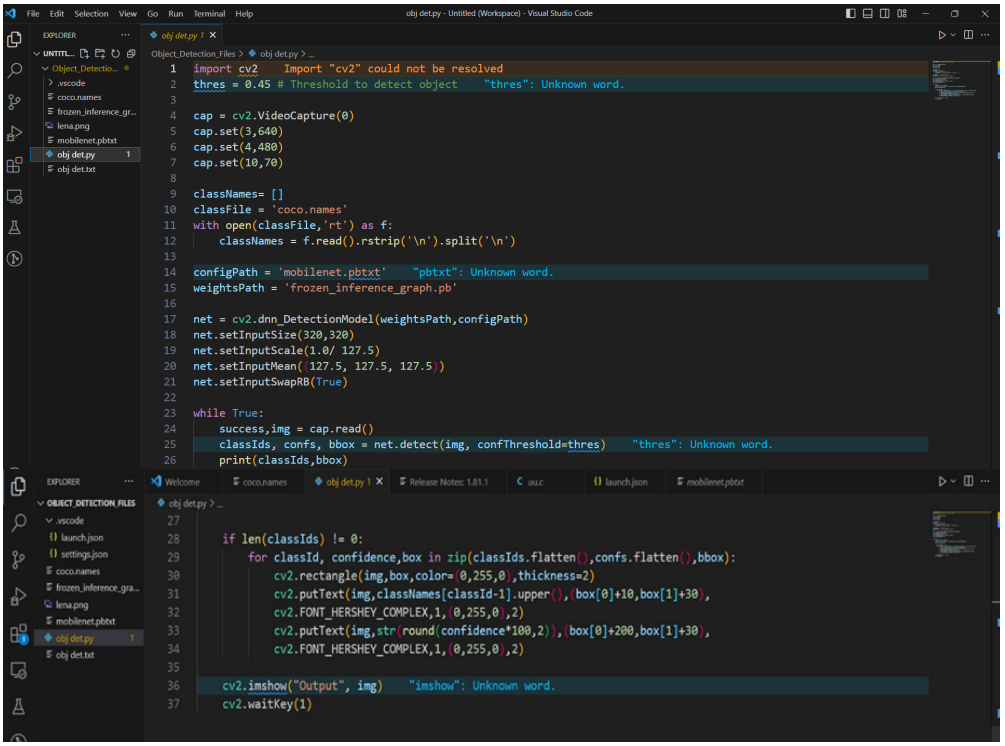


Figure 6.1

Output:

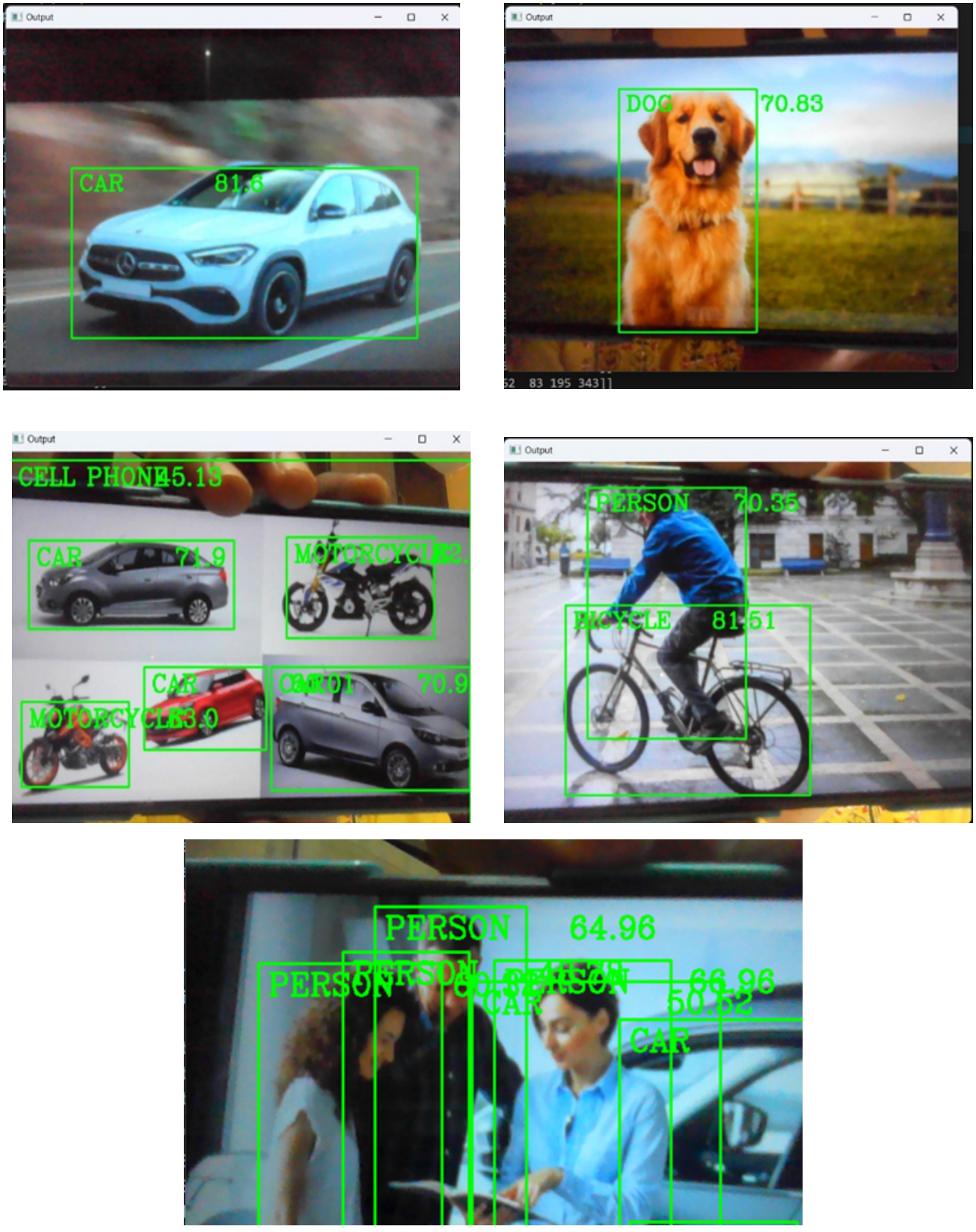


Figure 6.2

7. HAAR CASCADE - CUSTOMIZED

Haar Cascade is a machine learning-based object detection method used to identify objects in images or video frames. It was introduced by Viola and Jones in their 2001 paper titled "Rapid Object Detection using a Boosted Cascade of Simple Features." Haar Cascade is particularly effective for real-time object detection tasks and has been widely used for tasks such as face detection, pedestrian detection, and more.

How Haar Cascade Works:

The Haar Cascade method works by using a set of simple, rectangular features known as Haar-like features. These features are essentially rectangular boxes that are placed over a region of interest in an image. Each Haar-like feature calculates the difference between the sum of pixel values in white and black regions of the rectangle. This difference is used to determine whether the feature is present in that region.

The training process for Haar Cascade involves a machine learning algorithm, specifically the AdaBoost algorithm. The AdaBoost algorithm works by sequentially training a series of weak classifiers, where each classifier focuses on a particular Haar-like feature. These weak classifiers are combined into a strong classifier, which is used to make decisions about the presence of objects in an image.

The Haar Cascade training process involves the following steps:

1. **Positive and Negative Samples:** Positive samples are images containing the object you want to detect (e.g., faces), while negative samples are images without the object. These samples are used to train the algorithm.

2. **Feature Selection:** The algorithm selects a subset of Haar-like features to use for training. These features capture different aspects of the object's appearance, such as edges, corners, and texture.

3. **Feature Evaluation:** For each selected feature, the algorithm evaluates how well it separates positive and negative samples. Features that perform well are given higher weights.

4. **Weak Classifier Training:** The AdaBoost algorithm sequentially trains weak classifiers using the selected features. Each weak classifier focuses on one feature and tries to classify samples as positive or negative.

5. **Classifier Combination:** The weak classifiers are combined into a strong classifier using weighted voting. The weights are determined based on the performance of each weak classifier.

6. **Classifier Thresholding:** A threshold is set to determine the presence of the object. The strong classifier checks each region of the image, and if the weighted sum of weak classifiers' responses surpasses the threshold, the object is detected.

Applications of Haar Cascade:

Haar Cascade has been successfully applied to various object detection tasks, including:

1. **Face Detection:** One of the most well-known applications is face detection. The Haar Cascade-based face detectors can quickly and accurately identify faces in images and video streams.

2. **Pedestrian Detection:** Haar Cascade has been used to detect pedestrians in surveillance footage or traffic monitoring systems.

3. **Object Tracking:** By integrating Haar Cascade with other algorithms, objects can be tracked across multiple frames in video streams.

4. **Gesture Recognition:** It can also be applied for recognizing hand gestures or specific body parts in computer vision applications.

Limitations:

While Haar Cascade is efficient and suitable for real-time applications, it has some limitations:

1. **Sensitivity to Lighting:** Haar Cascade models might struggle with varying lighting conditions or image quality.

2. **Specific Training Data:** Effective training requires a diverse dataset with positive and negative samples.

3. **Limited to Simple Features:** Haar-like features are simple and might not capture complex object structures.

4. **False Positives and Negatives:** Like any object detection method, Haar Cascade may produce false positives (detecting an object when it's not there) and false negatives (not detecting an object when it's present).

Despite its limitations, Haar Cascade remains a foundational technique for real-time object detection and has paved the way for more advanced methods like deep learning-based object detection models.

Haar cascade Custom object detection:

Objective:

To train a custom object detection model using the Cascade Trainer GUI for detecting cars and bikes.

Process:

1. Dataset Preparation:

Collected a dataset with positive samples and negative samples.

2. Setup:

Installed OpenCV and set up Cascade Trainer GUI with the dataset.

3. Usage of Cascade Trainer GUI:

Link for GUI: https://amin-ahmadi.com/cascade-trainer-gui/

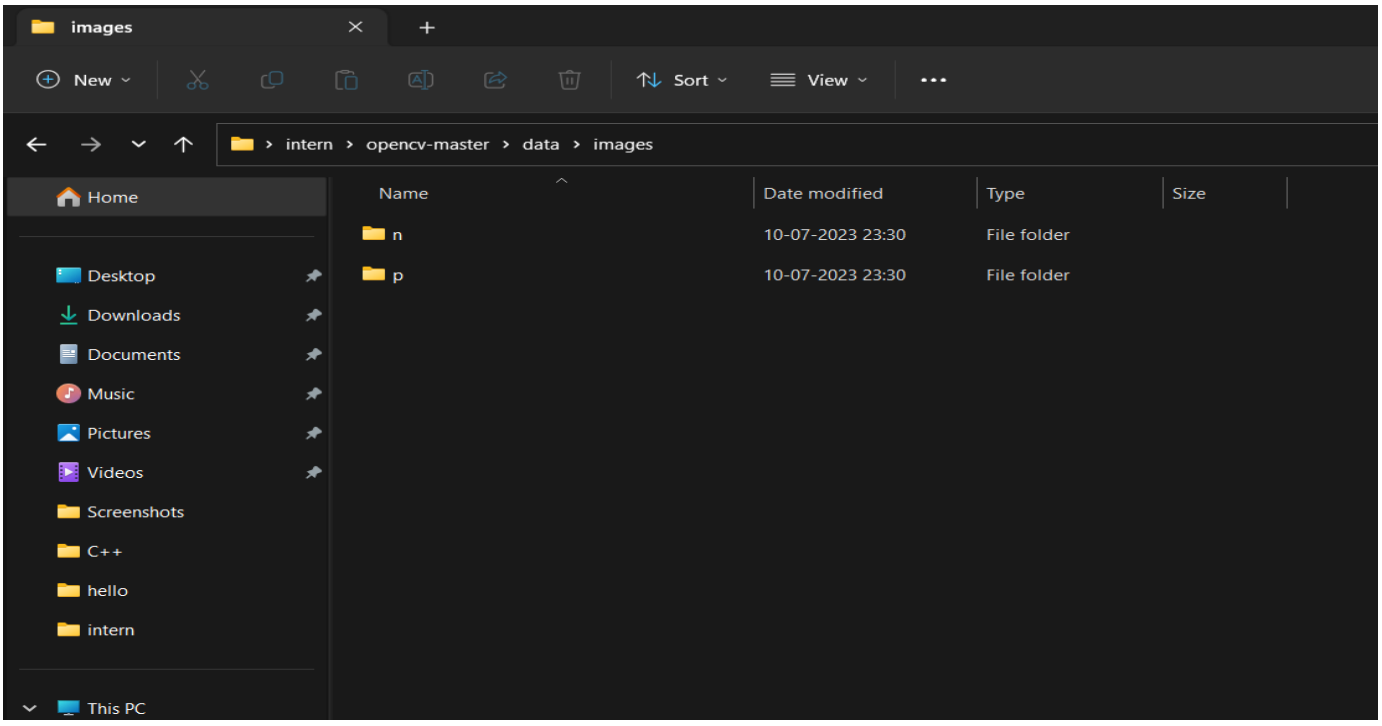


Figure 7.1

3.1 Positive Description folder:

Created an image folder (p) listing positive sample image paths. The folder should contain only one class of the image that is to be detected.

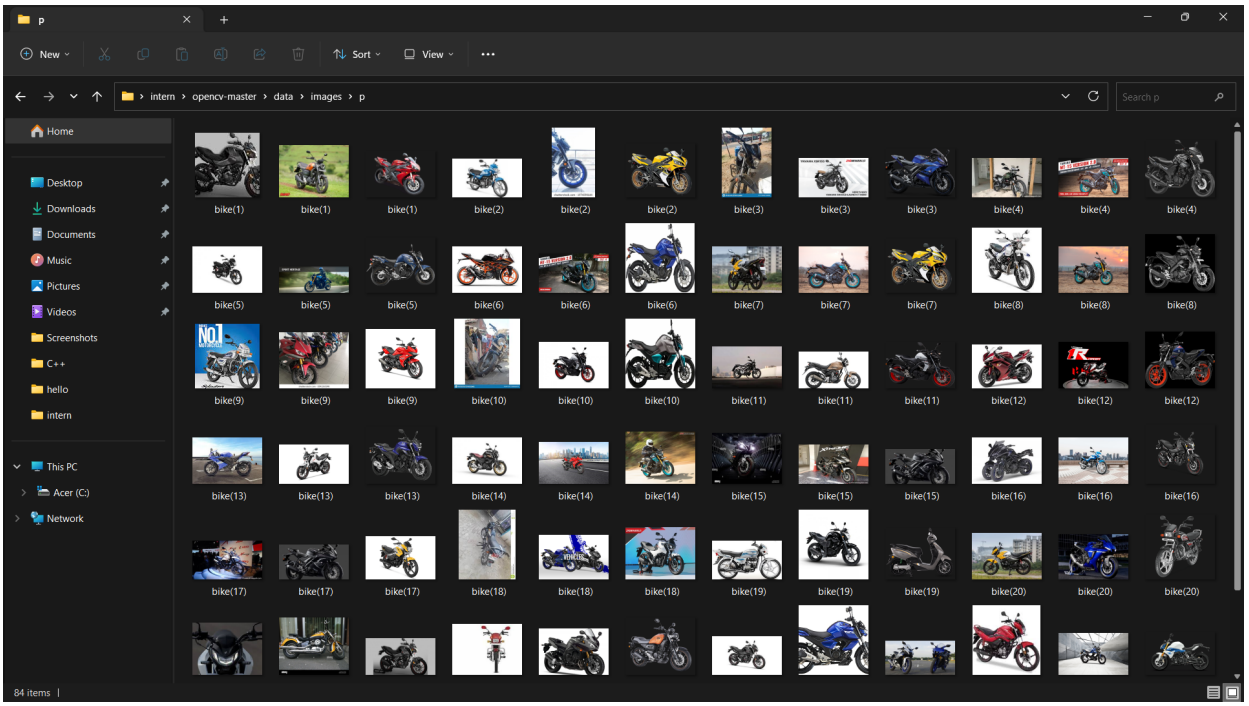


Figure 7.2

3.2 Negative Description File:

Created an image folder (n) listing negative sample image paths. It should contain other than the image in the ‘p’ folder such as background or other classes.

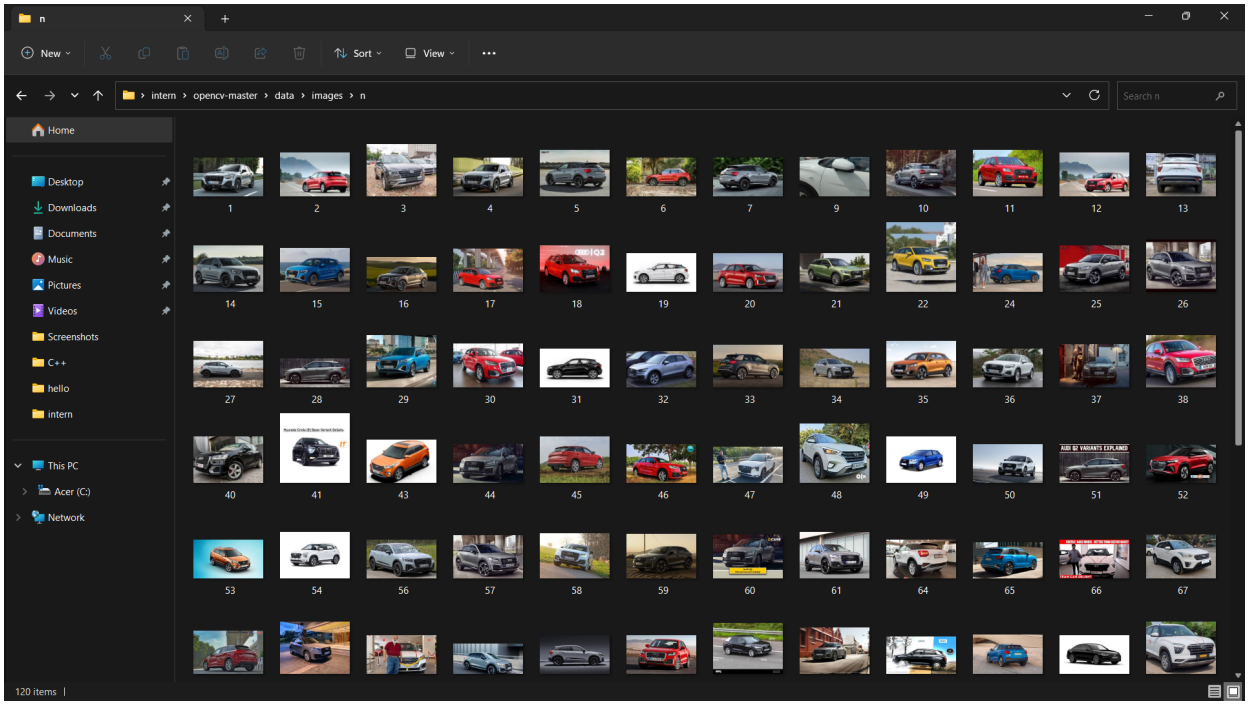


Figure 7.3

3.3 Cascade Trainer Parameters:

Defined training parameters, including stages, false-positive rate, and minimum detection window size.

3.4 Training:

Launched Cascade Trainer GUI, provided paths to description files, and initiated training. Browse and set the folder path which contains ‘p’ and ‘n’ folders in the ‘Samples folder’.
Set number of positive and negative images contained in the folder.

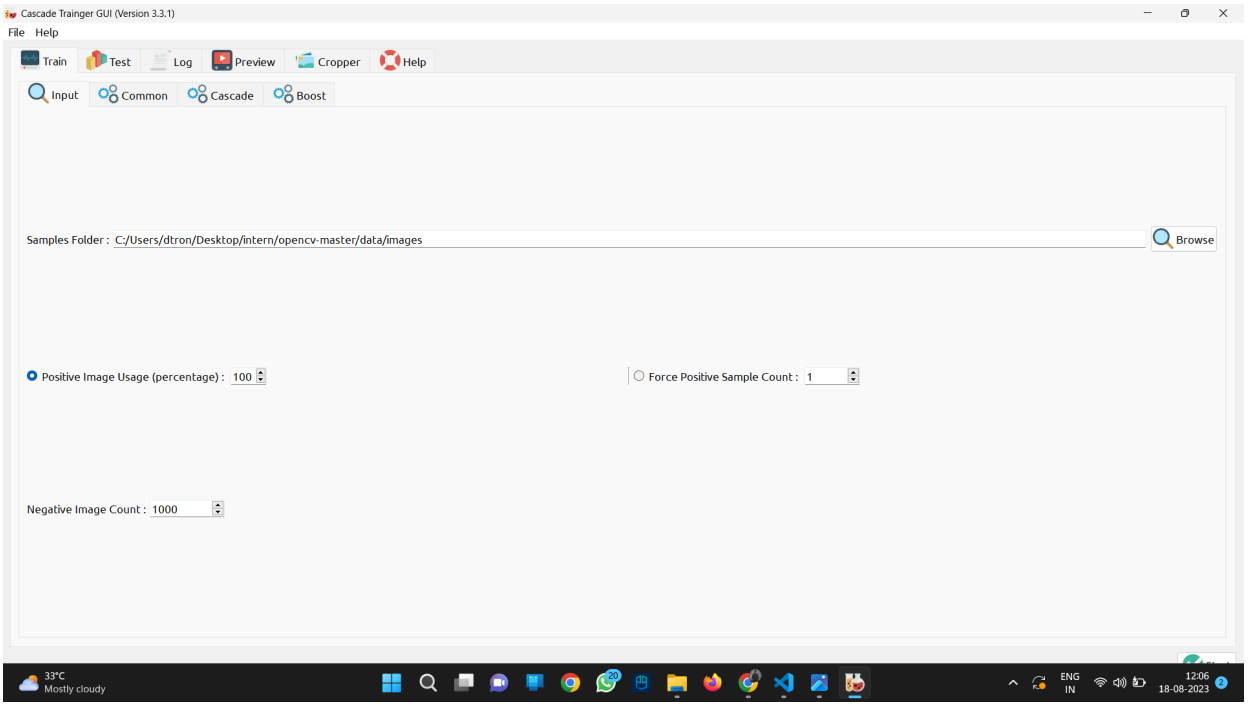
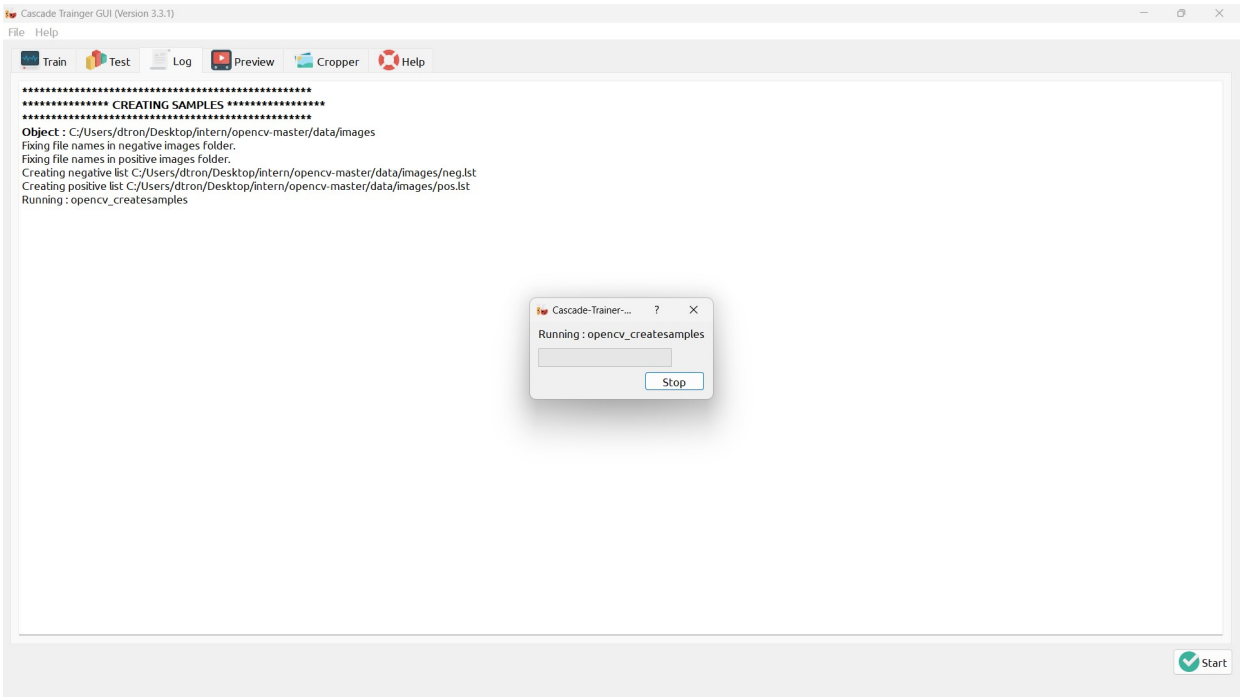


Figure 7.4

Click Start at the Bottom of the right corner to train the model.



4. Evaluation:

Used the trained cascade.xml model to detect cars and bikes in test images or videos. The trained xml file will be found in the classifier folder.

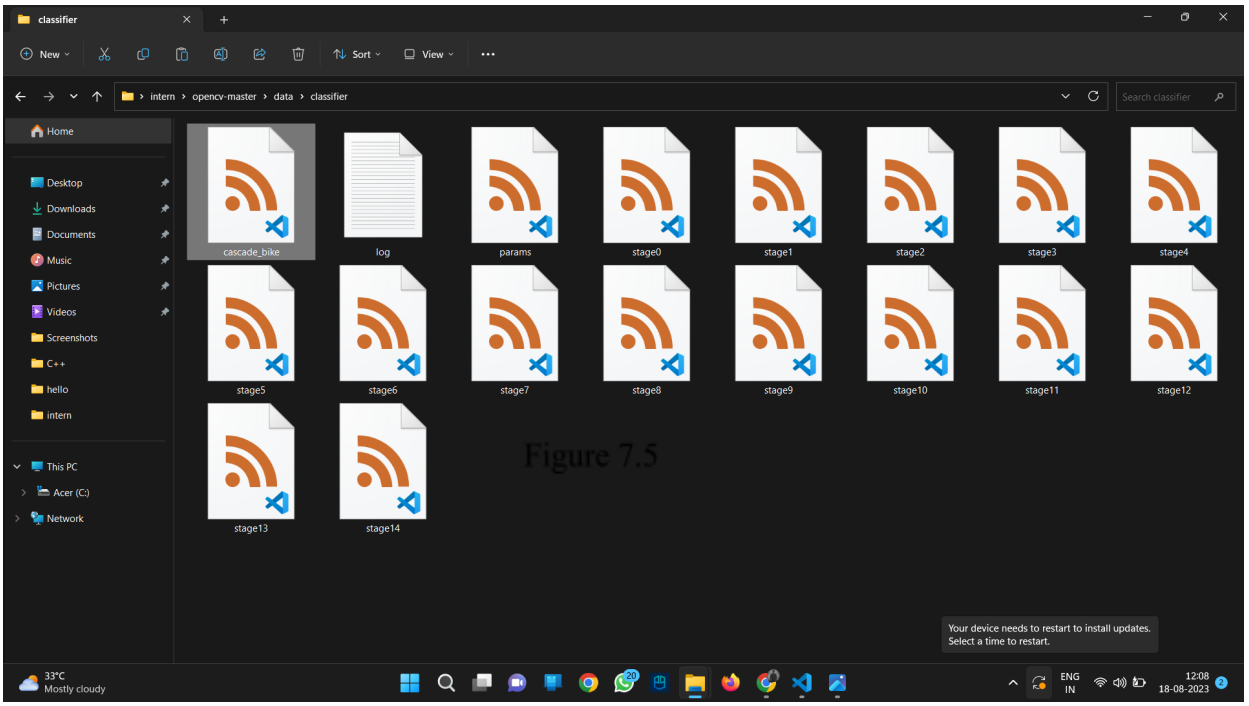


Figure 7.5

Figure 7.6

Using a trained xml file for bike and car the object is detected.

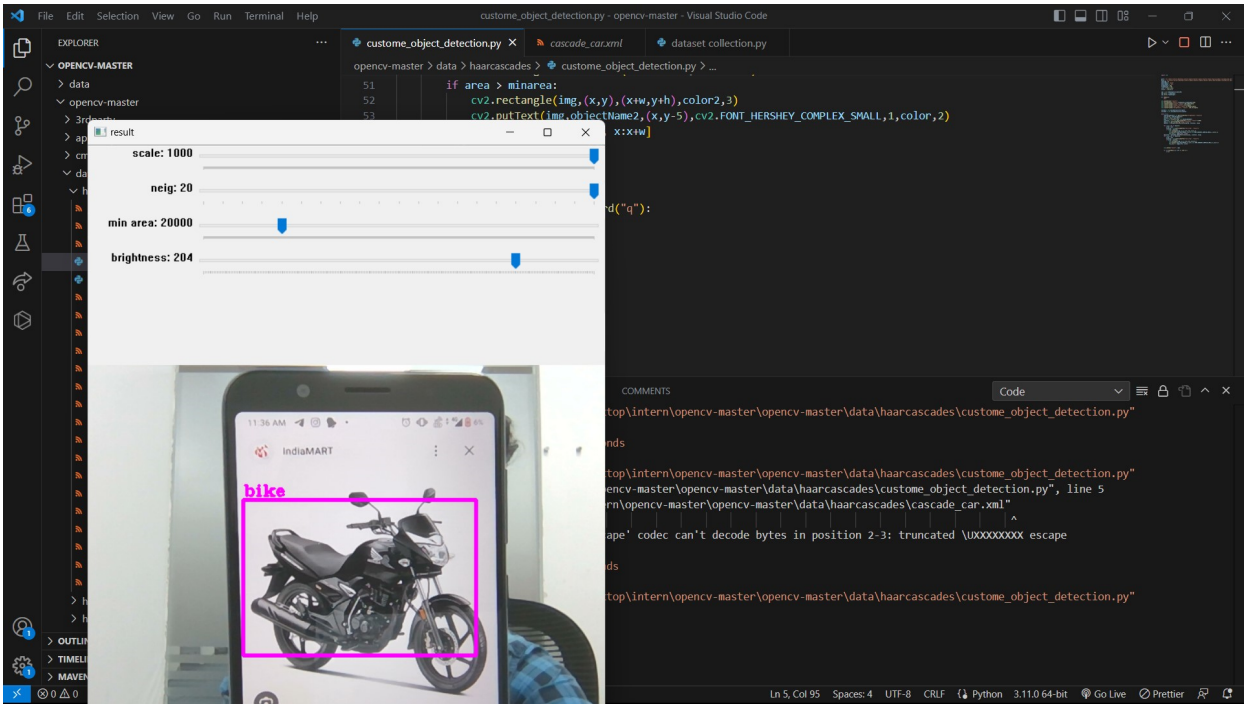


Figure 7.7

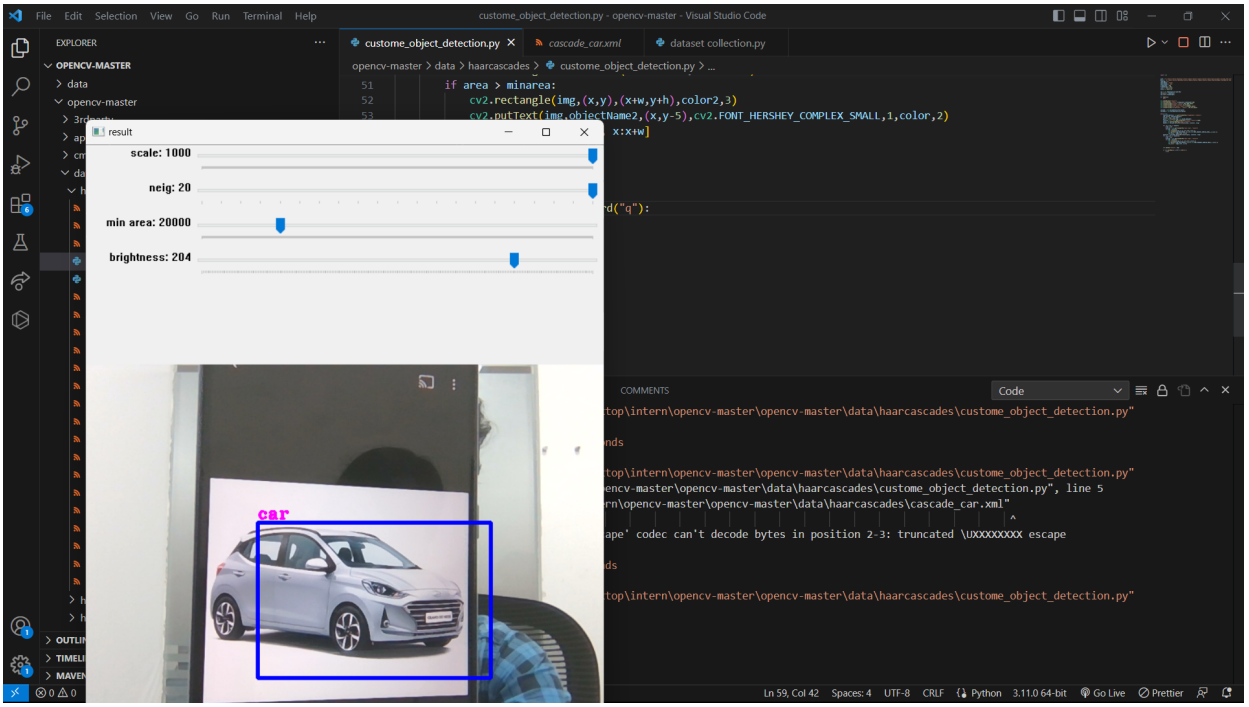


Figure 7.8

5. Conclusion:

The Cascade Trainer GUI streamlined the custom object detection model training process, resulting in a model proficient at detecting cars and bikes.

Limited Flexibility: The Cascade Trainer GUI is suitable for relatively straightforward object detection tasks but might struggle with more complex scenarios or varying object orientations. It doesn't have the capability to train more than 600+ images for one object at a time.

Performance Constraints: Haarcascade-based models, including those trained using the Cascade Trainer GUI, can have limitations in terms of accuracy and performance, especially when compared to modern deep learning-based object detection methods.

Manual Tuning: While the GUI simplifies training, fine-tuning parameters may still require some manual adjustment for optimal results.

Resource Intensive: Training a cascade classifier can be computationally intensive, requiring substantial processing power and time, especially as the number of stages increases.

8. YOLOv5 – PRETRAINED & CUSTOMIZED

YOLOv5 is a popular deep learning model for object detection, and it is an evolution of the YOLO (You Only Look Once) family of models. It is designed to be more lightweight, efficient, and easier to use than its predecessors. Custom object detection using YOLOv5 involves training the model on a specific dataset containing images of your custom objects.

Training Pre Trained data

Step 1: Open the terminal in the vscode to clone the YOLOv5 repository from GitHub (https://github.com/ultralytics/yolov5) and install the required dependencies. Type the below commands in the terminal.

git clone https://github.com/ultralytics/yolov5
cd yolov5

Step 2: This YOLOv5 will have pre trained data(coco128.yaml) file which we will use to detect the objects with the command

python detect.py --source 0

Figure 8.1

Training Customized data

1. Prepare Custom Dataset:

1.1 Collect Images

Collecting high-quality data is essential for training an accurate object detection model. Our team captures real-world images from rear-view cameras mounted on vehicles. We annotate these images with bounding boxes around vehicles of five different classes: car, bus, two-wheeler, auto-rickshaw, and truck. Some images are taken from the “google images”.

Dataset Link:

https://universe.roboflow.com/new-workspace-mekbw/bicycle-and-motor-13nsu/dataset/1/images (for bicycle and motor vehicle)

https://universe.roboflow.com/car-dataset/car-datasets (for cars)

1.2 Annotations

Link: https://www.makesense.ai/

Step 1: Go to above link and click Get started in the right bottom corner of the screen

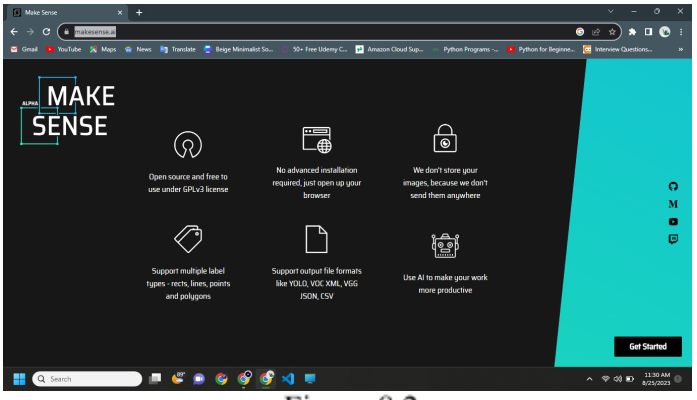


Figure 8.2

Step 2: upload or drag and drop the images that you are going to train and click object detection.

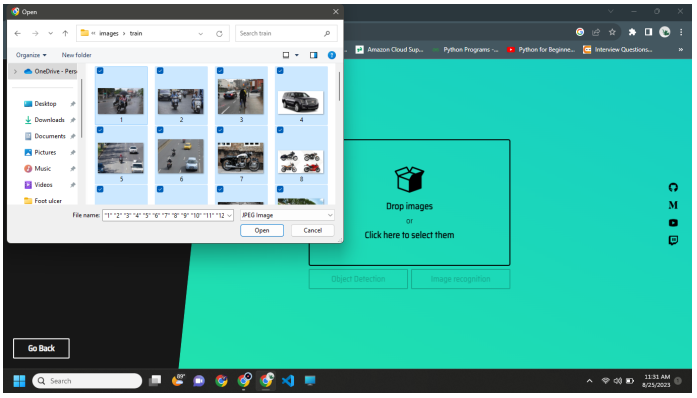


Figure 8.3

Step 3: Type label names and click start project.

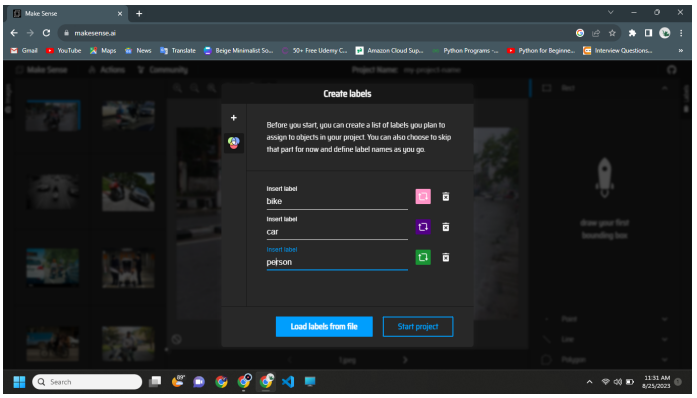


Figure 8.4

Step 4: Now start to annotate the images and select the label at the right corner.

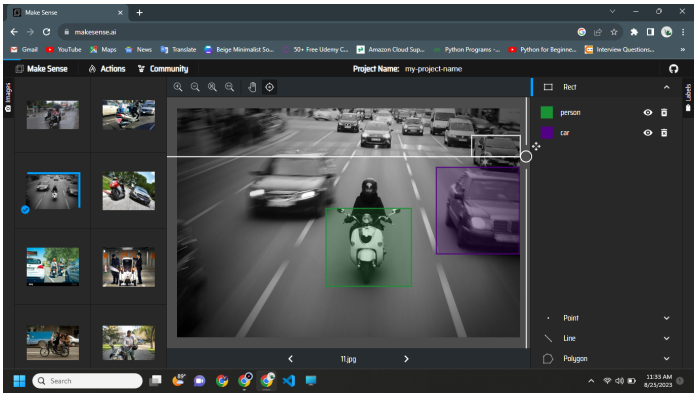


Figure 8.5

Step 5: After completion of the process now click action and click export annotations.

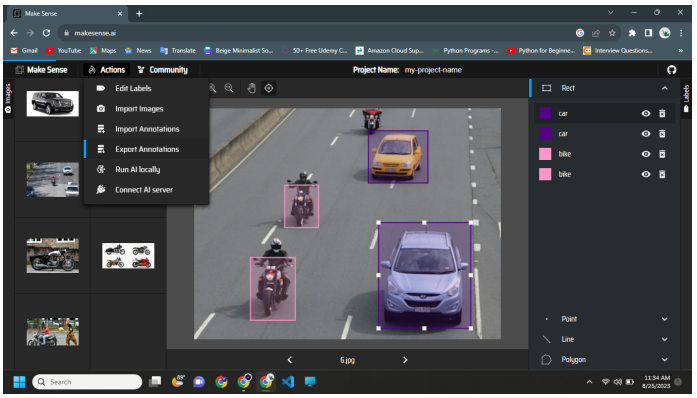


Figure 8.6

Step 6: Click Yolo format and export the zip file.

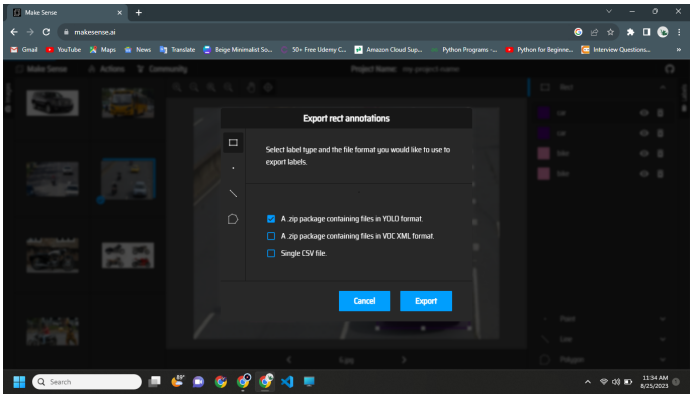


Figure 8.7

Step 7: Now extract the zip file and find the text file in it.

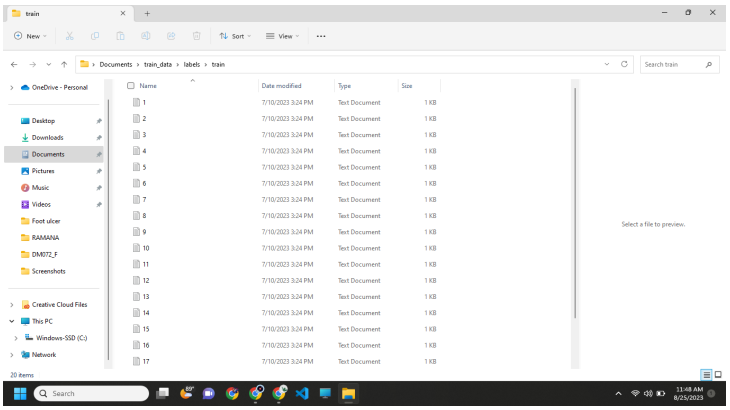


Figure 8.8

Step 8: Now create a folder named traindata1 in the desktop and add images and labels as sub folders in it(NOTE: images and labels must be in small letters). Again create train and val as sub folders for both the folders as described by the below image.

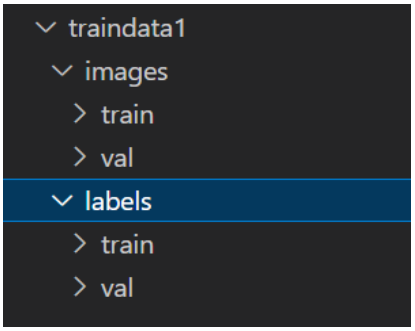


Figure 8.9

Step 9: Add first 80% of images collected in images->train folder and 20% images in images->val folder. Similarly add first 80% of its corresponding text file generated by makesence.ai to labels->train folder and 20% of text file to labels->val folder.

2. Set Up YOLOv5 Environment:

Step 1: Open vscode and create a new window. Add the created folder(traindata1) to the workspace

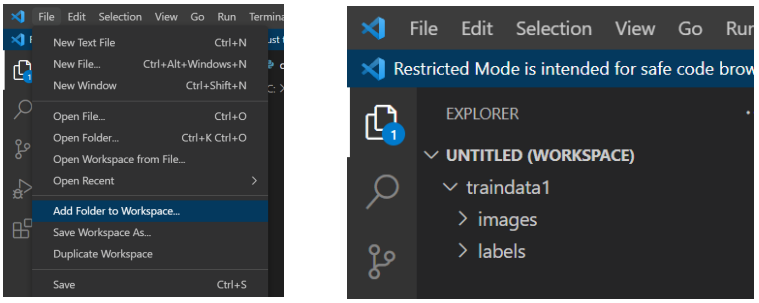


Figure 8.10

Step 2: Open the terminal in the vscode to clone the YOLOv5 repository from GitHub (https://github.com/ultralytics/yolov5) and install the required dependencies. Type the below commands in the terminal.

git clone https://github.com/ultralytics/yolov5 # clone
cd yolov5

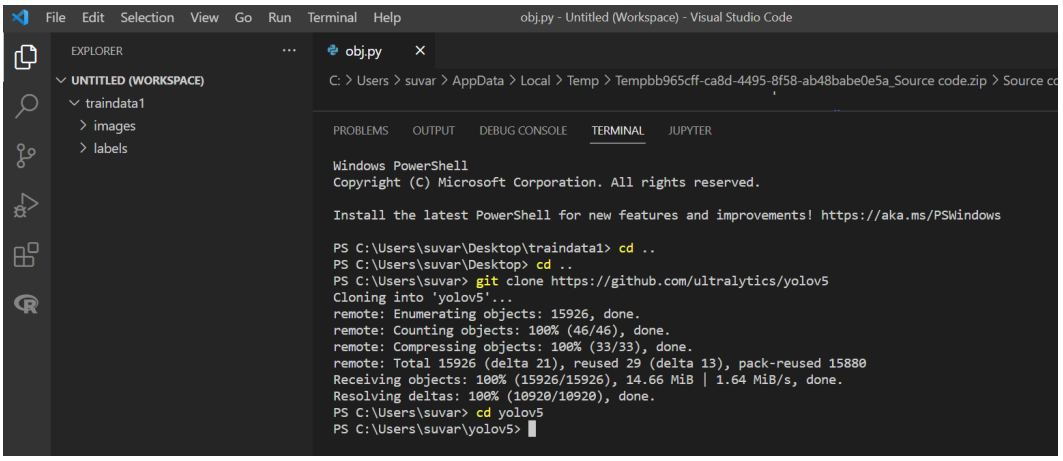


Figure 8.11

3. Create the custom_data.yaml

Step 1: Go to yolov5->data folder in your yolov5 folder in file explorer and create a copy of coco128.yaml file and rename it as custom_data.yaml file.

Step 2: Give the relative path of train and val of images folder and the number of classes(nc) annotated and their names along with it. It should look like:

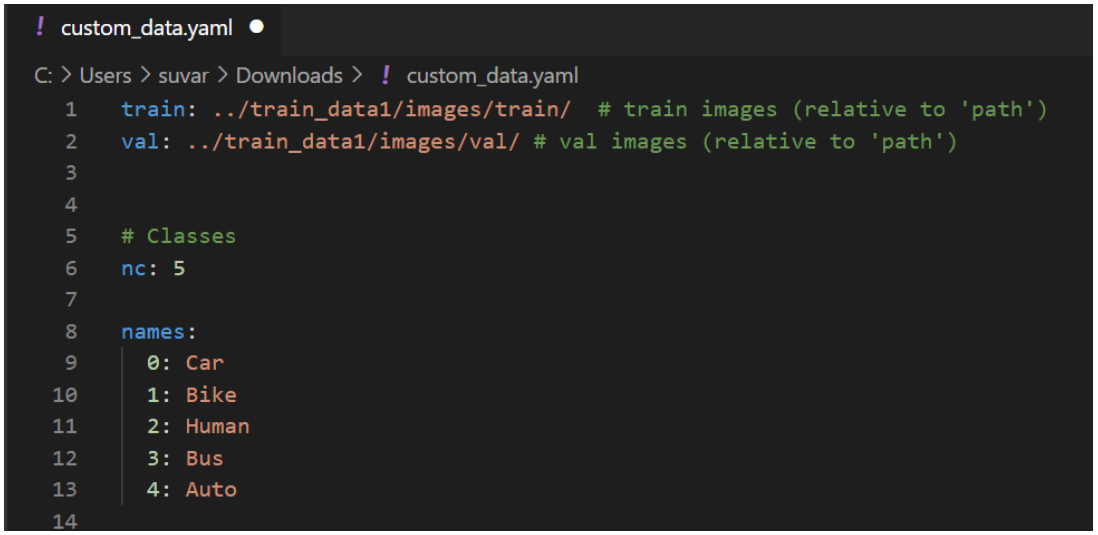
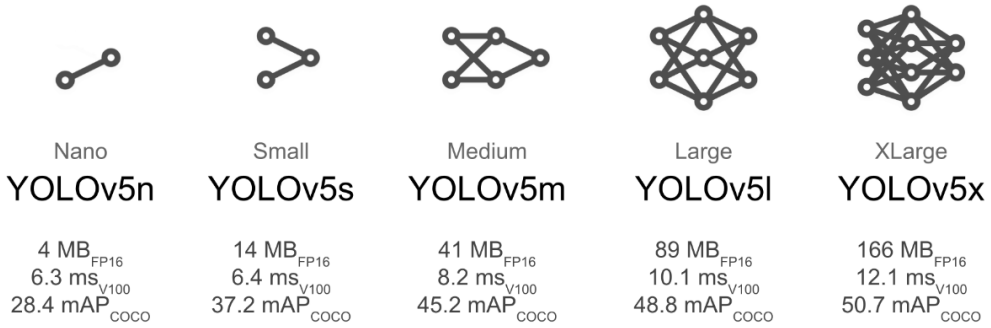


Figure 8.12

Step 3: Select the yolov5 model



Here we are using the YOLOv5s which is the second-smallest and fastest model available to train our data which will be more efficient than medium and large sizes.

4. Creating conda environment:

Step 1: Creation of conda environment as yolov5 with python version as 3.8 using the below command

conda create --name yolov5 python=3.8

This will create the base environment such that original python version and other dependencies will not crash with the one we are using with)
(NOTE: If this pops up an error message download anaconda navigator and use the vscode from it and repeat from step 2)

Step 2: For activating the created conda environment use: ***conda activate yolov5***

Step3: To install the required requirements in requirements.txt use the following
pip install -r requirements.txt

4.1. Training the collected data

To train the collected data using custom_data.yaml file type the following code

python train.py --img 640 --batch 2 --epochs 100 --data custom_data.yaml --weights yolov5s.pt --cache

Where the epoch represents the number of iterations it has to perform to get the accurate results. This will generate the best.pt file which can be used to detect the test images.

5. Evaluate the model:

After training the model's performance use detect.py to analyze the trained data. For this create a separate folder called test and include it in the same workspace. To detect use the following command

python detect.py --weights path/to/best.pt --source 0

This source 0 is used to connect to webcam in pc in order to test the data stored in test folder use:

python detect.py --weights path/to/best.pt --source path/to/test

Inference and output:

It's essential to have a reasonably large and diverse dataset for effective training. The quality of annotations and the balance of classes in the dataset also play a crucial role in the model's performance.
These are some of the outputs from the above trained datasets

9. CONCLUSION

9.1. Comparison between models of Haar Cascade, Mobilenet SSD and YOLOv5

Parameters	Haar Cascade	Mobilenet SSD	YOLOv5
Algorithm	Haar Cascade is an older object detection algorithm which utilizes Haar-like features to detect objects.	MobileNet SSD combines MobileNet's lightweight architecture with SSD for real-time object detection.	YOLOv5 excels in fast and accurate object detection by directly predicting bounding boxes and class probabilities with a deep neural network from input images.
Model Availability	Popular in OpenCV. Usable for Object detection.	Pretrained, accessible models for object detection.	Models (n, s, m, l, x) trained on COCO and more.
Model Size	The data given for training is Limited to only 600 images.	Smaller, efficient models for resource-limited devices.	Different sizes, larger for more accuracy but higher resource demand.
Inference Speed	They have fast inference speed, for resource-constrained devices	Real-time detection on less powerful devices	Fast inference, smaller versions optimized for speed.
Accuracy	Less accurate than CNNs. Good for simple tasks like face detection, struggles with complexity and lighting changes.	Good accuracy with efficiency, not top-tier.	Higher accuracy due to deeper architecture and design.
Deployment Flexibility	Easily deployable via OpenCV, suitable for embedded devices. Enables real-time detection on low-resource devices.	Ideal for mobile, IoT, and embedded devices.	Versatile but potentially more resource-intensive.
Best Fit	Limited Flexibility: The Cascade Trainer GUI is suitable for relatively straightforward	The mobilenet is the best suit for the Iot devices. The pretrained objects were well trained for	The YOLOv5 is also the best fit. The YOLO needs high computational power with Higher graphics

	object detection tasks but might struggle with more complex scenarios or varying object orientations. It doesn't have the capability to train more than 600+ images for one object at a time.	the application. There are a total of 91 objects that have been pre trained using this algorithm. For our application we integrate this to the board, mobilenet is the best fit algorithm. Because of its lightweight architecture with SSD for real-time object detection. At last we used the pretrained datasets by reducing the classes to our needs.	card. Though the board doesn't have the required specifications we aren't able to use the YOLO. If we use the cloud, it is possible to use it. And also in our case due to insufficient datasets with required resolution the model is not accurate. If we use the datasets with the correct resolution the YOLO will perform high.

In conclusion, when evaluating the performance of MobileNet, Haar Cascade, and YOLOv5 on IoT devices, it becomes evident that MobileNet emerges as the superior choice. While each algorithm has its own strengths and weaknesses, MobileNet excels in striking a balance between accuracy and efficiency, making it the optimal solution for resource-constrained IoT devices.

Haar Cascade, although a pioneering technique, often struggles to maintain accuracy when faced with complex scenarios due to its simplistic feature representation. On the other hand, YOLOv5 exhibits impressive accuracy but demands substantial computational resources, limiting its practicality for IoT devices with constrained processing capabilities.

MobileNet, through its depthwise separable convolutions and lightweight architecture, showcases remarkable performance in real-time object detection while consuming fewer computational resources compared to YOLOv5. This characteristic aligns well with the resource limitations of IoT devices, allowing them to execute object detection tasks effectively without compromising on accuracy.