ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH

PHÁT TRIỂN ỨNG DỤNG INTERNET OF THINGS (CO3038)

LAB 2 : INTEGRATING GOOGLE AI TEACHABLE MACHINE MODEL AND READING SENSORS' DATA VIA UART

SV thực hiện:
Nguyễn Trọng Tín 2012215

Thành phố Hồ Chí Minh, Ngày 23 Tháng 02 Năm 2024

Contents

List of Figures

1 INTRODUCTION

1.1 GOOGLE TEACHABLE MACHINE

Google Teachable Machine is an online platform developed by Google that allows users to easily train machine learning models without requiring extensive coding knowledge. It provides a user-friendly interface for creating custom machine learning models for image, sound, or pose recognition.

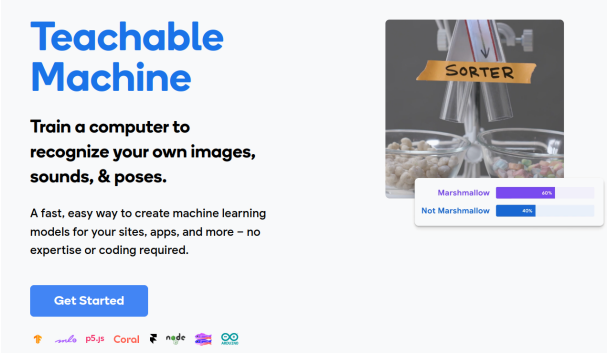


Figure 1: *Google Teachable Machine*

Key features of Google Teachable Machine include:

- User-Friendly Interface: The platform is designed to be accessible to users with little or no coding experience. It simplifies the process of training machine learning models through an intuitive graphical interface.
- Image, Sound, and Pose Recognition: Teachable Machine supports various types of machine learning models, including image classification, sound classification, and pose estimation. Users can train models to recognize patterns in images, sounds, or poses based on their specific use cases.
- Real-Time Training: The platform allows users to see the training results in real time, enabling them to iteratively improve their models by refining training data or adjusting parameters.
- Export Models: Once a model is trained, users can export it for use in their own applications, websites, or projects. The exported models can be integrated into various environments that support TensorFlow or other machine learning frameworks.
- Educational Tool: Google Teachable Machine serves as an educational tool to introduce individuals, especially students and beginners, to the concepts of machine learning in a hands-on and interactive manner.

By democratizing the process of creating machine learning models, Google Teachable Machine aims to make artificial intelligence more accessible and inclusive for a broader audience. It provides a practical

and engaging way for users to explore the capabilities of machine learning without delving into complex coding and algorithm development.

1.2 Null-modem emulator: com0com

The Null-modem emulator (com0com) is a kernel-mode virtual serial port driver for Windows. You can create an unlimited number of virtual COM port pairs and use any pair to connect one COM port based application to another. The HUB for communications (hub4com) allows to receive data and signals from one COM or TCP port, modify and send it to a number of other COM or TCP ports and vice versa.

Features

- com0com: baud rate emulation
- com0com: pinouts customization
- com0com: noised line emulation
- com0com: paired port settings monitoring (baud rate, data bits, stop bits, parity)
- com0com: hiding 'unavailable' ports from user
- hup4com: spliting data from one serial device into several virtual serial ports
- hup4com: redirecting serial to TCP/IP and TCP/IP to serial
- hup4com: RFC2217 - Telnet Com Port Control Option (client and server)

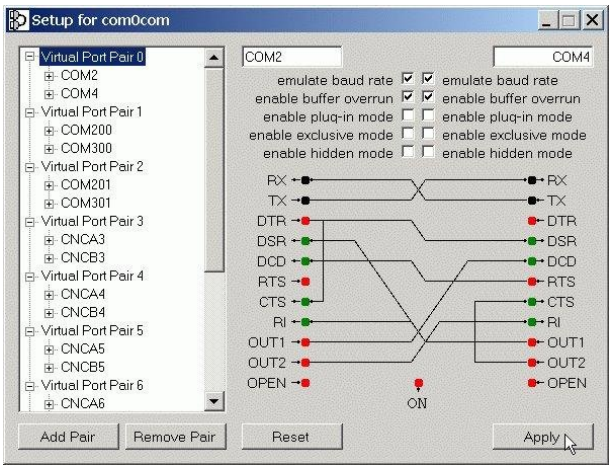


Figure 2: *com0com Interface*

1.3 HERCULES TERMINAL

Hercules SETUP utility is useful serial port terminal (RS-485 or RS-232 terminal), UDP/IP terminal and TCP/IP Client Server terminal. It was created for HW group internal use only, but today it's includes

many functions in one utility and it's Freeware! With our original devices (Serial/Ethernet Converter, RS-232/Ethernet Buffer or I/O Controller) it can be used for the UDP Config.

Features:

- Complete support for Windows 7, 8, 8.1 and 10
- All basic TCP and UDP utilities in one file, no installation required (just one .EXE file)
- Implemented Serial Port Terminal is working with the Virtual Serial Ports (COM12 for example). You can check and control all serial port lines (CTS, RTS, DTR, DSR, RI, CD) Simple TCP client (like the Hyperterminal) with the TEA support, view format, file transfers, macros..
- Easy to use TCP Server with the TEA support, view format, file transfers, macros..
- Hercules contains simple UDP/IP "Terminal" with view formats, echo, file transfers, macros..
- Support the NVT (Network Virtual Terminal) in the Test mode tab, as like as NVT debuging features..
- Using Telnet extended with NVT allows serial port configuration (RFC2217), device identification, confirmation of data sequence, etc.

2 SYSTEM ARCHITECTURE

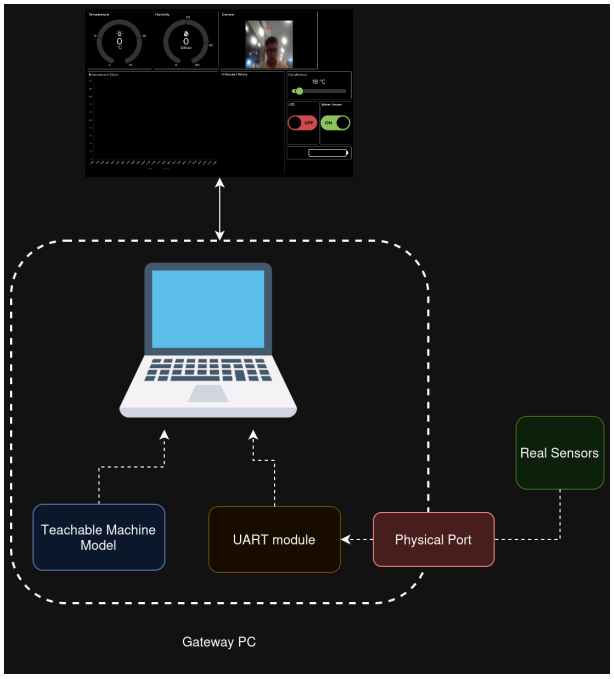


Figure 3: *Real life Architecture*

The above Image illustrates the architecture diagram of our system, but in this lab, it is restricted by hardware conditions. We proposed an alternative architecture that will use virtual ports instead of physical ones, and data gathered by real sensors will be produced by using scripts, virtual terminals, or whatever we can via UART protocol.

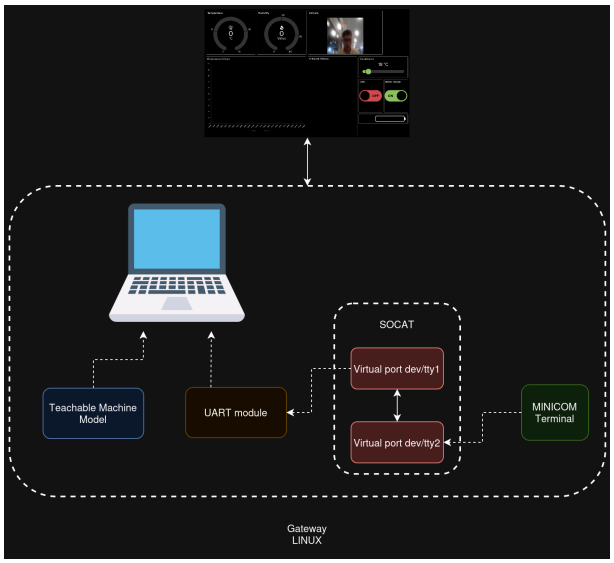


Figure 4: *Alternative Architecture on LINUX*

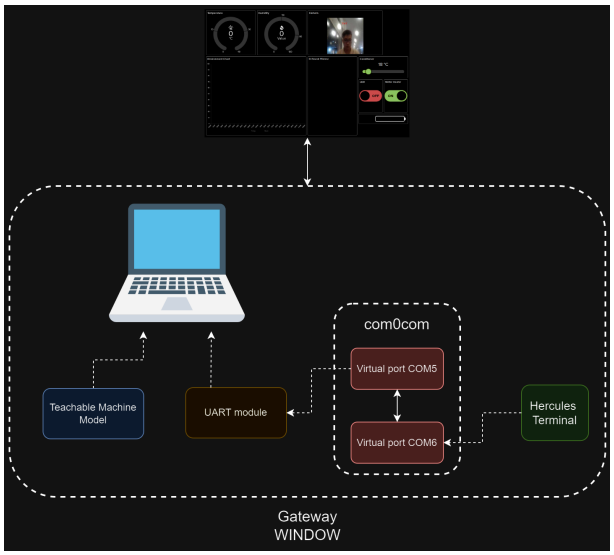


Figure 5: *Algternative Architecture on Window*

3 IMPLEMENTATION

3.1 TEACHABLE MACHINE AI

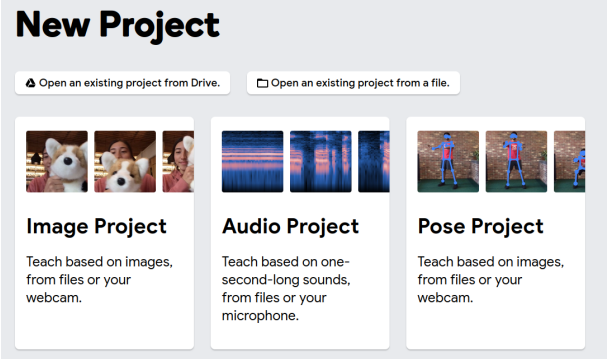
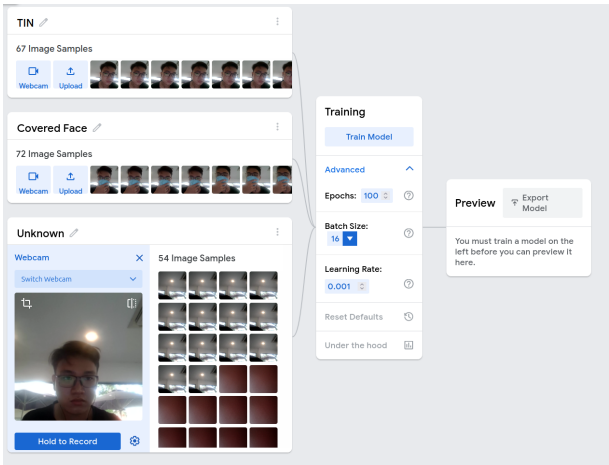


Figure 6: *Supported Models on TEACHABLE MACHINE*

There are many types of models supported by TeachableMachine, in this lab, we only use **Image Project** to develop a Face Recognition feature which can be embedded into many application such as SmartHome, SmartOffice, Smart Attendance...



We will create many classes we want to produce classification model, in this case, I produce 3 class which included Myface, Covered face, and Unknown.

Next, the training setting should be modified such as **Epoch**, **Batch size**, **Learning Rate**.

- **Epoch**: That every and each sample of dataset training should fed through our model at least one. In this case, I increased the Epochs from 50 (by default) to 100, that means all my sample will pass through the model at least 100 times, it will boost the predict abilities but take longer for training.
- **Batch size**: That indicates the size of samples will be fed to model. Let say it is 16, and my dataset includes 80 images, so there are 5 batches (80/16). Once 5 batches have been fed through model, one Epochs will be counted. Don't need to configure this parameter, it's used to separate our dataset into small pieces of group to avoid overload memory and computation.

- **Learning rate**: Measuring the size of steps taken, which is how aggressively we want to train our AI. Increasing it will make the model reach the goal faster, but it may overshooting. On the contrary, training takes longer to reach optimal weight values. Google recommends modifying this parameter if you're an expert in this field.

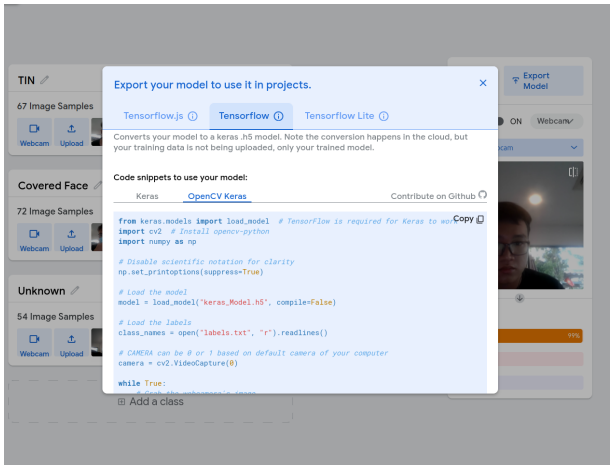


Figure 7: *Enter Caption*

GoogleAI also provides a sample code, which is very easy to understand and embed into our applications.

Firstly, we need to install some site packages, and the most vital is TensorFlow. Below is my system prerequirements for accelerating hardware by using GPU (NVIDIA RTX 2050 in my case). I strongly recommend that we should use LINUX for optimizing supports from TensorFlow and AI model.

```
readme.md
PREREQUIREMENTS:
    ubuntu: 23.10
    python: 3.11.6
    cudnn_version: 8
    cuda_version: 12.2

BUILD:
    sudo apt install nvidia-cuda-toolkit
    pip install tensorflow[and-cuda]
```

Figure 8: *Requirements*

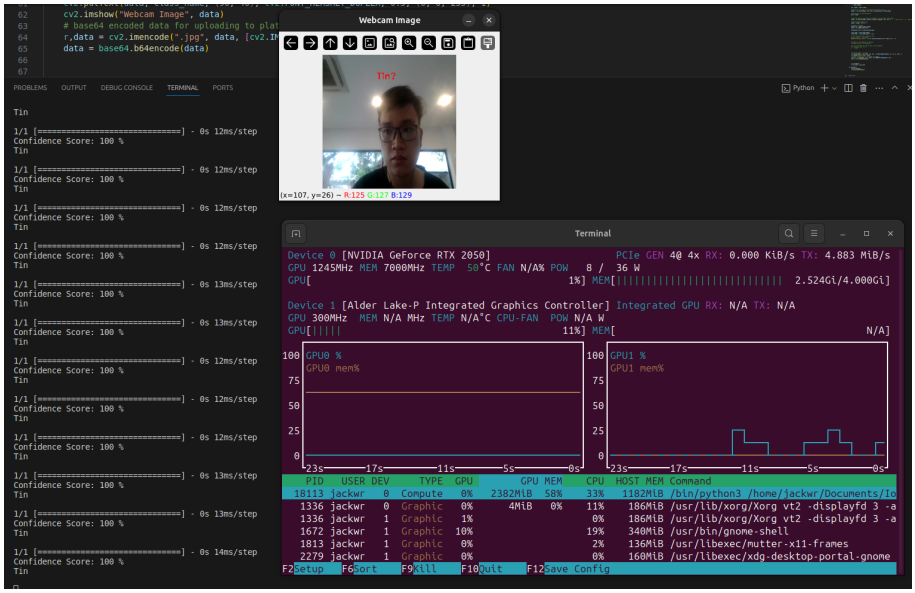


Figure 9: *Face Recognition Results*

I packages this AI model into one class, and very easy to use. It includes **AI_driver.py**, **keras_model.h5** and **labels.txt**. From which, we easily call **AI_driver.AI_Execute()**, that returns the labels or class name defined before and a **Base64 image** which is directly able to publish into AdafruitIO dashboard.

```
from AI_driver import AI_driver


...
while True:
    base64img, class_name = AI_driver.AI_Execute()
    client_1.MQTT_Publish("camera", base64img)
    print("Camera detect: ", class_name)
    time.sleep(5)
...
```

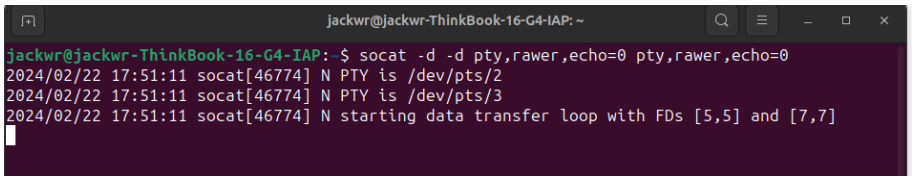
After implemented on both Window and Linux, based on performance, I evaluate that we should develop this application on LINUX kernel platform. I found it's very difficult to use Tensorflow-GPU on Window, and according to official Tensorflow documents, it says that Tensorflow no more support accelerating GPU on Window after version 2.10, now official Tensorflow releases are greater than version 2.12.

That leads to the performance of AI application on Window so poor. It takes more than **89ms per step** to process, but **only 15ms** in case of building on Linux kernel. Because the application running on Window uses CPU to process instead of using GPU as Linux. See Figure 9 and Figure 11 for performance comparison.

3.2 READ SENSORS DATA USING UART (LINUX VERSION)

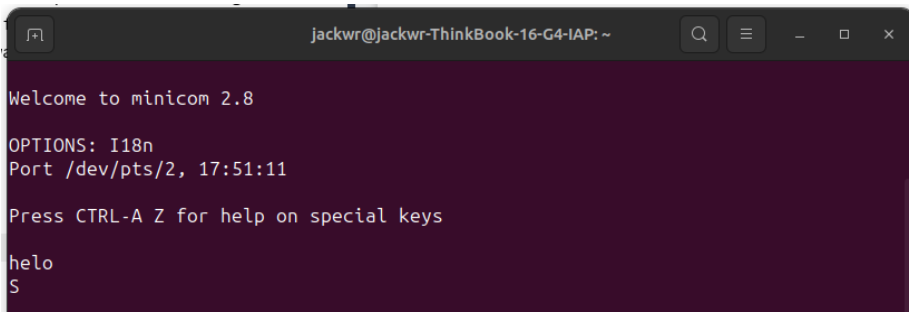
Firstly, I will create 2 virtual ports connecting with each other by using **SOCAT**. See below that it generates port **dev/pts/2** and **dev/pts/3**

```
sudo apt install socat
socat -d -d pty,rawer,echo=0 pty,rawer,echo=0
```



Secondly, use any terminal to connect one virtual port, and the other port will be assigned to our Gateway. In this case, I'm using **MINICOM** terminal. I configure **baudrate to 115200** and connect port **dev/pts/2**, which means our gateway will connect to **dev/pts/3**.

```
sudo apt install minicom
sudo minicom -b 115200 -D /dev/pts/2
```



Finally, I have implement a **UART_driver**, that's very simple to use:

- **UART_ConnectPort(yourPort)**: pass your name of port here. We also leave None, and it will automatically find the available one to connect. But i recommend that we should indicate the port's name for correctly operating.
- **UART_DataProcessing(data, mySensor)**: **don't need to call it**, it will call automatically. **data** is raw text received via UART, **mySensor** is a shared variable or class. The goal is to modify code to parse **raw data** and convert it to **mySensor** which is used by **main program**.
- **UART_loop_background(mySensor)**: Must call it once; it will receive data and process in the background, so we don't need to worry about control UART works anymore.

Use this code below in our main program at initialization field, we also should have a look in UART_DataProcess. Very simple code, and no need to worry about UART anymore.

```
from UART_driver import *
...

mySensor = MQTT_Sensors()
UART_ConnectPort("/dev/pts/3")
UART_loop_background(mySensor)
...
```

Notice that the data format of sensors is defined as follows:

```
    !1:temp:29#
    !2:humi:80#

```

where, every message must begin with " ! " and terminate by " # "; every parameter must be separated by " : ".

The first parameter is order number; the second one is a type of sensor (there have been already 2 defined sensors: **temp** and **humi**) and followed by their value parameter.

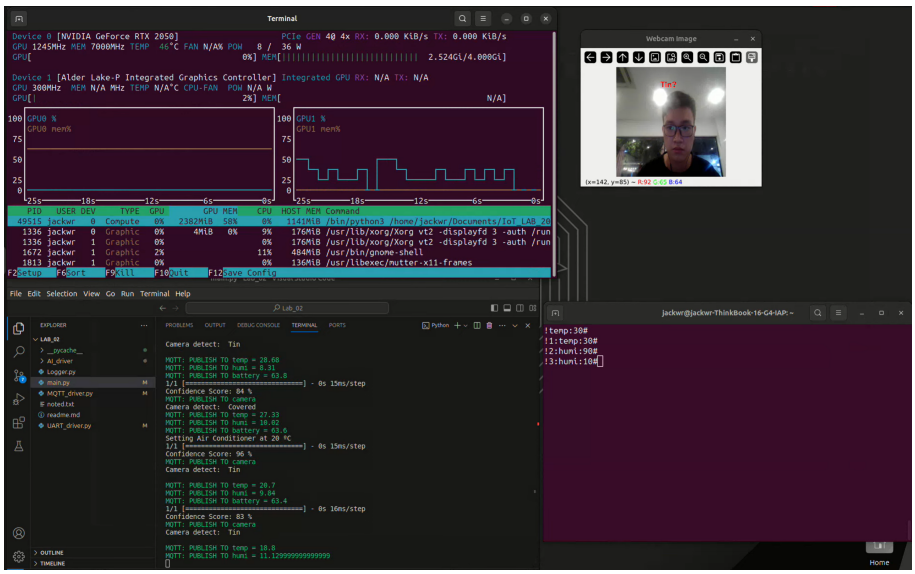
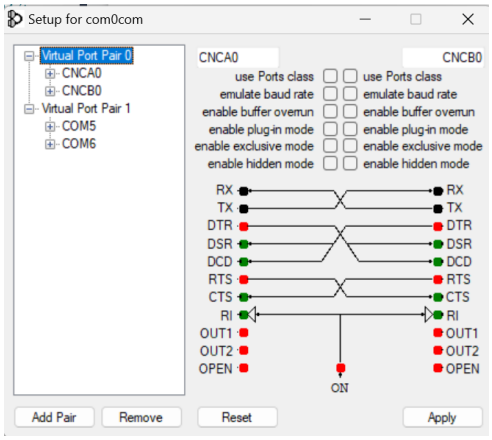


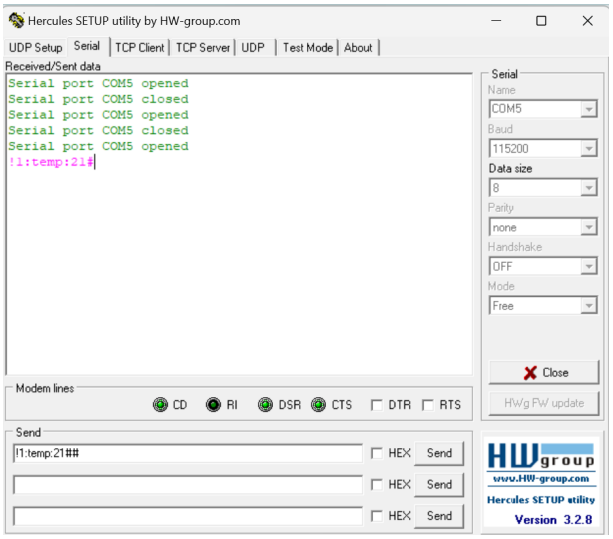
Figure 10: *Result on Linux*

3.3 READ SENSORS DATA USING UART (WINDOW VERSION)

In the first step, we would open **com0com** to create 2 virtual ports. There are **COM5** and **COM6** in my case.



Secondly, open Hercules Terminal, choose Serial Tab and Connect to one of two virtual ports. I connected to COM5.



Then, in the main program, particularly **main.py**, either passing the name of the other virtual port to **UART_ConnectPort()** or leaving it None is totally fine.

```
# In your main.py
mySensor = MQTT_Sensors()
UART_ConnectPort()          # leave it None will force auto-finding port
UART_loop_background(mySensor)
```

Finally, we can send data to the gateway via the Hercules terminal. The format of data is defined as: **"!1:temp:29#"**.

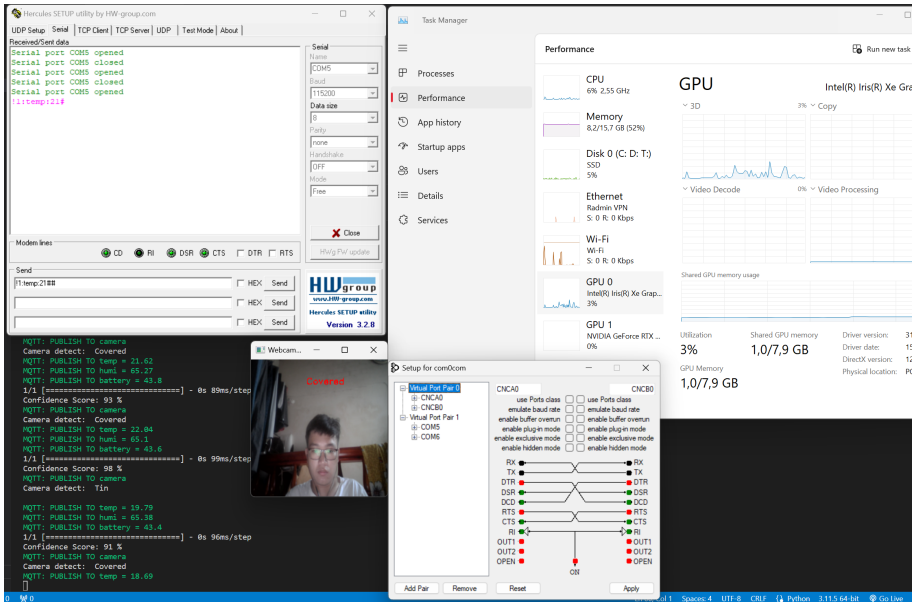


Figure 11: *Result on Window*

4 REFERENCES

Video Demo upload here: https://youtu.be/iVC1UfYrC3M

Please check my github for more information about source code and building problems

https://github.com/JackWrion/IoT_LAB_2024/tree/main/Lab_02