
Internship Report

By: Jubesh Joseph

Index

Chapter 1: Drone signal classification using YOLO v8	2-9
Chapter 2: Test Results for Drone Signal Classification using YOLOv8	10-17
Chapter 3: Packet info Header extractor using python	18-21
Chapter 4: User Manual for Wrapper	22
Chapter 5: Solution-Approach	23-34

“Drone Signal Classification using YOLO v8”

Chapter 1: TABLE OF CONTENTS

1. INTRODUCTION.....	4
1.1 Module overview	4
2. REQUIREMENTS.....	8
2.1. Required modes of operation.....	8
2.2. Software requirements	8
2.2.1. Requirements of Prediction mode	8
2.2.1.1. Getting prediction for incoming signal	8
2.2.1.1.1. (E) Command Line Script	8
2.2.1.1.2. (D) Design of algorithm/architecture	8
2.2.2. Requirements of Dataset Expansion mode	9
2.2.2.1. Adding support for more controllers	9
2.3. Hardware Requirements	9
2.3.1. Hardware platform requirement.....	9
2.3.2. FPGA/Processor estimated (approx) Resource requirements.....	9
2.4. Software environment requirements.....	9
2.4.1. Exception Handling and error responses related requirements	9
FUTURE IMPLEMENTATION REQUIREMENTS:.....	9

Table of Figures

Fig1: Interface block diagram.....	4
Fig2: Overview of working of the project.....	5
Fig3: Taking 1 crore samples from FS-GT2 + Wi-Fi data.....	6
Fig4: Labeling using YOLO label.....	6
Fig5: “YOLO friendly manner”.....	7
Fig6: Working of the python-C++ Wrapper.....	7

Introduction

The document outlines the requirements and methodology to correctly classify drone controller signals using a spectrogram based approach and the YOLOv8 model.

1.1 Module overview

The interface block diagram looks like this:

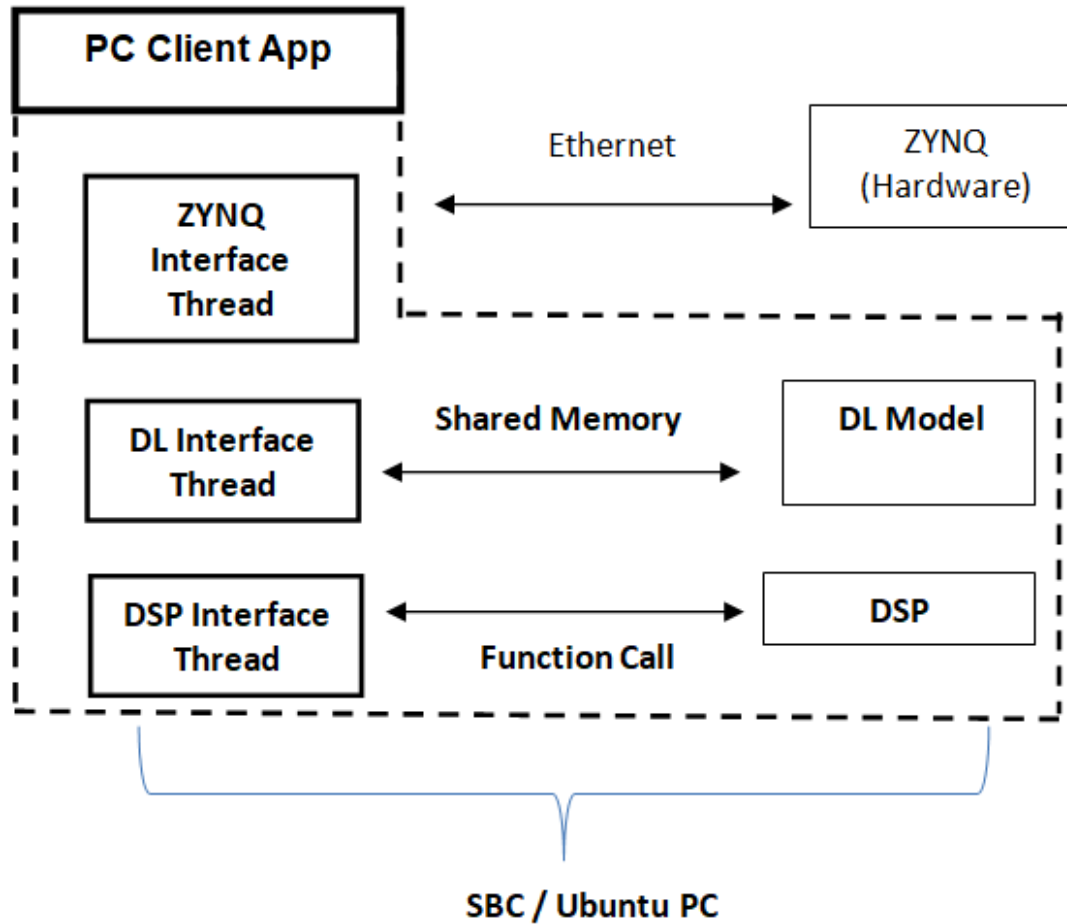


Fig 1: Interface Block Diagram

The project involves three main phases:

1. Generating a dataset of spectrogram images from signals emitted by various drone controllers. Label/annotate the images using YOLO Label software.
2. Training a YOLO v8 model using the annotated dataset of controller signals.
3. Developing a wrapper in C++ to receive signals, process them, invoke python functions and store the results.

About the dataset:

The dataset collected has signals of FS-GT2, Radio link AT9S pro, Wi-Fi (802.11 n, 802.11 g), RFD900 Telemetry module, HolyBro module. The data is sampled at 245.76 MHz sampling frequency and 200 MHz bandwidth.

For Wi-Fi, FSGT2, and AT9S pro: the center frequency is 2450 MHz

For RFD900 and Holybro: the center frequency is 900 MHz

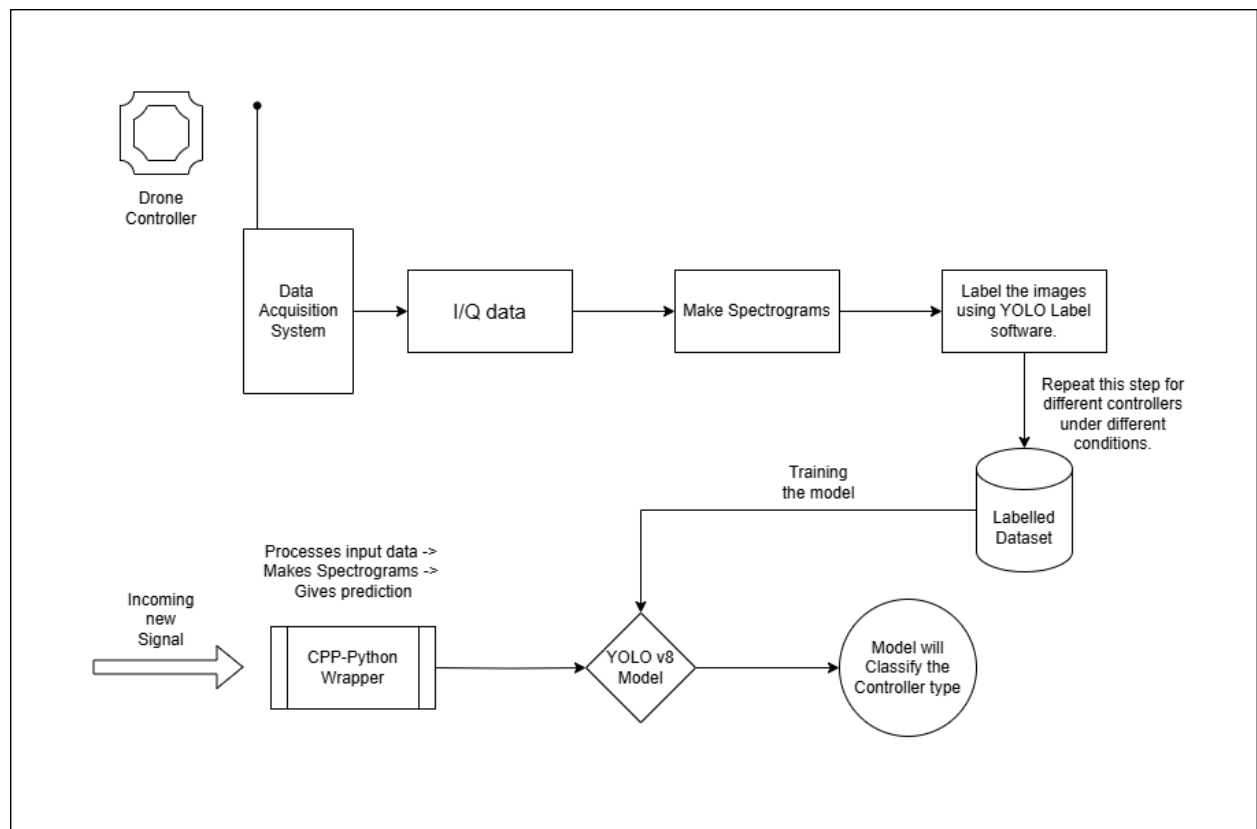


Fig2: Overview of working of the Project

The dataset has 3200 Million (320 crore) samples for each data-category mentioned above. From this data, Spectrogram images are plotted taking 53 Million (5.3 crore) samples in each image. This spectrogram will have time-frequency-energy information of signal of 217 ms. this number of 5.3 crore is derived after doing experimentation of plotting spectrograms with different sample size. It was observed that 200 ms of capture is a good enough to make patterns and classify them.

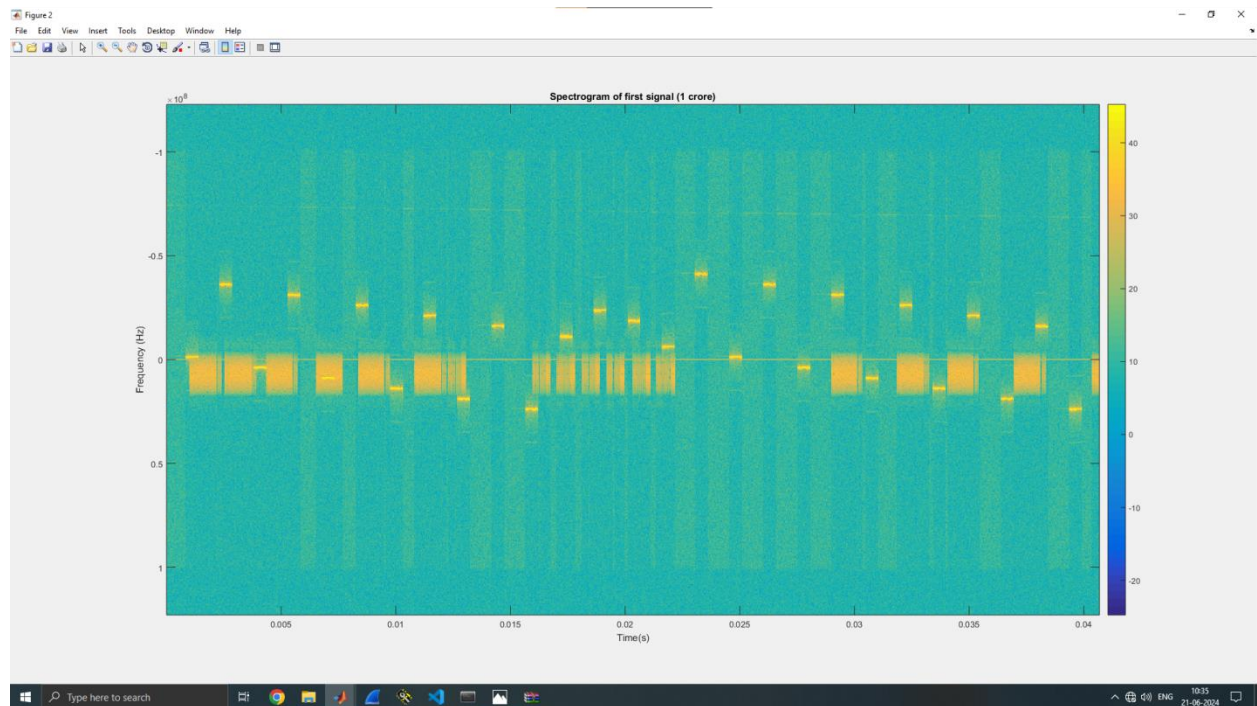


Fig3: Taking 1 crore samples from FS-GT2 + Wi-Fi data (Notice there is no pattern)

These images are then manually labeled using the YOLO Label software based on the controller names and stored in a “YOLO friendly manner”.

About manual labeling: This is done using the YOLO Label software:

After selecting the folder where all images are present, we select a .txt file having all the class names. These class names are then imported in the software directly and we can start with labeling.

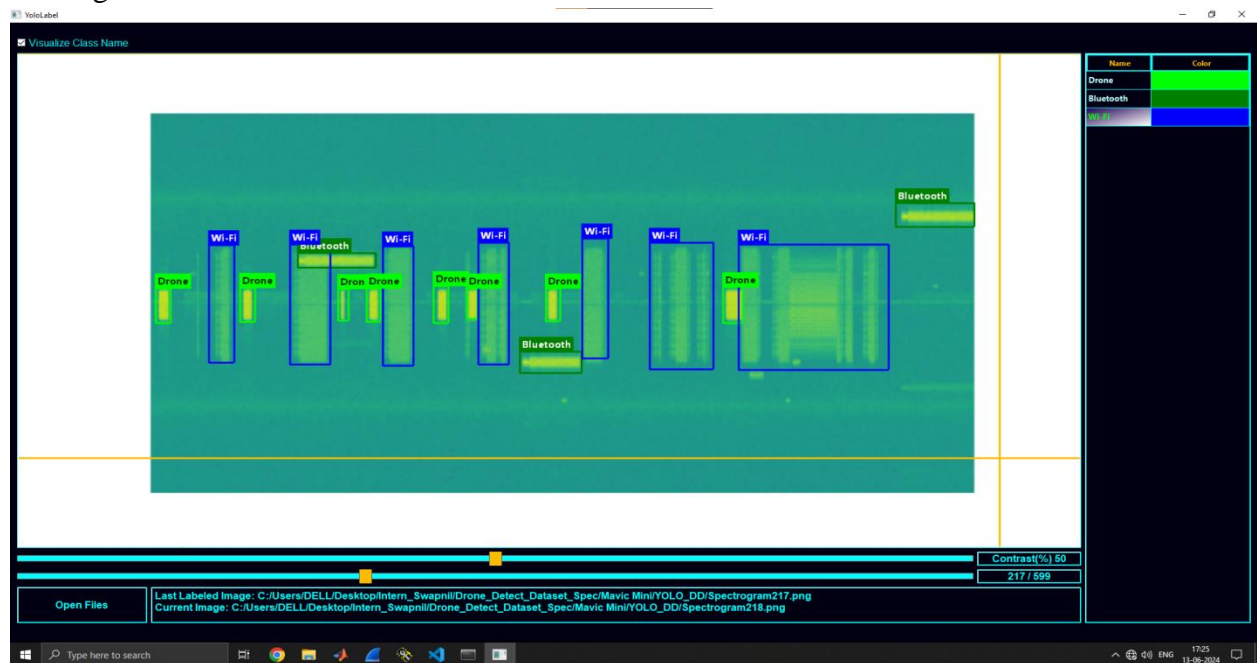


Fig4: Labeling using YOLOlabel (class names are on top left)

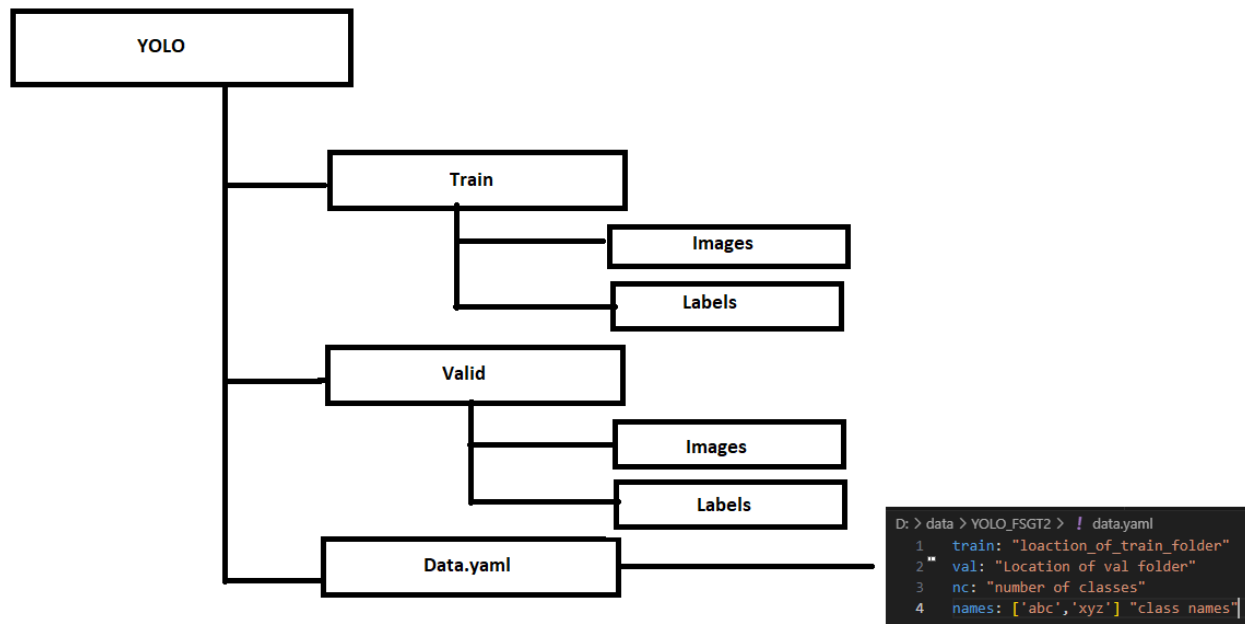


Fig5: This is the “YOLO friendly manner”

About the Model:

YOLOv8-nano model is then imported and trained on this data which is finally saved so that it can be loaded as and when required in the future.

About the C++-Python Wrapper:

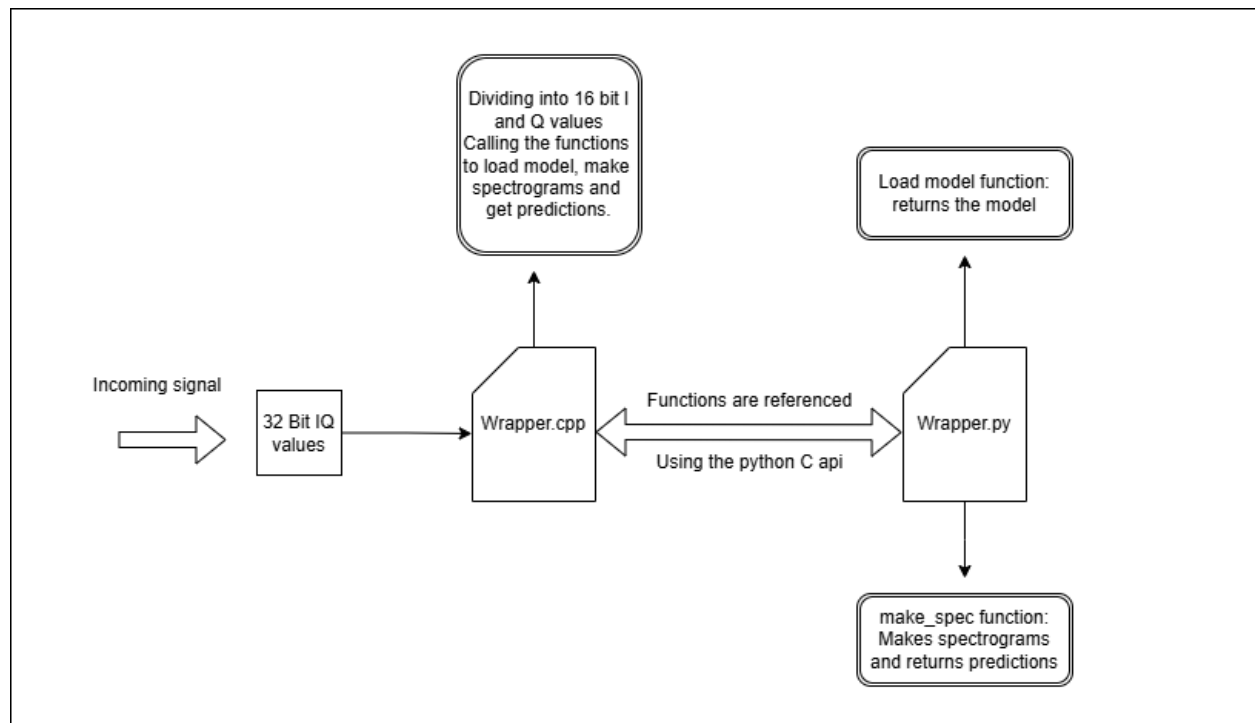


Fig6: Working of the python-C++ Wrapper

Final integration happens via the C++ code, but since all the libraries are imported and functionality is implemented using python, rather than converting the full code to C++, we link the corresponding functions in python to C++ using the Python-C api.

1. Requirements

1.1. Required modes of operation

The work in this module can be divided into 2 modes:

- Dataset Expansion: To add more data to the dataset. Data can be from other remote controllers or data of varying SNR range from remote controllers already added.
- Prediction mode: Wherein classes of incoming signal can be predicted using the trained model in realtime.

1.2. Software requirements

1.2.1. Requirements of Prediction mode

1.2.1.1. Getting prediction for incoming signal

- This part of the module, i.e., the C++-python wrapper is used to get inference/prediction from the incoming signal.

1.2.1.1.1. (E) Command Line Script

To use the wrapper, run this command in your terminal:

```
g++ -o model Wrapper.cpp -L "E:\anaconda\libs" -I"E:\anaconda\include" -I"E:\anaconda\Lib\site-packages\numpy\core\include\numpy" -lpython311
```

- -L "E:\anaconda\libs" is where the anaconda libraries are present
- -lpython311 means that I only want to link the python311.lib library (stored in libs folder)
- -I"E:\anaconda\include" will include all the necessary header files
- -I"E:\anaconda\include" -I"E:\anaconda\Lib\site-packages\numpy\core\include\numpy" will link the numpy header files

1.2.1.1.2. (D) Design of algorithm/architecture

- Running the C++ wrapper code will take input of the 32 crore IQ samples. These will be 32 bit values where the first 16 bits (from MSB) represent the Q values and the next 16 bits represent the I values.
- The wrapper function will extract the I and Q values and convert them into complex numbers. It will then send the reference of these complex values to the python function in Wrapper.py, which will then start plotting spectrograms of this data.
- While plotting, the spectrograms are converted into array format as .npy files.

- After Conversion is done, prediction is done using the already loaded model.
- Inference results are stored in a text file.

1.2.2. Requirements of Dataset Expansion mode

1.2.2.1. Adding support for more controllers

- To add support for more controllers, sample the data at 245.76 MHz sampling frequency and 200 MHz bandwidth. Keep center frequency according to the operating frequency of the controller. Use python code to extract I,Q data and plot spectrograms taking 53 Million samples in each spectrogram. Collect the data and save in “YOLO friendly manner”. Re-train the model on that data.

1.3. Hardware Requirements

1.3.1. Hardware platform requirement

ZYNQ Hardware, SBC/ Ubuntu ,PC, Ethernet

1.3.2. FPGA/Processor estimated (approx) Resource requirements

A CPU with clock frequency of 3 GHz and more.

1.4. Software environment requirements

The libraries used by C++ code include:

- Python-C header files, numpy header files, python311.lib file. These can be linked from the anaconda environment or if python is installed in the base environment and environment variables are changed accordingly.

The libraries required to plot and save the spectrograms are:

- Matplotlib, cv2, numpy, os, time

The libraries required by the prediction module are:

- Ultralytics, time, numpy, os

1.4.1. Exception Handling and error responses related requirements

Debug statements have been added in the codes.

Future Implementation requirements:

- Add more data and support for controllers by expanding the dataset.
- Add support for automated labeling.
- Use GPU for plotting of spectrograms.

TEST RESULTS

FOR

Drone Signal classification using YOLO v8

Table of Contents

1.	INTRODUCTION.....	12
1.1.	Module Overview	12

2.	SOFTWARE TEST REQUIREMENTS.....	12
2.1.	Name of the test site and test environment.....	12
2.1.1.	Software Items	12
2.1.2.	Installation. Testing and Control	12
2.2.	Testing against Software requirements	12
2.2.1.	Tests for the Requirements of Dataset expansion mode	16
2.2.1.1.	Results.....	13
2.2.2.	Tests for requirement of Prediction mode.....	14
2.2.2.1.	Results.....	14
2.2.2.2.	Inference time tests.....	16
2.2.2.3	Accuracy and recall scores.....	17

Table of Figures:

Fig1: I and Q values are separated and stored in a csv file.....	13
Fig2: Single spectrogram image of 53 million samples.....	13
Fig3: Dataset: Images are autogenerated and saved.....	14
Fig4: Results of prediction.....	14
Fig5: The visualized result.....	15
Fig6: Inference time analysis.....	16
Fig7: Time analysis for inference on 1 spectrogram.....	17

1. Introduction

- This chapter will give us an idea about how the model is performing in test environments.

1.1. Module Overview

The purpose of this system is to efficiently detect the presence of drone and classify the drone type using spectrograms as the fingerprint and YOLOv8 as the Deep learning model

2. Software Test requirements

2.1. Name of the test site and test environment

Lab testing for this is done on PC (i5 processor) with data captured from anechoic chamber.

2.1.1. Software Items

Python environment with C++ compiler installed. Matplotlib, cv2, numpy, os, time, Ultralytics libraries should be pre – installed. Header files for python and numpy should also be added.

2.1.2. Installation, Testing and Control

- After installing the said libraries in your python environment, set up the minGW compiler for 64 bit architecture.
- Run the following command in your terminal:

```
g++ -o model Wrapper.cpp -L "E:\anaconda\libs" -  
I"E:\anaconda\include" -I"E:\anaconda\Lib\site-  
packages\numpy\core\include\numpy" -lpython311
```

- Make sure to keep the iq_data file (that is used for testing) in the same folder, the model weight file in the same folder.

2.2. Testing against Software requirements

2.2.1. Tests for the Requirements of Dataset expansion mode

- For expanding the dataset, efficient extraction of data from .txt files and conversion to csv file and subsequent plotting is tested.
- IQ data captured using the ZYNQ hardware is given as input in the code and spectrograms of given size of signal are plotted and saved.

2.2.1.1. Results

	A	B
L	I	Q
2	15	4
3	12	10
4	8	14
5	2	16
5	-5	16
7	-10	13
8	-15	8
9	-17	2
10	-17	-4
11	-14	-10
12	-10	-14
13	-4	-17
14	3	-17
15	8	-14

Fig1. I and Q values are separated and stored in a csv file

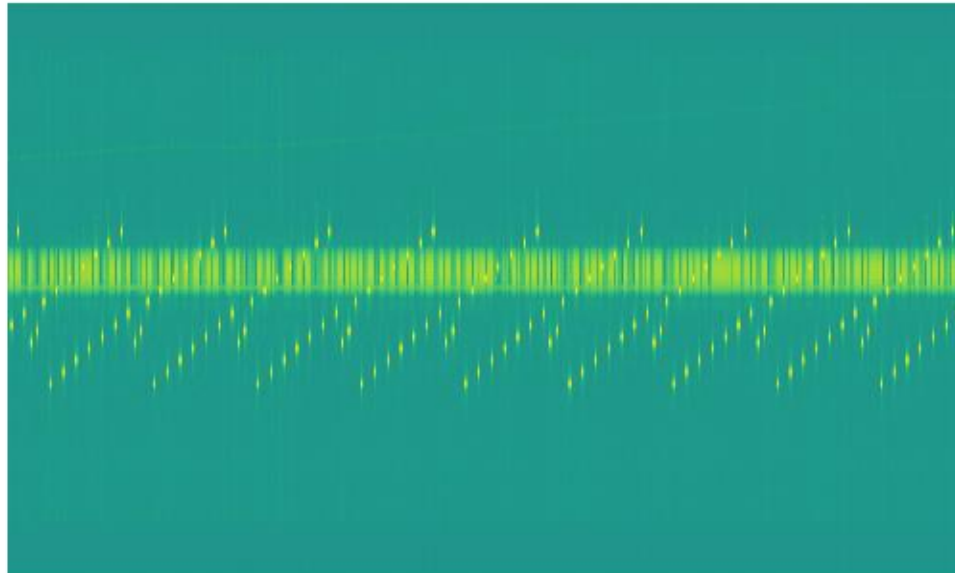


Fig2: Single spectrogram image of 53 Million samples

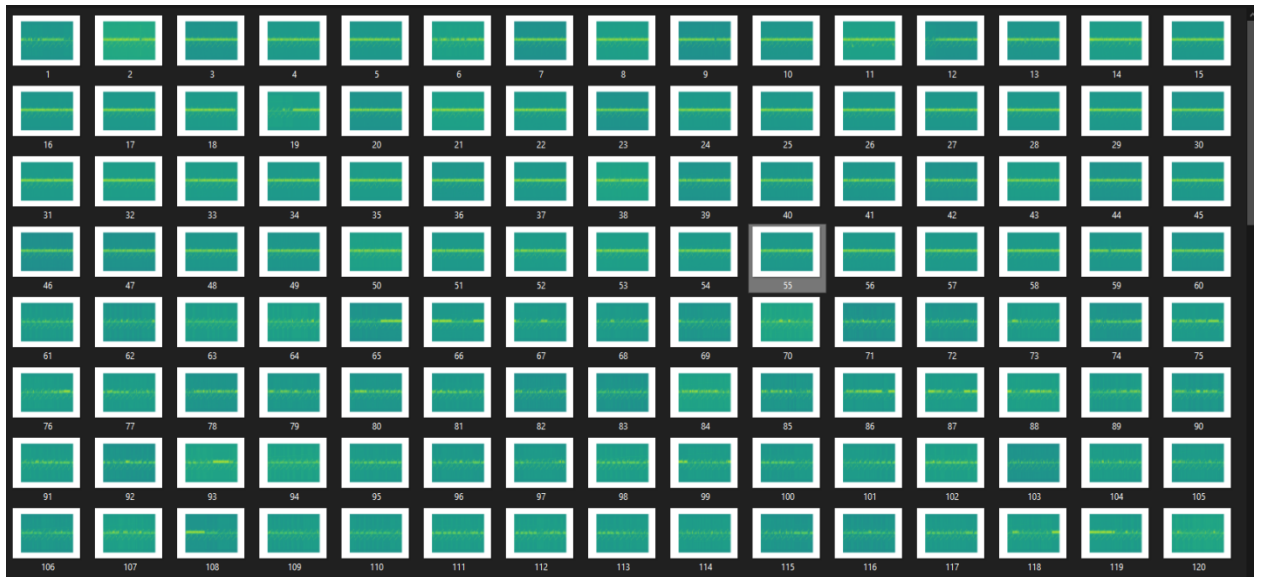


Fig3: Images are saved at a location

2.2.2. Tests for requirement of Prediction mode

2.2.2.1. Results

In prediction mode, we use the wrapper to input txt file having IQ data and the inference is finally saved in a result.txt file.

Inference on the full 32 crore samples gives results like:

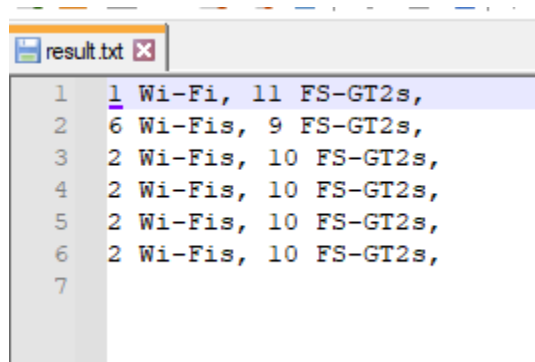


Fig4: Results of prediction

Additionally, we can also save the inference images having bounding boxes for visualization purposes. They will look like this:

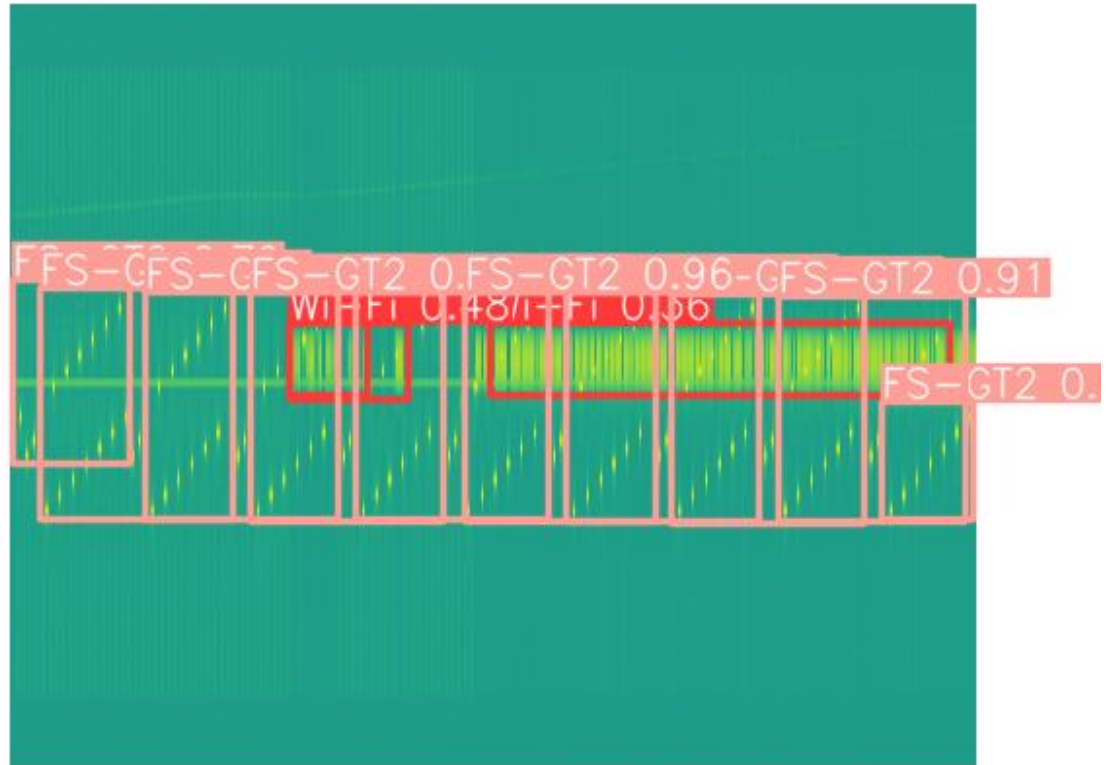


Fig5: The visualized result: The number shows the confidence score

2.2.2.2. Inference Time tests

When using the wrapper to plot the full 32 crore samples, this is how the time analysis looks like:

```
Module loaded successfully
Model loaded from python at 1720172646.7457206
Loading the .txt file
IQ numpy made
Save_spec function called at 1720172697.2520688
Spectrogram 0 plotting started at 1720172697.2520688

0: 480x640 1 Wi-Fi, 11 FS-GT2s, 109.4ms
Speed: 15.6ms preprocess, 109.4ms inference, 0.0ms postprocess per image at shape (1, 3, 480, 640)
Spectrogram 0 plotting ended at 1720172707.9095285
Spectrogram 1 plotting started at 1720172707.9095285

0: 480x640 6 Wi-Fis, 9 FS-GT2s, 159.1ms
Speed: 22.0ms preprocess, 159.1ms inference, 15.6ms postprocess per image at shape (1, 3, 480, 640)
Spectrogram 1 plotting ended at 1720172721.572714
Spectrogram 2 plotting started at 1720172721.572714

0: 480x640 2 Wi-Fis, 10 FS-GT2s, 107.5ms
Speed: 0.0ms preprocess, 107.5ms inference, 15.6ms postprocess per image at shape (1, 3, 480, 640)
Spectrogram 2 plotting ended at 1720172738.7243612
Spectrogram 3 plotting started at 1720172738.7243612

0: 480x640 2 Wi-Fis, 10 FS-GT2s, 121.5ms
Speed: 0.0ms preprocess, 121.5ms inference, 0.0ms postprocess per image at shape (1, 3, 480, 640)
Spectrogram 3 plotting ended at 1720172757.7589447
Spectrogram 4 plotting started at 1720172757.7589447

0: 480x640 2 Wi-Fis, 10 FS-GT2s, 108.8ms
Speed: 0.0ms preprocess, 108.8ms inference, 0.0ms postprocess per image at shape (1, 3, 480, 640)
Spectrogram 4 plotting ended at 1720172780.9089143
Spectrogram 5 plotting started at 1720172780.9089143

0: 480x640 2 Wi-Fis, 10 FS-GT2s, 122.3ms
Speed: 0.0ms preprocess, 122.3ms inference, 0.0ms postprocess per image at shape (1, 3, 480, 640)
Spectrogram 5 plotting ended at 1720172808.0922377
All Spectrograms plotted at 1720172808.0922377
prediction ended and saved in a file at 1720172808.0922377
1 Wi-Fi, 11 FS-GT2s,
6 Wi-Fis, 9 FS-GT2s,
2 Wi-Fis, 10 FS-GT2s,
2 Wi-Fis, 10 FS-GT2s,
2 Wi-Fis, 10 FS-GT2s,
2 Wi-Fis, 10 FS-GT2s,

Execution completed
Fri Jul 5 15:16:48 2024
```

Fig 6: Inference time analysis

As can be seen, the inference time (plotting spectrograms + getting inference) is 110 seconds. It is from the time save_spec function is called, up to the time when prediction is saved in a file.


```
Module loaded successfully
Model loaded from python at 1720174760.6972983
Loading the .txt file
IQ numpy made
Save_spec function called at 1720174819.2675211
Spectrogram 0 plotting started at 1720174819.2987697

0: 480x640 1 Wi-Fi, 11 FS-GT2s, 203.2ms
Speed: 15.6ms preprocess, 203.2ms inference, 31.2ms postprocess per image at shape (1, 3, 480, 640)
Spectrogram 0 plotting ended at 1720174833.2938418
All Spectrograms plotted at 1720174833.2938418
prediction ended and saved in a file at 1720174833.3060515
1 Wi-Fi, 11 FS-GT2s,

Execution completed
Fri Jul 5 15:50:33 2024
```

Fig7: Time analysis for inference of 1 spectrogram

Plotting for 1 spectrogram (53 million samples) takes 14 seconds as can be seen above.

2.2.2.3. Accuracy and Recall Scores:

For the RFD900 +Wi-Fi + Holybro model:

Precision: 92

MAP50: 90

Recall: 80

For the FSGT2+WiFi + AT9S pro model:

Precision: 88

MAP50: 88

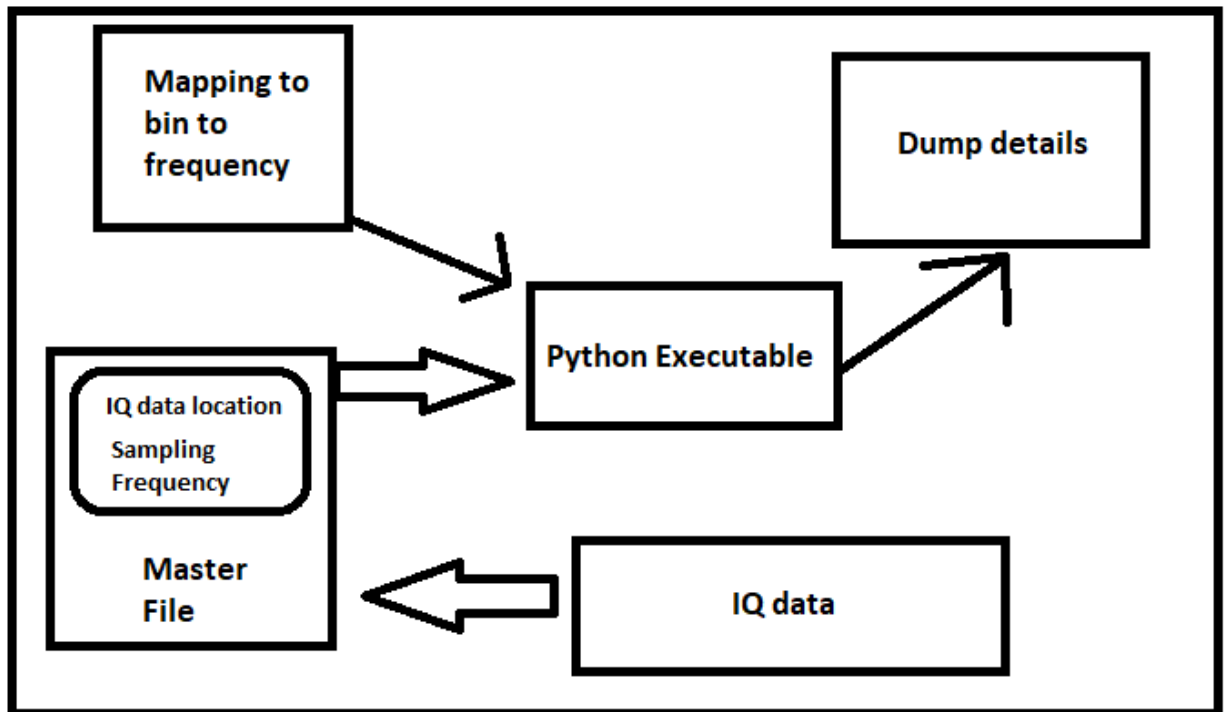
Recall: 82

Chapter 3: Packet info Header extractor using python

Overview:

Built a python application, which takes IQ data as input and processed the data to search for packet header, then the timestamp information is processed and saved. Additionally, spectrogram plot of the data is plotted and saved at a location.

Directory



Requirements:

- The files required are the mapping to bin values and a master file that will have information about the location of IQ data and the sampling frequency on which spectrogram will be plotted

```
C:\Users\DELL\Desktop\to_swapnil  
iq_data_0_1.txt  
245.76
```

Fig: Master file

- The location mentioned is where IQ data is saved.

- The mapping from bin to frequency looks like this:

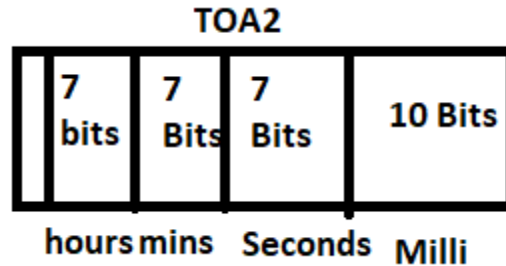
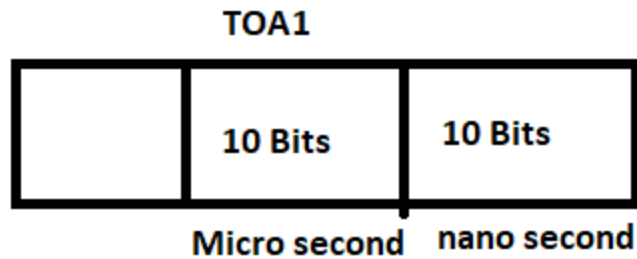
Bins	Freq
0	-5.2E+07
1	-1.2E+08
2	-1.9E+08
3	-2.6E+08
4	-3.3E+08
5	-4E+08
6	-4.5E+08
7	-5.2E+08
8	-5.9E+08
9	-6.6E+08
10	-7.3E+08
11	-8E+08
12	-8.6E+08
13	-9.3E+08
14	-1E+09
15	-1.1E+09

- After this the code parses the data and looks for “CFAD9371” value in the IQ data. If it is found, then the next 6 values are to be saved separately and removed from the original data.
- Spectrogram should be made from the leftover data.

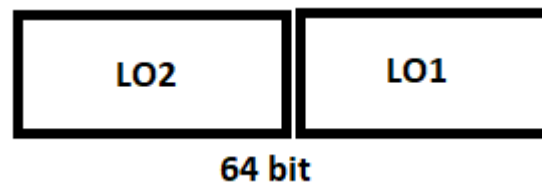
TOA1
TOA2
LO1
LO2
Freq
Bin

- These values are then stored separately in arrays.
- The data from the bin_mapping file is saved as a dictionary for faster access.

Details of how these captured values were extracted:
For TOA1 and TOA2



The LOA1 and LOA2 are 32 bit values each, they are appended and converted to int values. Then the twice of this value is written in the dump file.



This processing is done in the code, and the final details are written in a dump file that looks like this:

```
Header-1
0 hrs: 3 mins: 96 seconds: 9 milli-seconds: 946 micro-seconds: 303 nanoseconds : 2 : -52428800.0 : -122333866.7
Header-2
0 hrs: 7 mins: 75 seconds: 109 milli-seconds: 950 micro-seconds: 283 nanoseconds : 2 : -122333866.7 : -192238933.3
Header-3
0 hrs: 2 mins: 84 seconds: 15 milli-seconds: 953 micro-seconds: 269 nanoseconds : 2 : -192238933.3 : -262144000.0
```

The code snippet for data extraction is:

```

# write in final file
with open(f'{cur_dir}\\dump.txt', 'w') as f:
    for i in range(len(id)):
        f.write(f'Header-{id[i]} \n')

    # process toa1
    toa1_num = toa1[i]
    time1 = toa1_num & 0x3FF # extract the 10 bits from LSB
    time2 = (toa1_num>>10) & 0x3FF # extract the next 10 bits

    # process toa2
    toa2_num = toa2[i]
    time3 = toa2_num & 0x3FF # extract the 10 bits
    time4 = (toa2_num >> 10) & 0x7f # next 7
    time5 = (toa2_num >> 17) & 0x7f # next 7
    time6 = (toa2_num >> 24) & 0x7f # next 7

    # process loa1 and loa2
    loa1_value = loa1[i]
    loa2_value = loa2[i]

    loa_processed = (loa2_value << 16) + loa1_value # appending the values
    loa_processed = loa_processed*2

    # process bins
    bin_value = bin[i]
    matched_freq = -1
    if bin_value in mapping:
        matched_freq = mapping[bin_value]

    #frequency value
    frequency = freq[i]

```

This is then finally written as:

```

f.write(f'{time6} hrs: {time5} mins: {time4} seconds: {time3} milli-seconds: {time2} micro-seconds: {time1} nanoseconds : {loa_processed} : {frequency} : -{abs(matched_freq)}\n')

```

Chapter 4: User Manual for Wrapper

Step1: Head over to Wrapper.cpp and change this location to path where .txt file for testing is saved. [Line 105]

```
// Generate example signal in C++
vector< complex <double>> signal = load_input_data("C:/Users/DELL/Desktop/to_swapnil/iq_data_2.txt");
```

Step2: Now change the wrapper-helper.py. Put location where the model weights are stored: [Line 10]

```
model = YOLO('C:\\Users\\DELL\\Desktop\\test_cpp\\final_testing\\best.pt')
print(f"Model loaded from python at {time.time()}")
```

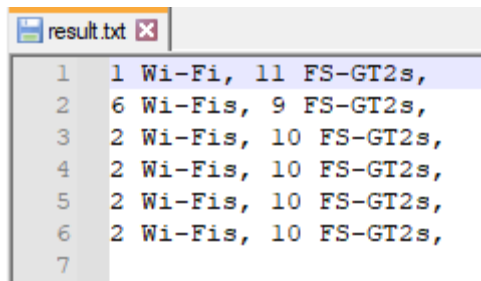
Step3: Run the following command in terminal:

```
des> g++ -std=c++17 -o model Wrapper.cpp -L "E:\\anaconda\\libs" -I"E:\\anaconda\\include" -I"E:\\anaconda\\Lib\\site-packages\\numpy\\core\\include\\numpy" -lpython311
```

This command is also written on page number 10.

Step4: Then run the executable by running “./model”

The inference will be stored in a result.txt file and will look like:



The screenshot shows a text editor window titled 'result.txt'. It contains a list of seven items, each on a new line. Each item consists of a number followed by a label and a value. The labels are 'Wi-Fi' and 'FS-GT2s'. The values are 11, 9, 10, 10, 10, 10, and 10. The first line is highlighted in blue.

Line	Label	Value
1	Wi-Fi	11
2	Wi-Fi	9
3	Wi-Fi	10
4	Wi-Fi	10
5	Wi-Fi	10
6	Wi-Fi	10
7	Wi-Fi	10

Chapter 5: Solution - Approach

How the approach was finalized?

Before coming to the conclusion of plotting spectrograms at 245 Msps and plotting 53 million samples per spectrogram image, the following preliminary work was done:

- Worked on testing the approach of collecting the drone signal using software defined radios (HackRF & PlutoSDR) and capture the data using universal radio hacker software.
- Studied about the concepts of Signal Processing.
- Did literature review of numerous papers related to drone signal identification.
- Worked and tested the approach using open-sourced datasets – DroneDetect, Pure Wi-Fi data, VTI Drone dataset.
- Selection of the object detection approach to classify.
- Selection of the model.
- Collecting the data at 245 MSPS and proceeding with dataset collection followed by model training.
- Building the wrapper to use the model in real-time using the python-C api

The subsequent part of chapter discusses how each of these was performed.

Work and conclusions on Universal Radio hacker

- I started with learning about the various functionalities and working of the URH software
Design of algorithm of URH:
 - 1.1.Selection of **SDR** (Software design radio)
 - 1.1.1. It helps us in capturing the raw radio signals from airwaves.
 - 1.2.Scanning the spectrum to find the target frequency and record signal.

1.3.Interpretation:

- 1.3.1. It involves converting the received sine waves into bits. This process is called demodulation.
- 1.3.2. Received signal is converted into a stream of messages of varying bits.

1.4.Analysis:

- 1.4.1. Implement protocol reverse engineering. We can analyse the headers, find patterns or use the URH built-in decoder to find the protocol associated with the signal.
- 1.4.2. Also perform decoding if the signals are encoded before transmission.
- 1.4.3. Add/Remove participants from the signal if we detect varying RSSI (Received signal strength indicator).
- 1.4.4. Assign labels to parts of the messages to identify fields like length, address, etc.

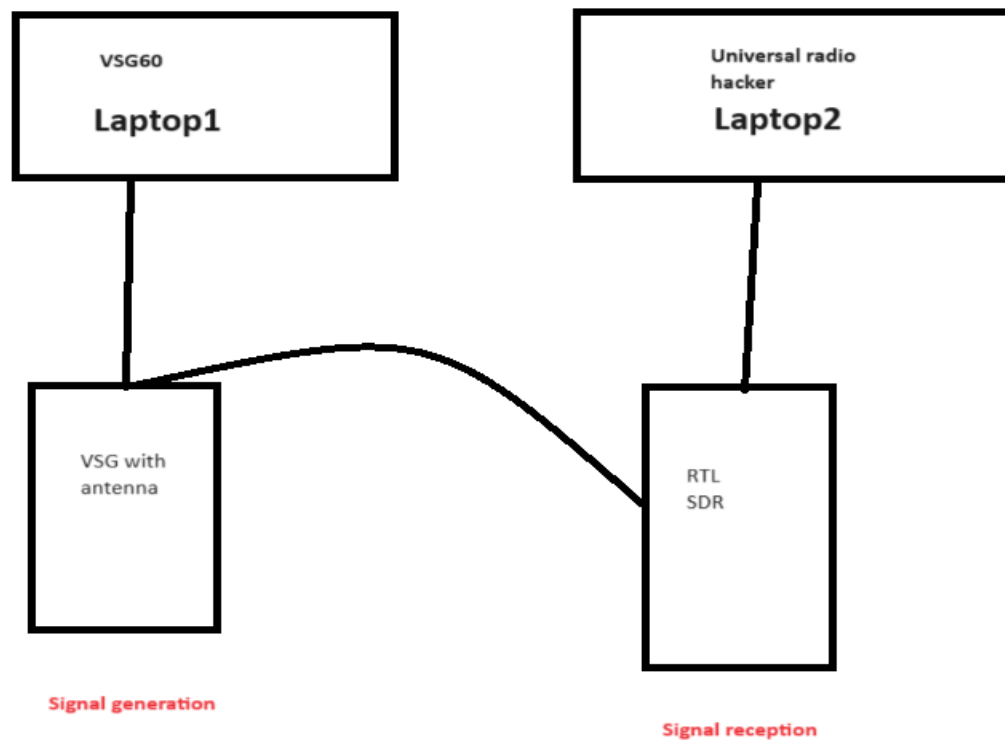
1.5.Generator:

- 1.5.1. Start manipulating the messages either by using fuzzing operation (where we enter a range of values for the target label and automate the process) or doing it manually.

1.6.Simulation:

- 1.6.1. For stateful protocols, sequence numbers are also checked by the receiving device, so we need to modify it in the fuzzed message.

- The initial work was to see and find out if I could detect the modulation type of signal using the URH software:
- URH needed to be fed signals, which needed to be captured using software defined radios (SDRs).
- I learned about signal terminologies like Bandwidth, sampling rate, FFT plots, signal analysers, IQ data.
- After this, I learned using the SDRs and captured real signals using Pluto-SDR, RTL-SDR, HackRF.
- This is how the setup looks like:



- These were the equipment:



Learnings:

After doing this, what I learned was:

- RTL SDR has to be discarded because its range of capturing frequency is 500 KHz to 1.75 MHz, the drone signals that need to be demodulated work at 2.4 GHz. Hence it should be discarded as is.
- HackRF & Pluto SDR can capture the signals. But URH only has support for ASK, FSK and PSK demodulation types, and some drone signals use OFDM as well. OFDM signals could not be decoded using URH.
- HackRF has a bandwidth of 20 MHz and PlutoSDR has a bandwidth of 61 MHz. The drone hopping range is more than both of these, so to capture it fully, both of these could not be used.
- Additionally, a lot of times, URH was not correctly able to detect the ASK, FSK as well.
- Since URH is an open-source software, I went through its codebase to find out the problem. I realized that the code logic that it used is very naïve and any diversion from ideal condition (which happens in real life) caused it to fail.
- This is how the code worked:

How to detect demodulation type of each message?

- Remove data with 0 magnitude
 - If samples removed is >3 , then **OOK**
 - Apply wavelet analysis, this will give us the variance of Magnitudes of wavelet (x) and variance of magnitudes of normalized wavelet (y).
 - If $x > 1.5*y$ ->**ASK**
 - If $x > 10*y$ ->**PSK**
 - Else: Apply FFT
 - If peaks are present: **FSK**, else **OOK**
 - OOK is interpreted as ASK in the end.
- I also captured the signals of flysky drone controller, but URH was not able to demodulate or decode them.

So, the next part of the study was to search for alternate SDR software that could efficiently find the modulation type.

For this:

I read about and tested the following software and made this conclusion:

Proposed Solution:

1. Use GNU Radio with custom blocks like gr-inspector for OFDM decoding and demodulating. It is a typical approach and will consume time. It estimates the parameters subcarrier spacing, symbol time, FFT size and cyclic prefix length for an input signal.
2. Other available software:
 - Gqrx: supports AM, SSB, CW, FM-N and FM-W (mono and stereo) demodulators only.
 - SDRangel: No concrete info about available demodulators, but some sources say that it supports OFDM.
 - HDSDR, CubicSDR, SDR# don't support OFDM modulation.

Supported Modulations of various SDR software:

1. **GQRX**: AM, SSB, CW, FM-N, FM-W (mono and stereo), and special FM mode for NOAA APT
2. **SodiaSDR**: Decodes AMSS, DCF77, RDS, and RF
3. **HDSDR**: AM, ECSS, FM, SSB, and CW
4. **SDRTouch**: Demodulates broadcast FM, AM, narrowband FM, USB, and LSB signals.

This work concluded finally with the following learnings:

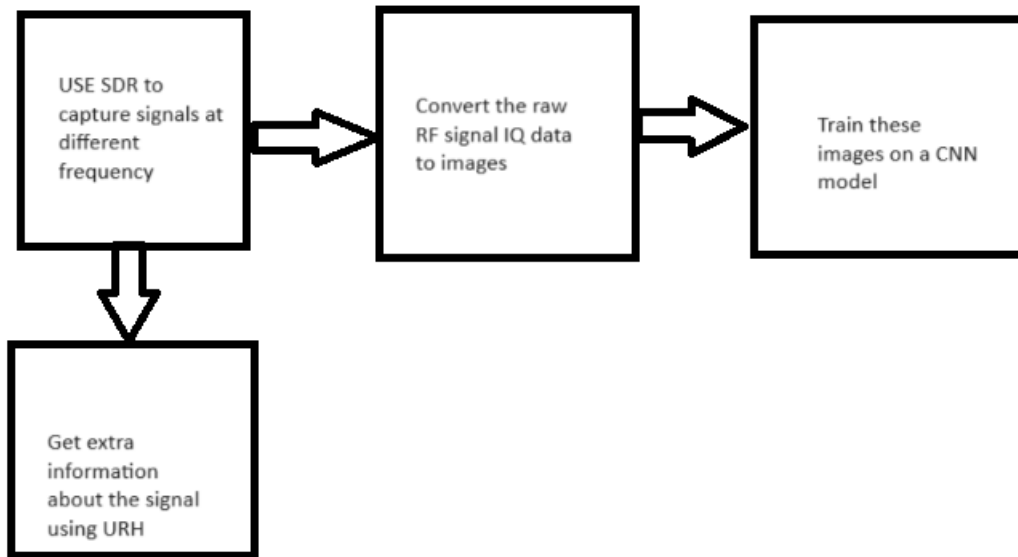
- Universal radio hacker cannot be used for demodulating or decoding the signals.
- It can still be used for collecting the signals using software defined radios though (HackRF, PlutoSDR).

Concepts of Signal Processing

1. **IQ data of RF signals:** It refers to the in-phase (I) and quadrature (Q) components, which represent the amplitude and phase of the signal. It is a 32-bit number with the first 16 bits representing I and the other 16 bits showing Q.
2. **Importance of digitising analog signal:**
 - It becomes compatible with digital systems.
 - Ease of storage and transmission
 - Becomes easier to implement signal analysis
3. **Sampling:**
 - It is the process of converting a continuous signal into a series of discrete values by measuring the signal's amplitude at regular intervals.
 - sampling is performed by using analog to digital converters along with software defined radios. The sampling rate determines how frequently the ADC samples the analog signal per second.
 - Sampling frequency: it refers to the number of samples taken from a continuous signal within a given time frame.
4. **Bin value:** The bin value of each sample refers to the magnitude of the signal's frequency content within a particular bin after the FFT computation. This magnitude represents the strength or amplitude of the signal at that frequency range.
5. **FFT:** Fast Fourier transform helps us in converting time domain signal into its frequency domain representation. Using this, we can analyse the phase and amplitude of each frequency component separately.
6. **Calculating frequency from Bin value:** The frequency resolution of each bin can be calculated as the sampling frequency divided by the FFT size. The bin with the highest magnitude or power corresponds to the dominant frequency component of the signal.
7. **Demodulation** of signals: It is the process of extracting the original information signal from a modulated carrier signal.
 - AM: Used in broadcast radio receivers
 - FM: Used in FM radio receivers
 - PSK: used in digital communication systems such as Wi-Fi, Bluetooth, and satellite communication
 - ASK: Used in Keyless entry systems, Remote sensors
 - OFDM: Used in Wi-Fi and digital video broadcasting

Literature Review

At that time, the approach to solve the drone classification problem was something like this:

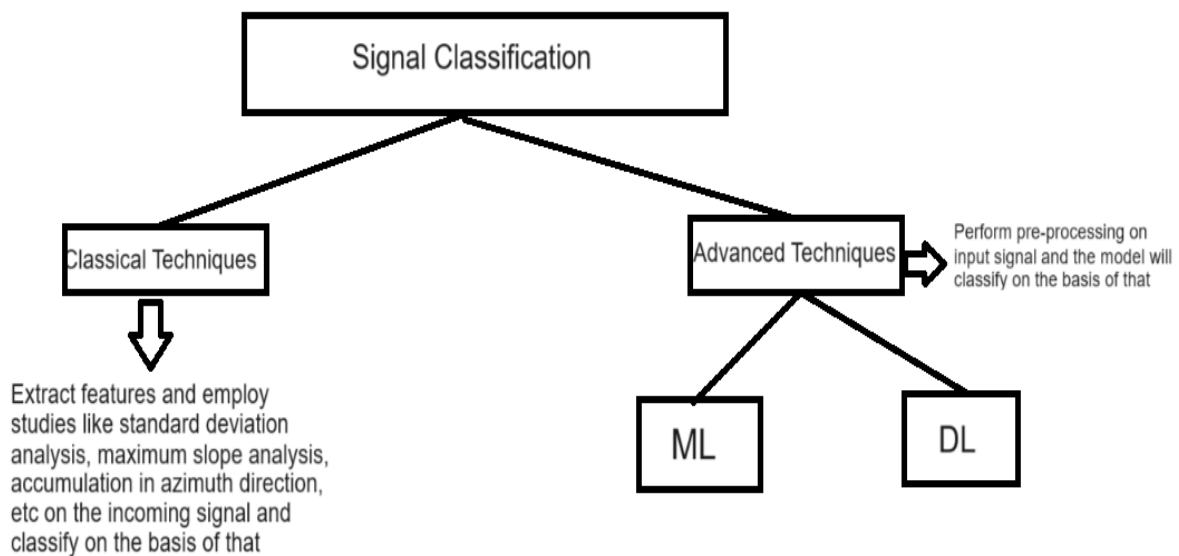


As time progressed, this approach eventually changed.

This is when I started looking at research papers, to know what is it that can be classified as images, meaning what can be used as a fingerprint to extract best feature for the model.

After going through many research papers, these were my conclusions:

The overall drone classification problem looked like this:



This was the split between different approaches:

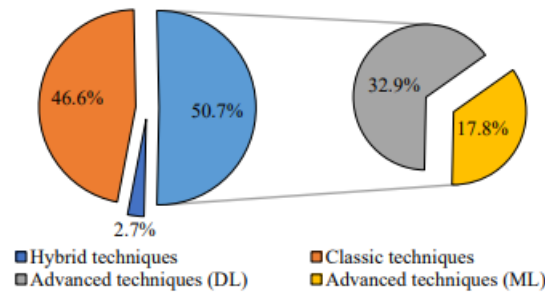
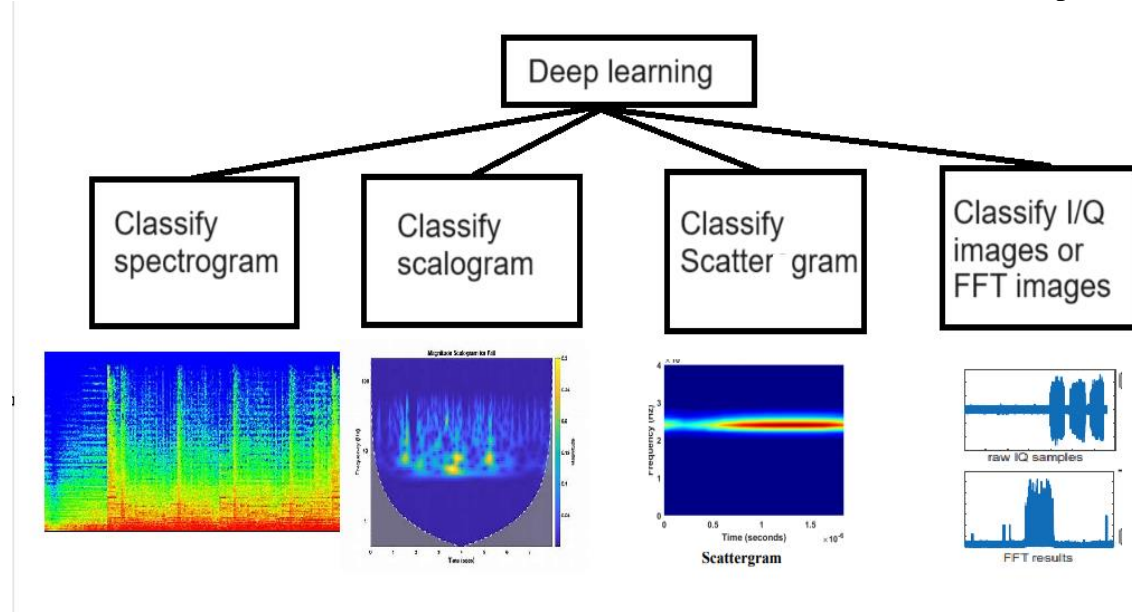


Figure 1. Quantitative comparison of RF techniques according to the method of processing input data.

After Full literature review, these were the features that could be extracted and exploited.

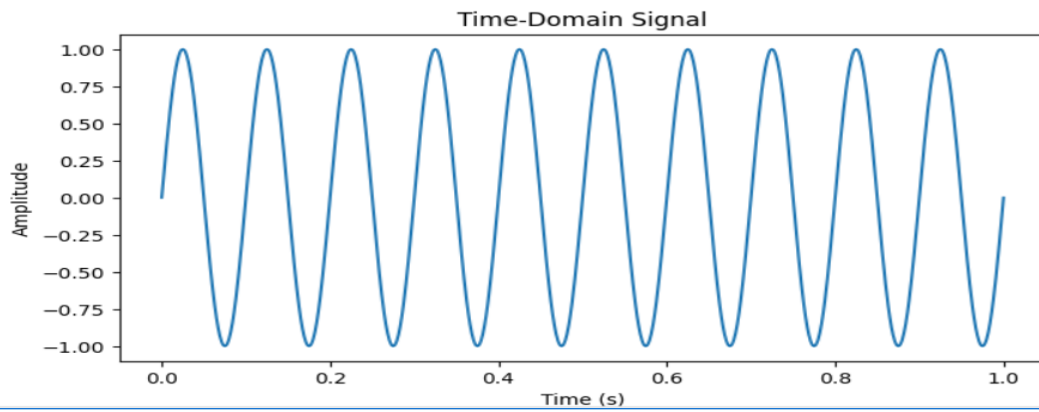


So, I started with plotting spectrograms, scalograms and FFT plots for randomly generated signals.

This is when I learned about various representations like the Amplitude-time, frequency-Amplitude, frequency-time, learned the basics of Wavelet scattering transform.

```
[55]: import numpy as np
import matplotlib.pyplot as plt
t = np.linspace(0, 1, 1000)
f = 10
signal = np.sin(2 * np.pi * f * t)

plt.figure(figsize=(8, 4))
plt.plot(t, signal)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Time-Domain Signal')
plt.show()
```

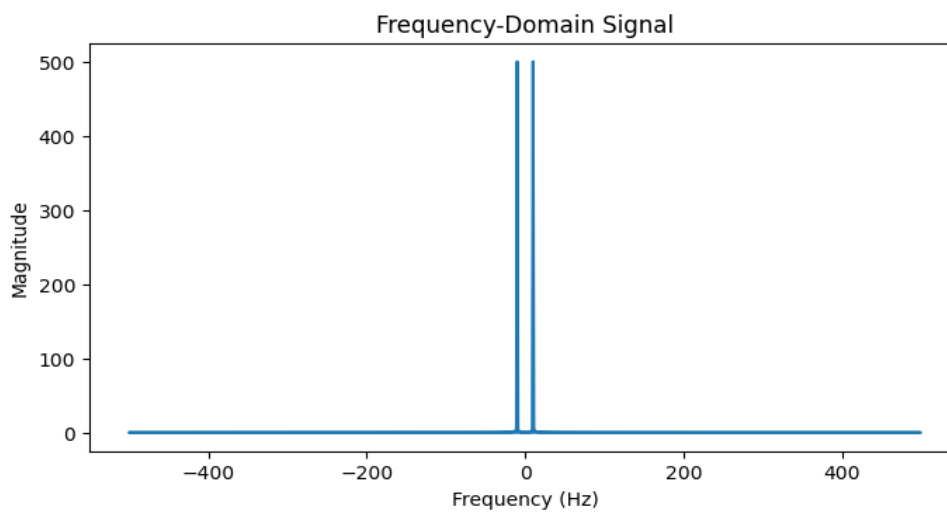


```
[56]: import numpy as np
import matplotlib.pyplot as plt

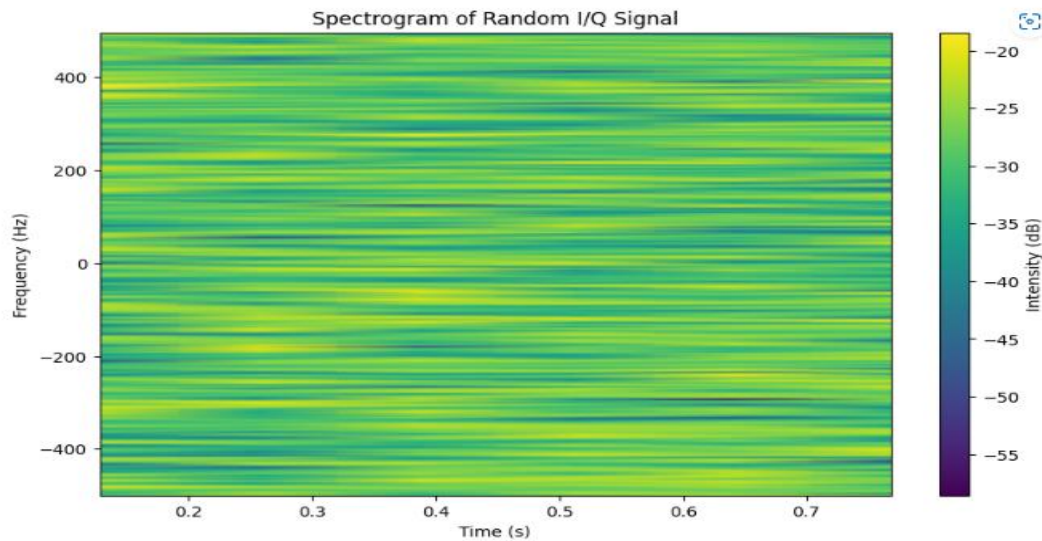
t = np.linspace(0, 1, 1000)
f = 10
signal = np.sin(2 * np.pi * f * t)
sample_rate = 1000

# Perform FFT
fft_result = np.fft.fft(signal)
freq_axis = np.fft.fftfreq(len(signal), 1/sample_rate)

plt.figure(figsize=(8, 4))
plt.plot(freq_axis, np.abs(fft_result))
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Frequency-Domain Signal')
plt.show()
```



For spectrograms:



After this, I narrowed down on plotting spectrograms for our problem because of two reasons:

- Availability of multiple Libraries to do the plotting.
- Spectrogram is always better than amplitude-time and power-frequency plots because of the additional information that it provides. It tells us the power of every frequency at each point of time. Scattergram vs Spectrogram was the final choice. Went along with second one for the previous reason and since most papers were using the same.

*To read more about this, refer to the research papers provided with this document.

Started working with open-source datasets

The type of data available online for drone signals include acoustic data, drone images, RF signal data. The datasets for RF signal data include:

- CardRF
- VTI Drone dataset
- Pure open sourced wifi data
- Drone Detect dataset
- DroneRF dataset

Other than this, We had data collected for drone controllers by our own acquisition system which was sampled at 30.72 MHz. The controllers covered were:

- Radio link AT9S
- Flysky CT6B
- Flysky FSGT2
- FlySky i6
- Flysky i6s
- Mavic Pro drone
- Radio Link RFD900
- Radio link T8FB

Spectrogram plotting for all these datasets was completed and this is when the code for data processing was written.

Drone Detect dataset was the most useful amongst all these since it had data from Drone (Mavic mini), from Wifi, bluetooth and all the three. Plotting these data strengthened the idea that spectrogram can differentiate between incoming signals and help us classify the signals efficiently.

My first model that was trained came from the Drone Detect Dataset, wherein I classified between the Drone, Wi-Fi and Bluetooth signals.

Selection of the Object detection approach

To classify the controller type using spectrogram, either I could use the object detection approach or image classification approach.

I chose the object detection approach for the scalability offered by it. The image classification approach could detect and classify amongst labels on which it was already trained and since in our case, there could be multiple drones in an area, collecting and training on this data was a tiresome task.

Hence, I took the object detection approach in which the model first detects the presence of a signal and then classifies it.

The laborious task in this approach is to label the data manually before training. I did it using the YOLO label software.

After that was done, the model could efficiently detect and classify the incoming signal.

Why YoloV8

Yolo models are the best object detection models that are open sourced currently. The question was to select the model of YOLO that will best suite our interest. The challenge was to select between V8 and V10. The benefits offered by V10 were that it could detect the presence of very small objects whereas V8 offered faster inference time. Both of these used different architectures. Since for us, the drone signal would cover good part of the bandwidth, detecting very small objects was not a question.

Hence, weighing the benefits, I moved along with YOLO v8.

Collection of Final Dataset

After this, since we wanted to capture the full bandwidth, I decided to go along with 245 MSPS sampling frequency. This would capture the whole 200 MHz bandwidth with centre frequency at 2450 MHz.

Suman sir collected the data from anechoic chamber. He collected data for the following:

- Pure Flysky FS-GT2, Pure Wi-Fi (802.11 n, 802.11 g), interference of all these signals.
- Radio Link HolyBro, Radio link RFD900

Then plotting and YOLO labelling for all of this data was completed. Then, a YOLO nano model was trained and predictions were made. It was able to detect the presence of signal (FSGT2) in the test data.

Making a Wrapper

The Code and functions written for the work done till now was done in python, to link this module to C++ (requirement of hardware), I wrote a wrapper in C++ that could efficiently process the real-time incoming signals and give inference. More about this is already written in the SRS doc.