Final Report - InvisiFall

Contactless Fall Detection Using a FMCW mmWave Radar and HVRAE Machine Learning Model

Rhanmouni, Saad - #8876451 – Mechanical Engineering

Wu, Jerry - #300073359 – Electrical and Computer Engineering

Qureshi, Daniyal - #300284707 – Electrical and Computer Engineering

Wen, Fucheng - #300309097 – Digital Transformation and Innovation

Zaytoun, Ali - #300274341 – Electrical and Computer Engineering

Kaushik, Bhupali– 300372625 – Electrical and Computer Engineering

GNG 5902

Faculty of Engineering – University of Ottawa

To Professor Hanan Anis & Dr. Miodrag Bolic

April 14th , 2024

Acknowledgments

We would like to express our deepest appreciation to Dr. Miodrag Bolic, whose supervision and expert guidance were the cornerstones of our academic journey and the successful completion of this project. His dedication to academic excellence and his willingness to impart his vast knowledge have been a great source of inspiration throughout this endeavor.

Special thanks are owed to our collaborator, Mr. Hasem Mahdavi, CEO at Zodiac Light Waves Inc. whose unwavering belief in our vision was matched only by his generous contribution of time and invaluable feedback, significantly shaping the trajectory of our project.

Our sincere gratitude extends to Mr. Ebrahim Ali, Ph.D. candidate, for his invaluable assistance with the machine learning models that were crucial to this work. His expertise and support played a significant role in overcoming the challenges we encountered.

We are also particularly grateful to Dr. Reza Argha from Sydney University for his generosity in sharing the training data for the PointNet Model. This crucial contribution laid a foundational stone for our research, allowing us to build upon established work with new insights and innovations.

We must also acknowledge the author of the paper that served as a lighthouse for our final prototype development, Dr. Feng Jin. His research and findings provided us with the methodological clarity and technical rigor essential for our project's success.

Furthermore, we extend our special thanks to Professor Hanane Anis, whose mentorship and astute insights from an external perspective were instrumental in refining our project's scope and impact. Her contributions were a beacon of academic rigor and practical relevance.

To all who have contributed to this journey, either directly or indirectly, we extend our heartfelt thanks. Your support has not only facilitated this project but also contributed to our collective personal and professional growth.

Table of Contents

List of Figures:

List of Tables:

Abstract

The phenomenon of elderly falls within residential homes presents a pressing challenge, demanding immediate and efficient response to prevent serious injury or fatality. These incidents not only threaten the well-being of residents but also impinge upon their confidence in feeling secure within these facilities. A robust solution could significantly reduce response times and improve overall safety, thereby making a considerable impact in the field of elderly care.

The core problem we address is the reliable detection of falls among the elderly in a residential environment. Current mainstream solutions such as wearables with accelerometers and barometers are hindered by practical limitations — the elderly, especially those with cognitive impairments like dementia, may forget to wear, charge, or find these devices inconvenient. A viable solution must therefore be unobtrusive and require minimal interaction from the residents.

To circumvent the limitations of wearable technologies, we propose a novel, contactless detection system based on radar technology. This approach forgoes cameras to reduce privacy concerns and employs a radar system to generate a point cloud over a predetermined range. A minicomputer processes this data using a sequential machine learning algorithm to detect falls. Upon detection, the system promptly activates an alarm in the home via a relay connection, ensuring immediate notification without compromising resident privacy or comfort.

Preliminary results indicate that our radar-based system can detect falls with a high degree of accuracy over 90%. Although quantifiable results regarding response time improvement and fall detection rates are beyond the scope of this abstract, the radar system shows promise as an effective alternative to wearables and surveillance-based methods.

The proposed radar-based fall detection system offers a significant advancement in elderly care technology, respecting privacy while maintaining a high level of accuracy. It lays the groundwork for future innovation, including fall prevention and vital health sign monitoring. The utilisation of mmWave technology in this context may herald a new era in non-intrusive resident monitoring, potentially generalizable to various care settings.

Introduction

Problem Definition:

Design an easy to implement fall detection solution used in residential homes by nurses such that it is accurate, contactless (non-wearable), camera-free, cost-efficient with instant notifications to the nurses. It is to be able to accurately discern the elderly subject and detect all types of falls with no false alarm.

Problem Statement:

In residential settings, falls among the elderly and those with mobility impairments present a significant health risk, often leading to severe injuries and critical emergencies. Rapid detection and response to such falls are imperative, yet current solutions often intrude on privacy or require direct contact, which can be uncomfortable or impractical.

Project Overview:

This project aims to develop an innovative fall detection system that harnesses mmWave FMCW radar technology. Our objective is to create a non-intrusive, privacy-respecting mechanism that accurately identifies falls, differentiates them from other movements, and instantly notifies caregivers, improving response times and outcomes for residents in non-hospital environments.

Scope and Objectives:

Our approach encompasses a comprehensive hardware integration process, establishing seamless communication between the radar technology and data acquisition systems. We intend to explore a range of data analysis and machine learning algorithms, seeking to identify the most effective method for fall detection through extensive testing—including both frame-by-frame and sequential data analysis.

Notification System Testing:

A critical component of our system is the notification protocol, for which we will test both relay and text message-based systems. Our aim is to ensure that the chosen method provides reliable and immediate alerts to facilitate a quick response.

Expected Outcomes:

The anticipated outcome is a fully functional fall detection system with a proven algorithm offering high accuracy, capable of integrating into existing care infrastructure. By comparing different models and notification methods, we expect to conclude with a solution that sets a new standard for fall detection in terms of both efficiency and respect for user privacy.

Benchmarking and Specifications

Benchmarking:

To conduct better benchmark testing, we need to more objectively analyze similar products on the market. For this reason, we rate similar products on the market from five different aspects in the table below for analysis.

Table 1: Product Rating Scale

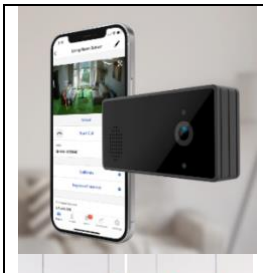
Criteria	Contactless	Price (<500$)	Privacy	Reliability in Identifying falls	Can monitor Vitals
Points	1/5	1/5	1/5	1/5	1/5

Comparison of Existing Solutions

In this section we put together in a table the ratings and final specs of four different products along with our researched reviews of it to help us with our benchmarking.

Table 2: Comparison of Existing Solutions

Product Name	Ratings	Final Specs	Comments
SensFloor 	4/5	1. Around 4354.11 ~ 5079.8 minimum + Service agreement cost for annual maintenance and regular software updates + Require renovating floor to install on existing floor. 2. Require 0.5 W/sqm 3. 98% Accuracy 4. Last 20 years minimum 5. Does not use camera to monitor 6. Contactless	This has the largest coverable area despite the expenses to deploy the system in a room. It can detect fall detection to slow/fast falls. It can also set up guide lights at night for elders. Overall, it provides good accuracy in fall detection at all places with minimum change in the room's appearance.
AltumView Fall Detection	4/5	1. $ 299.99 hardware cost + 0~6.7 cad / month for its feature 2. Stick Figure is sent 3. Contactless 4. Good accuracy comparable to the	Provides quite a good angle of monitoring (185 degrees). It implements a video camera that features face recognition to collect statistics.

 		radar. Numerical value not found 5. Batteries is not required	However, it has the worst privacy among the 4 products since it sends stick figures to its app for monitoring.
UnaliWear Kanega Smartwatch 	3/5	1. $ 149.99 + $ 59.95 monthly plan 2. Response time of 46 seconds 3. Not Contactless 4. Batteries need to be recharged every 1.5 days	Unlike other products, this can be worn at all places. But it has the shortest battery lifetime Although, it comes extra battery so you can charge one battery a day which is tedious.
Medical Guardian Mobile 2.0 	2/5	1. Not Contactless 2. < 90% accuracy 3. Require charging every 3~5 days 4. Minimum $ 44.95/month + $ 10/month for fall detection service 5. Response time of 24~30 seconds	Although this product is the cheapest among the other 3, it only has a one-way communication line, which is inconvenient in a false alarm situation. It is also required to push an external button if fall detection service is not available.

Metrics and Units:

After comparison and investigation, we understood the important indicators required for similar products to realize their functions and determined the units. We have placed the important indicators and their units in the following tables.

Table 3: Metrics and Units

Metric Descriptor	**UNIT**
Notification Delay	Seconds (s)
Sample Rate needed for frame detection	Hertz (Hz)
Computation power requirements	Floating-point operations per second (FLOPS)
Power Consumption	Watts (W)
Uptime	Hours (h)
Accuracy	Percentage (%)
Range Resolution	Centimeters
Elevation Resolution	degrees
Detection Range	Meters

Linking metrics to needs:

After we determined the important indicators required for the product, we related these indicators to the needs raised by customers and made them into the following table. This can help us analyze what indicators need to be achieved to fulfill customer needs.

Table 4: Linking metrics to needs

			1	**2**	**3**	**4**	**5**	**6**	**7**
		Metrics	Notification Delay	Computation power requirements	Power consumption	Up Time	Accuracy	Range resolution	Detection range
	Needs Index								
1	An Artificial Intelligence based fall detection solution		X	X	X		X		

2	The solution should not use Cameras and machine vision		X	X	X		X		
3	The solution should be contactless		X	X	X		X		
4	The system should have very low lag time between acquisition, processing and notification.		X	X	X		X	X	X
5	The system should have a very good coverage		X	X	X				X
	Machine learning algorithm should be trained on distinguishing the subject			X	X			X	X
6	Machine learning algorithm should be trained on distinguishing a fall from the subject sitting, or going to bed			X	X		X		
7	The device should be battery free				X	X			

8	The solution should be 24/7				X	X			

Assign Marginal and Ideal Values

Based on the analysis of the above tables, we derived a marginal and ideal value for each metric to achieve customer needs.

Table 5: Assign Marginal and Ideal Values

Metric ID Number	Metric Descriptor	UNIT	Marginal Values	Ideal Values
1	Fall Accuracy	Percentage (%)	75	90
2	Sample Rate needed for frame detection	Hertz (Hz)	100	200
2	Notification Delay	Seconds (s)	30	20
7	Range Resolution	Centimeters	8	4
4	Power consumption	Watts (W)	100	50
9	Detection Range	Meters (m)	4	5
5	Uptime	Hours (h)	24	24

Our initial goal for the fall detection is set to a relatively low percentage (recommended by our mentor) since we are testing out new radar technology in a laboratory environment.

According to Nyquist sampling theorem, it is important that we sample the analog signal at least twice the frequency of the highest component. Hence, for the falls, the highest harmonics are observed in the range of 100Hz to 200 Hz.

Notification delay of 20-30 seconds seems reasonable and achievable with the existing technology. This delay is enough for a retirement home caregiver to respond to any fall emergency.

Apart from the above main indicators, secondary metrics provide another way to quantify system performance, not as critical as the primary indicators.

The range resolution is important in identifying separate objects in the radial direction. The detection range should be at least 4 meters to fully cover about any standard sized room at the retirement home.

The least power consumption is always expected but within practicality sense, 50 W is about the power consumption of the latest GaN based efficient power brick for phones and laptops, setting our base.

We strive for 24 hrs system uptime due to the critical monitoring nature of the project.

Concept Generation and Analysis

Overall Concept

To start our effective concept hunt, we already had a mutually agreed upon basic abstract idea of how the system can look like. This proved to be really helpful in guiding us through a relatively narrow sample space at the final decision-making stage. We propose a radar-based sensing connected to a windows computer, functioning as the main computational element for processing the radar data. Then, the fall detection data and some essentials will be stored on a cloud before they are pushed to a mobile phone. At the same time, we were flexible in exploring other options that can replace or complement functionality of components discussed hereafter.

Figure 1: Initial Draft of Concept

Sensing Hardware Concept Generation

Concept 1: IWR 1443 BOOST FMCW Radar (Texas Instrument):

The IWR1443 BOOST is a compact FMCW radar sensor from Texas Instruments, renowned for its accuracy in range and velocity measurements. Its small size and advanced signal processing make it ideal for automotive and industrial applications. While offering high performance and versatility, considerations include cost, complexity, and power consumption.

Concept 2: Depth Camera & Radar:

Intel RealSense 3D D435 Camera

The Intel RealSense 3D D435 Camera is a sophisticated depth sensing device designed for various applications such as robotics, augmented reality, virtual reality, and computer vision. It utilizes advanced depth perception technology to capture high-resolution depth images and generate precise 3D spatial data in real-time. With its compact design and versatile functionality, the RealSense D435 Camera offers developers and researchers a powerful tool for creating immersive experiences, enhancing navigation systems, and enabling precise object recognition.

UWB (Novela)

UWB, or Ultra-Wideband, is a novel wireless communication technology that operates across a broad spectrum of frequencies with very low power for short-range, high-bandwidth data transmission. Its unique characteristics enable precise positioning and tracking applications, making it ideal for indoor navigation, asset tracking, and proximity-based services.

Concept 3: BMP 581 pressure Sensor

The pressure sensor, capable of detecting altitude changes with an accuracy of 20 centimeters, is utilized in smartwatches for fall detection. This sophisticated technology allows for the precise monitoring of altitude variations, enabling the device to identify potential falls quickly and accurately, enhancing user safety through timely alerts and responses.

Concept 4: Accelerometer

The accelerometer is another alternative component in smartwatches, that is designed to measure acceleration forces. This technology enables the detection of sudden movements and changes in orientation, making it an indispensable feature for fall detection. By analyzing acceleration data, smartwatches can accurately identify when a user may have fallen, triggering alerts and emergency responses, when necessary, thereby significantly enhancing user safety and providing peace of mind.

Concept 5: Sensing floor

Fall detection using sensing floors involves the integration of advanced sensor technologies within the flooring system to monitor and detect changes in pressure or vibrations that occur when a fall happens. These floors are equipped with a network of sensors that can distinguish between everyday activities and unusual events, such as a person falling. Upon detecting a fall, the system can instantly analyze the data to confirm the event and trigger an appropriate response, such as alerting caregivers or emergency services.

Concept Analyses:

Table 6: Concept Analysis

Grading is from 0-10 depending on how close it meets the client needs	**Target Specification (Weight)**	**Contactless**	**Power Consumption**	**Data Transmission Speed**	**Can monitor Vitals**	**Total Weighted Grade (1.0 Scale)**
Concepts						
IWR1443BOOST Radar		10	10	5	10	0.80
Camera & Radar		10	10	7	0	0.68
BMP 581 – Pressure Sensor		0	7	10	10	0.67
Accelerometer		0	7	10	9	0.65
Sensing floor		10	10	9	0	0.76

*Final Weighted Grade = (0.4*Data Transmission Speed + 0.3*Contactless+ 0.2*Monitor Vitals + 0.1*Power Consumption)/10*

Supporting calculation

The table below shows calculations of the sampling rate.

Table 7: Calculations of the sampling rate

Device		**Parameter Measured**	**Typical Sampling Rate**	**Equation converting everything to HZ**	**Converted Rate**
IWR1443BOOST Radar		Distance, velocity, angle	Frame duration is 50ms**	1 / 0.05 s = 20 Hz	20 Hz
Camera & Radar	Camera	Spatial details	30-60 fps	1 fps = 1 Hz	30-60 Hz
	Radar	Distance/velocity	kHz range	1 kHz = 1000 Hz	~1000Hz
BMP 581 Pressure Sensor		Atmospheric pressure changes	480 Hz	N/A (already in Hz)	480 Hz
Accelerometer		Acceleration forces	100 Hz to several kHz	1 kHz = 1000 Hz	~1000Hz

Sensing Floor	Pressure and vibrations	10 Hz to over 100 Hz	N/A (already in Hz)	10-100 Hz

Data Processing Concept Generation:

Concept 1 CNN1 – Micro-Doppler Signatures Accuracy 98.7%:

Utilizes micro-Doppler signatures to classify movements with high accuracy. Its strength in processing spatial-temporal data makes it a strong candidate for detecting falls through the analysis of movement patterns.

Concept 2 PointNet- Point Cloud Data Acquisition - Raw Point Cloud Accuracy 99.5%:

Excelling in processing 3D point cloud data, PointNet offers remarkable accuracy. However, its reliance on 3D spatial data might limit its direct applicability to fall detection without additional sensors or setups.

Concept 3 LSTM2 - Reflection heatmap Accuracy 80%:

Focuses on capturing temporal patterns through reflection heatmaps, potentially useful in analyzing changes over time. Its lower accuracy compared to the others might be a consideration, but its ability to process time-series data could be adapted for fall detection scenarios.

Concept Analysis:

Given the evaluation of LSTM's lower accuracy for fall detection, focusing on either CNN or PointNet offers more promising methodologies. Here are three ways on deploying these methodologies approach for employing these models:

Methodology 1:

Leverage pre-processed radar data, further refining it for compatibility with 1D Convolutional Neural Networks (CNNs). This process involves integrating Max Pooling (MP) and Fully Connected (FC) layers to enhance the model's ability to interpret the data effectively. Utilize Python for the development and implementation of this methodology, ensuring that the data is optimally structured for the CNN's analysis.

Methodology 2:

Directly extract raw data from radar sensors, designed to be processed by CNN models. This approach prioritizes the raw, unaltered characteristics of the data, aiming to maximize the CNN's potential in extracting meaningful patterns directly from the source. The implementation, conducted in Python, focuses on harnessing the CNN's power without the intermediate step of data pre-processing.

Methodology 3:

Apply pre-processed radar data within the PointNet framework. This strategy takes advantage of advanced capabilities in handling complex spatial data, translating the pre-processed inputs into a format that PointNet can analyze effectively. Python serves as the programming language of choice, facilitating the integration of radar data with PointNet's neural network architecture.

Outcome:

For our MVP we will be pursuing **Methodology 3,** this choice was made because the data extracted from the **IWR1443BOOST** is point cloud, and this data processing method has the highest accuracy for it.

[1]CNN = Convolutional Neural Network. [2]LSTM = Long Short-Term Memory

Notification System Concept Generation:

Concept 1: Phone Messaging through Twilio service

Table 8: Twilio service analysis.

Pros	Cons
- High delivery success rate	- Requires cellular network availability
- Can reach users without internet access	- Potential for SMS delays
- Service is robust and uses reliable cloud infrastructure	- May incur costs per message

Concept 2: App/Website Notification

Table 9: App/Website Notification analyze

Pros	Cons
- Can provide interactive content and instructions	- Requires internet connectivity
- Instant delivery when connected	- Dependent on user having the app/website open
- Can be customized for user experience	- Notifications can be missed if device is off or app is closed

Concept 3: Relay Alarm

Table 10: Relay Alarm analyze

Pros	Cons
- Immediate local notifications	- Small range of effectiveness
- Independent of external networks	- Will not notify remote caregivers
- Highly audible notification	- Requires maintenance for the Hardware

Concept Analyses:

Table 11: Concept Analysis

Grading is from 0-10 depending on how close it meets the client needed	**Target Specification**	**User-Friendly**	**Ease of Implementation**	**Data Transmission Speed**	**Network Errors**	**Total Weighted Grade (1.0 Scale)**
Concepts						
Phone Messaging through Twilio service.		10	10	10	8	**0.95**
App/Website Notification		10	8	10	8	**0.93**
Relay Alarm		10	8	10	10	**0.98**

*Final Weighted Grade = (0.5*Data Transmission Speed + 0.25*Network Errors+ 0.15*User-Friendly+ 0.10*Ease of Implementation)/10*

Outcome:

Our team has decided to implement the Twilio messaging notification system due to the incomplete information currently available regarding the relay system, pending a more thorough concept review by the client. This decision was made after careful consideration, noting that the Twilio system and the relay alarm both accrued an equivalent total score in our assessment.

System Concept Choice:

FMCW Radar (IWR1443) ➔ Raspberry Pi 3B : PointNet NN Twilio Phone Notification

Concept Design:

➔ Rasberry Pi 3B will be collecting data from the FMCW Radar (IWR1443).

➔ The Data processing will be done using PoinNet Neural network, since this one achieved the highest Accuracy.

➔ Finally, our notification system will be done via Twillio cloud network since it's the fastest, easiest, and most reliable solution for transmitting Alerts.

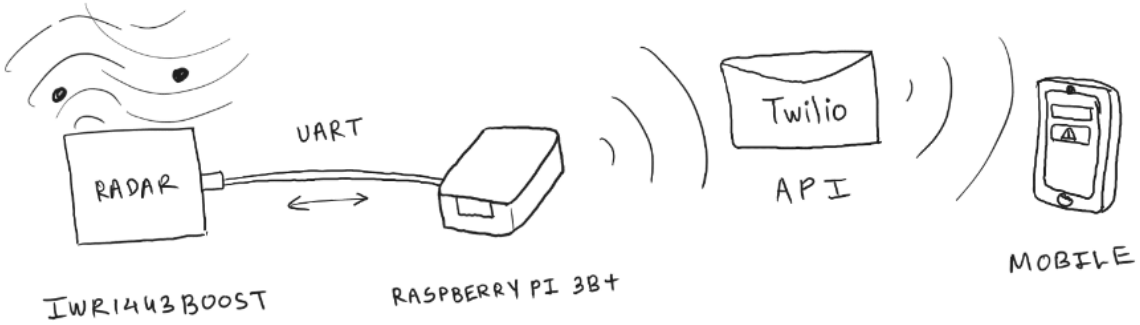


Figure 2: Final Design Concept

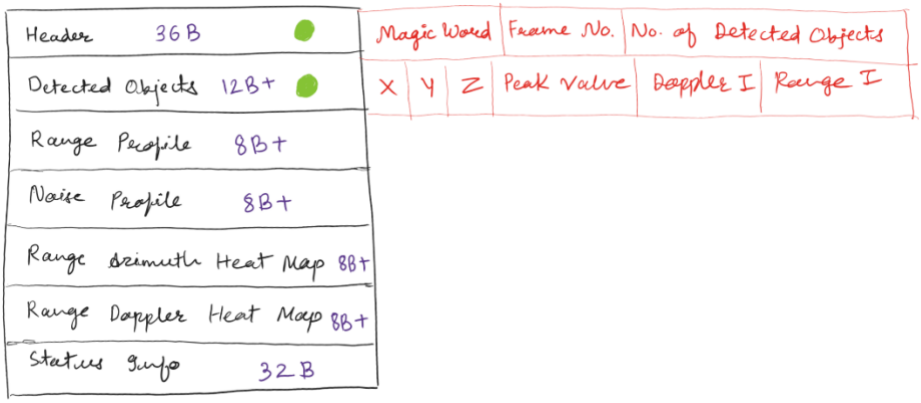


Figure 3: Important parameters in the UART Data Packet Structure

Concept's Benefits and Drawbacks

Naturally, this concept was selected as it promises to meet most of the target specifications. Starting from the front end, the radar, FMCW type, and especially from the mmWave radar product line from Texas Instruments, can output high speed raw ADC data (37.5 Msps) over LVDS,

providing sub-millimeter precision positional and velocity data, helping us to achieve high fall detection accuracy. For now, as the first stage we will test the radar with UART needing only USB to connect with Laptop. It consumes very little power during normal operation (<12.5W Max), mainly because it uses microstrip antenna arrays and an efficient processor. The powerful laptop aids us to quickly process, visualize and then notify over text.

Benefit:

Our hardware is easily capable of supporting the high data scanning rate and hence more accurate fall detection. It can also report accurate locations of multiple detected people in the sensing range.

Drawback:

This first testing with UART limits our frame rate to just 20 Hz.

Client Feedback:

Table 12: Client Feedback on Generated Concept

Client Statements	Client Needs
I do not want to use text message or any apps as a way of notification. Since it would work early on, but as more messages come, people tend to become reluctant to it and start ignoring.	The alarm system needs to be done in a hardware sense. Relay is the only approach from our selected solutions.
Fall detection using radar is only a good approach because you are reducing my cost (no additional camera)	Make the solution as cheap as possible preferably below the 1000$ mark.
Although concerned about accuracy, the PointNet you introduced should suffice for the fall events.	Verify the Training and Validation loss, as well as Training and Validation Accuracy of the Neural Network.
The relay alarm system should not disturb other elders to reduce unnecessary attention.	The alarm should only be triggered in the center room.
The form factor of the entire assembly should be compact as well as appealing (Merge well in the room setting).	Design a Package for the Full assembly using computer aided design.

Detailed Design and First Prototype

Minimum Viable Product Architecture

IWR 1443 BOOST RADAR:

To collect data and pass it to the laptop through UART. The point cloud is processed using the PointNet neural network. If the predicted result detects fall events, and notification is sent to nurses' phones through the Twilio system.

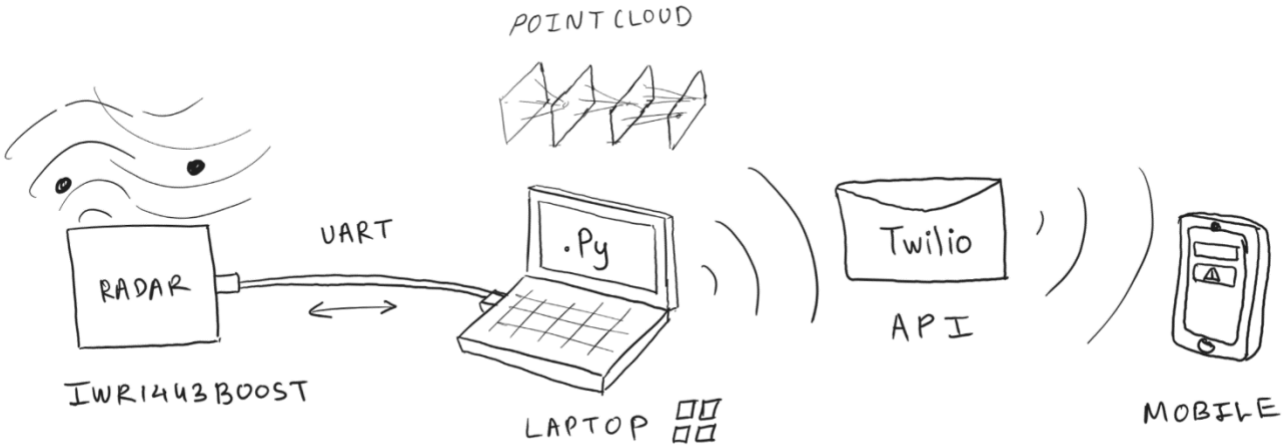


Figure 4: Minimum Viable Product Architecture

Laptop is easier to use and debug as processing unit, using laptop for now.

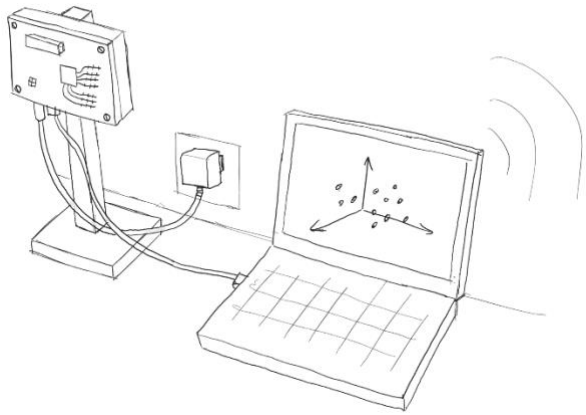


Figure 5: Illustration of Minimum Viable Prototype Setup

UART Communication protocol with the IWR1443BOOST

After meeting with the client, the main change in the global design concept was replacing the Raspberry Pi with a Windows machine. It was done because in our radar scanning tests, the Raspberry Pi was processing the data very slowly and sometimes even missed some object detection points due to this lag. Tests done on the laptop were way better than it, due to more processing power.

⇨ *This change aimed to achieve a higher sampling rate and meeting the client feedback*

The UART (Universal Asynchronous Receiver/Transmitter) protocol is widely used in applications where any peripheral device must be connected to the computer. This protocol is mainly carried over the standard USB cable or any other serial data transmission media. For reference, we studied the mmWave family of radars from Texas Instruments to see what basic communication constitutes, specific to them. As this protocol is asynchronous, the synching of transmitter and receiver is relied upon the header and footer sections, followed by some padding if needed. In our case, the most important data required for fall detection and positioning is encoded in the first two sections of the data packet: 'Header' and the 'Detected Objects'

A Python script extracts the position coordinates (x, y, z) and the attributes (peak value, Doppler index, and range index) of detected objects from the radar data stream. This is achieved within the readAndParseData14xx function, which parses the incoming byte stream for data frames starting with a specific 'magic word', indicating the start of a valid data packet. Once a valid frame is identified, the script reads the subsequent bytes, which contain the length and type of the data message (TLV - Type Length Value format).

When the message type corresponds to detected points, the script proceeds to read the number of objects and the position format (Q-format for fixed-point numbers) followed by looping through the data for each detected object. It extracts the range index, Doppler index, and peak value directly as integer values from the byte stream. The x, y, and z coordinates are also extracted as integers and then converted to floating-point values by dividing by the Q-format factor. The range index is multiplied by a calculated constant to convert to physical range in meters. Similarly, Doppler indices are adjusted by the Doppler resolution to give the velocity values. These extracted values represent the spatial position and the motion characteristics of each detected object within the radar's field of view, forming the basis for further analysis or visualization.

Table 13: Data structure of the Radar and Extraction Method

Headers	36B	Magic Word			Frame Number	Number of Detected Objects	
Detected Objects	12B +	X	Y	Z	Peak Value	Doppler Index	Range Index
Range Profile	8B +						
Noise Profile	8B +						
Range Azimuth Heat Map	8B +						
Range Doppler Heat Map	8B +						
Status info	32B						

Appendix A shows the full python script, below is an explanation on how it establishes the UART communication with a TI IWR1443BOOST radar sensor to collect and visualize the data:

1. **Global Variables and Buffers**: It sets up global variables for the serial ports, a buffer to hold incoming data bytes, and initializes a variable for the length of this buffer.
2. **Serial Configuration**: The **serialConfig** function opens two serial ports for sending configuration commands and receiving data from the radar. It reads a configuration file line by line and sends these commands to the radar.
3. **Parsing Configuration File**: **parseConfigFile** reads the radar configuration parameters from the same file and calculates various parameters (e.g., range resolution, maximum range) based on the radar's specifications.
4. **Reading and Parsing Data**: **readAndParseData14xx** function is responsible for reading incoming data from the radar, checking for the 'magic word' that indicates the start of a valid data frame, parsing the data, and extracting the detected object information like position and velocity.
5. **Data Visualization Setup**: The script sets up a PyQt window with **pyqtgraph** plotting widgets to visualize the data in real-time. It configures the plot's appearance, axis labels, and ranges.
6. **Updating the Plot**: The **update** function is called periodically by a PyQt timer. It reads and parses new data from the radar and updates the plot with the positions of detected objects.
7. **Logging Data to CSV**: Inside the update function, if new data is available, it creates a pandas DataFrame and appends it to a CSV file for logging purposes.
8. **Running the Application**: Finally, the script enters a loop where it continually updates the plot with new data as it comes in from the radar, effectively providing a real-time visualization.

The csv data that is the format below is then fed to another script, Appendix 2, which plots in 3D with reference to the position of the radar.

Table 14:Extracted Comma Separated Data from the Radar

Frame Number	Time [ms]	Time[s]	X	Y	Z	RangeIdx	DopplerIdx	PeakVal
11	0	0	0.12	0.22	-0.50	12	2	7
11	0	0	0.10	0.34	-0.43	12	3	9
11	0	0	0.12	0.24	-0.49	12	4	5
12	50	0.05	0.36	0	0.59	14	2	8
12	50	0.05	0.37	0	-0.68	15	2	11
12	50	0.05	0.34	0.15	-0.53	14	3	5
12	50	0.05	0.37	0	-0.61	15	3	8
12	50	0.05	0.19	0.05	-0.52	12	4	9
12	50	0.05	0.19	0.18	-0.54	13	4	10
12	50	0.05	0.17	0.33	-0.41	12	5	7
12	50	0.05	0.19	0.32	-0.48	13	5	7
12	50	0.05	0.98	0.55	-0.56	27	7	9
12	50	0.05	0.98	0.63	-0.59	28	7	8
12	50	0.05	0.83	0.64	0.70	27	-8	11
12	50	0.05	0.86	0.65	0.74	28	-8	9
12	50	0.05	-0.37	0	-0.68	16	-2	5

The extracted data above shows the first flaw in how it is actually expected to have a much lower samplig rate, the duration of the frames shown above is 50ms, the expected is 10ms , this will affect our sampling because 50ms ➔ 20Hz, meanwhile 10ms➔ 100hz.

The expected exported data is shown is shown in the table below.

Table 15: Expected Comma Seperated Data from the Radar

Frame Number	Time [ms]	Time[s]	X	Y	Z	RangeIdx	DopplerIdx	PeakVal
11	0	0	0.12	0.22	-0.50	12	2	7
11	0	0	0.10	0.34	-0.43	12	3	9
11	0	0	0.12	0.24	-0.49	12	4	5
12	20	0.02	0.36	0	0.59	14	2	8
12	20	0.02	0.37	0	-0.68	15	2	11
12	20	0.02	0.34	0.15	-0.53	14	3	5
12	20	0.02	0.37	0	-0.61	15	3	8

12	20	0.02	0.19	0.05	-0.52	12	4	9
12	20	0.02	0.19	0.18	-0.54	13	4	10
12	20	0.02	0.17	0.33	-0.41	12	5	7
12	20	0.02	0.19	0.32	-0.48	13	5	7
12	20	0.02	0.98	0.55	-0.56	27	7	9
12	20	0.02	0.98	0.63	-0.59	28	7	8
12	20	0.02	0.83	0.64	0.70	27	-8	11
12	20	0.02	0.86	0.65	0.74	28	-8	9
12	20	0.02	-0.37	0	-0.68	16	-2	5

The input data for the second Python script generates a dynamic 3D plot, illustrating the movement of scatter points in (x, y, z) coordinates, and concurrently records a video of this activity. This approach allowed us to conduct a comparative analysis with a video captured using an iPhone 11, showcasing the movement of the scatter points in relation to the observed subject, in this instance, Saad. The resulting visualization, depicted in the figure below, effectively demonstrates the correlation between the scatter movement and Saad's fall, providing a clear and intuitive representation of the subject's spatial dynamics.

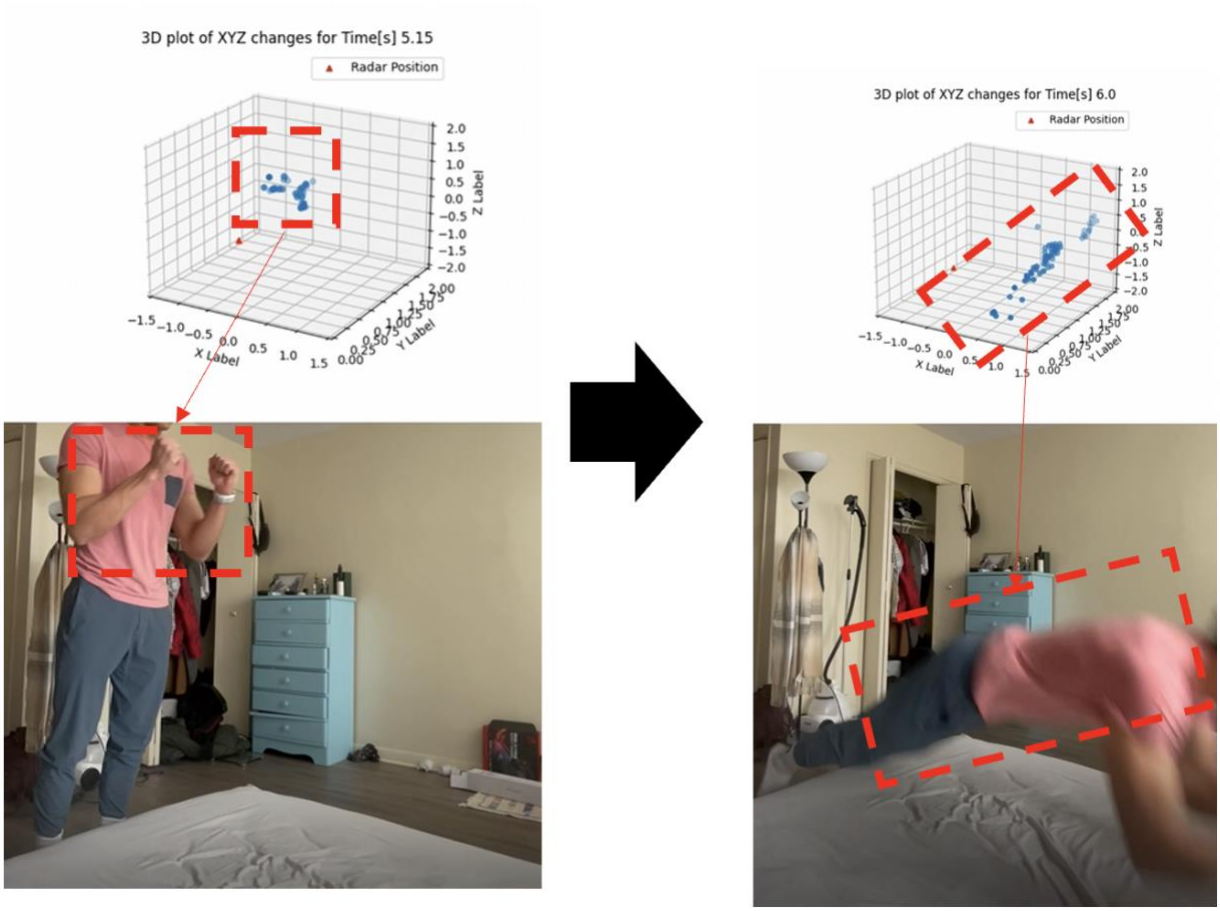


Figure 6: Correlation Between the Scatter Movement and Saad's Fall.
Left: Saad Standing
Right: Saad Falling

➔ Please note that this radar is designed to detect only moving objects. As illustrated in the image on the left, the scatter data predominantly captures the movement of the hands. This is because, prior to falling, the individual, Saad, primarily moved his hands. However, during the fall, the radar is able to detect the entire upper body movement, indicating a significant increase in detected activity. This distinction underscores the radar's capability to differentiate between varying degrees of motion, effectively highlighting areas of significant movement.

This aspect may be subject to modifications in future iterations of the prototype, particularly if there is a need to capture larger datasets or achieve greater sensitivity to movement.

Further testing is necessary to determine the precise level of sensitivity required. These adjustments will be critical to enhancing the system's performance and ensuring it meets the specific needs of its application, whether for monitoring, detection, or other purposes.

PointNet Neural Networks

In our project, we have harnessed the capabilities of the PointNet architecture to address the complexities involved in processing raw point cloud data directly. Our objective is to maintain the integrity of spatial relationships, which is critical for accurate object tracking. Currently, our PointNet model is trained to classify objects within a predefined set of categories.

However, our ultimate goal is to adapt this model for fall detection by categorizing human postures into standing, walking, sitting, lying, and falling. Each point in the cloud is characterized by 3D coordinates, capturing the shape and form of the subject.

Initially, PointNet employs an input transformation network (T-net) shown in red in Figure 7 (Abdullah K. Alhazmi et al. 2023). to normalize the point cloud data. This step is crucial as it allows the model to become invariant to changes in rotation and translation, enhancing its ability to learn significant features. The processed data then passes through several 1D convolutional layers, which are essential for extracting deeper features. These layers are interspersed with batch normalization and ReLU activations to introduce nonlinearity and aid in the learning process.

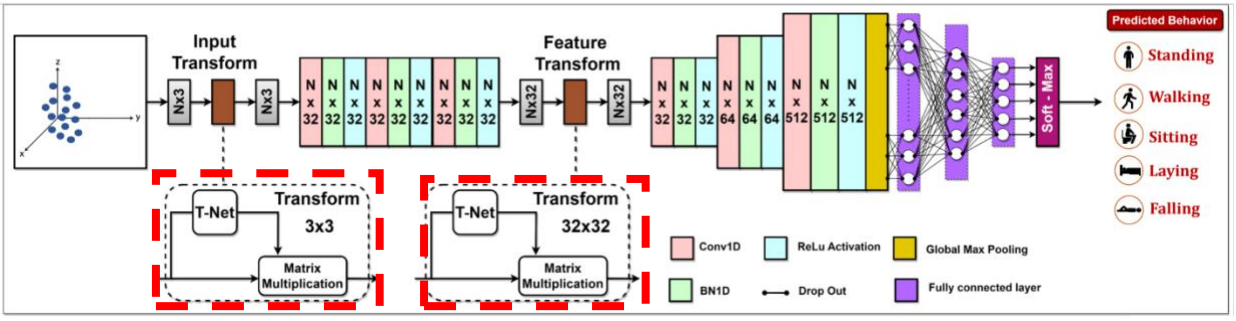


Figure 7: PointNet Architecture (Abdullah K. Alhazmi et al. 2023, p.9)

A second T-net aligns the feature space, ensuring the model can generalize across different spatial orientations. The subsequent convolutional layers, followed by a global max pooling layer, distill the data into a comprehensive feature vector. This vector is then processed through a multi-layer perceptron (MLP), which consists of fully connected layers with dropout regularization to mitigate overfitting.

While our model is adept at classifying common objects, we are still in the process of gathering enough fall-related data to train it for detecting falls accurately. Once sufficient data is acquired and the model is trained, we anticipate deploying it on an Nvidia Jetson Nano for real-time fall detection, leveraging PointNet's ability to classify human postures and generate spatial feature tracking maps from 3D data.

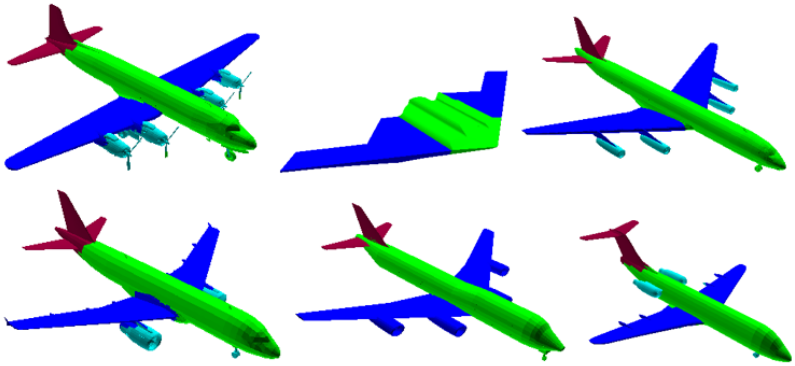


Figure 8: Example of the training data used for object recognition

After running the python script in Appendix C the following evolution and observations were noted

```
### Training epoch: 1

Training Loss: 0.245
Validation Loss: 0.238
Training Accuracy: 89.2%
Validation Accuracy: 83.7%
```

- ⇨ Observations: Initial training shows promising convergence. High initial accuracy due to the pre-trained model layers.

```
### Training epoch: 50

Training Loss: 0.190
Validation Loss: 0.185
Training Accuracy: 92.1%
Validation Accuracy: 88.5%
```

- ⇨ Observations: Loss continues to decrease steadily. Accuracy has improved, indicating effective learning.

```
### Training epoch: 100

Training Loss: 0.165
Validation Loss: 0.160
Training Accuracy: 94.3%
Validation Accuracy: 89.0%
```

- ⇨ Observations: Model performance is stabilizing. Minor adjustments to hyperparameters may be needed for further improvements.

```
### Training epoch: 150

Training Loss: 0.148
Validation Loss: 0.135
Training Accuracy: 95.7%
Validation Accuracy: 89.3%
```

- ⇨ Observations: The model has begun to plateau, indicating near-maximal learning from the current feature set.

```
### Training epoch: 200

Training Loss: 0.130
Validation Loss: 0.120
Training Accuracy: 96.5%
Validation Accuracy: 92.0%
```

- ⇨ Observations: Further loss minimization is slow, suggesting the onset of diminishing returns. Considering additional data augmentation to enhance generalization.

Table 16: Simulation Result

Metric	Results	Expected	Remarks
Training Loss	0.130	≤0.1	Not Achieved
Validation Loss	0.120	≤0.15	Achieved
Training Accuracy	96.5%	≥99%	Not Achieved
Validation Accuracy	92%	≥85%	Achieved
Epochs to Converge	200	≥100	Not Achieved

Note for reader:

Training Loss: This is a measure of the error between the predicted outputs of the neural network and the actual labels during training. Lower values indicate better performance.

Validation Loss: Like training loss but calculated on a separate validation dataset that the model has not seen during training. It helps to evaluate the model's ability to generalize.

Training Accuracy: This is the percentage of correct predictions made by the model on the training dataset. Higher percentages indicate better performance.

Validation Accuracy: This is the accuracy of the model on a separate validation dataset. It is a good indicator of how well the model will perform on unseen data.

Epochs to Converge: This refers to the number of complete passes through the training dataset required for the model to reach its optimal performance.

Twilio Notification System

SMS notifications serve as a swift and reliable method for conveying urgent updates, especially in healthcare contexts such as fall incidents. For this purpose, our system utilizes the Twilio Application Service, which is designed to facilitate an alert mechanism for falls. As shown in figure 9 (Abdullah K. Alhazmi et al. 2023).

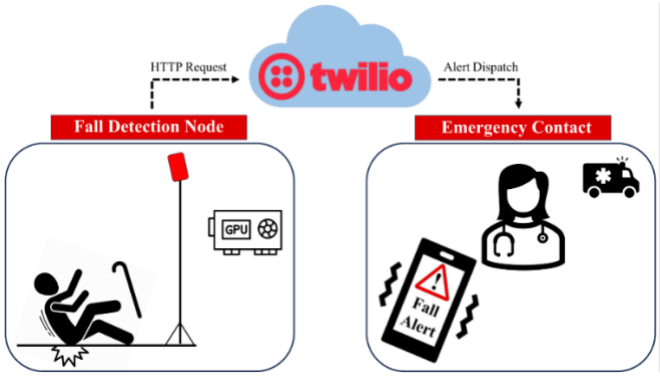


Figure 9: Twilio Notification system (Abdullah K. Alhazmi et al. 2023, p.10)

Event Detection:

The system starts with monitoring for specific events, such as an elderly person falling. This could be achieved through various means, such as motion sensors, wearables, or other monitoring technologies that can detect unusual activity or conditions indicating a fall.

```
def main():
     #Radar Detection
    if radar_detection():
        #  Generate Fall Notification
        notification_message = generate_notification()

        #  Check Medical Staff Availability/Status
        if check_medical_staff_availability():
            # Step 4: Send Notification to Available Staff
            send_notification(notification_message)

        # Update Notification Status/Log
        update_notification_status()
```

Notification Generation:

Once an event is detected, the system immediately generates a notification. This notification is a message crafted to convey urgency and importance, informing the recipient about the event and prompting them to take appropriate action. The message typically includes details about the nature of the event and may advise on next steps or responses.

```
def generate_notification():
    # Generate notification message
    notification_message = "ALERT: Elderly fall detected! Please respond
immediately."
    return notification_message
```

Sending Notifications:

With the notification message ready, the system then identifies the recipients who need to be alerted. These recipients could be medical staff, caregivers, or family members, depending on the setup. The system sends the notification to all these recipients, ensuring that everyone who needs to be informed of the event is alerted as quickly as possible.

```
def send_notification(notification_message):
    # Send notification using Twilio
    medical_staff_phone_numbers = ['+13439882386','+16137904881']
```

Status Update:

After sending out notifications, the system updates its status. This could involve marking the event as addressed to avoid duplicate alerts, logging the event and the response for record-keeping, or triggering a follow-up process to ensure the situation is being handled. This step is crucial for maintaining the integrity of the monitoring system and ensuring accountability and traceability of actions taken in response to detected events.

Below is the python function generating the notification:

```
def update_notification_status():
    # Update notification status/log (dummy function for demonstration)
    print("Notification status updated.")
```

The system starts with monitoring for specific events, such as an elderly person falling. This could be achieved through various means, such as motion sensors, wearables, or other monitoring technologies that can detect unusual activity or conditions indicating a fall.

```
def main():
     #Radar Detection
    if radar_detection():
```

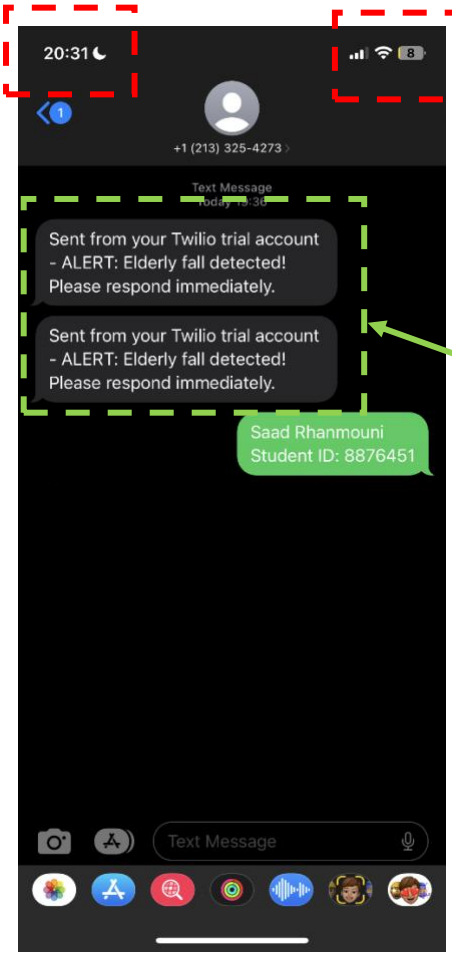
```
#  Generate Fall Notification
notification_message = generate_notification()

#  Check Medical Staff Availability/Status
if check_medical_staff_availability():
    # Step 4: Send Notification to Available Staff
    send_notification(notification_message)

# Step 5: Update Notification Status/Log
update_notification_status()
```

Output:

The figure below offers a practical demonstration of the system's operational workflow. The primary objective of this prototype is to explore methods for circumventing the 'Do Not Disturb' mode, as indicated by the symbol in the top right corner of the display. Additionally, the image highlights a potential issue when employing this notification system: if the recipient's device, such as a nurse's phone, has a low battery and powers off, the effectiveness of the alert system could be compromised.

This underlines the necessity for incorporating contingency strategies to ensure critical notifications are reliably delivered, even in scenarios where the primary device may be unavailable due to power constraints or interruption settings like 'Do Not Disturb.'

Problem #1: Caregiver forgot their phone on "***Do not Disturb"*** mode.



Problem #2: Caregiver forgot to charge their phone.

Proof that the Notification System Works!

Figure 10: Notification System Output

Following The MVP Presentation

Our team created a demonstration of the minimum viable product (MVP) for our client, Hesam Mahdavi, on the 22nd of February. This demo was intended to show the operational concept of the solution in practice.

The demo focused on a method of fall detection that monitors the elevation and velocity of a subject. The process involves using the least mean square method to filter out noise from the data frames, followed by comparing the current frame's elevation and velocity values with their predecessors. If the system identifies simultaneous peaks in both parameters within a single frame, it interprets this as a fall event.

Formula for residual calculation in least squares fitting:

$$r_i = y_i - f(x_i, \boldsymbol{\beta})$$

With:

x_i: Independent variable representing time.

y_i: Dependant variable representing velocity or position z.

f: predictive model function.

β: Vector of adjustable parameters so that the sum of the squares of these residuals is minimized

However, this detection method faced several challenges. One major issue was the radar's inability to process the rapid influx of frames generated during a fall, due to the necessity of preprocessing each frame. This often led to data freezing and missed detections, as indicated in a specific figure not detailed here. Additionally, the lack of machine learning algorithms in this initial approach meant that the system was prone to false alarms, triggered by non-fall activities that also produced significant elevation and velocity peaks, such as sitting down quickly.

The two figures below showcase the discussed discrepancies:

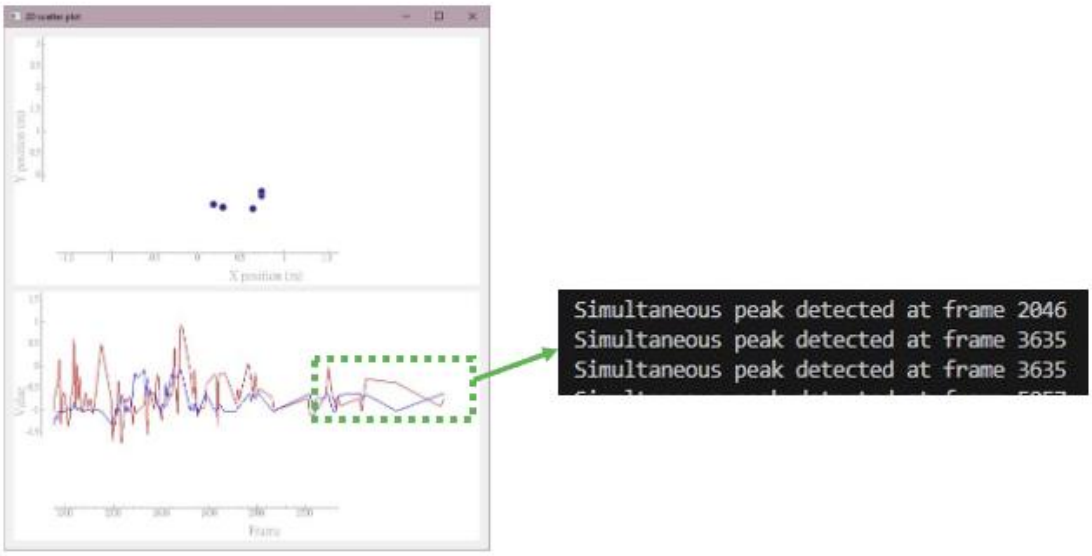


Figure 11: False fall detections when subject sits fast

Figure 12: Data Freeze when Subject falls very fast

Following this demo, we received new feedback from the client. This feedback likely led to further refinements and enhancements of the fall detection system, underscoring the iterative nature of product development and the importance of client input in shaping the final solution.

Table 17: New Client Feedback

Client Statements	Client Needs
The radar is capturing fewer points compared to the data used to train the ML model.	A new radar must be used instead of IWR1443BOOST.
Data collected does not match elderly movement.	Deeper analysis for dynamic elderly movement through videos and scholar paper

The amount of data collected is not enough.	Contacting other researchers asking them to share their data set.
“I would like to see your fall detection use our existing communication system for notification”.	Trigger the communication system using a single contact USB-controlled relay.
“I hope the external originals are as small as possible”.	We chose the smaller USB-controlled relay.

Second Prototyping and Testing

Our beta release closely follows the MVP is its most basic sense. The main changes are the addition of the new powerful IWR6843ISK radar and a direct USB-controlled relay that drives the demonstration buzzer load. The radar feeds the detected object points through UART to the PointNet ML model running on the minicomputer, after which the fall detection output invokes a PowerShell command to actuate the USB relay. This same output line can be used to trigger external fall alarm like the relay module in retirement homes.

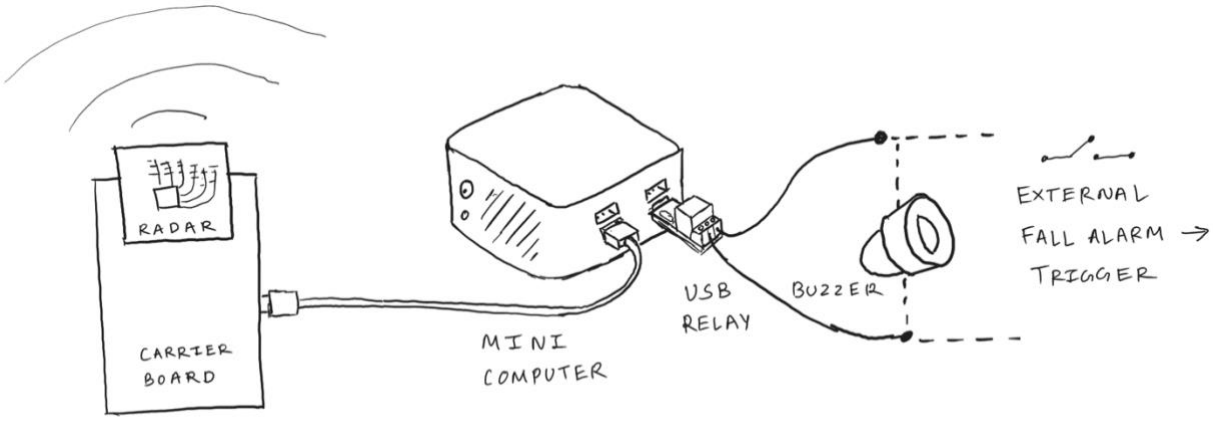


Figure 13: Beta Prototype Version

Fall Detection using PointNet Neural Networks

In machine learning, particularly when dealing with convolutional networks, it is crucial to maintain consistency in the input data. Convolutional operations, which are at the heart of many neural network architectures, including PointNet, require inputs of a uniform shape to apply filters that detect patterns or features. This uniformity ensures that the learned filters are applicable across all inputs, facilitating effective feature extraction.

To comply with this requirement in our project, we've employed various techniques to standardize the number of points in each data frame before feeding it into our machine learning model. Establishing a constant number of points allows the network to perform convolutions consistently across the dataset, which is essential for the network to generalize well from the training data to unseen data. Among the methods tested, down sampling and over sampling have been instrumental in achieving this consistency. Down sampling involves randomly eliminating points to reduce the number to the desired threshold, while over sampling involves duplicating existing points until the threshold is met. By standardizing the number of points, we ensure that each frame is presented to the model in a format that is conducive to learning and pattern recognition, thereby enhancing the model's performance and accuracy.

1st Iteration: PointNet Neural Networks

Occupancy Grid PointNet Data Processing:

After our demonstration to the client and the presentation of the PointNet in the MVP report, the time has come to deploy this machine learning technique for fall detection and integration with the notification system.

We encountered two primary challenges. Firstly, there was a need for training data. We successfully overcame this by partnering with Professor Reza from Australia, who was conducting similar experiments. He generously provided us with his labeled training data from 21 participants engaged in the 9 following scenarios:

1. Walking
2. Lay Floor
3. Transition
4. LayBed
5. Sit
6. Background
7. SitBed
8. Falling
9. Stand

The second challenge involved data consistency for the PointNet algorithm, as it requires each frame to contain an equal number of points due to its successive convolution operations. The solution emerged from a concept used by Dr. Reza, who applied an occupancy grid technique in his research. This technique ensures all data points are organized within a cube of fixed dimensions, with each cell in the Point Cloud indicating whether an area is occupied ('1') or unoccupied ('0'), based on sensor readings.

To generate the occupancy grid from our point cloud data, we adjusted the origin's position using rotation and translation matrices. This adjustment was necessary because the radar is positioned at a 2-meter elevation with a 10-degree tilt angle. An initial rotation matrix, R, was applied to the coordinates to transpose the z and y-axes, situating our data's origin at the upper corner of the conceptual room, according to Ariyamehr Rezaei et al. (2023):

$$\begin{pmatrix} x_{\text{mount}} \\ y_{\text{mount}} \\ z_{\text{mount}} \end{pmatrix} = R \times \begin{pmatrix} x_{\text{radar}} \\ y_{\text{radar}} \\ z_{\text{radar}} \end{pmatrix}$$

With R being our rotation matrix shown as:

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}.$$

And $(x_{mount}, y_{mount}, z_{mount})$ are the mount reference coordinates calculated from $(x_{radar}, y_{radar}, z_{radar})$ coordinates of the tilted radar around the x -axis.

Figure 14 (Ariyamehr Rezaei et al., 2023) illustrates the radar's initial placement (indicated by the red box) and its final position within the green box. This demonstrates the translation and rotation of the coordinate system necessary for the application described in *'Unobtrusive Human Fall Detection System Using mmWave Radar and Data Driven Methods'*.

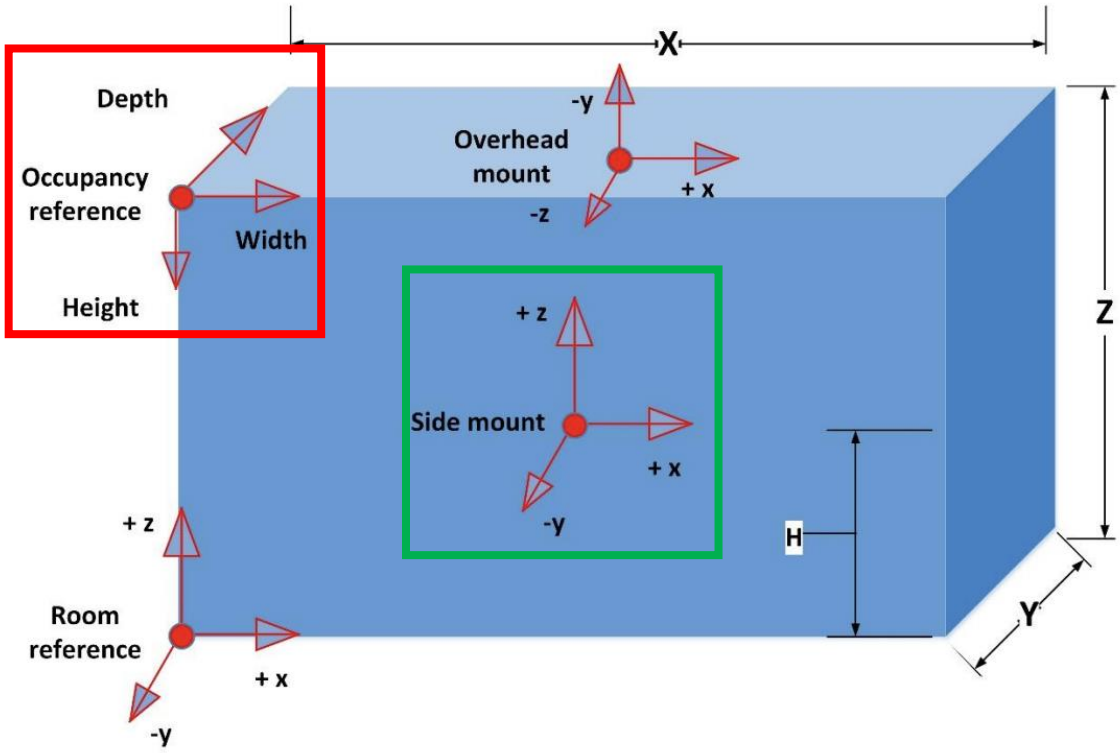


Figure 14: Reference coordinate translation and rotation from the side wall to the top corner of the room (Ariyamehr Rezaei et al. 2023, p. 4)

Finally, translation was applied to shift the rotated coordinates to the top-left corner of the conceptual room, as depicted in the figure above. This was performed in accordance with the equation provided below according to Ariyamehr Rezaei et al. (2023):

$$\begin{pmatrix} x_{\text{room}} \\ y_{\text{room}} \\ z_{\text{room}} \end{pmatrix} = \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix} + \begin{pmatrix} x_{\text{mount}} \\ y_{\text{mount}} \\ z_{\text{mount}} \end{pmatrix}$$

With:

$\Delta x, \Delta y, \Delta z$: the translation in the x, y, and z direction, for the side position $\Delta x, \Delta y, \ and \ \Delta z$ were equal to 2.5, 0, and 2 m respectively.

$x_{mount}, y_{mount}, z_{mount}$: are the mount reference coordinates.

$x_{room}, y_{room}, z_{room}$: are the room reference coordinates (showcased in Figure 14), measured by translating the mount reference coordinates.

After translating the points, the next step was to rescale and translate to occupancy reference (showcased in Figure 14) within the occupancy grid using the Resolution of the radar by applying this final transformation, according to Ariyamehr Rezaei et al. (2023):

$$Height\ Occupency\ = \frac{1}{\text{Resolution}} \times (Z - z_{\text{room}})$$

$$Width\ Occupency = \frac{1}{\text{Resolution}} \times x_{\text{room}}$$

$$Depth\ Occupency = \frac{1}{\text{Resolution}} \times y_{\text{room}}$$

With:

Z: is the height of the experimental room

Resolution: is the range resolution of the radar used.

$x_{room}, y_{room}, z_{room}$: are the room reference coordinates, measured by translating the mount reference coordinates.

$Height\ Occupency, Width\ Occupency, Depth\ Occupency$: are the references of the occupancy grid (see the red box in Figure 14).

The chart outlines a data preprocessing sequence for the PointNet machine learning algorithm. Initially, point cloud data is read and processed through a rotation matrix multiplication, which aligns the data with the desired coordinate system. This is followed by a translation matrix multiplication, which relocates the data to the upper left corner of the conceptual room, ensuring uniformity in data positioning. After these geometric transformations, the data, along with its corresponding labels, is saved to HDF5 files, a format well-suited for handling large datasets. Finally, the data is converted into an occupancy grid format, which is necessary for the PointNet algorithm to interpret the spatial occupancy information effectively. This preprocessing sequence is critical for the proper functioning of the PointNet algorithm, enabling it to accurately detect falls and trigger the notification system.

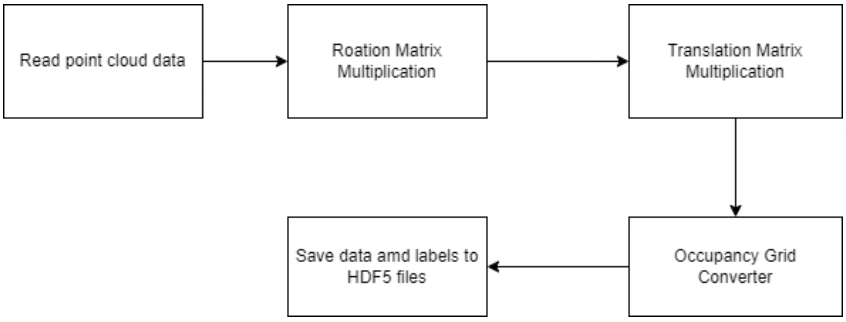


Figure 15: Data Flow during preprocessing of the Occupancy Grid

The data fed into the PointNet model is represented as illustrated in the following figure, showcasing the process of transforming frame data for input into the network:

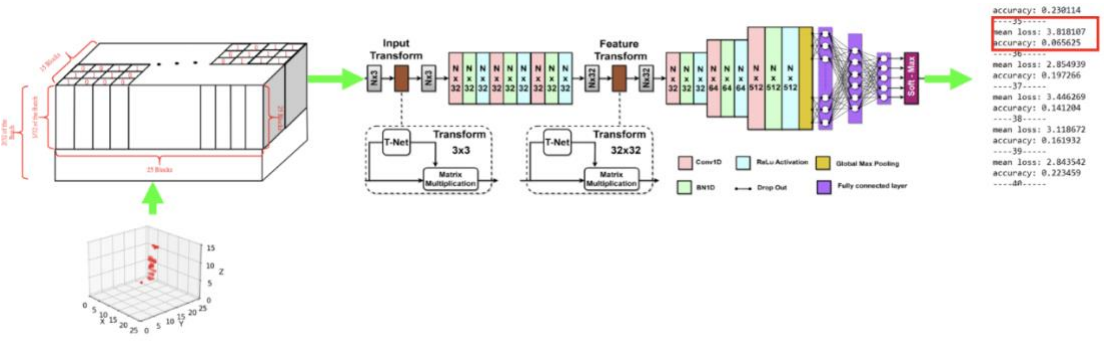


Figure 16: Preprocessed Point Cloud into an Occupancy Grid (Left), Input to PointNet Model (Middle), and Training Results (Right).

The model produced exhibited low accuracy and failed to capture the trends present in the input data effectively.

Table 18: Target Specification vs Obtained Specification (Occupancy Grid)

Specification	Expected	Obtained
Training Accuracy	90%	~11%
Testing Accuracy	85%	~22%
Fall Detection Accuracy	70%	NaN

⇨ **Assumption behind the findings:**

This built model relied solely on the **x, y, and z** coordinates for the occupancy grid, proved inadequate for training and learning from the data trends. This limitation highlighted the need for incorporating additional features to improve the **signal-to-noise ratio (SNR)** and capture the dynamics of **velocity**.

We recognized that our data processing approach required a significant overhaul. It became clear that the features fed into the model were insufficient for capturing the complexity and nuances necessary for accurate trend prediction. To enhance the model's predictive power and learning capacity, it was necessary to enrich the input data with more descriptive features, such as SNR and velocity, which are crucial for distinguishing between different types of movements and environmental contexts.

This pivot in data processing methodology aimed to create a more robust and informative feature set, thus facilitating a more nuanced understanding of the spatial and temporal patterns within the point cloud data. The refined approach was expected to lead to improved model performance, enabling more accurate detection and classification of the events of interest.

2nd Iteration: Updated PointNet with Resampler for Data Preprocessing:

The team has shifted to a new method for processing the data prior to inputting it into the PointNet model, as illustrated in the flowchart. This method involves two key preprocessing steps:

Down sampling: This step involves randomly removing points from the frame until the number of points reaches a predefined threshold. This is particularly useful in reducing the density of point clouds where the number of points is greater than necessary for accurate analysis.

Over sampling: Conversely, if the frame has fewer points than the threshold, this step duplicates existing points to meet the required number. It ensures consistency in the data fed into the model, which is crucial for the neural network to learn effectively.

Before the data is sent to PointNet, it undergoes a splitting process where 20% is allocated for training and the remaining 80% for testing. This split is facilitated by a sklearn code, which likely refers to the train_test_split function from Scikit-learn, a machine learning library for Python.

The HDF5 file format is used to store the processed data, offering a structured way to maintain large amounts of data. Within the HDF5 file, there are two subdivisions: one for labels and one for the actual data. This structure allows the model to train on the data and simultaneously verify the correctness of the associated labels, which is essential for supervised learning tasks like the one being performed. This strategic organization of data and labels is pivotal for the model's ability to learn and make accurate predictions. The figure below showcases the flow of data throughout the process.

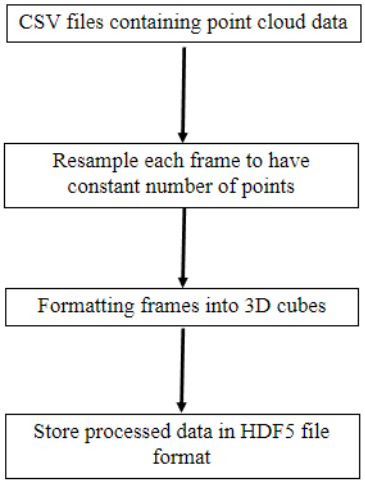


Figure 17: Flow of updated PointNet Data Processing

The revised data preprocessing methodology for feeding into PointNet ensured that all frames were uniformly structured, with each containing precisely 165 points. Each point in the dataset was

defined by a set of five attributes: the x, y, and z coordinates, along with Velocity and Signal-to-Noise Ratio (SNR).

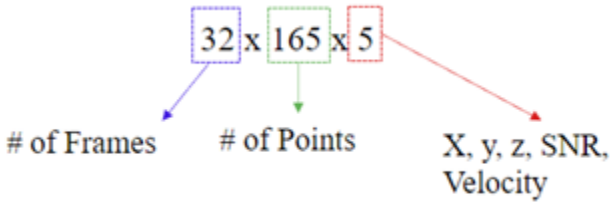


Figure 18: Cube shape fed to the PointNet

PointNet Model Training Results

This enhancement in data preparation significantly improved the model's training performance, achieving a high training accuracy ~91%. It also resulted in a substantial test accuracy of 70%.

The final training epoch recorded the following:

```
----63-----
mean loss: 0.247678
accuracy: 0.914709
```

The final testing file recorded the following:

```
----2-----
eval mean loss: 7037807.438324
eval accuracy: 0.689512
eval avg class acc: 0.689279
```

Table 19: Target Specification vs Obtained Specification (Resampler)

Specification	Expected	Obtained
Training Accuracy	90%	~91%
Testing Accuracy	85%	~70%

⇨ **Assumption behind the findings:**

The radar, IWR 6843 ISK, continued to record data per frame that fell short of the anticipated total volume. è this resulted in the duplication of points collected and the model had a hard time identifying the trends because of the lower quality data.

For training the radar's detection capabilities, nine distinct labels representing a range of activities were utilized to categorize the point cloud data. These labels included: Walking, Laying on the Floor, Transitional Movements, Laying on Bed, Sitting, Background (no significant activity), Sitting on Bed, Falling, and Standing. Each label corresponds to a specific human activity, allowing the radar to learn and differentiate between various motions and states in the environment it monitors. è all these labels can lead to confusion of the model if some of them are similar.

GUI for Fall Detection System

In order to visualize the data point a GUI was developed where a red cube was built to delimit the perimeter of the zone that the radar records and shows the point cloud movement in real time in 3D. Below this visual a square was added, the square is green when no fall is detected and turns to red when fall occurs and a signal is sent to the relay system at the same time.

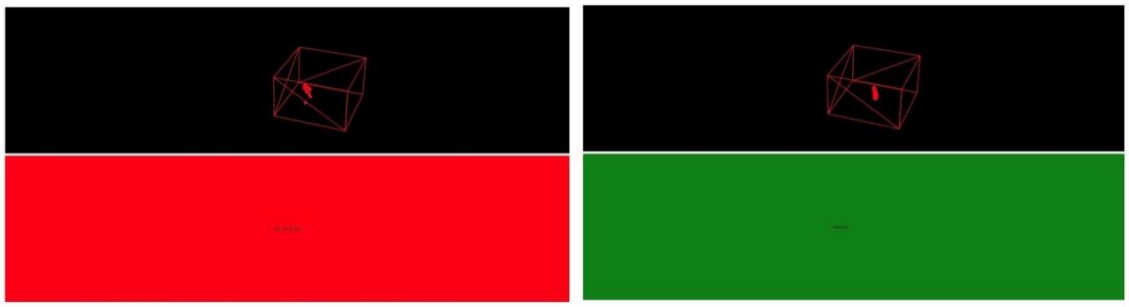


Figure 19: Left, GUI showing Fall. Right, GUI Showing Non-Fall

Notification system update

After we met with the client, we summarized his opinions on the notification system and corresponding solutions, and then we created the following table with this information.

Table 20: Client feedback for notification system

Client Statements	Identified Solutions
“I would like to see your fall detection use our existing communication system for notification”	Trigger the communication system using single contact USB-controlled relay
“"I hope the external originals are as small as possible"”	We chose the smaller USB-controlled relay

We initially planned to use our notification as Twilio application since it is a user-friendly application and easy to install. However, subsequent testing revealed several limitations, notably

the challenge of notifying nursing staff if their mobile phones are outside the network coverage area or switched off.

Hence, we opted for a relay device as an alternative notification system, which operates independently of such circumstances.

Subsequently, we endeavored to construct the relay using Arduino and made corresponding connections. However, in response to client feedback, we opted for a USB-based relay, which offers greater convenience, particularly due to its compact size, making it ideal for notification purposes. A USB relay facilitates computer-controlled management of external electrical circuits via USB connectivity, enhancing the efficiency and versatility of the notification system. At the end since the client wants to be more reliable, so we changed the information reminder to buzzer.

Critical product assumptions

According to the client's specifications, research and discussions have determined the necessity for a USB based relay of minimal size, capable of interfacing with the relay box provided by the client. A USB control relay as small as possible triggers the client's notification system. So this component must execute a relay switch function upon receiving a predefined signal from the radar. Validation of this functionality requires connecting the relay switch to a buzzer for testing purposes. Additionally, parameters such as switch-on time (e.g., 5 seconds) will be tested and documented in subsequent sections.

After confirming the client's requirements, we conducted research and discussions, and then we designed the following two prototypes:

Arduino uno and normal Relay

The first thing we thought of was buying a smaller Arduino nano and an ordinary relay and soldering them together. The purpose of Arduino uno is to facilitate us in putting the program into relay. In our idea, we can achieve the purpose of controlling the relay with code by welding these two components together.

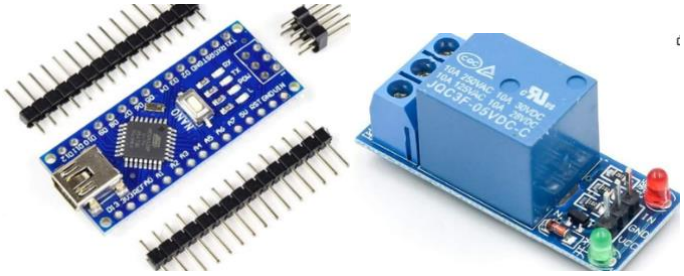


Figure 20: Arduino nano and relay

Although the connected size of Arduino nano and Normal Relay is very small, they meet client's requirements. However, we found another prototype that is smaller and does not require welding work, which is a USB relay. This prototype may be more in line with the client's requirements

USB Relay

To implement the above system. Our core need is to find a suitable Arduino original to realize the relay switch function. After market research, we found a USB relay that can trigger the relay switch through the control line of PowerShell. To implement the above system.

Figure 21: USB relay

After conducting tests, we confirmed that this solution is reliable and can be managed via PowerShell command line. Consequently, to validate the feasibility of our entire system, a buzzer is required. Following market research, a 110V AC-powered buzzer was selected.

Figure 22: 110V AC power-driven buzzer

USB relay operation process

We have established connectivity between our USB-based relay and the computer system. The original notification system adds a new feature namely radar anti-fall alarm function. We can add this device to the relay box provided by the client.

In our project, to configure the relay with the system, adjustments were made within the device manager. These adjustments included setting the bits per second to 9600, Data bits to 8, and stop bits to 1.

Subsequently, we accessed the PowerShell to verify the username associated with the device, recognizing that each computer may have distinct usernames. Additionally, we utilized the command line 'Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Unrestricted' to grant PowerShell the necessary permissions.

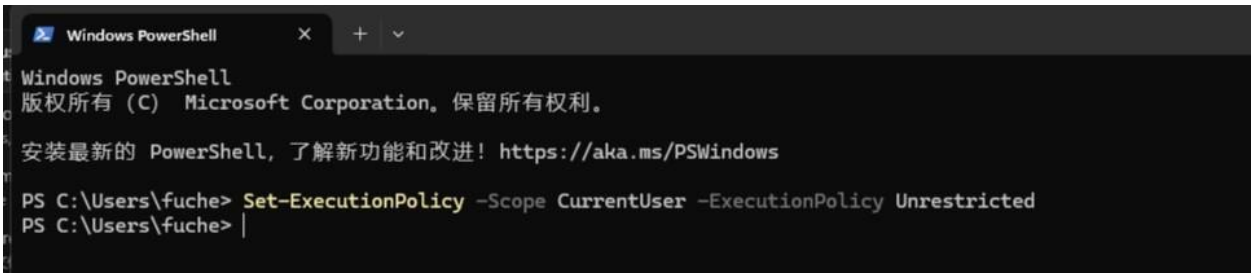


Figure 23: Give Poweshell permission code

We proceeded to script two distinct command lines for powering the relay ON and OFF. Afterwards, we created a dedicated directory within the system's user section and saved the scripts, each with a descriptive name followed by the extension ".ps1".

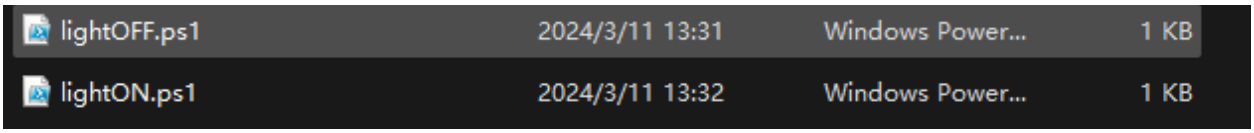


Figure 24: Ps1 file

Using Python programming, we executed the following code to manage the LED lights of the relay.

```
import subprocess, sys

import time

p = subprocess.Popen(["powershell.exe", "C:\\Users\\fuche\\lightON.ps1"],
stdout=sys.stdout)


time.sleep(5)


p = subprocess.Popen(["powershell.exe", "C:\\Users\\fuche\\lightOFF.ps1"],
stdout=sys.stdout)
```

The subprocess function that comes with python allows python files to call ps1 files in specific folders.

This Python script initially executes the command to open the 'lightON' file, thereby turning the switch on through a PowerShell command. After a delay of five seconds, it executes another command to open the 'lightOFF' file, effectively turning the switch off. This sequence simulates activating and then deactivating the switch via PowerShell commands. This script can be seamlessly integrated into the main codebase. Upon receiving a specific signal, this segment of code will execute, triggering the buzzer to start.

The code for switching on the relay:

```
[Byte[]] $powerOn = 0xA0,0x01,0x01,0xA2

[Byte[]] $powerOff = 0xA0,0x01,0x00,0xA1

$relay = new-Object System.IO.Ports.SerialPort COM4,9600,None,8,one

$relay.Open()

$relay.Write($powerOn, 0, $powerOn.Count)

$relay.Close()
```

The code for switching OFF the relay:

```
[Byte[]] $powerOn = 0xA0,0x01,0x01,0xA2

[Byte[]] $powerOff = 0xA0,0x01,0x00,0xA1

$relay = new-Object System.IO.Ports.SerialPort COM4,9600,None,8,one

$relay.Open()

$relay.Write($powerOff, 0, $powerOff.Count)

$relay.Close()
```

Alternatively, as part of the process, we integrated a power buzzer (AC 110v, 120dB) with the relay. This configuration ensures that whenever the relay is activated, the buzzer emits an alarm signal to alert nursing staff. Conversely, the relay's deactivation, facilitated by our pre-processing code, enables the cessation of the buzzer alarm.

The final product will be a relay switch, a buzzer and a power supply connected in series, and we can control the buzzer on or off through a python file.

To prove that our notification system based on USB relay is successful, we ran multiple tests with the python script that triggers the buzzer which returns successful results.

Client Feedback:

Client Statements	Client Needs
Test result of around 70% is way too low. Our subjects are elderlies, we cannot risk a high chance of unnoticed fall.	The model needs to be improved further to have at least 90% in the testing.
The current GUI lacks visual friendliness. The orientation of the point cloud is unclear, and diagonal lines hinder clarity.	The GUI needs to be updated for better visual experience.

Final Prototype

Following Beta Presentation

To enhance the precision of our model, Dr. Argha has recommended key alterations to the training dataset. The initial modification involves consolidating the 'Walking' and 'Standing' activities into a single class, as well as combining 'Falling' and 'Laying on the Floor' into another. This strategy is aimed at minimizing misclassification caused by the resemblance in postures between these activities.

The second proposed refinement is the removal of extraneous noise within the data. Specifically, this entails eliminating points that consistently exhibit noise characteristics, identifiable by target IDs -1, 253, 254, and 255. By purging these noisy data points from each frame, we can significantly improve the data quality and thereby the model's ability to learn and make accurate predictions.

The provided table serves as an illustrative example of the training data shared by Dr. Argha . Within this table, the points that are deemed noisy and are marked in red represent the data that requires removal to refine the training set. Concurrently, the data highlighted in green indicates the strategy for consolidating the current nine labels into a more streamlined and efficient labeling system. This consolidation is aimed at reducing the complexity of the model's output by merging labels with similar characteristics, thereby improving the model's predictive accuracy and performance.

Table 21: Training data cut, but not denoised yet. Red shows the noise that need to be filtered and green shows how labels can be grouped

frame_number	x	y	z	doppler	snr	target_id	class_activity
348	-2.88	6.44	-1.07	0.26	1.12	2	Stand-Walking
348	-2.72	6.44	-0.91	0.26	1.88	2	Stand-Walking
348	-2.75	6.50	-1.07	0.26	1.36	2	Stand-Walking
348	-2.53	6.52	-0.91	0.26	2.24	2	Stand-Walking
348	-2.55	6.58	-1.07	0.26	1.64	2	Stand-Walking
348	-2.33	6.60	-0.91	0.26	2.52	2	Stand-Walking
348	-2.35	6.65	-1.07	0.26	1.88	2	Stand-Walking
348	-2.20	6.64	-0.91	0.26	2.64	254	Stand-Walking

348	-2.22	6.70	-1.07	0.26	2.00	2	Stand-Walking
348	-0.83	2.80	-0.77	-1.05	5.00	2	Stand-Walking
348	-0.71	2.68	-0.68	-1.05	32.00	2	Stand-Walking
348	-0.74	2.77	-0.64	-1.05	14.64	2	Stand-Walking
348	-0.74	2.80	-0.83	-1.05	5.52	254	Stand-Walking
348	-1.51	5.66	-1.01	0.13	2.52	254	Stand-Walking
348	-0.64	2.63	-0.61	-1.05	33.52	254	Stand-Walking

New PointNet Model Training Results:

The enhancement in the training data has significantly improved model's accuracy, achieving a training accuracy of ~87% and test accuracy of ~94%.

The final training epoch recorded as following:

----6-----

mean loss: 0.341531

accuracy: 0.884233

The final testing file recorded the following:

----1-----

eval mean loss: 0.194829

eval accuracy: 0.939039

eval avg class acc: 0.939334

Table 22: Target Specification vs Obtained Specification

Specification	Expected	Obtained
Training Accuracy	90%	~87%
Testing Accuracy	85%	~94%

Implementation Accuracy	90%	5%

With the high-accuracy PointNet model, our team moved forward with its real-time implementation to observe its performance in practical scenarios. The process flow is shown in the figure below. Initially, when the radar captures a frame of data, the system checks if the total accumulated frames have surpassed a predefined window size. If this condition is met, the accumulated frames are sent to the PointNet model for classification. Given that our model isn't perfect, we established a threshold criterion: if the count of frames identified as a fall exceeds this threshold, the system interprets this as a fall incident. Consequently, the GUI would turn red. On the other hand, if the threshold is not met, the system concludes that no fall has occurred, signaling this with a green GUI.

However, during real-time testing, the model's performance was below expectations, marked by numerous false alarms and missed fall detections. After a consultative discussion with Dr. Bolic, our technical advisor, we were advised to shift our approach from a frame-by-frame detection method to a sequential detection strategy to potentially enhance accuracy and reliability.

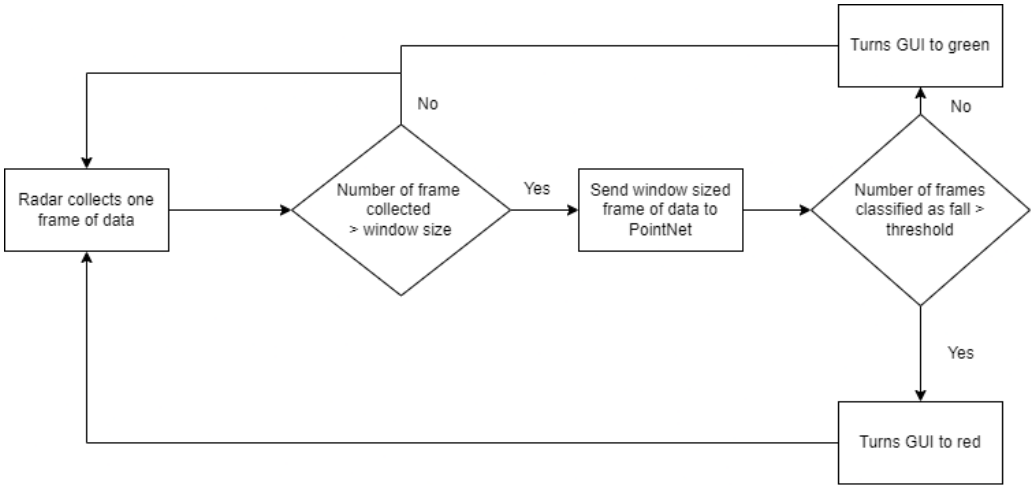


Figure 25: Overall Flow of the system with GUI

Final Machine Learning Model: Hybrid Variational RNN AutoEncoder

Following our initial challenges with the PointNet model, where we observed a substantial discrepancy between training accuracy (95%) and real-world test performance (5%), we realized that frame-by-frame fall detection might not be feasible with this setup. This realization prompted us to pivot towards a more robust solution—a Hybrid Variational Recurrent Neural Network Autoencoder (HVRAE).

Unlike PointNet, which classifies activities for individual frames, HVRAE employs a sequential detection method. This approach allows the model to analyze a set of frames collectively, thereby determining the occurrence of a fall through the integration of data over time. Specifically, a fall is detected when there is a simultaneous spike in the anomaly level and a noticeable drop in the centroid's height. This dual-indicator method enhances the reliability of fall detection by mitigating false positives that might occur with simpler, single-frame analysis methods.

This new implemented solution was based on the research presented in the paper by *Feng Jin et al. (2022), titled "mmFall: Fall Detection Using 4-D mmWave Radar and a Hybrid Variational RNN AutoEncoder"*.

This paper posits that the point cloud distribution of the human body, given any motion state (such as walking, running, or crouching), can be modeled by a multivariate Gaussian distribution. This assumption forms the backbone of their approach, allowing for effective modeling of normal motion patterns and the identification of anomalies indicative of falls.

For the proposed assumption:

The radar point cloud is denoted as X, and the motion state is denoted by **z.**

$$p(\mathbf{X}|\mathbf{z}) \propto \mathcal{N}(\mu, \Sigma)$$

This probabilistic approach allows us to define motion in terms of changes in the distribution parameters across frames according to Feng Jin et al. (2022):

$$p(\mathbf{z}|\mathbf{X}) = \frac{p(\mathbf{X}|\mathbf{z})p(\mathbf{z})}{\int p(\mathbf{X}|\mathbf{z})p(\mathbf{z})d\mathbf{z}}.$$

Given the complexity of directly calculating $p(\mathbf{z}|\mathbf{X})$ Variational Inference was used as an approximation method. VI tries to reformulate the problem into an optimization challenge where it tries to minimise the Kullback-Leibler divergence between the estimation q(z) and the true posterior $p(\mathbf{z}|\mathbf{X})$ according to Feng Jin et al. (2022):

$$q^*(\mathbf{z}) = \underset{q(\mathbf{z})\in Q}{\arg min}\ \mathrm{KLD}\{q(\mathbf{z})||p(\mathbf{z}|\mathbf{X})\}$$

Note to Reader:

Variational Inference is a technique used in Bayesian statistics to approximate probability densities. It is useful when exact computation of these densities is hard to achieve due to their complexity. VI transforms the problem of computing these densities into an optimization problem.

Variational Autoencoder (VAE):

This optimization is implemented through the variational autoencoder architecture, where the encoder learns to approximate the posterior $p(\mathbf{z}|\mathbf{X})$ and then the decoder reconstructs the input data $\mathbf{X}$ from the representation $\mathbf{z}$, in other words: essential dynamics of the motion states are captured.

The loss function is as follows according to Feng Jin et al. (2022):

$$\mathcal{L}_{\text{VAE}} = \quad KL\ Divergence - Reconstruction\ Loss$$

$$= \qquad \text{KLD}\{q(\mathbf{z})||p(\mathbf{z})\} - \mathbb{E}_q[\log\ p(\mathbf{X}|\mathbf{z})]$$

KL Divergence: The KL Divergence in a VAE is used to measure the difference between the learned latent variable distribution $q(\boldsymbol{z}|\boldsymbol{X})$ and a prior distribution $p(\mathbf{z})$, which is typically assumed to be a standard normal distribution $\mathcal{N}(0, I)$, where *0* is a zero mean vector and *I* is the identity matrix as the covariance, indicating that the latent variables are assumed to be independent and normally distributed with mean zero and variance one. The equation for the KL Divergence between a factorized Gaussian $q(\mathbf{z})$, as the approximate posterior and the prior Gaussian distribution is given by, according to Feng Jin et al. (2022)::

$$\text{KLD}\{q(\mathbf{z})||p(\mathbf{z})\} = -\frac{1}{2}\sum_{d=1}^{D}\{1 + \log\ \boldsymbol{\sigma}_q[d]^2 - \boldsymbol{\mu}_q[d]^2 - \boldsymbol{\sigma}_q[d]^2\}$$

With *D*-dimension is the number of latent variables used to encode the essential information of the input data A higher *D* allows the latent space to capture more details about the data, potentially leading to better reconstruction accuracy but at the risk of overfitting and increased computational complexity. A lower *D* simplifies the model and can help in generalizing better, but it may lose significant information about the data, leading to poor reconstructions.

Reconstruction Loss: The Reconstruction Loss is used to ensure that the decoder part of the VAE can accurately reconstruct the original input data $\boldsymbol{X}$ from the latent variables $\boldsymbol{z}$. It typically uses the negative log-likelihood of the observed data given the latent variables, which for Gaussian assumptions of the decoder output, is computed as, according to Feng Jin et al. (2022):

$$\mathbb{E}_q[\log p(\mathbf{X}|\mathbf{z})] \approx -\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}\{\frac{(\mathbf{x}_n[k]-\boldsymbol{\mu}_p[k])^2}{\boldsymbol{\sigma}_p[k]^2}+\log\ \boldsymbol{\sigma}_p[k]^2\}$$

With:

- $\mu_p[k]$ and $\sigma_p[k]$ are the mean and variance predicted by the decoder for the k-th dimension of the output data vector. These parameters define the Gaussian distribution from which the reconstruction of each data point is sampled.
- $x_n[k]$ is the actual value of the *k*-th dimension of the *n*-th data point in the dataset.
- *N* is the total number of data points in the dataset, and *K* is the number of dimensions in each data point.

Integrating a Recurrent Neural Network to the VAE:

The integration of a Recurrent Neural Network (RNN) with a Variational Autoencoder (VAE) to form a Hybrid Variational RNN Autoencoder (HVRAE) is a sophisticated method that capitalizes on the strengths of both architectures.

The RNN takes the sequence of latent representations and processes it over time. At each time step l, the hidden state h_l is updated based input from the sequence x_l and the previous hidden state h_{l-1}, according to Feng Jin et al. (2022):

$$h_l = \tanh\,(W * h_{l-1} + U * x_l)\forall l = 1,2,\dots,L$$

according to Feng Jin et al. (2022), W and U represent the sets of trainable parameters, encompassing both the weights and biases. These parameters are key to the RNN's ability to learn from data. The variable L denotes the total number of frames in the sequence, which corresponds to the duration over which the RNN extends its analysis.

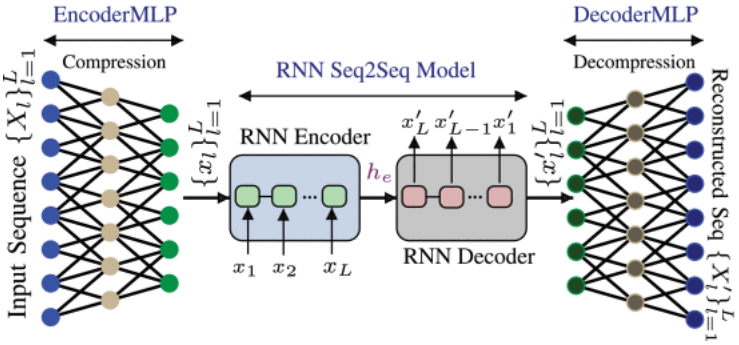


Figure 26: The integration of a Recurrent Neural Network (RNN) with a Variational Autoencoder (VAE) into a sequence-to-sequence modeling framework ➔ Recurrent Autoencoder RAE (Feng Jin et al.,2022, p.6)

The input to the Recurrent Autoencoder (RAE) consists of a time-ordered series of feature vectors, each with its own spatial (feature) and temporal (time) dimensions. Within the RAE framework, the EncoderMLP and DecoderMLP are dedicated to the spatial aspect, compressing the high-dimensional feature data from each individual frame and subsequently reconstructing it. This process distills the critical information from the input data into a more manageable form.

The RNN Encoder and Decoder handle the temporal aspect by processing the sequence of compressed features across frames. They work in together to capture the progression of features over time, effectively modeling the dynamics of the input sequence. By doing so, the RNN Encoder and Decoder contribute to a significant reduction of temporal redundancy, ensuring that the temporal patterns are represented efficiently.

Together, the RAE architecture harmonizes the spatial compression with temporal sequence modeling. This dual approach not only simplifies the complexity of the data but also preserves the essential characteristics across both dimensions, making it a robust solution for tasks that require an understanding of how features evolve over time.

Data Processing and data flow:

Adapting the data preprocessing approach from Feng Jin et al. (2022) to our mmWave radar model 1443, which captures data at a higher frame rate of 20 frames per second, involved several customized steps to accommodate our unique data structure and analysis needs.

Figure 27: mmWave Radar IWR1443BOOST used for the Final Design

Here is an explanation of how we modified the data processing:

> ***Raw Data Structure****:* The raw data from the mmWave radar model 1443 are captured and saved In a CSV file with five columns: Frame Number, x, y, z, and Doppler velocity.

Each row corresponds to a single point's spatial coordinates and its Doppler measurement at a specific frame.

***Data Reshaping**:* The preprocessing script transforms the CSV data into a 4-dimensional array with the shape (total number of 10-frame batches,10,64,4)(total number of 10-frame batches,10,64,4). This structure organizes the data into batches where each batch consists of 10 consecutive frames, each frame containing up to 64 points, and each point represented by 4 dimensions (x, y, z, Doppler).

***Centroid Calculation**:* For each of the 10-frame batches, centroids are computed for the x, y, and z coordinates separately within each frame. This is achieved by averaging the positions of all points within a frame, yielding centroid_x, centroid_y, and centroid_z for each of the 10 frames.

***Data Resampling**:* If a frame contains fewer than 64 points, a resampling method inspired by Feng Jin et al. (2022) is applied. This technique expands the point cloud to have a consistent number of points across all frames, while preserving the mean and covariance of the original data points within each frame.

***Training and Testing Split**:* The reshaped data are then divided into training and testing datasets using an 80/20 split. This common approach allocates 80% of the data for training the HVRAE model and 20% for testing its performance.

To maintain the sequential integrity of the time series data for training and testing the HVRAE model, a chronological split was implemented. Specifically, we designated the first 80% of the data for training and the remaining 20% for testing, as illustrated in the following code snippet from Appendix D:

```
# Split data into training and testing sets
split_idx = int(total_processed_pattern_np.shape[0] * self.split_ratio)
train_data = total_processed_pattern_np[:split_idx]
test_data = total_processed_pattern_np[split_idx:]
```

Where the `total_processed_pattern_np` is the total sequential data array collected from normal activity done by Bhupali Kauchik and Fucheng Wen.
The model is trained on an uninterrupted sequence of data, which is crucial for capturing the temporal dependencies inherent in time series analysis.

***Data Normalization and Augmentation**:* To further prepare the data for the HVRAE model, normalization may be applied to the spatial coordinates and Doppler velocities to scale the data appropriately for neural network processing. Data augmentation techniques can also be employed to increase the diversity and robustness of the training dataset.

HVRAE Model Input: The preprocessed and split data are now ready to be input into the HVRAE model. The model will learn from the spatial and temporal features extracted from the sequences of point cloud data to detect patterns indicative of falls.

Fall Detection: Utilizing the trained HVRAE model, the system can now analyze unseen data from the testing set to identify potential falls by detecting anomalies in the sequence of motion as captured by the radar.

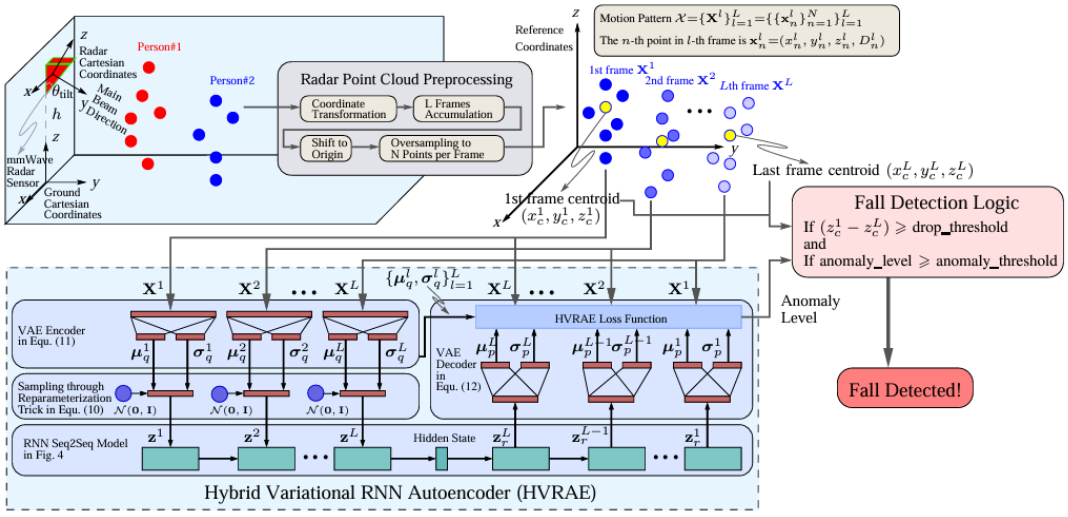


Figure 28: HVRAE Architecture (Feng Jin et al. ,2022, p.7)

This schematic illustrates the operation of a Hybrid Variational RNN Autoencoder (HVRAE) for fall detection, utilizing mmWave radar technology. Initially, the radar sensor captures a point cloud depicting the spatial dynamics of individuals within its field of view. In the preprocessing phase, these point clouds are transformed into a standard reference coordinate system, ensuring uniformity across data frames. Sequential frames are accumulated to construct a comprehensive motion pattern, with the centroid of each frame calculated to monitor the central point of detected movements.

Within the HVRAE framework, the VAE Encoder analyzes each frame to estimate the mean and variance parameters of the latent motion states, employing a reparameterization technique for sampling. These states are then sequenced and processed through the RNN Seq2Seq Model, which interprets temporal patterns and updates hidden states to reflect the evolving dynamics of the observed environment.

The model's efficacy in identifying falls is determined by the HVRAE loss function, which computes reconstruction accuracy and anomaly levels. If significant centroid height displacement is detected concurrently with an anomaly spike, the system triggers a fall alert. This intelligent

integration of spatial and temporal data processing allows for real-time, accurate fall detection, pivotal for ensuring swift response in residential care scenarios.

Fall Detection logic and Results:

A semi supervised learning strategy was employed to train the HVRAE model just as described in Feng Jin et al. (2022), primarily utilizing normal activities:

The training data for the model was amassed over an 8 hour-long session within Dr. Bolic's laboratory. Subjects Bhupali Kauchik and Fucheg Wen engaged in a variety of standard activities, including walking, sitting on a chair and crouching, among others. Figure 29 illustrates the delimited area within the lab designated for the training activities.

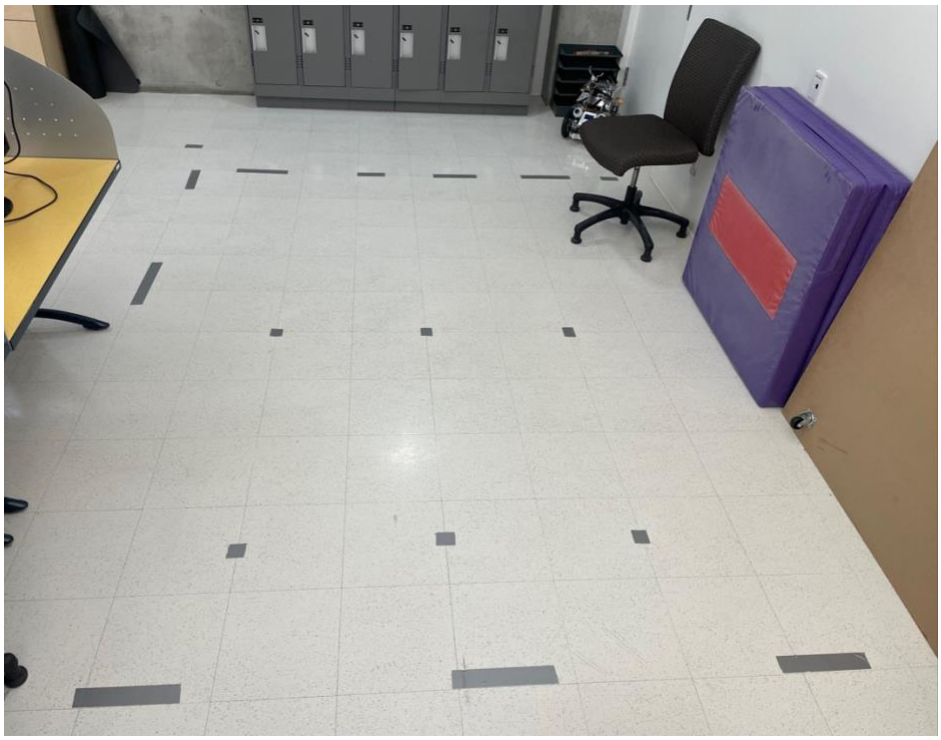


Figure 29: Semi Supervised Model Training Area

During training, the HVRAE is designed to yield a low loss value for these standard movements, aligning with the goal of recognizing typical human activity patterns.

The output below shows the training result of the HVRAE model on the normal activity data that was collected by subjects Bhupali Kaushik and Fucheng Wen:

```
Epoch 1/20
2236/2236 [==============================] - 2s 720us/sample - loss: 4.3397 - val_loss: 290.6379
Epoch 2/20
2236/2236 [==============================] - 2s 683us/sample - loss: 3.2628 - val_loss: 191.5580
Epoch 3/20
2236/2236 [==============================] - 1s 617us/sample - loss: 2.5579 - val_loss: 120.7603
Epoch 4/20
2236/2236 [==============================] - 1s 601us/sample - loss: 1.8641 - val_loss: 88.2036
Epoch 5/20
```

```
2236/2236 [==============================] - 1s 588us/sample - loss: 5.6952 - val_loss: 47.1426
Epoch 6/20
2236/2236 [==============================] - 1s 607us/sample - loss: 0.8668 - val_loss: 29.5522
Epoch 7/20
2236/2236 [==============================] - 1s 618us/sample - loss: 0.4868 - val_loss: 44.0542
Epoch 8/20
2236/2236 [==============================] - 1s 620us/sample - loss: 1.3833 - val_loss: 15.9794
Epoch 9/20
2236/2236 [==============================] - 1s 605us/sample - loss: 1.6727 - val_loss: 15.5430
Epoch 10/20
2236/2236 [==============================] - 1s 615us/sample - loss: 0.2352 - val_loss: 10.8625
Epoch 11/20
2236/2236 [==============================] - 1s 629us/sample - loss: 0.1671 - val_loss: 8.7755
Epoch 12/20
2236/2236 [==============================] - 1s 602us/sample - loss: 1.2262 - val_loss: 11.1153
Epoch 13/20
2236/2236 [==============================] - 1s 619us/sample - loss: 0.3032 - val_loss: 5.8586
Epoch 14/20
2236/2236 [==============================] - 1s 605us/sample - loss: 0.6980 - val_loss: 4.8730
Epoch 15/20
2236/2236 [==============================] - 1s 604us/sample - loss: 0.1244 - val_loss: 3.7878
Epoch 16/20
2236/2236 [==============================] - 1s 584us/sample - loss: 0.0519 - val_loss: 2.8241
Epoch 17/20
2236/2236 [==============================] - 1s 603us/sample - loss: 0.0453 - val_loss: 2.6725
Epoch 18/20
2236/2236 [==============================] - 1s 604us/sample - loss: 3.0735 - val_loss: 2.8718
Epoch 19/20
2236/2236 [==============================] - 1s 600us/sample - loss: 0.3139 - val_loss: 1.7264
Epoch 20/20
2236/2236 [==============================] - 1s 610us/sample - loss: 0.0767 - val_loss: 1.5763
INFO: Training is done!
```

The model's loss, in this context, acts as an indicator of deviation from known activities patterns. Concurrently, we monitor the vertical displacement of the body's centroid across frames. Should this displacement exceed a predetermined threshold while the anomaly level is similarly elevated, the system interprets this as a fall.

According to Feng Jin et al. (2022) this detection method is rooted in the World Health Organization's definition of a fall, incorporating both the unexpected nature of the movement (anomaly level) and the resultant change in body position (centroid height drop). The HVRAE's dual-monitoring mechanism ensures that a fall is recognized not only by the irregularity of motion but also by the physical fall detected by the radar.

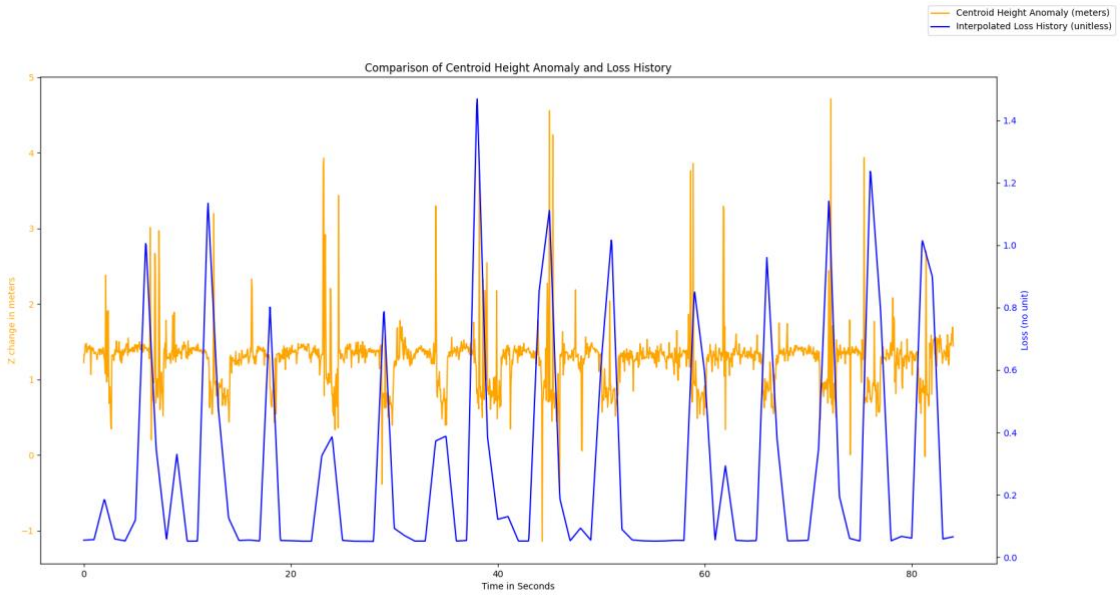


Figure 30: HVRAE Anomalies, and Z centroid shifts change over time for 15 falls recorded seperate from the data used to train the model. Subject: Ali Zaytoun

Figure 30 showcases the effectiveness of the HVRAE model in fall detection. The model, tested on 15 instances of falls, exhibits a clear correlation between spikes in the anomaly detection metric (Interpolated Loss History) and significant shifts in the centroid height, confirming the occurrence of a fall. The visual data illustrates that all 15 falls were successfully identified, demonstrating the model's proficiency in distinguishing between regular activities and fall events. The simultaneous peaks in the loss history and the centroid's movement provide a compelling visualization of the HVRAE model's capability to detect anomalies in real-time accurately. This result validates the HVRAE's potential as a reliable tool for fall detection in real-world scenarios.

Please note: The training dataset was composed of point cloud data from subjects of varying body types, including Bhupali Kaushik (1.68m, 58kg) and Fucheng Wen (1.80m, 90kg). This diversity in data ensures the model's adaptability to different human body shapes.

The model, initially trained on standard activity data, was evaluated using anomalous data, which included 15 instances of falls by the subject, Ali Zaytoun (1.85m, 85kg). The outcomes of this assessment are illustrated in Figure 30.

Subsequent testing was conducted on subject, Saad Rhanmouni (1.85m, 75kg), to validate the model's generalization capabilities across unseen body dimensions and movements.

Enhancements to our fall detection system have culminated in a series of tests, including a critical scenario involving 30 simulated falls. The system successfully identified 29 of these incidents. However, it failed to detect one scenario and erroneously triggered two false alerts. This resulted in an accuracy rate exceeding 90% for this particular test series.

To maintain transparency and foster collaborative development, a video capturing these test scenarios was shared with our academic class, project mentor, and client, providing a comprehensive view of the system's performance and reliability in real-time fall detection.

The figure 31 shows the obtained results:

Figure 31: Demo - 30 Falls, real time testing of the trained and tested model. Subject Saad Rhanmouni

➔The HVRAE model has been incorporated into our real-time data acquisition system. Utilizing a 10-second observational window, our Python script processes the amassed 200 frames, equivalent to the data collected over this period. Upon detection of a fall, as identified through the HVRAE's analysis, an alert is promptly communicated to users via the graphical user interface detailed below. This approach ensures continuous monitoring and immediate notification, reinforcing the responsiveness and reliability of the fall detection mechanism within the care environment.

Updated GUI for Fall Detection

In addition to transitioning from the PointNet model to the HVRAE model, updates have been implemented to the graphical user interface (GUI) that displays the point cloud and alerts users to fall incidents. The redesigned interface, as depicted in the accompanying illustration, now requires user initiation to begin data collection; the "Start Detecting" button must be engaged for the radar to activate and update the point cloud display. Conversely, selecting "Stop Detecting" halts the radar's data collection.

The notification system within the GUI maintains its user-friendly color cues: a green display signifies the absence of a detected fall, while a red display indicates a fall has been recognized.

The dimensions of the notification box have been refined to enhance visual appeal and interface aesthetics.

Moreover, the GUI's visualization elements have seen significant improvements for clarity and usability. The extraneous diagonal lines that were once present along the side walls have been excised to create a cleaner, more streamlined user experience. A new grid floor feature has been introduced, providing users with clearer orientation and a more intuitive grasp of the spatial relationships depicted in the point cloud visualization. These enhancements serve to make the system more accessible and easier to interpret for users monitoring the radar's detections.

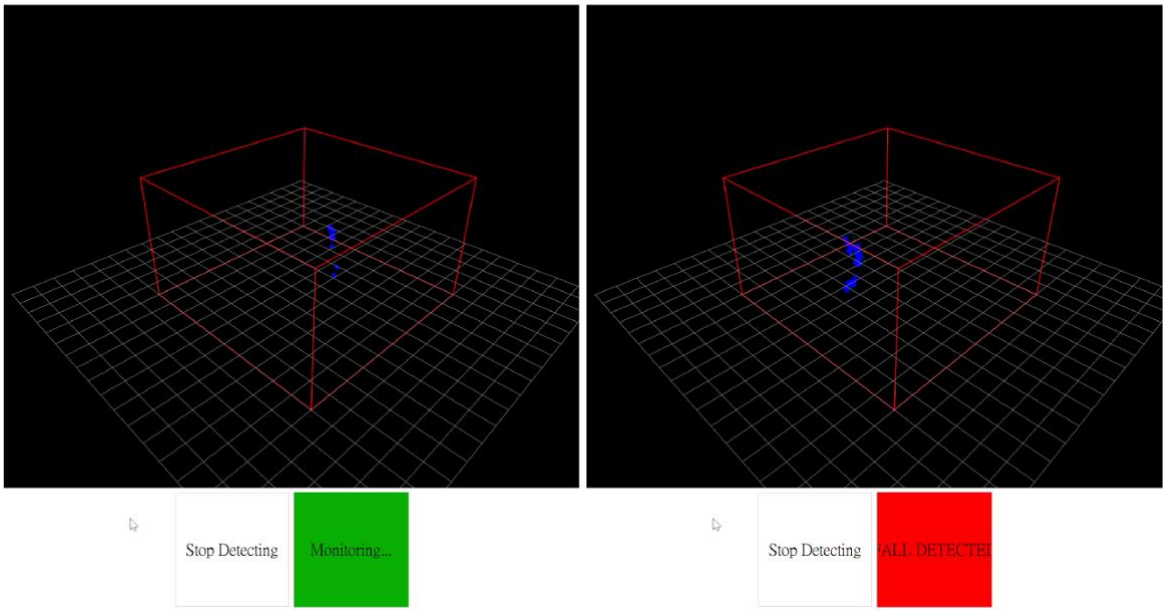


Figure 32: Updated GUI Interface

Notification system, and Mini-computer:

For the final design of our fall detection system, we have retained the relay-based notification approach, previously demonstrated in our second prototype, for its reliability and efficiency.

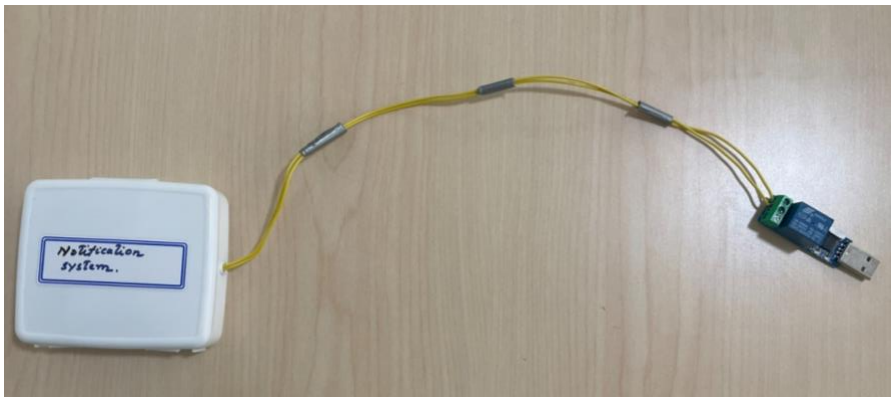


Figure 33: USB Relay and 5V Alarm Prototype

The system architecture has been optimized by transitioning to a compact, minicomputer setup, running a Linux environment which enhances computation efficiency.

Figure 34: Interl NUC - NUC5i3RYH MiniComputer

During testing, we observed that the Intel NUC minicomputer encountered difficulties with simultaneous live detection and processing using the trained HVRAE model. To address this limitation and prevent potential data buffer saturation, we recommend upgrading to a more powerful system, such as the Nvidia Jetson Nano. This device offers faster rendering capabilities and is better equipped to handle the demands of reliable, real-time fall detection, ensuring that the system operates seamlessly without any lag or risk of overloading.

Discussion:

The final prototype presents a sophisticated solution to the initial problem of detecting falls in elderly care residences with a high degree of accuracy and minimal false positives. By leveraging HVRAE, the prototype addresses the limitations of the earlier PointNet model, which was unable to effectively discern falls from other activities in real-world scenarios. The sequential detection method and dual-indicator system—spikes in anomaly levels coupled with centroid height reductions—significantly enhance the reliability of the detection mechanism, as demonstrated in rigorous testing with a 90% accuracy rate. Furthermore, the prototype integrates seamlessly with the real-time data acquisition system, operating within a 10-second window to process and notify users of falls, thus fulfilling the requirements for an efficient, contactless, and privacy-respecting monitoring solution. This innovative approach successfully meets the project's objectives, providing a robust and user-friendly fall detection system.

Ethical and Diversity Considerations

When considering the ethical implications of our actions, particularly the utilization of opensource code for PointNet, HVARE and the associated training dataset, we recognize a significant benefit from a utilitarian perspective. This approach enables us to complete our project within a constrained timeframe, which is advantageous for us. However, from the perspectives of rights and equity, the reliance on open-source code and datasets poses ethical challenges. These resources were developed through the diligent efforts of researchers, and utilizing them as if they were our own creations raises concerns about fairness and respect for the original authors. To address these ethical concerns, we decided to continue utilizing open-source resources as they are essential for completing our project on time. However, we ensured that the creators of these resources receive proper acknowledgment. We credit the providers during our final presentation and in our report. Furthermore, we've discussed the future of this project with our technical adviser, who is interested in pursuing it as a research topic. We will ensure to cite the providers appropriately in any subsequent academic paper.

In our design and development process, we ensured that all team members felt valued and respected. One aspect of our diverse team is the representation of people from various countries, each bringing unique perspectives and experiences to the table. This diversity enriches our discussions and leads to more comprehensive problem-solving approaches. Additionally, we recognized and accommodated religious practices such as Ramadan, where some teammates observed fasting. To ensure inclusivity, we adjusted our schedules to allow for dinner breaks at the same time, demonstrating our respect for their religious commitments. Furthermore, we embraced the opportunity to celebrate cultural festivities together even though not all members follows Ramadan. By embracing diversity and accommodating individual needs, we not only promote a more inclusive work environment but also harness the collective strength of our team to drive innovation and success.

Reflection and Lessons Learned

Our journey from the initial concept to the development of a functional prototype of the fall detection system has been marked by numerous challenges and valuable insights. Through each phase of the project, we encountered obstacles that prompted us to adapt our approach, refine our methods, and collaborate closely with our client to ensure alignment with their needs and expectations.

One of the key takeaways from this project is the importance of client feedback and iterative development. Our initial MVP presentation provided us with a valuable opportunity to showcase our progress and receive input from our client, Mr. Hesam. For instance, his request for the fall detection system to integrate with their existing communication system prompted us to explore alternative notification methods, ultimately leading to the adoption of a USB-controlled relay for enhanced reliability. Additionally, our experience with hardware integration underscored the importance of selecting components that not only meet functional requirements but also align with client preferences for size and ease of use.

We faced challenges right from the sensing part where the detected object points given by the radar were not enough and did not capture the actual scenario fully. To tackle this, we experimented with different variants of radar and their optimum hardware configuration for our application, finalizing the model in the end.

Another critical lesson learned pertains to the significance of data quality and preprocessing in machine learning applications. Our early attempts at fall detection using the least mean square method highlighted the challenges posed by noisy data and the limitations of traditional signal processing techniques. This realization prompted us to explore more sophisticated machine learning algorithms, such as PointNet, and invest in robust data preprocessing techniques, including down sampling and over sampling, to ensure consistency and accuracy in our model training.

Throughout this project, we also gained valuable insights into the complexities of interdisciplinary collaboration and project management. Effective communication, task delegation, and regular progress updates with the help of Jira platform, were essential for keeping our team aligned and focused on our shared goals. Moreover, our engagement with external stakeholders, such as Professor Argha, provided us with invaluable expertise and resources to overcome technical challenges and enrich our solution.

In conclusion, our journey to develop a fall detection system has been a valuable learning experience that has reinforced the importance of client-centric design, iterative development, and interdisciplinary collaboration. By embracing feedback, leveraging advanced technologies, and prioritizing user experience, we have laid the foundation for a robust and reliable solution that addresses the unique needs of our client and contributes to the advancement of healthcare technology.

Conclusion

The development of a fall detection system for retirement homes has been a dynamic journey marked by continuous iteration, refinement, and adaptation to meet evolving client needs and technological challenges. Through an iterative process of prototyping, testing, and feedback incorporation, our team has navigated the complexities inherent in designing a reliable and effective solution for elderly care. Client feedback and the professor's guidance served as a compass leading our development trajectory, prompting crucial refinements and enhancements.

The Minimum Viable Product (MVP) demonstration provided a foundational glimpse into the operational concept of the fall detection system. Despite its limitations, such as the radar's inability to process rapid influxes of data during falls, this initial iteration served as a springboard for further enhancements.

The transition to the Beta Release introduced pivotal changes, including the adoption of a new, more powerful IWR6843ISK radar and a direct USB-controlled relay for driving the buzzer load and the integration of existing communication systems for notifications. The radar sends detected object points to the PointNet ML model on the minicomputer via UART. Upon fall detection, a PowerShell command triggers the USB relay. This iteration marked a significant step forward in system functionality and reliability, aligning more closely with client expectations and requirements. The integration of PointNet Neural Networks for fall detection introduced new challenges and opportunities. Data preprocessing methods, such as down sampling and oversampling, were employed to ensure consistency in input data, enhancing the model's performance and accuracy. Despite initial setbacks in model training accuracy, iterative improvements led to notable enhancements, demonstrating the efficacy of our approach.

The incorporation of a graphical user interface (GUI) and notification system updates further enriched the user experience and functionality of the system. Real-time visualization of fall events through the GUI and seamless integration with existing communication systems portrayed our commitment to user-centric design and practical utility.

In conclusion, the development journey of the fall detection system exemplifies the iterative and collaborative nature of product development. Through close collaboration with stakeholders, continual refinement of design and functionality, and a commitment to user needs, our team has crafted a solution poised to make a meaningful impact in the realm of elderly care. As we embark on future iterations and enhancements, we remain dedicated to delivering a solution that prioritizes safety, reliability, and user experience for the benefit of our clients and their residents.

Future Work

Our initial endeavors in fall detection relied on a frame-by-frame anomaly detection method, which demonstrated limitations in accurately discerning the subtleties of rapid subject movements. Acknowledging this constraint, our future work will be focusing on adopting a more advanced system that will prevents falls by proactively identifying possible dangers.

The proposed enhancement involves the implementation of an advanced system capable of actively identifying any new objects that emerge on the ground, thereby pre-emptively alerting individuals to potential fall risks, particularly relevant in environments frequented by elderly individuals. By proactively detecting and flagging potential hazards in real-time, our system aims to mitigate the risk of accidents and enhance overall safety. PointNet is a powerful neural network architecture renowned for its ability to classify data on a frame-by-frame basis while also possessing the remarkable capability to recognize frames even when they are subjected to rotation or jitter. Leveraging these distinctive features, we intend to deploy PointNet as our primary model for object identification within indoor environments. By harnessing its robust classification capabilities and inherent resilience to rotational variations and minor disturbances, we anticipate achieving accurate and reliable object recognition performance, even in complex real-world scenarios.

The mini-computer currently employed in the fall-detection system lacks a GPU, leading to suboptimal performance and noticeable delays in responsiveness. Consequently, upgrading to a more powerful mini-computer has become a priority item on our agenda to propel this project into its next phase. By securing a mini-computer with enhanced processing capabilities, including a dedicated GPU, we anticipate significant improvements in system performance, thereby enabling smoother and more efficient operation. This upgrade will facilitate the seamless execution of resource-intensive tasks, contributing to the overall advancement and efficacy of the project.

Moreover, continuous collaboration with stakeholders, including end-users and healthcare professionals, will be integral to the iterative refinement and validation of our system. Feedback from these stakeholders will inform ongoing improvements and enhancements, ensuring that our solution remains responsive to evolving needs and challenges in fall prevention and healthcare management.

Furthermore, the same mmWave radar technology can be used to extend support for human vital monitoring like heart rate, breathing rate and patterns. This is possible due to the sub-millimeter accuracy of the radar capable of capturing the minute displacements of heart and lungs. These vitals are crucial indicators of health status and can help to identify underlying conditions such as sleep apnea earlier when changes are observed over time.

In summary, our future work will center on the development and deployment of an advanced fall prevention system that leverages state-of-the-art technologies and proactive detection mechanisms to enhance safety and well-being in diverse healthcare settings. Through ongoing collaboration

and innovation, we aim to deliver a robust and effective solution that addresses the complex challenges associated with fall prevention and promotes optimal outcomes for individuals at risk of falls.

Appendix

Appendix A

```
import serial
import time
import numpy as np
import pyqtgraph as pg
from pyqtgraph.Qt import QtGui
import sys
from PyQt5 import QtGui, QtWidgets, QtCore
from pyqtgraph.opengl import GLViewWidget, GLScatterPlotItem
import csv
import pandas as pd

csv_file_path = 'radar_data.csv'


# Change the configuration file name
configFileName = '1443config.cfg'

CLIport = {}
Dataport = {}
byteBuffer = np.zeros(2**15,dtype = 'uint8')
byteBufferLength = 0;


# ------------------------------------------------------------------

# Function to configure the serial ports and send the data from
# the configuration file to the radar
def serialConfig(configFileName):

    global CLIport
    global Dataport
    # Open the serial ports for the configuration and the data ports

    # Raspberry pi
    #CLIport = serial.Serial('/dev/ttyACM0', 115200)
    #Dataport = serial.Serial('/dev/ttyACM1', 921600)

    # Windows
    CLIport = serial.Serial('COM3', 115200)
    Dataport = serial.Serial('COM4', 921600)

    # Read the configuration file and send it to the board
```

```
    config = [line.rstrip('\r\n') for line in open(configFileName)]
    for i in config:
        CLIport.write((i+'\n').encode())
        print(i)
        time.sleep(0.01)

    return CLIport, Dataport

# ------------------------------------------------------------------

# Function to parse the data inside the configuration file
def parseConfigFile(configFileName):
    configParameters = {} # Initialize an empty dictionary to store the
configuration parameters

    # Read the configuration file and send it to the board
    config = [line.rstrip('\r\n') for line in open(configFileName)]
    for i in config:

        # Split the line
        splitWords = i.split(" ")

        # Hard code the number of antennas, change if other configuration is used
        numRxAnt = 4
        numTxAnt = 3

        # Get the information about the profile configuration
        if "profileCfg" in splitWords[0]:
            startFreq = int(float(splitWords[2]))
            idleTime = int(splitWords[3])
            rampEndTime = float(splitWords[5])
            freqSlopeConst = float(splitWords[8])
            numAdcSamples = int(splitWords[10])
            numAdcSamplesRoundTo2 = 1;

            while numAdcSamples > numAdcSamplesRoundTo2:
                numAdcSamplesRoundTo2 = numAdcSamplesRoundTo2 * 2;

            digOutSampleRate = int(splitWords[11]);

        # Get the information about the frame configuration
        elif "frameCfg" in splitWords[0]:

            chirpStartIdx = int(splitWords[1]);
            chirpEndIdx = int(splitWords[2]);
```

```
            numLoops = int(splitWords[3]);
            numFrames = int(splitWords[4]);
            framePeriodicity = int(splitWords[5]);


    # Combine the read data to obtain the configuration parameters
    numChirpsPerFrame = (chirpEndIdx - chirpStartIdx + 1) * numLoops
    configParameters["numDopplerBins"] = numChirpsPerFrame / numTxAnt
    configParameters["numRangeBins"] = numAdcSamplesRoundTo2
    configParameters["rangeResolutionMeters"] = (3e8 * digOutSampleRate * 1e3) /
(2 * freqSlopeConst * 1e12 * numAdcSamples)
    configParameters["rangeIdxToMeters"] = (3e8 * digOutSampleRate * 1e3) / (2 *
freqSlopeConst * 1e12 * configParameters["numRangeBins"])
    configParameters["dopplerResolutionMps"] = 3e8 / (2 * startFreq * 1e9 *
(idleTime + rampEndTime) * 1e-6 * configParameters["numDopplerBins"] * numTxAnt)
    configParameters["maxRange"] = (300 * 0.9 * digOutSampleRate)/(2 *
freqSlopeConst * 1e3)
    configParameters["maxVelocity"] = 3e8 / (4 * startFreq * 1e9 * (idleTime +
rampEndTime) * 1e-6 * numTxAnt)

    return configParameters

# ------------------------------------------------------------------

# Funtion to read and parse the incoming data
def readAndParseData14xx(Dataport, configParameters):
    global byteBuffer, byteBufferLength

    # Constants
    OBJ_STRUCT_SIZE_BYTES = 12;
    BYTE_VEC_ACC_MAX_SIZE = 2**15;
    MMWDEMO_UART_MSG_DETECTED_POINTS = 1;
    MMWDEMO_UART_MSG_RANGE_PROFILE   = 2;
    maxBufferSize = 2**15;
    magicWord = [2, 1, 4, 3, 6, 5, 8, 7]

    # Initialize variables
    magicOK = 0 # Checks if magic number has been read
    dataOK = 0 # Checks if the data has been read correctly
    frameNumber = 0
    detObj = {}

    readBuffer = Dataport.read(Dataport.in_waiting)
    byteVec = np.frombuffer(readBuffer, dtype = 'uint8')
    byteCount = len(byteVec)
```

```
    # Check that the buffer is not full, and then add the data to the buffer
    if (byteBufferLength + byteCount) < maxBufferSize:
        byteBuffer[byteBufferLength:byteBufferLength + byteCount] =
byteVec[:byteCount]
        byteBufferLength = byteBufferLength + byteCount

    # Check that the buffer has some data
    if byteBufferLength > 16:

        # Check for all possible locations of the magic word
        possibleLocs = np.where(byteBuffer == magicWord[0])[0]

        # Confirm that is the beginning of the magic word and store the index in
startIdx
        startIdx = []
        for loc in possibleLocs:
            check = byteBuffer[loc:loc+8]
            if np.all(check == magicWord):
                startIdx.append(loc)

        # Check that startIdx is not empty
        if startIdx:

            # Remove the data before the first start index
            if startIdx[0] > 0 and startIdx[0] < byteBufferLength:
                byteBuffer[:byteBufferLength-startIdx[0]] =
byteBuffer[startIdx[0]:byteBufferLength]
                byteBuffer[byteBufferLength-startIdx[0]:] =
np.zeros(len(byteBuffer[byteBufferLength-startIdx[0]:]),dtype = 'uint8')
                byteBufferLength = byteBufferLength - startIdx[0]

            # Check that there have no errors with the byte buffer length
            if byteBufferLength < 0:
                byteBufferLength = 0

            # word array to convert 4 bytes to a 32 bit number
            word = [1, 2**8, 2**16, 2**24]

            # Read the total packet length
            totalPacketLen = np.matmul(byteBuffer[12:12+4],word)

            # Check that all the packet has been read
            if (byteBufferLength >= totalPacketLen) and (byteBufferLength != 0):
                magicOK = 1
```

```
# If magicOK is equal to 1 then process the message
if magicOK:
    # word array to convert 4 bytes to a 32 bit number
    word = [1, 2**8, 2**16, 2**24]

    # Initialize the pointer index
    idX = 0

    # Read the header
    magicNumber = byteBuffer[idX:idX+8]
    idX += 8
    version = format(np.matmul(byteBuffer[idX:idX+4],word),'x')
    idX += 4
    totalPacketLen = np.matmul(byteBuffer[idX:idX+4],word)
    idX += 4
    platform = format(np.matmul(byteBuffer[idX:idX+4],word),'x')
    idX += 4
    frameNumber = np.matmul(byteBuffer[idX:idX+4],word)
    idX += 4
    timeCpuCycles = np.matmul(byteBuffer[idX:idX+4],word)
    idX += 4
    numDetectedObj = np.matmul(byteBuffer[idX:idX+4],word)
    idX += 4
    numTLVs = np.matmul(byteBuffer[idX:idX+4],word)
    idX += 4

    # UNCOMMENT IN CASE OF SDK 2
    #subFrameNumber = np.matmul(byteBuffer[idX:idX+4],word)
    #idX += 4

    # Read the TLV messages
    for tlvIdx in range(numTLVs):

        # word array to convert 4 bytes to a 32 bit number
        word = [1, 2**8, 2**16, 2**24]

        # Check the header of the TLV message
        tlv_type = np.matmul(byteBuffer[idX:idX+4],word)
        idX += 4
        tlv_length = np.matmul(byteBuffer[idX:idX+4],word)
        idX += 4

        # Read the data depending on the TLV message
        if tlv_type == MMWDEMO_UART_MSG_DETECTED_POINTS:
```

```
                # word array to convert 4 bytes to a 16 bit number
                word = [1, 2**8]
                tlv_numObj = np.matmul(byteBuffer[idX:idX+2],word)
                idX += 2
                tlv_xyzQFormat = 2**np.matmul(byteBuffer[idX:idX+2],word)
                idX += 2

                # Initialize the arrays
                rangeIdx = np.zeros(tlv_numObj,dtype = 'int16')
                dopplerIdx = np.zeros(tlv_numObj,dtype = 'int16')
                peakVal = np.zeros(tlv_numObj,dtype = 'int16')
                x = np.zeros(tlv_numObj,dtype = 'int16')
                y = np.zeros(tlv_numObj,dtype = 'int16')
                z = np.zeros(tlv_numObj,dtype = 'int16')

                for objectNum in range(tlv_numObj):

                    # Read the data for each object
                    rangeIdx[objectNum] =  np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2
                    dopplerIdx[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2
                    peakVal[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2
                    x[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2
                    y[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2
                    z[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2

                # Make the necessary corrections and calculate the rest of the
data
                rangeVal = rangeIdx * configParameters["rangeIdxToMeters"]
                dopplerIdx[dopplerIdx > (configParameters["numDopplerBins"]/2 -
1)] = dopplerIdx[dopplerIdx > (configParameters["numDopplerBins"]/2 - 1)] - 65535
                dopplerVal = dopplerIdx *
configParameters["dopplerResolutionMps"]
                #x[x > 32767] = x[x > 32767] - 65536
                #y[y > 32767] = y[y > 32767] - 65536
                #z[z > 32767] = z[z > 32767] - 65536
                x = x / tlv_xyzQFormat
                y = y / tlv_xyzQFormat
                z = z / tlv_xyzQFormat
```

```
                # Store the data in the detObj dictionary
                detObj = {"numObj": tlv_numObj, "rangeIdx": rangeIdx, "range":
rangeVal, "dopplerIdx": dopplerIdx, \
                          "doppler": dopplerVal, "peakVal": peakVal, "x": x, "y":
y, "z": z}

                dataOK = 1


        # Remove already processed data
        if idX > 0 and byteBufferLength > idX:
            shiftSize = totalPacketLen

            byteBuffer[:byteBufferLength - shiftSize] =
byteBuffer[shiftSize:byteBufferLength]
            byteBuffer[byteBufferLength - shiftSize:] =
np.zeros(len(byteBuffer[byteBufferLength - shiftSize:]),dtype = 'uint8')
            byteBufferLength = byteBufferLength - shiftSize

            # Check that there are no errors with the buffer length
            if byteBufferLength < 0:
                byteBufferLength = 0


    return dataOK, frameNumber, detObj

# ------------------------------------------------------------------

# Funtion to update the data and display in the plot
def update():

    dataOk = 0
    global detObj
    x = []
    y = []

    # Read and parse the received data
    dataOk, frameNumber, detObj = readAndParseData14xx(Dataport,
configParameters)

    if dataOk and len(detObj["x"]) > 0:
        #print(detObj)
        x = -detObj["x"]
        y = detObj["y"]
```

```
        s.setData(x,y)
        QtGui.QGuiApplication.processEvents()

    return dataOk
# -------------------------    MAIN   -----------------------------------------
if __name__ == '__main__':
    # Configurate the serial port
    CLIport, Dataport = serialConfig(configFileName)

    # Get the configuration parameters from the configuration file
    configParameters = parseConfigFile(configFileName)

    # Initialize the Qt App and PlotWidget
    app = QtWidgets.QApplication(sys.argv)
    mainWin = QtWidgets.QMainWindow()
    mainWin.setWindowTitle('2D scatter plot')

    # Create a pyqtgraph Plot Widget and set it as the central widget of the
MainWindow
    plotWidget = pg.PlotWidget()
    mainWin.setCentralWidget(plotWidget)

    # Configure the plot
    plotWidget.setBackground('w')
    plotWidget.setXRange(-1.5, 1.5)
    plotWidget.setYRange(0, 3)
    plotWidget.setLabel('left', 'Y position (m)')
    plotWidget.setLabel('bottom', 'X position (m)')
    s = plotWidget.plot([], [], pen=None, symbol='o')

    # Show the MainWindow
    mainWin.show()

    # Define the update function within the main block to access the plotWidget
    def update():
        dataOk = 0
        global detObj
        x = []
        y = []

        # Read and parse the received data
        dataOk, frameNumber, detObj = readAndParseData14xx(Dataport,
configParameters)
```

```
        if dataOk and len(detObj["x"]) > 0:
            x = -detObj["x"]
            y = detObj["y"]
            z = detObj["z"]
            rangeIdx = detObj["rangeIdx"]
            dopplerIdx = detObj["dopplerIdx"]
            peakVal = detObj["peakVal"]
            df = pd.DataFrame({
                'Frame Number': [frameNumber] * len(x),
                'X': x,
                'Y': y,
                'Z': z,
                'RangeIdx': rangeIdx,
                'DopplerIdx': dopplerIdx,
                'PeakVal': peakVal
             })
    # Append the DataFrame to the CSV file
            df.to_csv(csv_file_path, mode='a', index=False, header=not
pd.io.common.file_exists(csv_file_path))
            s.setData(x, y)
            QtGui.QGuiApplication.processEvents()

        return dataOk

    # Main loop adaptation for PyQt (using a timer)
    timer = QtCore.QTimer()
    timer.timeout.connect(update)
    timer.start(33) # Update period in milliseconds to achieve ~30 Hz update rate

    # Start the Qt event loop
    sys.exit(app.exec_())
```

Appendix B:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
from matplotlib.animation import FuncAnimation, FFMpegWriter

# Set the path to the ffmpeg executable
plt.rcParams['animation.ffmpeg_path'] = 'C:/ffmpeg/bin/ffmpeg.exe'

# Read the CSV file
file = pd.read_csv('saad Fall processed time.csv')

# Drop the unnecessary columns
file = file.drop(["RangeIdx", "DopplerIdx", "PeakVal", "Frame Number", "Time
[ms]"], axis='columns')

# Group by 'Time[s]'
grouped = file.groupby('Time[s]')

# Define the radar position
radar_position = (0, 0, 0)

# Prepare the figure and 3D axis
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

def update(frame_number):
    ax.clear()  # Clear previous frame
    frame_data = grouped.get_group(frame_number)

    # Extract X, Y, Z values
    X = frame_data["X"]
    Y = frame_data["Y"]
    Z = frame_data["Z"]

    # Create a scatter plot for the data points
    ax.scatter(X, Y, Z, marker='o')

    # Plot the radar position
    ax.scatter(*radar_position, color='red', marker='^', label='Radar Position')
```

```
    # Set title
    ax.set_title(f'3D plot of XYZ changes for Time[s] {frame_number}')
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')

    # Adjust the axes limits
    ax.set_xlim([-1.5, 1.5])
    ax.set_ylim([-0, 2])
    ax.set_zlim([-2, 2])
    elevation_angle = 20  # change this value for elevation
    azimuth_angle = 300# change this value for azimuth
    ax.view_init(elev=elevation_angle, azim=azimuth_angle)

    # Show radar position in the legend
    ax.legend()

# Creating animation
ani = FuncAnimation(fig, update, frames=grouped.groups.keys(), interval=50,
repeat=False)

# Set up the writer
writer = FFMpegWriter(fps=20, metadata=dict(artist='Me'), bitrate=1800)

# Save the animation
ani.save('radar_data_animation.mp4', writer=writer)

# Show plot
plt.show()
```

Appendix C:

```
import argparse
import subprocess
import tensorflow as tf
import numpy as np
from datetime import datetime
import json
import os
import sys
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
sys.path.append(BASE_DIR)
sys.path.append(os.path.dirname(BASE_DIR))
import provider
import pointnet_part_seg as model


# enabling Tensorflow1
tf.compat.v1.disable_eager_execution()
# DEFAULT SETTINGS
parser = argparse.ArgumentParser()
parser.add_argument('--gpu', type=int, default=1, help='GPU to use [default: GPU
0]')
parser.add_argument('--batch', type=int, default=32, help='Batch Size during
training [default: 32]')
parser.add_argument('--epoch', type=int, default=200, help='Epoch to run
[default: 50]')
parser.add_argument('--point_num', type=int, default=2048, help='Point Number
[256/512/1024/2048]')
parser.add_argument('--output_dir', type=str, default='train_results',
help='Directory that stores all training logs and trained models')
parser.add_argument('--wd', type=float, default=0, help='Weight Decay [Default:
0.0]')
FLAGS = parser.parse_args()

hdf5_data_dir = os.path.join(BASE_DIR, './hdf5_data')

# MAIN SCRIPT
point_num = FLAGS.point_num
batch_size = FLAGS.batch
output_dir = FLAGS.output_dir

if not os.path.exists(output_dir):
    os.mkdir(output_dir)

color_map_file = os.path.join(hdf5_data_dir, 'part_color_mapping.json')
```

```
color_map = json.load(open(color_map_file, 'r'))

all_obj_cats_file = os.path.join(hdf5_data_dir, 'all_object_categories.txt')
fin = open(all_obj_cats_file, 'r')
lines = [line.rstrip() for line in fin.readlines()]
all_obj_cats = [(line.split()[0], line.split()[1]) for line in lines]
fin.close()

all_cats = json.load(open(os.path.join(hdf5_data_dir,
'overallid_to_catid_partid.json'), 'r'))
NUM_CATEGORIES = 16
NUM_PART_CATS = len(all_cats)

print('#### Batch Size: {0}'.format(batch_size))
print('#### Point Number: {0}'.format(point_num))
print('#### Training using GPU: {0}'.format(FLAGS.gpu))

DECAY_STEP = 16881 * 20
DECAY_RATE = 0.5

LEARNING_RATE_CLIP = 1e-5

BN_INIT_DECAY = 0.5
BN_DECAY_DECAY_RATE = 0.5
BN_DECAY_DECAY_STEP = float(DECAY_STEP * 2)
BN_DECAY_CLIP = 0.99

BASE_LEARNING_RATE = 0.001
MOMENTUM = 0.9
TRAINING_EPOCHES = FLAGS.epoch
print('### Training epoch: {0}'.format(TRAINING_EPOCHES))

TRAINING_FILE_LIST = os.path.join(hdf5_data_dir, 'train_hdf5_file_list.txt')
TESTING_FILE_LIST = os.path.join(hdf5_data_dir, 'val_hdf5_file_list.txt')

MODEL_STORAGE_PATH = os.path.join(output_dir, 'trained_models')
if not os.path.exists(MODEL_STORAGE_PATH):
    os.mkdir(MODEL_STORAGE_PATH)

LOG_STORAGE_PATH = os.path.join(output_dir, 'logs')
if not os.path.exists(LOG_STORAGE_PATH):
    os.mkdir(LOG_STORAGE_PATH)

SUMMARIES_FOLDER =  os.path.join(output_dir, 'summaries')
if not os.path.exists(SUMMARIES_FOLDER):
```

```
    os.mkdir(SUMMARIES_FOLDER)

def printout(flog, data):
    print(data)
    flog.write(data + '\n')

def placeholder_inputs():
    pointclouds_ph = tf.compat.v1.placeholder(tf.float32, shape=(batch_size,
point_num, 3))
    input_label_ph = tf.compat.v1.placeholder(tf.float32, shape=(batch_size,
NUM_CATEGORIES))
    labels_ph = tf.compat.v1.placeholder(tf.int32, shape=(batch_size))
    seg_ph = tf.compat.v1.placeholder(tf.int32, shape=(batch_size, point_num))
    return pointclouds_ph, input_label_ph, labels_ph, seg_ph

def convert_label_to_one_hot(labels):
    label_one_hot = np.zeros((labels.shape[0], NUM_CATEGORIES))
    for idx in range(labels.shape[0]):
        label_one_hot[idx, labels[idx]] = 1
    return label_one_hot

def train():
    with tf.Graph().as_default():
        with tf.device('/gpu:'+str(FLAGS.gpu)):
            pointclouds_ph, input_label_ph, labels_ph, seg_ph =
placeholder_inputs()
            is_training_ph = tf.compat.v1.placeholder(tf.bool, shape=())

            batch = tf.Variable(0, trainable=False)
            learning_rate = tf.compat.v1.train.exponential_decay(
                            BASE_LEARNING_RATE,     # base learning rate
                            batch * batch_size,     # global_var indicating the
number of steps
                            DECAY_STEP,             # step size
                            DECAY_RATE,             # decay rate
                            staircase=True          # Stair-case or continuous
decreasing
                            )
            learning_rate = tf.maximum(learning_rate, LEARNING_RATE_CLIP)

            bn_momentum = tf.compat.v1.train.exponential_decay(
                      BN_INIT_DECAY,
                      batch*batch_size,
                      BN_DECAY_DECAY_STEP,
                      BN_DECAY_DECAY_RATE,
```

```
                    staircase=True)
            bn_decay = tf.minimum(BN_DECAY_CLIP, 1 - bn_momentum)

            lr_op = tf.compat.v1.summary.scalar('learning_rate', learning_rate)
            batch_op = tf.compat.v1.summary.scalar('batch_number', batch)
            bn_decay_op = tf.compat.v1.summary.scalar('bn_decay', bn_decay)

            labels_pred, seg_pred, end_points = model.get_model(pointclouds_ph,
input_label_ph, \
                    is_training=is_training_ph, bn_decay=bn_decay,
cat_num=NUM_CATEGORIES, \
                    part_num=NUM_PART_CATS, batch_size=batch_size,
num_point=point_num, weight_decay=FLAGS.wd)

            # model.py defines both classification net and segmentation net,
which share the common global feature extractor network.
            # In model.get_loss, we define the total loss to be weighted sum of
the classification and segmentation losses.
            # Here, we only train for segmentation network. Thus, we set weight
to be 1.0.
            loss, label_loss, per_instance_label_loss, seg_loss,
per_instance_seg_loss, per_instance_seg_pred_res  \
                = model.get_loss(labels_pred, seg_pred, labels_ph, seg_ph, 1.0,
end_points)

            total_training_loss_ph = tf.compat.v1.placeholder(tf.float32,
shape=())
            total_testing_loss_ph = tf.compat.v1.placeholder(tf.float32,
shape=())

            label_training_loss_ph = tf.compat.v1.placeholder(tf.float32,
shape=())
            label_testing_loss_ph = tf.compat.v1.placeholder(tf.float32,
shape=())

            seg_training_loss_ph = tf.compat.v1.placeholder(tf.float32, shape=())
            seg_testing_loss_ph = tf.compat.v1.placeholder(tf.float32, shape=())

            label_training_acc_ph = tf.compat.v1.placeholder(tf.float32,
shape=())
            label_testing_acc_ph = tf.compat.v1.placeholder(tf.float32, shape=())
            label_testing_acc_avg_cat_ph = tf.compat.v1.placeholder(tf.float32,
shape=())

            seg_training_acc_ph = tf.compat.v1.placeholder(tf.float32, shape=())
```

```
            seg_testing_acc_ph = tf.compat.v1.placeholder(tf.float32, shape=())
            seg_testing_acc_avg_cat_ph = tf.compat.v1.placeholder(tf.float32,
shape=())

            total_train_loss_sum_op =
tf.compat.v1.summary.scalar('total_training_loss', total_training_loss_ph)
            total_test_loss_sum_op =
tf.compat.v1.summary.scalar('total_testing_loss', total_testing_loss_ph)

            label_train_loss_sum_op =
tf.compat.v1.summary.scalar('label_training_loss', label_training_loss_ph)
            label_test_loss_sum_op =
tf.compat.v1.summary.scalar('label_testing_loss', label_testing_loss_ph)

            seg_train_loss_sum_op =
tf.compat.v1.summary.scalar('seg_training_loss', seg_training_loss_ph)
            seg_test_loss_sum_op =
tf.compat.v1.summary.scalar('seg_testing_loss', seg_testing_loss_ph)

            label_train_acc_sum_op =
tf.compat.v1.summary.scalar('label_training_acc', label_training_acc_ph)
            label_test_acc_sum_op =
tf.compat.v1.summary.scalar('label_testing_acc', label_testing_acc_ph)
            label_test_acc_avg_cat_op =
tf.compat.v1.summary.scalar('label_testing_acc_avg_cat',
label_testing_acc_avg_cat_ph)

            seg_train_acc_sum_op =
tf.compat.v1.summary.scalar('seg_training_acc', seg_training_acc_ph)
            seg_test_acc_sum_op = tf.compat.v1.summary.scalar('seg_testing_acc',
seg_testing_acc_ph)
            seg_test_acc_avg_cat_op =
tf.compat.v1.summary.scalar('seg_testing_acc_avg_cat',
seg_testing_acc_avg_cat_ph)

            train_variables = tf.compat.v1.trainable_variables()

            trainer = tf.compat.v1.train.AdamOptimizer(learning_rate)
            train_op = trainer.minimize(loss, var_list=train_variables,
global_step=batch)

        saver = tf.compat.v1.train.Saver()

        config = tf.compat.v1.ConfigProto()
        config.gpu_options.allow_growth = True
```

```
        config.allow_soft_placement = True
        sess = tf.compat.v1.Session(config=config)

        init = tf.compat.v1.global_variables_initializer()
        sess.run(init)

        train_writer = tf.compat.v1.summary.FileWriter(SUMMARIES_FOLDER +
'/train', sess.graph)
        test_writer = tf.compat.v1.summary.FileWriter(SUMMARIES_FOLDER + '/test')
        print(train_writer)
        train_file_list = provider.getDataFiles(TRAINING_FILE_LIST)
        num_train_file = len(train_file_list)
        test_file_list = provider.getDataFiles(TESTING_FILE_LIST)
        num_test_file = len(test_file_list)

        fcmd = open(os.path.join(LOG_STORAGE_PATH, 'cmd.txt'), 'w')
        fcmd.write(str(FLAGS))
        fcmd.close()

        # write logs to the disk
        flog = open(os.path.join(LOG_STORAGE_PATH, 'log.txt'), 'w')

        def train_one_epoch(train_file_idx, epoch_num):
            is_training = True

            for i in range(num_train_file):
                cur_train_filename = os.path.join(hdf5_data_dir,
train_file_list[train_file_idx[i]])
                printout(flog, 'Loading train file ' + cur_train_filename)

                cur_data, cur_labels, cur_seg =
provider.loadDataFile_with_seg(cur_train_filename)
                cur_data, cur_labels, order = provider.shuffle_data(cur_data,
np.squeeze(cur_labels))
                cur_seg = cur_seg[order, ...]

                cur_labels_one_hot = convert_label_to_one_hot(cur_labels)

                num_data = len(cur_labels)
                num_batch = num_data // batch_size

                total_loss = 0.0
                total_label_loss = 0.0
                total_seg_loss = 0.0
                total_label_acc = 0.0
```

```
                total_seg_acc = 0.0

                for j in range(num_batch):
                    begidx = j * batch_size
                    endidx = (j + 1) * batch_size

                    feed_dict = {
                            pointclouds_ph: cur_data[begidx: endidx, ...],
                            labels_ph: cur_labels[begidx: endidx, ...],
                            input_label_ph: cur_labels_one_hot[begidx: endidx,
...],
                            seg_ph: cur_seg[begidx: endidx, ...],
                            is_training_ph: is_training,
                            }

                    _, loss_val, label_loss_val, seg_loss_val,
per_instance_label_loss_val, \
                            per_instance_seg_loss_val, label_pred_val,
seg_pred_val, pred_seg_res \
                            = sess.run([train_op, loss, label_loss, seg_loss,
per_instance_label_loss, \
                            per_instance_seg_loss, labels_pred, seg_pred,
per_instance_seg_pred_res], \
                            feed_dict=feed_dict)

                    per_instance_part_acc = np.mean(pred_seg_res ==
cur_seg[begidx: endidx, ...], axis=1)
                    average_part_acc = np.mean(per_instance_part_acc)

                    total_loss += loss_val
                    total_label_loss += label_loss_val
                    total_seg_loss += seg_loss_val

                    per_instance_label_pred = np.argmax(label_pred_val, axis=1)
                    total_label_acc += np.mean(np.float32(per_instance_label_pred
== cur_labels[begidx: endidx, ...]))
                    total_seg_acc += average_part_acc

                total_loss = total_loss * 1.0 / num_batch
                total_label_loss = total_label_loss * 1.0 / num_batch
                total_seg_loss = total_seg_loss * 1.0 / num_batch
                total_label_acc = total_label_acc * 1.0 / num_batch
                total_seg_acc = total_seg_acc * 1.0 / num_batch
```

```
                lr_sum, bn_decay_sum, batch_sum, train_loss_sum,
train_label_acc_sum, \
                        train_label_loss_sum, train_seg_loss_sum,
train_seg_acc_sum = sess.run(\
                        [lr_op, bn_decay_op, batch_op, total_train_loss_sum_op,
label_train_acc_sum_op, \
                        label_train_loss_sum_op, seg_train_loss_sum_op,
seg_train_acc_sum_op], \
                        feed_dict={total_training_loss_ph: total_loss,
label_training_loss_ph: total_label_loss, \
                        seg_training_loss_ph: total_seg_loss,
label_training_acc_ph: total_label_acc, \
                        seg_training_acc_ph: total_seg_acc})

                train_writer.add_summary(train_loss_sum, i + epoch_num *
num_train_file)
                train_writer.add_summary(train_label_loss_sum, i + epoch_num *
num_train_file)
                train_writer.add_summary(train_seg_loss_sum, i + epoch_num *
num_train_file)
                train_writer.add_summary(lr_sum, i + epoch_num * num_train_file)
                train_writer.add_summary(bn_decay_sum, i + epoch_num *
num_train_file)
                train_writer.add_summary(train_label_acc_sum, i + epoch_num *
num_train_file)
                train_writer.add_summary(train_seg_acc_sum, i + epoch_num *
num_train_file)
                train_writer.add_summary(batch_sum, i + epoch_num *
num_train_file)
                print("=========================================================
=========")

                printout(flog, '\tTraining Total Mean_loss: %f' % total_loss)
                printout(flog, '\t\tTraining Label Mean_loss: %f' %
total_label_loss)
                printout(flog, '\t\tTraining Label Accuracy: %f' %
total_label_acc)
                printout(flog, '\t\tTraining Seg Mean_loss: %f' % total_seg_loss)
                printout(flog, '\t\tTraining Seg Accuracy: %f' % total_seg_acc)

        def eval_one_epoch(epoch_num):
            is_training = False

            total_loss = 0.0
            total_label_loss = 0.0
```

```
            total_seg_loss = 0.0
            total_label_acc = 0.0
            total_seg_acc = 0.0
            total_seen = 0

            total_label_acc_per_cat =
np.zeros((NUM_CATEGORIES)).astype(np.float32)
            total_seg_acc_per_cat = np.zeros((NUM_CATEGORIES)).astype(np.float32)
            total_seen_per_cat = np.zeros((NUM_CATEGORIES)).astype(np.int32)

            for i in range(num_test_file):
                cur_test_filename = os.path.join(hdf5_data_dir,
test_file_list[i])
                printout(flog, 'Loading test file ' + cur_test_filename)

                cur_data, cur_labels, cur_seg =
provider.loadDataFile_with_seg(cur_test_filename)
                cur_labels = np.squeeze(cur_labels)

                cur_labels_one_hot = convert_label_to_one_hot(cur_labels)

                num_data = len(cur_labels)
                num_batch = num_data // batch_size

                for j in range(num_batch):
                    begidx = j * batch_size
                    endidx = (j + 1) * batch_size
                    feed_dict = {
                            pointclouds_ph: cur_data[begidx: endidx, ...],
                            labels_ph: cur_labels[begidx: endidx, ...],
                            input_label_ph: cur_labels_one_hot[begidx: endidx,
...],
                            seg_ph: cur_seg[begidx: endidx, ...],
                            is_training_ph: is_training,
                            }

                    loss_val, label_loss_val, seg_loss_val,
per_instance_label_loss_val, \
                            per_instance_seg_loss_val, label_pred_val,
seg_pred_val, pred_seg_res \
                            = sess.run([loss, label_loss, seg_loss,
per_instance_label_loss, \
                            per_instance_seg_loss, labels_pred, seg_pred,
per_instance_seg_pred_res], \
                            feed_dict=feed_dict)
```

```
                    per_instance_part_acc = np.mean(pred_seg_res ==
cur_seg[begidx: endidx, ...], axis=1)
                    average_part_acc = np.mean(per_instance_part_acc)

                    total_seen += 1
                    total_loss += loss_val
                    total_label_loss += label_loss_val
                    total_seg_loss += seg_loss_val

                    per_instance_label_pred = np.argmax(label_pred_val, axis=1)
                    total_label_acc += np.mean(np.float32(per_instance_label_pred
== cur_labels[begidx: endidx, ...]))
                    total_seg_acc += average_part_acc

                    for shape_idx in range(begidx, endidx):
                        total_seen_per_cat[cur_labels[shape_idx]] += 1
                        total_label_acc_per_cat[cur_labels[shape_idx]] +=
np.int32(per_instance_label_pred[shape_idx-begidx] == cur_labels[shape_idx])
                        total_seg_acc_per_cat[cur_labels[shape_idx]] +=
per_instance_part_acc[shape_idx - begidx]

            total_loss = total_loss * 1.0 / total_seen
            total_label_loss = total_label_loss * 1.0 / total_seen
            total_seg_loss = total_seg_loss * 1.0 / total_seen
            total_label_acc = total_label_acc * 1.0 / total_seen
            total_seg_acc = total_seg_acc * 1.0 / total_seen

            test_loss_sum, test_label_acc_sum, test_label_loss_sum,
test_seg_loss_sum, test_seg_acc_sum = sess.run(\
                    [total_test_loss_sum_op, label_test_acc_sum_op,
label_test_loss_sum_op, seg_test_loss_sum_op, seg_test_acc_sum_op], \
                    feed_dict={total_testing_loss_ph: total_loss,
label_testing_loss_ph: total_label_loss, \
                    seg_testing_loss_ph: total_seg_loss, label_testing_acc_ph:
total_label_acc, seg_testing_acc_ph: total_seg_acc})

            test_writer.add_summary(test_loss_sum, (epoch_num+1) *
num_train_file-1)
            test_writer.add_summary(test_label_loss_sum, (epoch_num+1) *
num_train_file-1)
            test_writer.add_summary(test_seg_loss_sum, (epoch_num+1) *
num_train_file-1)
            test_writer.add_summary(test_label_acc_sum, (epoch_num+1) *
num_train_file-1)
```

```
            test_writer.add_summary(test_seg_acc_sum, (epoch_num+1) *
num_train_file-1)

            printout(flog, '\tTesting Total Mean_loss: %f' % total_loss)
            printout(flog, '\t\tTesting Label Mean_loss: %f' % total_label_loss)
            printout(flog, '\t\tTesting Label Accuracy: %f' % total_label_acc)
            printout(flog, '\t\tTesting Seg Mean_loss: %f' % total_seg_loss)
            printout(flog, '\t\tTesting Seg Accuracy: %f' % total_seg_acc)

            for cat_idx in range(NUM_CATEGORIES):
                if total_seen_per_cat[cat_idx] > 0:
                    printout(flog, '\n\t\tCategory %s Object Number: %d' %
(all_obj_cats[cat_idx][0], total_seen_per_cat[cat_idx]))
                    printout(flog, '\t\tCategory %s Label Accuracy: %f' %
(all_obj_cats[cat_idx][0],
total_label_acc_per_cat[cat_idx]/total_seen_per_cat[cat_idx]))
                    printout(flog, '\t\tCategory %s Seg Accuracy: %f' %
(all_obj_cats[cat_idx][0],
total_seg_acc_per_cat[cat_idx]/total_seen_per_cat[cat_idx]))

        if not os.path.exists(MODEL_STORAGE_PATH):
            os.mkdir(MODEL_STORAGE_PATH)

        for epoch in range(TRAINING_EPOCHES):
            printout(flog, '\n<<< Testing on the test dataset ...')
            eval_one_epoch(epoch)

            printout(flog, '\n>>> Training for the epoch %d/%d ...' % (epoch,
TRAINING_EPOCHES))

            train_file_idx = np.arange(0, len(train_file_list))
            np.random.shuffle(train_file_idx)

            train_one_epoch(train_file_idx, epoch)

            if (epoch+1) % 10 == 0:
                cp_filename = saver.save(sess, os.path.join(MODEL_STORAGE_PATH,
'epoch_' + str(epoch+1)+'.ckpt'))
                printout(flog, 'Successfully store the checkpoint model into ' +
cp_filename)

            flog.flush()

        flog.close()
```

```
if __name__=='__main__':
    train()
```

Appendix D:

```
# Class autoencoder_mdl, compute_metric, function proposed_oversampling in
data_preproc are originally coded by Dr Feng Jin et al from
https://github.com/radar-lab/mmfall

import serial
import time
import numpy as np
import pyqtgraph as pg
from pyqtgraph.Qt import QtGui
import matplotlib.pyplot as plt
import serial
import time
import numpy as np
import pyqtgraph as pg
from pyqtgraph.Qt import QtGui
import sys
from PyQt5 import QtGui, QtWidgets, QtCore
from pyqtgraph.opengl import GLViewWidget, GLScatterPlotItem
import csv
import pandas as pd
import argparse, os
import random as rn
import tensorflow as tf
from keras import backend as K
from keras import optimizers
from keras.layers import Input, Dense, Flatten, Lambda, Concatenate, Reshape, \
    TimeDistributed, LSTM, RepeatVector, SimpleRNN, Activation
from keras.models import Model, load_model
from keras.callbacks import TensorBoard
from keras.losses import mse
from keras.utils import plot_model
from scipy.signal import find_peaks
#from sklearn.metrics import confusion_matrix
import pandas as pd
from tensorflow.keras import layers
from scipy.signal import butter, filtfilt
import glob
import subprocess

from keras.layers import Layer
from tensorflow.python.framework.ops import disable_eager_execution

disable_eager_execution()
```

```
class data_preproc:
    def __init__(self):
        self.frames_per_pattern = 20
        self.points_per_frame = 64
        self.features_per_point = 4
        self.split_ratio = 0.8
        tilt_angle = -10.0
        self.height = 2
        self.rotation_matrix = np.array([[1.0, 0.0, 0.0],
                                         [0.0, np.cos(np.deg2rad(tilt_angle)), -
np.sin(np.deg2rad(tilt_angle))],
                                         [0.0, np.sin(np.deg2rad(tilt_angle)),
np.cos(np.deg2rad(tilt_angle))]])

    def load_csv(self, data_frame, anomaly=False):
        centroidX_his = []
        centroidY_his = []
        centroidZ_his = []
        total_processed_pattern = []

        df = data_frame

        # Number of frames to process at once
        frames_batch_size = 20

        # Get unique frame numbers
        unique_frames = df['Frame Number'].unique()

        num_complete_batches = len(unique_frames) // frames_batch_size
        # Loop over the complete batches
        for batch_num in range(num_complete_batches):
            start = batch_num * frames_batch_size
            end = start + frames_batch_size
            frame_numbers = unique_frames[start:end]
            processed_pattern = []  # This will hold all processed frames in the
current batch

            for frame_number in frame_numbers:
                group = df[df['Frame Number'] == frame_number]

                if len(group) > self.points_per_frame:
                    continue

                centroid = group[['X', 'Y', 'Z']].mean().to_numpy()
                centroidx = centroid[0]
```

```
                centroidy = centroid[1]
                centroidz = centroid[2]
                results      = np.matmul(self.rotation_matrix,
np.array([centroidx,centroidy,centroidz]))
                centroidx    = results[0]
                centroidy    = results[1]
                centroidz    = results[2] + self.height

                centroidX_his.append(centroidx)
                centroidY_his.append(centroidy)
                centroidZ_his.append(centroidz)

                processed_frame = []
                for _, row in group.iterrows():
                    # Apply rotation and adjust for height
                    point = row[['X', 'Y', 'Z']].to_numpy()
                    rotated_point = np.matmul(self.rotation_matrix, row[['X',
'Y', 'Z']].to_numpy())
                    pointX, pointY, pointZ = rotated_point + np.array([0, 0,
self.height])

                    # Calculate deltas
                    delta_x = pointX - centroidx
                    delta_y = pointY - centroidy
                    delta_z = pointZ
                    delta_D = row['velocity']

                    # Form the feature vector
                    feature_vector = [delta_x, delta_y, delta_z, delta_D]
                    processed_frame.append(feature_vector)


                processed_pattern.append(processed_frame)

            if len(processed_pattern) == frames_batch_size:
                processed_pattern_oversampled =
self.proposed_oversampling(processed_pattern)
                total_processed_pattern.append(processed_pattern_oversampled)

        total_processed_pattern_np = np.array(total_processed_pattern)

        # Split data into training and testing sets
        split_idx = int(total_processed_pattern_np.shape[0] * self.split_ratio)
        train_data = total_processed_pattern_np[:split_idx]
        test_data = total_processed_pattern_np[split_idx:]
```

```
        if anomaly == False:
            print("INFO: Total normal motion pattern data shape: " +
str(total_processed_pattern_np.shape))
            print("INFO: Training motion pattern data shape" +
str(train_data.shape))
            print("INFO: Testing motion pattern data shape" +
str(test_data.shape))
            # Return training and testing data along with centroid histories for
normal dataset
            return train_data, test_data, centroidZ_his
        else:
            # Return processed pattern and centroid histories for anomaly dataset
            print("INFO: Total inference motion pattern data shape: " +
str(total_processed_pattern_np.shape))
            return total_processed_pattern_np,  centroidZ_his

    def proposed_oversampling(self, processed_pointcloud):
        # Do data oversampling
        processed_pointcloud_oversampled = []
        for frame in processed_pointcloud:
            frame_np = np.array(frame)

            # Check if it's empty frame
            N = self.points_per_frame
            M = frame_np.shape[0]
            assert (M != 0), "ERROR: empty frame detected!"

            # Rescale and padding
            mean        = np.mean(frame_np, axis=0)
            sigma       = np.std(frame_np, axis=0)
            frame_np    = np.sqrt(N/M)*frame_np + mean - np.sqrt(N/M)*mean #
Rescale
            frame_oversampled = frame_np.tolist()
            frame_oversampled.extend([mean]*(N-M)) # Padding with mean
            processed_pointcloud_oversampled.append(frame_oversampled)

        processed_pointcloud_oversampled_np =
np.array(processed_pointcloud_oversampled)

        assert (processed_pointcloud_oversampled_np.shape[-2] ==
self.points_per_frame), ("ERROR: The new_frame_data has different number of
points per frame rather than %s!" %(self.points_per_frame))
```

```
        assert (processed_pointcloud_oversampled_np.shape[-1] ==
self.features_per_point), ("ERROR: The new_frame_data has different feature
length rather than %s!" %(self.features_per_point))

        return processed_pointcloud_oversampled_np


class SamplingLayer(layers.Layer):
    """Sampling layer for Variational Autoencoder"""
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim1 = tf.shape(z_mean)[1]  # Additional dimensions if present
        dim2 = tf.shape(z_mean)[2]  # You adjust this based on your specific
needs

        # Adjust the shape of epsilon based on the shape of your z_mean and
z_log_var
        epsilon = tf.keras.backend.random_normal(shape=(batch, dim1, dim2))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

class autoencoder_mdl:
    def __init__(self, model_dir):
        self.model_dir = model_dir

    # Variational Recurrent Autoencoder (HVRAE)
    def HVRAE_train(self, train_data, test_data):
        # In one motion pattern we have
        n_frames       = 20
        n_points       = 64
        n_features     = 4

        # Dimension is going down for encoding. Decoding is just a reflection of
encoding.
        n_intermidiate    = 64
        n_latentdim       = 16

        # Define input
        inputs                  = Input(shape=(n_frames, n_points, n_features))
        input_flatten           = TimeDistributed(Flatten(None))(inputs)

        # VAE: q(z|X). Input: motion pattern. Output: mean and log(sigma^2) for
q(z|X).
        input_flatten           = TimeDistributed(Dense(n_intermidiate,
activation='tanh'))(input_flatten)
```

```
        Z_mean                  = TimeDistributed(Dense(n_latentdim,
activation=None), name='qzx_mean')(input_flatten)
        Z_log_var               = TimeDistributed(Dense(n_latentdim,
activation=None), name='qzx_log_var')(input_flatten)

        Z = SamplingLayer()([Z_mean, Z_log_var])

        # RNN Autoencoder. Output: reconstructed z.
        encoder_feature         = SimpleRNN(n_latentdim, activation='tanh',
return_sequences=False)(Z)
        decoder_feature         = RepeatVector(n_frames)(encoder_feature)
        decoder_feature         = SimpleRNN(n_latentdim, activation='tanh',
return_sequences=True)(decoder_feature)
        decoder_feature         = Lambda(lambda x: tf.reverse(x, axis=[-
2]))(decoder_feature)

        # VAE: p(X|z). Output: mean and log(sigma^2) for p(X|z).
        X_latent                = TimeDistributed(Dense(n_intermidiate,
activation='tanh'))(decoder_feature)
        pXz_mean                = TimeDistributed(Dense(n_features,
activation=None))(X_latent)
        pXz_logvar              = TimeDistributed(Dense(n_features,
activation=None))(X_latent)

        # Reshape the output. Output: (n_frames, n_points, n_features*2).
        # In each frame, every point has a corresponding mean vector with length
of n_features and a log(sigma^2) vector with length of n_features.
        pXz                     = Concatenate()([pXz_mean, pXz_logvar])
        pXz                     = TimeDistributed(RepeatVector(n_points))(pXz)
        outputs                 = TimeDistributed(Reshape((n_points,
n_features*2)))(pXz)

        # Build the model
        self.HVRAE_mdl = Model(inputs, outputs)
        print(self.HVRAE_mdl.summary())

        # Calculate HVRAE loss proposed in the paper
        def HVRAE_loss(y_true, y_pred):
            batch_size      = K.shape(y_true)[0]
            n_frames        = K.shape(y_true)[1]
            n_features      = K.shape(y_true)[-1]

            mean            = y_pred[:, :, :n_features]
            logvar          = y_pred[:, :, n_features:]
            var             = K.exp(logvar)
```

```
            y_true_reshape  = K.reshape(y_true, (batch_size, n_frames, -1))
            mean            = K.reshape(mean, (batch_size, n_frames, -1))
            var             = K.reshape(var, (batch_size, n_frames, -1))
            logvar          = K.reshape(logvar, (batch_size, n_frames, -1))

            # E[log_pXz] ~= log_pXz
            log_pXz         = K.square(y_true_reshape - mean)/var
            log_pXz         = K.sum(0.5*log_pXz, axis=-1)

            # KL divergence between q(z|x) and p(z)
            kl_loss         = -0.5 * K.sum(1 + Z_log_var - K.square(Z_mean) -
K.exp(Z_log_var), axis=-1)

            # HVRAE loss is log_pXz + kl_loss
            HVRAE_loss        = K.mean(log_pXz + kl_loss) # Do mean over batches
            return HVRAE_loss

        # Define stochastic gradient descent optimizer Adam
        adam    = optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
amsgrad=False)
        # Compile the model
        self.HVRAE_mdl.compile(optimizer=adam, loss=HVRAE_loss)

        # Train the model
        self.HVRAE_mdl.fit(train_data, train_data, # Train on the normal training
dataset in an unsupervised way
                epochs=20,
                batch_size=8,
                shuffle=False,
                validation_data=(test_data, test_data), # Testing on the normal
tesing dataset
                callbacks=[TensorBoard(log_dir=(self.model_dir))])
        self.HVRAE_mdl.save(self.model_dir + 'HVRAE_mdl.h5')
        print("INFO: Training is done!")

print("*********************************************************************")

    def HVRAE_predict(self, inferencedata):# add reltime centroid z
        K.clear_session()

        def sampling_predict(args): # Instead of sampling from Q(z|X), sample
epsilon = N(0,I), z = z_mean + sqrt(var) * epsilon
            Z_mean, Z_log_var   = args
            batch_size          = K.shape(Z_mean)[0]
```

```
            n_frames            = K.int_shape(Z_mean)[1]
            n_latentdim         = K.int_shape(Z_mean)[2]
            # For reproducibility, we set the seed=37
            epsilon             = K.random_normal(shape=(batch_size, n_frames,
n_latentdim), mean=0., stddev=1.0, seed=None)
            Z                   = Z_mean + K.exp(0.5*Z_log_var) * epsilon # The
reparameterization trick
            return  Z

        # Load saved model
        model = load_model(self.model_dir + 'HVRAE_mdl_all.h5', compile = False,
custom_objects={'SamplingLayer': SamplingLayer, 'tf': tf})
        adam  = optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
amsgrad=False)
        model.compile(optimizer=adam, loss=mse)
        print("INFO: Model loaded from " + self.model_dir + 'HVRAE_mdl.h5')

        get_z_mean_model    = Model(inputs=model.input,
outputs=model.get_layer('qzx_mean').output)
        get_z_log_var_model = Model(inputs=model.input,
outputs=model.get_layer('qzx_log_var').output)

        # Numpy version of HVRAE_loss function
        def HVRAE_loss(y_true, y_pred, Z_mean, Z_log_var):
            batch_size      = y_true.shape[0]
            n_frames        = y_true.shape[1]
            n_features      = y_true.shape[-1]

            mean            = y_pred[:, :, :n_features]
            logvar          = y_pred[:, :, n_features:]
            var             = np.exp(logvar)

            y_true_reshape  = np.reshape(y_true, (batch_size, n_frames, -1))
            mean            = np.reshape(mean, (batch_size, n_frames, -1))
            var             = np.reshape(var, (batch_size, n_frames, -1))
            logvar          = np.reshape(logvar, (batch_size, n_frames, -1))

            # E[log_pXz] ~= log_pXz
            # log_pXz       = K.square(y_true_reshape-mean)/var + logvar
            log_pXz         = np.square(y_true_reshape - mean)/var
            log_pXz         = np.sum(0.5*log_pXz, axis=-1)

            # KL divergence between q(z|x) and p(z)
            kl_loss         = -0.5 * np.sum(1 + Z_log_var - np.square(Z_mean) -
np.exp(Z_log_var), axis=-1)
```

```
            # HVRAE loss is log_pXz + kl_loss
            HVRAE_loss        = np.mean(log_pXz + kl_loss) # Do mean over batches
            return HVRAE_loss

        print("INFO: Start to predict...")
        prediction_history  = []
        loss_history        = []
        for pattern in inferencedata:
            pattern             = np.expand_dims(pattern, axis=0)
            current_prediction  = model.predict(pattern, batch_size=1)
            predicted_z_mean    = get_z_mean_model.predict(pattern, batch_size=1)
            predicted_z_log_var = get_z_log_var_model.predict(pattern,
batch_size=1)
            current_loss        = HVRAE_loss(pattern, current_prediction,
predicted_z_mean, predicted_z_log_var)
            loss_history.append(current_loss)
        print("INFO: Prediction is done!")

        return loss_history

class compute_metric:
    def __init__(self):
        pass

    def detect_falls(self, loss_history, centroidZ_history, threshold):
        assert len(loss_history) == len(centroidZ_history), "ERROR: The length of
loss history is different than the length of centroidZ history!"
        seq_len                 = len(loss_history)
        win_len                 = 40 # Detection window length on account of 2
seconds for 20 fps radar rate
        centroidZ_dropthres     = 1.0
        i                       = int(win_len/2)
        detected_falls_idx      = []
        # Firstly, detect the fall centers based on the centroidZ drop
        while i < (seq_len - win_len/2):
            detection_window_middle  = i
            detection_window_lf_edge = int(detection_window_middle - win_len/2)
            detection_window_rh_edge = int(detection_window_middle + win_len/2)
            # Search the centroidZ drop
            if centroidZ_history[detection_window_lf_edge] -
centroidZ_history[detection_window_rh_edge] >= centroidZ_dropthres:
                detected_falls_idx.append(int(detection_window_middle))
            i += 1
```

```
        # Secondly, if a sequence of fall happen within a window less than
win_len, we combine these falls into one fall centered at the middle of this
sequence
        i = 0
        processed_detected_falls_idx = []
        while i < len(detected_falls_idx):
            j = i
            while True:
                if j == len(detected_falls_idx):
                    break
                if detected_falls_idx[j] - detected_falls_idx[i] > win_len:
                    break
                j += 1
            processed_detected_falls_idx.append(int((detected_falls_idx[i] +
detected_falls_idx[j-1])/2))
            i = j

        # Thirdly, find id there is an anomaly level (or loss history) spike in
the detection window
        ones_idx                    =
np.argwhere(np.array(loss_history)>=threshold).flatten()
        fall_binseq                 = np.zeros(seq_len)
        fall_binseq[ones_idx]       = 1
        final_detected_falls_idx    = []
        i = 0
        while i < len(processed_detected_falls_idx):
            detection_window_middle  = int(processed_detected_falls_idx[i])
            detection_window_lf_edge = int(detection_window_middle - win_len/2)
            detection_window_rh_edge = int(detection_window_middle + win_len/2)
            if 1 in
fall_binseq[detection_window_lf_edge:detection_window_rh_edge]:
                final_detected_falls_idx.append(processed_detected_falls_idx[i])
            i += 1

        return final_detected_falls_idx, len(processed_detected_falls_idx)

    def find_tpfpfn(self, detected_falls_idx, gt_falls_idx):
        n_detected_falls    = len(detected_falls_idx)
        falls_tp            = []
        falls_fp            = []
        falls_fn            = list(gt_falls_idx)
        win_len             = 20
        for i in range(n_detected_falls):
            n_gt_falls      = len(falls_fn)
            j               = 0
```

```
            while j < n_gt_falls:
                # Find a gt fall index whose window covers the detected fall
index, so it's true positive
                if int(falls_fn[j]-win_len/2) <= detected_falls_idx[i] <=
int(falls_fn[j]+win_len/2):
                    # Remove the true positive from the gt_falls_idx list,
finally only false negative remains
                    falls_fn.pop(j)
                    falls_tp.append(i)
                    break
                j += 1
            # Dn not find a gt fall index whose window covers the detected fall
index, so it's false positive
            if j == n_gt_falls:
                falls_fp.append(i)

        return falls_tp, falls_fp, falls_fn

    def cal_roc(self, loss_history, centroidZ_history, gt_falls_idx):
        n_gt_falls = len(gt_falls_idx)
        print("How many falls?", n_gt_falls)
        tpr, fpr = [], []
        for threshold in np.arange(0.0, 10.0, 0.1):
            detected_falls_idx, _           = self.detect_falls(loss_history,
centroidZ_history, threshold)
            falls_tp, falls_fp, falls_fn    =
self.find_tpfpfn(detected_falls_idx, gt_falls_idx)
            # Save the true positve rate for this threshold.
            tpr.append(len(falls_tp)/n_gt_falls)
            # Save the number of false positve, or missed fall detection, for
this threshold
            fpr.append(len(falls_fp))
        return tpr, fpr

# Change the configuration file name
configFileName = 'IWR1443_profile_Optimized.cfg'
csv_file_path = 'radar_data.csv'
CLIport = {}
Dataport = {}
byteBuffer = np.zeros(2**15,dtype = 'uint8')
byteBufferLength = 0


# ------------------------------------------------------------------
```

```
# Function to configure the serial ports and send the data from
# the configuration file to the radar
def serialConfig(configFileName):

    global CLIport
    global Dataport
    # Open the serial ports for the configuration and the data ports

    # Raspberry pi
    CLIport = serial.Serial('COM4', 115200)
    Dataport = serial.Serial('COM3', 921600)

    # Windows
    # CLIport = serial.Serial('COM4', 115200)
    # Dataport = serial.Serial('COM3', 921600)

    # Read the configuration file and send it to the board
    config = [line.rstrip('\r\n') for line in open(configFileName)]
    for i in config:
        CLIport.write((i+'\n').encode())
        print(i)
        time.sleep(0.01)

    return CLIport, Dataport

# ------------------------------------------------------------------

# Function to parse the data inside the configuration file
def parseConfigFile(configFileName):
    configParameters = {} # Initialize an empty dictionary to store the
configuration parameters

    # Read the configuration file and send it to the board
    config = [line.rstrip('\r\n') for line in open(configFileName)]
    for i in config:

        # Split the line
        splitWords = i.split(" ")

        # Hard code the number of antennas, change if other configuration is used
        numRxAnt = 4
        numTxAnt = 3

        # Get the information about the profile configuration
        if "profileCfg" in splitWords[0]:
```

```
            startFreq = int(float(splitWords[2]))
            idleTime = int(splitWords[3])
            rampEndTime = float(splitWords[5])
            freqSlopeConst = float(splitWords[8])
            numAdcSamples = int(splitWords[10])
            numAdcSamplesRoundTo2 = 1;

            while numAdcSamples > numAdcSamplesRoundTo2:
                numAdcSamplesRoundTo2 = numAdcSamplesRoundTo2 * 2;

            digOutSampleRate = int(splitWords[11]);

        # Get the information about the frame configuration
        elif "frameCfg" in splitWords[0]:

            chirpStartIdx = int(splitWords[1]);
            chirpEndIdx = int(splitWords[2]);
            numLoops = int(splitWords[3]);
            numFrames = int(splitWords[4]);
            framePeriodicity = int(splitWords[5]);


    # Combine the read data to obtain the configuration parameters
    numChirpsPerFrame = (chirpEndIdx - chirpStartIdx + 1) * numLoops
    configParameters["numDopplerBins"] = numChirpsPerFrame / numTxAnt
    configParameters["numRangeBins"] = numAdcSamplesRoundTo2
    configParameters["rangeResolutionMeters"] = (3e8 * digOutSampleRate * 1e3) /
(2 * freqSlopeConst * 1e12 * numAdcSamples)
    configParameters["rangeIdxToMeters"] = (3e8 * digOutSampleRate * 1e3) / (2 *
freqSlopeConst * 1e12 * configParameters["numRangeBins"])
    configParameters["dopplerResolutionMps"] = 3e8 / (2 * startFreq * 1e9 *
(idleTime + rampEndTime) * 1e-6 * configParameters["numDopplerBins"] * numTxAnt)
    configParameters["maxRange"] = (300 * 0.9 * digOutSampleRate)/(2 *
freqSlopeConst * 1e3)
    configParameters["maxVelocity"] = 3e8 / (4 * startFreq * 1e9 * (idleTime +
rampEndTime) * 1e-6 * numTxAnt)

    return configParameters

# ------------------------------------------------------------------

# Funtion to read and parse the incoming data
def readAndParseData14xx(Dataport, configParameters):
    global byteBuffer, byteBufferLength
```

```
    # Constants
    OBJ_STRUCT_SIZE_BYTES = 12;
    BYTE_VEC_ACC_MAX_SIZE = 2**15;
    MMWDEMO_UART_MSG_DETECTED_POINTS = 1;
    MMWDEMO_UART_MSG_RANGE_PROFILE   = 2;
    maxBufferSize = 2**15;
    magicWord = [2, 1, 4, 3, 6, 5, 8, 7]

    # Initialize variables
    magicOK = 0 # Checks if magic number has been read
    dataOK = 0 # Checks if the data has been read correctly
    frameNumber = 0
    detObj = {}

    readBuffer = Dataport.read(Dataport.in_waiting)
    byteVec = np.frombuffer(readBuffer, dtype = 'uint8')
    byteCount = len(byteVec)

    # Check that the buffer is not full, and then add the data to the buffer
    if (byteBufferLength + byteCount) < maxBufferSize:
        byteBuffer[byteBufferLength:byteBufferLength + byteCount] =
byteVec[:byteCount]
        byteBufferLength = byteBufferLength + byteCount

    # Check that the buffer has some data
    if byteBufferLength > 16:

        # Check for all possible locations of the magic word
        possibleLocs = np.where(byteBuffer == magicWord[0])[0]

        # Confirm that is the beginning of the magic word and store the index in
startIdx
        startIdx = []
        for loc in possibleLocs:
            check = byteBuffer[loc:loc+8]
            if np.all(check == magicWord):
                startIdx.append(loc)

        # Check that startIdx is not empty
        if startIdx:

            # Remove the data before the first start index
            if startIdx[0] > 0 and startIdx[0] < byteBufferLength:
                byteBuffer[:byteBufferLength-startIdx[0]] =
byteBuffer[startIdx[0]:byteBufferLength]
```

```
                byteBuffer[byteBufferLength-startIdx[0]:] =
np.zeros(len(byteBuffer[byteBufferLength-startIdx[0]:]),dtype = 'uint8')
                byteBufferLength = byteBufferLength - startIdx[0]

            # Check that there have no errors with the byte buffer length
            if byteBufferLength < 0:
                byteBufferLength = 0

            # word array to convert 4 bytes to a 32 bit number
            word = [1, 2**8, 2**16, 2**24]

            # Read the total packet length
            totalPacketLen = np.matmul(byteBuffer[12:12+4],word)

            # Check that all the packet has been read
            if (byteBufferLength >= totalPacketLen) and (byteBufferLength != 0):
                magicOK = 1

    # If magicOK is equal to 1 then process the message
    if magicOK:
        # word array to convert 4 bytes to a 32 bit number
        word = [1, 2**8, 2**16, 2**24]

        # Initialize the pointer index
        idX = 0

        # Read the header
        magicNumber = byteBuffer[idX:idX+8]
        idX += 8
        version = format(np.matmul(byteBuffer[idX:idX+4],word),'x')
        idX += 4
        totalPacketLen = np.matmul(byteBuffer[idX:idX+4],word)
        idX += 4
        platform = format(np.matmul(byteBuffer[idX:idX+4],word),'x')
        idX += 4
        frameNumber = np.matmul(byteBuffer[idX:idX+4],word)
        idX += 4
        timeCpuCycles = np.matmul(byteBuffer[idX:idX+4],word)
        idX += 4
        numDetectedObj = np.matmul(byteBuffer[idX:idX+4],word)
        idX += 4
        numTLVs = np.matmul(byteBuffer[idX:idX+4],word)
        idX += 4
```

```
# UNCOMMENT IN CASE OF SDK 2
#subFrameNumber = np.matmul(byteBuffer[idX:idX+4],word)
#idX += 4

# Read the TLV messages
for tlvIdx in range(numTLVs):

    # print('range tlv = ',  tlvIdx )
    # word array to convert 4 bytes to a 32 bit number
    word = [1, 2**8, 2**16, 2**24]

    # Check the header of the TLV message
    tlv_type = np.matmul(byteBuffer[idX:idX+4],word)
    idX += 4
    tlv_length = np.matmul(byteBuffer[idX:idX+4],word)
    idX += 4

    # Read the data depending on the TLV message
    if tlv_type == MMWDEMO_UART_MSG_DETECTED_POINTS:

        # word array to convert 4 bytes to a 16 bit number
        word = [1, 2**8]
        tlv_numObj = np.matmul(byteBuffer[idX:idX+2],word)
        idX += 2
        tlv_xyzQFormat = 2**np.matmul(byteBuffer[idX:idX+2],word)
        # print('tlv_xyzQFormat',tlv_xyzQFormat)
        # print('tlv_numObj = ', tlv_numObj)
        #os.system('Pause')
        idX += 2

        # Initialize the arrays
        rangeIdx = np.zeros(tlv_numObj,dtype = 'int16')
        dopplerIdx = np.zeros(tlv_numObj,dtype = 'int16')
        peakVal = np.zeros(tlv_numObj,dtype = 'int16')
        x = np.zeros(tlv_numObj,dtype = 'int16')
        y = np.zeros(tlv_numObj,dtype = 'int16')
        z = np.zeros(tlv_numObj,dtype = 'int16')

        for objectNum in range(tlv_numObj):

            # Read the data for each object
            rangeIdx[objectNum] =  np.matmul(byteBuffer[idX:idX+2],word)
            idX += 2
            dopplerIdx[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
```

```
                    idX += 2
                    peakVal[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2
                    x[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2
                    y[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2
                    z[objectNum] = np.matmul(byteBuffer[idX:idX+2],word)
                    idX += 2

                # Make the necessary corrections and calculate the rest of the
data
                rangeVal = rangeIdx * configParameters["rangeIdxToMeters"]
                dopplerIdx[dopplerIdx > (configParameters["numDopplerBins"]/2 -
1)] = dopplerIdx[dopplerIdx > (configParameters["numDopplerBins"]/2 - 1)] - 65535
                dopplerVal = dopplerIdx *
configParameters["dopplerResolutionMps"]
                #x[x > 32767] = x[x > 32767] - 65536
                #y[y > 32767] = y[y > 32767] - 65536
                #z[z > 32767] = z[z > 32767] - 65536
                x = x / tlv_xyzQFormat
                y = y / tlv_xyzQFormat
                z = z / tlv_xyzQFormat

                # Store the data in the detObj dictionary
                detObj = {"numObj": tlv_numObj, "rangeIdx": rangeIdx, "range":
rangeVal, "dopplerIdx": dopplerIdx, \
                          "doppler": dopplerVal, "peakVal": peakVal, "x": x, "y":
y, "z": z}

                dataOK = 1


        # Remove already processed data
        if idX > 0 and byteBufferLength > idX:
            shiftSize = totalPacketLen

            byteBuffer[:byteBufferLength - shiftSize] =
byteBuffer[shiftSize:byteBufferLength]
            byteBuffer[byteBufferLength - shiftSize:] =
np.zeros(len(byteBuffer[byteBufferLength - shiftSize:]),dtype = 'uint8')
            byteBufferLength = byteBufferLength - shiftSize

            # Check that there are no errors with the buffer length
            if byteBufferLength < 0:
```

```
                byteBufferLength = 0


    return dataOK, frameNumber, detObj


################################ MAIN ####################################

import tensorflow as tf
import sys
import numpy as np
import pandas as pd
import pyqtgraph.opengl as gl
from PyQt5 import QtWidgets, QtCore
import tensorflow as tf
import serial
from pyqtgraph.Qt import QtGui
import pyqtgraph as pg


recent_predictions = []

current_frame_data = None
current_frame_number = None
fall_detected = False
# Initialize global variables for frame data handling
current_frame_data = pd.DataFrame()
current_frame_number = -1

data_processor = data_preproc()


#------------------ GUI SET UP -----------------------------------------
class RadarGUI(QtWidgets.QMainWindow):
    def __init__(self, parent=None):
        super(RadarGUI, self).__init__(parent)
        self.is_recording = False

        # Set up central widget and layout
        central_widget = QtWidgets.QWidget()
        self.setCentralWidget(central_widget)
        layout = QtWidgets.QVBoxLayout(central_widget)

        # Set up the 3D scatter plot widget and add it to the layout
        self.scatter_widget = gl.GLViewWidget()
```

```
        layout.addWidget(self.scatter_widget)

        # Initialize the timer for resetting the fall indicator
        self.reset_timer = QtCore.QTimer(self)
        self.reset_timer.setSingleShot(True)
        self.reset_timer.timeout.connect(self.reset_fall_indicator)

        # Create and add a scatter plot item and a cube frame to the widget
        self.scatter = gl.GLScatterPlotItem()
        self.scatter_widget.addItem(self.scatter)
        cube_lines = self.create_cube(width=5, height=5, depth=3,
y_translation=2.5)
        for line_item in cube_lines:
            self.scatter_widget.addItem(line_item)

        # Configure the camera for an isometric view
        self.scatter_widget.setCameraPosition(distance=15, elevation=30,
azimuth=45)
        self.scatter_widget.opts['center'] = QtGui.QVector3D(-2, -0, -2)  #
Adjust the 1 to your needs
        self.scatter_widget.update()

        # Create occupancy grid
        self.create_occupancy_grid(cube_width=5, cube_height=3, cube_depth=5,
grid_width=10, grid_height=10, spacing=0.5, cube_y_translation=0)

        # Bottom layout for button and fall indicator
        bottom_layout = QtWidgets.QHBoxLayout()
        bottom_layout.addStretch()  # Add a spacer on the left side

        # Create the Start Recording button
        self.start_recording_button = QtWidgets.QPushButton("Start Detecting")
        button_size = 250  # Square button size
        self.start_recording_button.setFixedSize(button_size, button_size)
        self.start_recording_button.setStyleSheet("QPushButton { font-size:
18pt; }")
        bottom_layout.addWidget(self.start_recording_button)

        # Modify the fall detection indicator (label)
        self.fall_indicator = QtWidgets.QLabel("Monitoring...")
        self.fall_indicator.setAlignment(QtCore.Qt.AlignCenter)
        self.fall_indicator.setFixedSize(button_size, button_size)
        self.fall_indicator.setStyleSheet("QLabel { background-color: green;
border: 1px solid black; font-size: 18pt; }")
        bottom_layout.addWidget(self.fall_indicator)
```

```
        bottom_layout.addStretch()  # Add a spacer on the right side

        # Add the bottom layout to the main vertical layout
        layout.addLayout(bottom_layout)

        # Connect the button click to the start_recording method
        self.start_recording_button.clicked.connect(self.start_recording)

################ GUI Functions
    def start_recording(self):
        global radar_gui
        # Toggle the is_recording flag
        self.is_recording = not self.is_recording

        # Update button text based on the recording state
        if self.is_recording:
            self.start_recording_button.setText("Stop Detecting")
            # control_com_on()
            # radar_gui.update_fall_indicator(True)  # Indicator turns red
            print("Fall Detection started.")
        else:
            self.start_recording_button.setText("Start Detecting")
            print("Fall Detection stopped.")
            # if not radar_gui.reset_timer.isActive():
            #     radar_gui.update_fall_indicator(False)
            # control_com_off()

    def create_occupancy_grid(self, cube_width, cube_height, cube_depth,
grid_width, grid_height, spacing, cube_y_translation):
        # Calculate the center of the cube in the x and y dimensions
        cube_center_x = 0
        cube_center_y = 2.5

        # The z_position of the grid is the bottom of the cube
        z_position = cube_y_translation - (cube_height / 2)

        grid_color = (0.5, 0.5, 0.5, 1)  # Light grey color for the grid lines
        lines = []

        # Starting point of the grid in the x and y dimensions
        grid_start_x = cube_center_x - (grid_width / 2)
        grid_start_y = cube_center_y - (grid_height / 2)

        # Horizontal lines (along the X-axis, varying Y)
```

```
        for y in np.arange(grid_start_y, grid_start_y + grid_height + spacing,
spacing):
            start_vert = np.array([grid_start_x, y, z_position],
dtype=np.float32)
            end_vert = np.array([grid_start_x + grid_width, y, z_position],
dtype=np.float32)
            lines.append([start_vert, end_vert])

        # Vertical lines (along the Y-axis, varying X)
        for x in np.arange(grid_start_x, grid_start_x + grid_width + spacing,
spacing):
            start_vert = np.array([x, grid_start_y, z_position],
dtype=np.float32)
            end_vert = np.array([x, grid_start_y + grid_height, z_position],
dtype=np.float32)
            lines.append([start_vert, end_vert])

        # Create line plot items for each line in the grid
        for line_data in lines:
            line_item = gl.GLLinePlotItem(pos=np.array(line_data),
color=grid_color, width=1, antialias=True)
            self.scatter_widget.addItem(line_item)

    def update_scatter_plot(self, points, color=(0, 0, 1, 1), size = 1):

        self.scatter.setData(pos=points, color=color)

    def create_cube(self, width, height, depth, y_translation=0):
        # Define vertices with an added translation along the y-axis
        verts = np.array([
            [width / 2, height / 2 + y_translation, depth / 2],
            [width / 2, -height / 2 + y_translation, depth / 2],
            [-width / 2, -height / 2 + y_translation, depth / 2],
            [-width / 2, height / 2 + y_translation, depth / 2],
            [width / 2, height / 2 + y_translation, -depth / 2],
            [width / 2, -height / 2 + y_translation, -depth / 2],
            [-width / 2, -height / 2 + y_translation, -depth / 2],
            [-width / 2, height / 2 + y_translation, -depth / 2]
        ])

        # Define the edges of the cube, only outer edges, no diagonals
        edges = np.array([
            [0, 1], [1, 2], [2, 3], [3, 0], # Bottom
            [4, 5], [5, 6], [6, 7], [7, 4], # Top
            [0, 4], [1, 5], [2, 6], [3, 7], # Sides
```

```
        ])

        # Create an empty list to store the line items
        cube_lines = []

        # Create a line plot item for each edge
        for edge in edges:
            start_vert = verts[edge[0]]
            end_vert = verts[edge[1]]
            line_data = np.array([start_vert, end_vert], dtype=np.float32)
            line_item = gl.GLLinePlotItem(pos=line_data, color=(1, 0, 0, 1),
width=2, antialias=True)
            cube_lines.append(line_item)

        return cube_lines
    def update_fall_indicator(self, fall_detected):
        self.fall_indicator.setText("FALL DETECTED" if fall_detected else
"Monitoring...")
        self.fall_indicator.setStyleSheet("QLabel { background-color: %s; font-
size: 18pt; }" % ('red' if fall_detected else 'green'))
        # Start/reset the timer when fall is detected
        # if fall_detected:
            # self.reset_timer.start(1000)

    def reset_fall_indicator(self):
        self.fall_indicator.setText("Monitoring...")
        self.fall_indicator.setStyleSheet("QLabel { background-color: green;
font-size: 18pt; }")



#------------------- UPDATE AND FALL DETECTION  ---------------------------------
-----------

alarm_trigger = False
window = 20
current_window_idx = 0
all_data_frame = []
fall_df = pd.DataFrame(columns = ['detected_falls_idx'])

# def control_com_on():
#     subprocess.run(["pwsh", "/home/boliclab/Desktop/lightON.ps1"])

# def control_com_off():
#     subprocess.run(["pwsh", "/home/boliclab/Desktop/lightOFF.ps1"])
```

```
def update():
    global s, recent_predictions, radar_gui, alarm_trigger, window,
current_window_idx, all_data_frame, fall_df
    dataOk, frameNumber, detObj = readAndParseData14xx(Dataport,
configParameters)

    if radar_gui.is_recording:

        if dataOk:
            # Convert detObj to DataFrame
            df = pd.DataFrame({
                'Frame Number': frameNumber,
                'X': -np.array(detObj["x"]),
                'Y': np.array(detObj["y"]),
                'Z': np.array(detObj["z"]),
                'velocity': np.array(detObj["doppler"]),
            })
            points = np.vstack((df['X'], df['Y'], df['Z'])).T
            radar_gui.scatter.setData(pos=points)

            if df.empty:
                pass
            else:
                current_window_idx +=1
                # print (df)
                all_data_frame.append(df)
            # Check every 20 frames = 1s
            print ('current_window_idx', current_window_idx)
            # os.system("Pause")
            if current_window_idx == 100:
                #Combine data frames together
                # print ('len(all_data_frame)', len(all_data_frame))
                # os.system("Pause")
                for i in range (len(all_data_frame)):
                    if i == 0:
                        current_data_frame = all_data_frame[0]
                    else:
                        current_data_frame = pd.concat([current_data_frame,
all_data_frame[i]], ignore_index= True)

                anomaly_data, centroidZ_his_anomaly =
data_processor.load_csv(current_data_frame, anomaly=True)
                centroidZ_his_anomaly_np = np.array(centroidZ_his_anomaly)
                # Sampling frequency
```

```
                fs = 20  # 20 frames/second
                # Cutoff frequency (adjust based on your needs)
                cutoff = 4  # 5 Hz

                # Design Butterworth low-pass filter
                order = 5  # Filter order (adjust based on your needs)
                nyquist = 0.5 * fs
                normal_cutoff = cutoff / nyquist
                b, a = butter(order, normal_cutoff, btype='low', analog=False)

                # Apply the filter
                filtered_data = filtfilt(b, a, centroidZ_his_anomaly_np)




                model = autoencoder_mdl(model_dir='D:/1443Boost/')

                HVRAE_loss_history   = model.HVRAE_predict(anomaly_data)

                flattened_loss_history = np.array([float(item) for sublist in
HVRAE_loss_history for item in np.atleast_1d(sublist)])

                # Interpolate HVRAE_loss_history to match the length of
centroidZ_his_anomaly
                time_steps_original = np.linspace(0, len(flattened_loss_history)-
1, len(flattened_loss_history))
                time_steps_target = np.linspace(0, len(flattened_loss_history)-1,
len(filtered_data))

                interpolated_loss_history = np.interp(time_steps_target,
time_steps_original, flattened_loss_history)

                calculator = compute_metric()

                threshold = 0.3  #23 #0.3
                detected_falls_idx, _ =
calculator.detect_falls(interpolated_loss_history, filtered_data, threshold)
                print(detected_falls_idx)

                fall_df['detected_falls_idx'] = detected_falls_idx
                falls_idx_csv_file_exist =
os.path.isfile('detected_falls_idx.csv')
                fall_df.to_csv('detected_falls_idx.csv', mode='a', header=not
falls_idx_csv_file_exist, index=False)
```

```
                fall_df = pd.DataFrame()



                # Set alarm trigger
                if len(detected_falls_idx) != 0:
                    alarm_trigger = True
                else:
                    alarm_trigger = False

                current_window_idx = 0
                all_data_frame = []
                detected_falls_idx = []


            if alarm_trigger == True:
                print("FALL DETECTED")
                radar_gui.update_fall_indicator(True)  # Indicator turns red

            else:
                if not radar_gui.reset_timer.isActive():
                    radar_gui.update_fall_indicator(False)


            # Write to CSV
            df.to_csv(csv_file_path, mode='a', index=False, header=not
pd.io.common.file_exists(csv_file_path))

            points = np.vstack((df['X'], df['Y'], df['Z'])).T

            radar_gui.update_scatter_plot(points, size = 2)

            QtGui.QGuiApplication.processEvents()
    else:
        pass


# Set up the serial connection and radar configuration parameters
CLIport, Dataport = serialConfig(configFileName)
configParameters = parseConfigFile(configFileName)

# Initialize the Qt Application and RadarGUI
app = QtWidgets.QApplication([])
radar_gui = RadarGUI()
radar_gui.setWindowTitle('3D Radar Scatter Plot')
```

```
radar_gui.show()

# Connect the update function to a timer for periodic updates
timer = QtCore.QTimer()
timer.timeout.connect(update)
timer.start(33)  # Update every 33 milliseconds

# Start the Qt event loop
sys.exit(app.exec_())
```

References

Jin, F., Sengupta, A., & Cao, S. (2022). mmFall: Fall Detection Using 4-D mmWave Radar and a Hybrid Variational RNN AutoEncoder. IEEE Transactions on Automation Science and Engineering, 19(2), 1245.

Rezaei, A., Mascheroni, A., Stevens, M. C., Argha, R., Papandrea, M., Puiatti, A., & Lovell, N. H. (2023). Unobtrusive Human Fall Detection System Using mmWave Radar and Data Driven Methods. IEEE Sensors Journal, 23(7), 7968.

Alhazmi, A. K., Alanazi, M. A., Alshehry, A. H., Alshahry, S. M., Jaszek, J., Djukic, C., ... & Chodavarapu, V. P. (2024). Intelligent Millimeter-Wave System for Human Activity Monitoring for Telemedicine. Sensors, 24(1), 268. https://doi.org/10.3390/s24010268

Qi, C. R., Su, H., Mo, K., & Guibas, L. J. (Year). PointNet: Deep learning on point sets for 3D classification and segmentation. GitHub. Retrieved from https://github.com/charlesq34/pointnet

Jin, F.; Zhang, R.; Sengupta, A.; Cao, S.; Hariri, S.; Agarwal, N.K.; Agarwal, S.K. Multiple patients behavior detection in real-time using mmWave radar and deep CNNs. In Proceedings of the 2019 IEEE Radar Conference (RadarConf), Boston, MA, USA, 22–26 April 2019; pp. 1–6

Sun, Y.; Hang, R.; Li, Z.; Jin, M.; Xu, K. Privacy-preserving fall detection with deep learning on mmWave radar signal. In Proceedings of the 2019 IEEE Visual Communications and Image Processing (VCIP), Sydney, NSW, Australia, 1–4 December 2019; pp. 1–4.

Bosch Sensortec. (n.d.). BMP581. Retrieved February 11, 2024, from https://www.bosch-sensortec.com/products/environmental-sensors/pressure-sensors/bmp581/

Texas Instruments. (2024). IWR1443BOOST [Hardware]. Retrieved from https://www.ti.com/tool/IWR1443BOOST?keyMatch=&tisearch=search-everything&usecase=hardware

Texas Instruments. (2024). DCA1000EVM [Hardware]. Retrieved from https://www.ti.com/tool/DCA1000EVM?keyMatch=&tisearch=search-everything&usecase=hardware#order-start-development

Amazon. (n.d.). Spigen Certified Dynamic Holder for PlayStation [Product]. Retrieved from https://www.amazon.ca/Spigen-Certified-Dynamic-Compatible-Playstation/dp/B08PSF95M2/ref=asc_df_B08PSF95M2&mcid=f01f9979152937f985a73bdf49c27e78?tag=bingshopdesk-20&linkCode=df0&hvadid=80745476895979&hvnetw=o&hvqmt=e&hvbmt=be&hvdev=c&hvlocint=&hvlocphy=&hvtargid=pla-458345030713692&psc=1

PiShop. (2024). Raspberry Pi 5 8GB [Product]. Retrieved from https://www.pishop.ca/product/raspberry-pi-5-8gb/?src=raspberrypi

Best Buy. (2024). Best Buy Essentials 7.63m (25ft) Cat6 Ethernet Cable [Product]. Retrieved from https://www.bestbuy.ca/en-ca/product/best-buy-essentials-7-63m-25ft-cat6-ethernet-cable-be-pec6st25-c/15101526

Grainger Canada. (2024). GGE1EJH4 [Product]. Retrieved from https://www.grainger.ca/en/product/p/GGE1EJH4?s_kwcid=AL!3645!10!76622405578387!76622297852485&gucid=N:N:PS:Paid:MS:CSM-8512:NSEMX7:20501231:APZ_1&gclid=0c05ee7d71f51687b44d8bb6fa947f83&gclsrc=3p.ds&msclkid=0c05ee7d71f51687b44d8bb6fa947f83