香港城市大學
City University of Hong Kong

Department of Computer Science

BSCCS Final Year Project Report
2024-2025

24CS125

Motion detection via mmWave radars

(Volume ___ of ___)

Student Name : **CHOWDHURY Zuhayr Rahman**

Student No. : **56885230**

Programme Code : **BSCEGU4**

For Official Use Only

Supervisor : **Prof YIN, Zhimeng**

1st Reader : **Prof QIU, Junqiao**

2nd Reader : **Prof HANCKE, Gerhard Petrus**

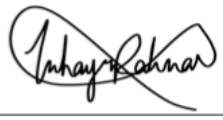
Student Final Year Project Declaration

I have read the project guidelines and I understand the meaning of academic dishonesty, in particular plagiarism and collusion. I hereby declare that the work I submitted for my final year project, entitled:

Motion detection via mmWave radars

does not involve academic dishonesty. I give permission for my final year project work to be electronically scanned and if found to involve academic dishonesty, I am aware of the consequences as stated in the Project Guidelines.

Student Name: CHOWDHURY Zuhayr Rahman

Signature: 

Student ID: 56885230

Date: March 31, 2025

Table of Contents

Abstract

This project focuses on motion detection using mmWave radar technology. Utilizing datasets available online and those acquired using the mmWave radar, the data is first processed and converted into point cloud data to make suitable for use in this project, after which three variants of the Recurrent Neural Network (RNN) Model - Traditional or Vanilla RNN, Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) Classification Models are trained and tested, and then the LSTM, which is the best-performing model of the three, is used to detect and classify motion. By leveraging the abilities of the LSTM Model in training, by learning from complex features in point cloud data, this project aims to achieve maximum accuracy and reliability in motion detection performance.

The report begins with a brief overview of mmWave radars and their advantages over other technologies used for motion detection. Following this, detailed steps regarding the methodology implemented in data collection and processing, and model and training and testing are discussed. Various RNN Models will be evaluated extensively, with their performance and accuracy compared, along with the strengths and weaknesses of each model, thus providing an extensive insight into motion detection using mmWave radar technology.

Summary of Revisions and Additions

Initially, a Random Forest (RF) Model was considered the most suitable for the project due to its robustness, simplicity and ease of implementation. However, after further research and discussions, it was determined that the RNN Model would be more appropriate for this project due to its ability to learn from sequential data.

The datasets have been processed, with all implemented steps being discussed in detail, including data collection, dataset preprocessing, and DataFrame creation. Furthermore, initial training and testing results have been evaluated, offering valuable insights into the development and performance of the model. These findings will greatly assist in future developments, which have also been discussed.

mmWave Demo Visualizer has been used instead of mmWave Studio because it provides data (.dat) files containing point cloud data, which were more suitable for me to convert to comma-separated values (csv) files and for training the RNN models for this project.

Significant modifications have been made to data collection and processing, including capturing six distinct types of motion and no-motion, updating the code to handle each type of motion, and assigning labels to entire point clouds rather than individual points.

Extensive comparison and analysis has been conducted for each RNN variant, as well as for the parameters and hyperparameters used for training the final LSTM Model to ensure that the optimal set was selected.

Acknowledgements

I would like to express my heartfelt gratitude to my family - my parents, siblings, and grandparents - for their unconditional support.

I extend my sincere thanks to my supervisor, Dr. Yin, for his feedback, support, guidance, and motivation, as well as Mr. Danei Gong, for his assistance throughout the project, especially in critical areas that I could not have resolved on my own.

Additionally, I am grateful to the Department of Computer Science at the City University of Hong Kong for providing essential resources, such as the mmWave Radar Sensor, which played a crucial role in my project.

Finally, I want to thank everyone who has supported me during my time at university.

1. Introduction

1.1 Background Information

In the modern world, motion detection has various applications in areas such as security, automation, and healthcare (Ameen et al., 2010). Motion sensors are devices that can detect motion with the help of technology such as computer vision (Aydin & Othman, 2017) and infrared sensors. They use complex algorithms and machine learning techniques to accurately detect motion, as studied by Yong et al. (2011) and Kavuncuoğlu et al. (2021) respectively. These advanced sensors are paired with security cameras, smart lights and other sensors to create surveillance systems (Song et al., 2008) and smart home systems (Fanti et al., 2017), and have a variety of other applications.

1.2 Problem Statement

The most common type of motion sensor is Passive Infrared (PIR) Sensors (Aydin & Othman, 2017). As the name suggests, and Mukhopadhyay et al. (2018) discuss, they detect motion by analyzing wavelengths of infrared radiation that pass its field of view. Although affordable (Frankiewicz & Cupek, 2013), Andrews et al. (2020) argue that these sensors struggle to detect stationary objects. They also have accuracy and reliability limitations (Naaraju et al., 2024). Furthermore, these sensors have a very limited detection range of only a few metres (Liddiard, 2007), and are prone to false alarms due to environmental changes such as temperature fluctuations (Hong et al., 2013), or wildlife (Damarla & Mehmood, 2019). So, despite the technological

advancements made in motion detection systems, there are still some obvious shortcomings.

1.3 Motivation

There is a need for motion sensors that can operate effectively and provide the same quality output irrespective of the environmental conditions. mmWave radars can detect signals accurately, regardless of lighting or weather conditions (Venon et al., 2022). This project seeks to use mmWave radars to develop a motion detection system that can provide real-time, accurate motion detection in all environments. mmWave radars have the potential to be crucial for future improvements in motion detection technology.

1.4 Objectives and Scope

The project aims to develop a motion detection system using mmWave radars. Section 2 provides a literature review of the existing technologies, as well as the new technology that will be implemented in this project. Section 3 outlines how the system will be developed, by selecting and using the appropriate hardware components, developing data algorithms, and combining them to create a well-integrated and fully functional system. The system will be evaluated on its accuracy, performance across different environments and false positive rate. Additionally, the project will produce a final report, presentation and demonstration for comprehensive documentation of the motion detection system's design, implementation and evaluation. Sections 3, 4, 5, 6 and 7 contain the initial design of the system, results and evaluation, conclusion, the tentative project schedule and monthly logs respectively.

2. Literature Review

2.1 Existing Technologies

This section offers an in-depth analysis of the technology that is currently used, i.e., Passive Infra-Red (PIR) sensors, the limitations of this technology, and the relevance of the new technologies implemented, i.e., mmWave radars, highlighting the project's relevance.

2.1.1 Passive Infra-Red (PIR) Sensors

Passive Infra-Red (PIR) motion sensors are the most common devices that are used for motion detection. They operate by sensing infrared radiation emitted by objects and have a simple and straightforward mechanism. PIR sensors consist of two or four pyroelectric elements connected in parallel. When an object moves, one of these elements detects more infrared radiation than another due to a change in infrared radiation, the PIR sensor is triggered. When there is no movement, all the elements detect the same amount of infrared radiation and the sensor is not triggered (Andrews et al., 2020). These sensors are affordable, power efficient, and extremely versatile as they can be used in various applications such as energy management in smart homes and buildings, and security systems, as discussed by Sharanbasappa et al. (2023) and Likhitha et al. (2019) respectively.

However, PIR sensors have a few limitations. Since these sensors use infrared radiation, they have a very limited range of 8 to 12 microns (Doctor, 1994). Furthermore, since these sensors are dependent on infrared radiation, temperature fluctuations in the environment can cause disruptions and lead to false positives, indicating motion when

there is none (Choubisa et al., 2016). Narayana et al. (2015) also state that even minimal disruptions such as leaves moving due to wind, or gusts of wind can trigger false alarms. So, while PIR sensors are popular due to their affordability, efficiency and versatility, they struggle to work effectively in adverse conditions due to the inherent range and environment-related limitations of infrared radiation. Addressing these shortcomings of PIR sensors is crucial for the advancement of motion detection systems. Therefore, there is a need for alternative sensors that do not have these limitations and can overcome these challenges.

2.2 mmWave Radar Motion Detection

2.2.1 Sensor Technology

Soumya et al. (2023) state that mmWave radars emit radio waves which reflect off objects, and Chen et al. (2023) highlight their ability to create radar point clouds. Soumya et al. (2023) further explain the radars' composition and mechanism as follows: each mmWave radar consists of two antennae, one for transmitting signals and one for receiving the reflected signals, and a signal processing system. An intermediate-frequency (IF) signal is obtained by combining the reflected signal with a transmitting signal, which is then processed to obtain information about the detected object.

By analyzing the reflected waves, the radar can determine whether an object is moving through the measurement of Doppler Frequency Shifts (DFS) (Airat et al., 2022). So, the mechanism of mmWave radars is much more complex and advanced than that of

PIR sensors. Although they are both used to detect motion, they operate in fundamentally different ways, with the former emitting mmWave radiation and then analyzing the reflected waves and the latter analyzing the infrared radiation emitted by objects in its field of view.

2.2.2 Advantages of mmWave Radars

mmWave radars offer significant advantages over traditional PIR sensors. mmWave radar-based sensing is ideal for motion detection, as its performance and accuracy are not affected by environmental factors such as rain, fog and snow (Chen et al., 2023), and even extremely bright or low light (Soumya et al., 2023). Jardak et al. (2019) highlight that mmWave sensors possess higher range accuracy due to their shorter wavelengths, which allow for smaller antenna footprints. The sensors provide micron-level accuracy in distance measurements, as observed by Piotrowsky et al. (2022), which is significantly higher than what is typically achievable with IR sensors. Advancements in radar technology and digital signal processing have also helped mmWave radars achieve higher accuracy and resolution than their counterparts (Soumya et al., 2023). Thus, mmWave radars are an ideal alternative for PIR sensors because they are not restrained by the limitations of PIR sensors and can work in diverse environmental conditions, while also providing superior range precision. These advantages make mmWave radars much more useful and reliable than their PIR counterparts for modern applications in motion detection.

3. Initial Design

3.1 System Design and Setup

3.1.1 Technology Overview

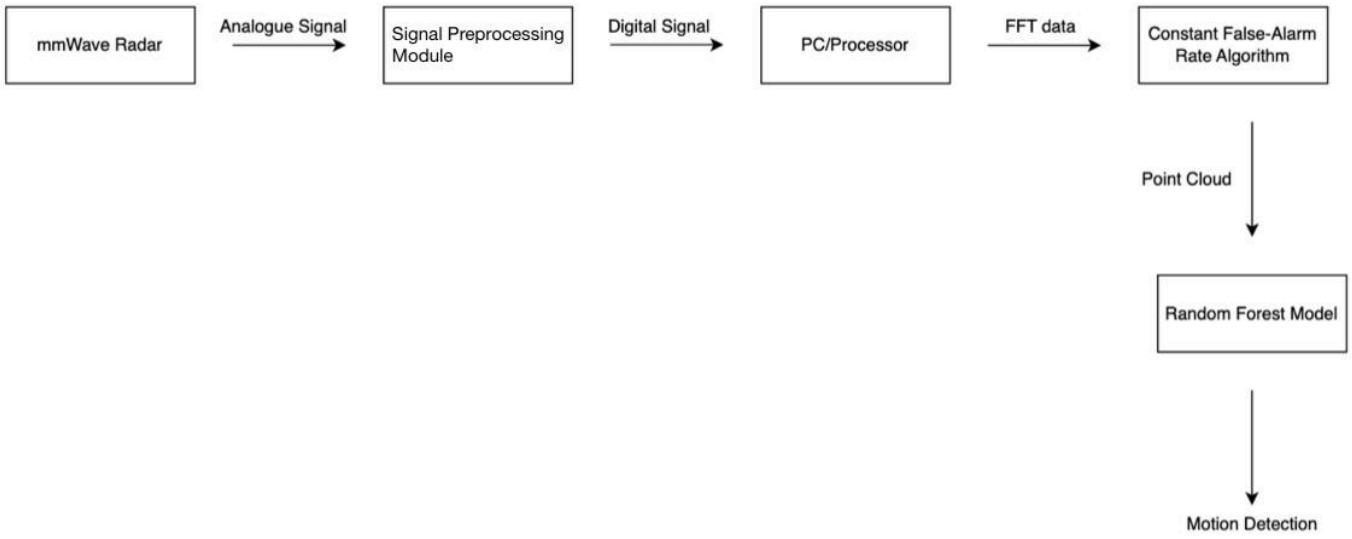


Figure 3.1.1: mmWave Motion Sensor System Flow Chart

The diagram provides a high-level overview of the mechanism of the proposed system. The mmWave radar transmits electromagnetic signals which bounce off the observed subject and return to the radar in the form of reflected waves. Both the transmitted and reflected waves are processed by the signal preprocessing module, converted to digital signals, and sent to the PC for further analysis and data post-processing.

The PC, which is both the receiver of the digital signals and the Digital Signal Processing (DSP) unit, uses the mmWave Demo Visualizer software to process the received digital signal. The PC performs a 3D Fast Fourier Transform (FFT) on the digital signals, converting time domain signals into the frequency domain. This process results in a 3-dimensional FFT comprising the range FFT, Doppler FFT and angle FFT (Abdu et al., 2021).

The Constant False-Alarm Rate (CFAR) Algorithm is then used to extract point clouds, which are then used as input for the Recurrent Neural Network (RNN) Model.

3.2 System and Software Components

This section talks about the components that will be used for data collection and processing, as well as their mechanisms. The information is based on Iovescu & Rao (2017), and Rao (2017).

3.2.1 mmWave Radar

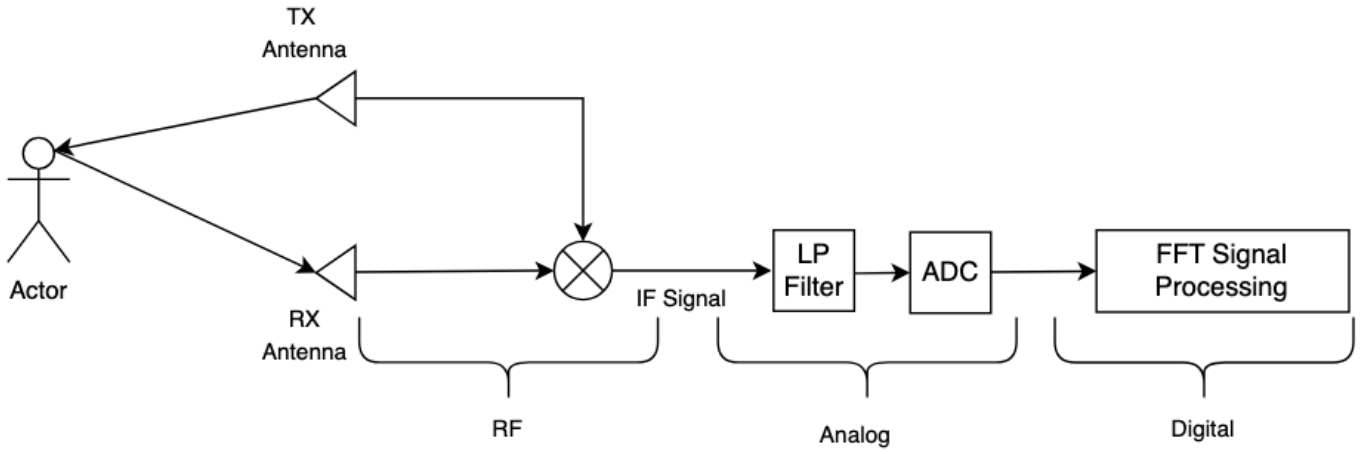


Figure 3.2.1a: mmWave Radar Mechanism

The figure illustrates the RF, analogue and digital components of a mmWave sensor. A mmWave radar transmits a signal known as a “chirp”, whose frequency increases linearly with time. The synthesizer (Synth) generates a chirp and the Transmitting (TX) Antenna transmits it. The chirp is reflected off an object back to the radar and is captured by the Receiving (RX) Antenna. The Mixer combines the TX and RX chirps,

creating an Intermediate Frequency (IF) signal, which is then transmitted to the PC, which then executes data post-processing.

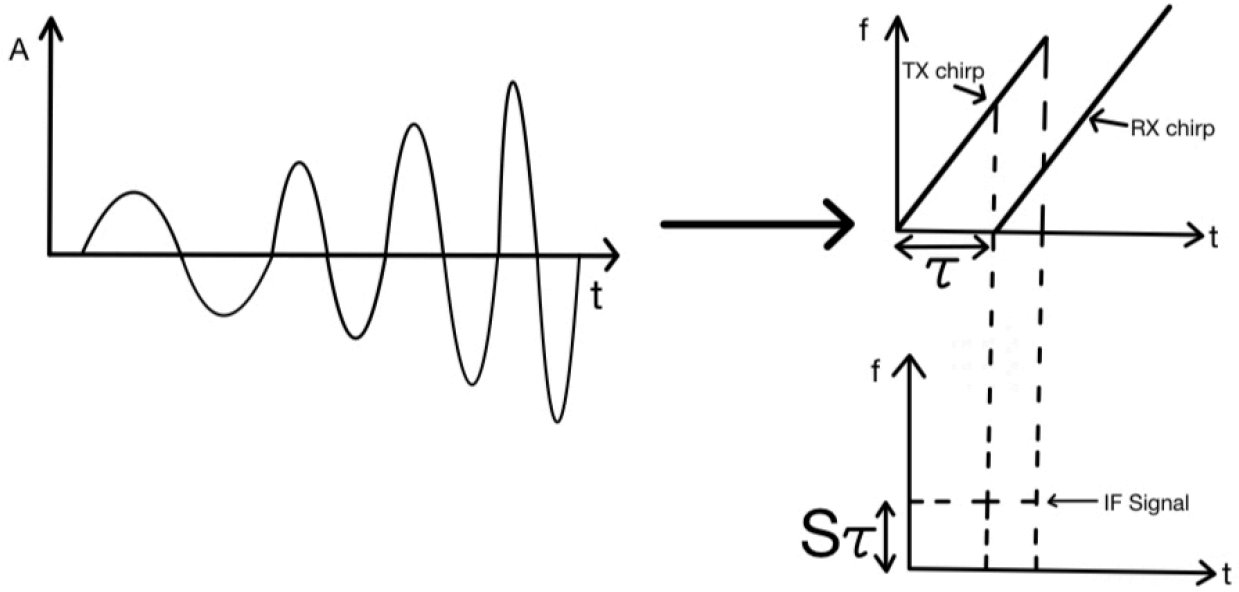


Figure 3.2.1b: Combination of TX and RX Chirps to create an IF Signal

This project will use Texas Instruments' AWR6843ISK mmWave sensor evaluation kit, which consists of a single chip compatible with 60-64 GHz frequencies, four receiving (RX) and three transmitting (TX) antennas. It uses its 60-pin high-speed connector to connect to the preprocessing module.

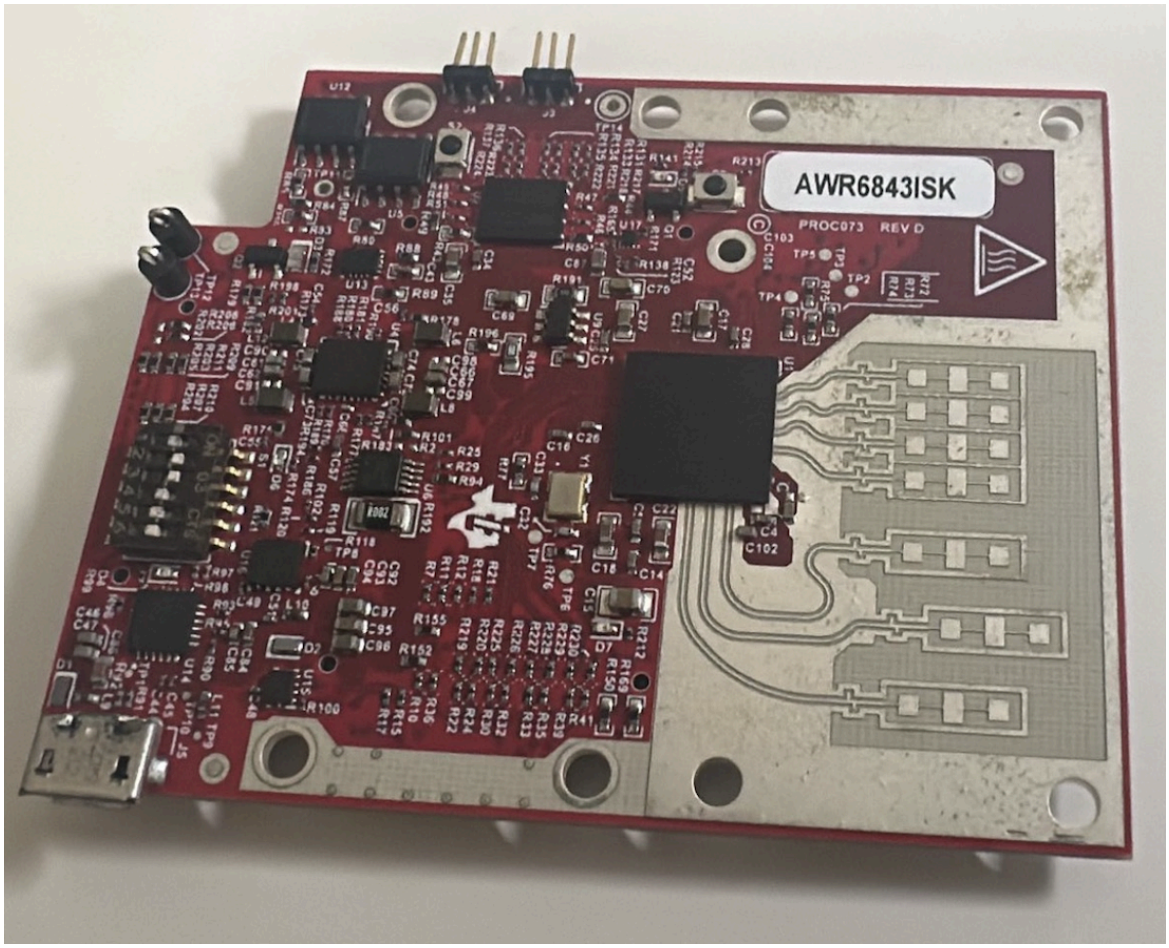


Figure 3.2.1c: AWR6843ISK mmWave Sensor Evaluation Kit

3.2.2 PC/Processor

The digital signals will be processed using an HP ENVY x360 Windows laptop with an AMD Ryzen™ 7 2700U Microprocessor with 4 cores, and 8 GB DDR4-2400 SDRAM. The setup is inspired by Maiwald et al. (2024).

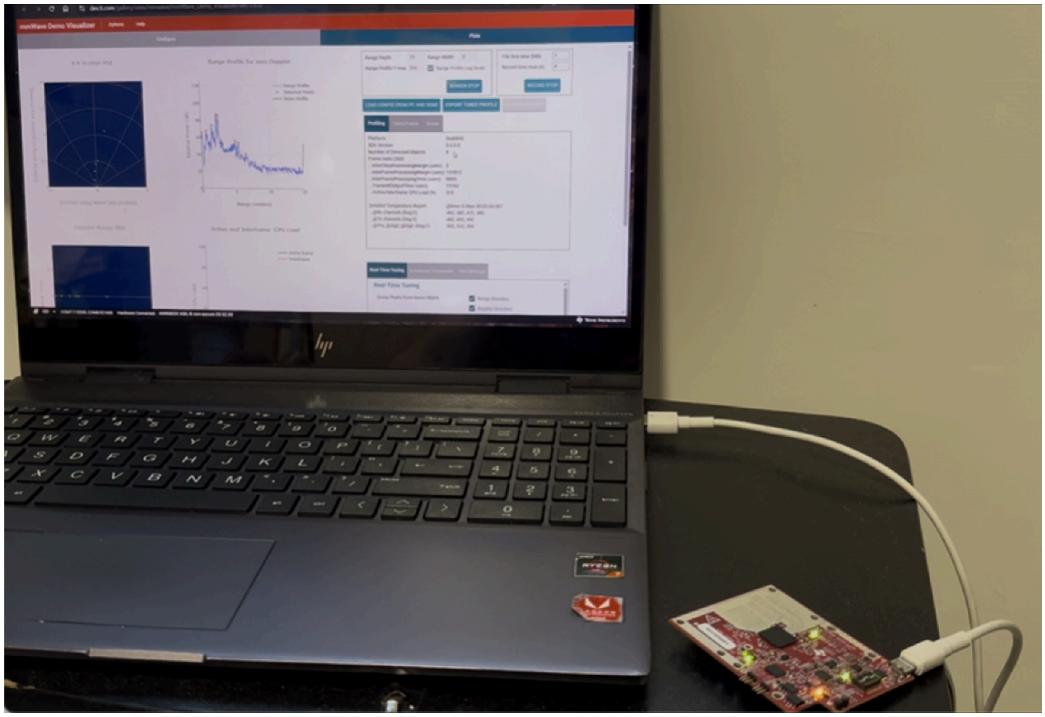


Figure 3.2.2: Laptop Connected to mmWave Radar System

3.2.3 Experimental Setup

Taking inspiration from Palipana et al. (2021), the setup is illustrated in the figure below. The radar will be positioned to capture the various motions that will be performed in front of it. After each motion is executed, I will verify its successful recording by visualizing the point cloud on a laptop. If the recording fails, the procedure will be repeated.

Figure 3.2.3a: Data Capture In Progress (Walking)

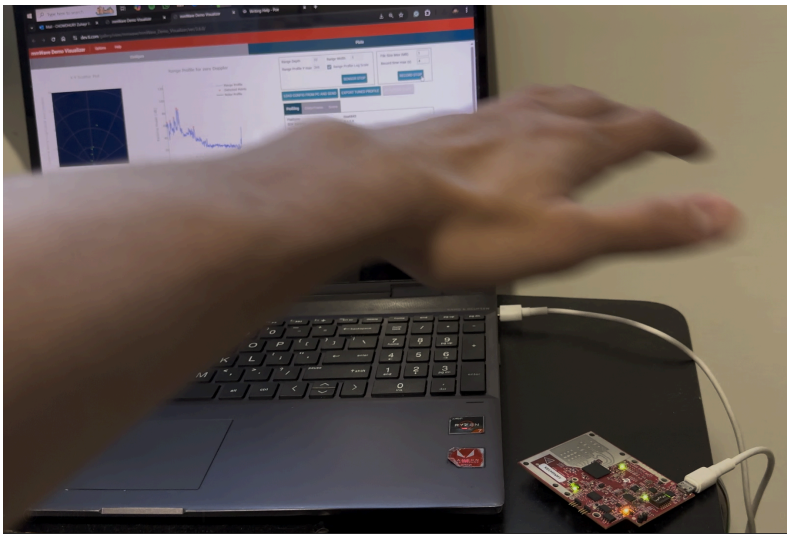


Figure 3.2.3b: Data Capture In Progress (Hand Waving)

3.2.4 3D Fast Fourier Transform

The PC executes a Fourier transform to produce a 3D FFT consisting of three components: range, velocity and angle of arrival.

Fourier transform converts the IF signal from its representation in the time domain to one in the frequency domain. Each sinusoid in the time domain corresponds to a single transmission chirp and receiving chirp. Fourier transform converts this to the frequency domain, and a Range-FFT is produced in which each detected object is represented by a single peak.

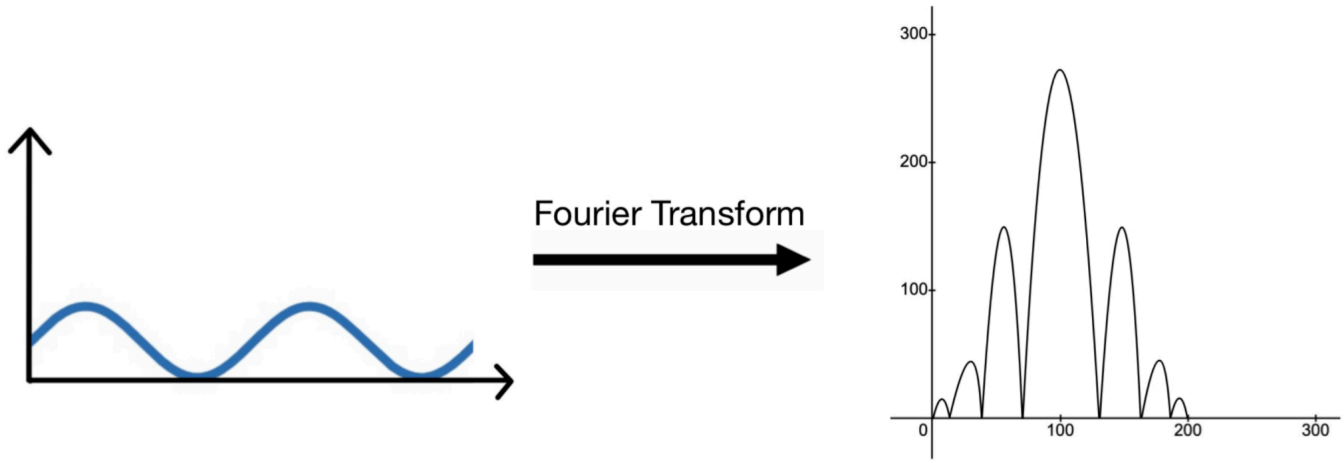


Figure 3.2.4a: Fourier Transform on IF Signal

By analyzing the Range-FFT output, the distance between the object and the radar can be calculated using the equation $f_{IF} = \frac{S2d}{c}$, where f_{IF} is the frequency of the IF signal, S is the slope of the IF signal on the frequency-time graph, d is the distance between the object and the radar and c is the speed of light.

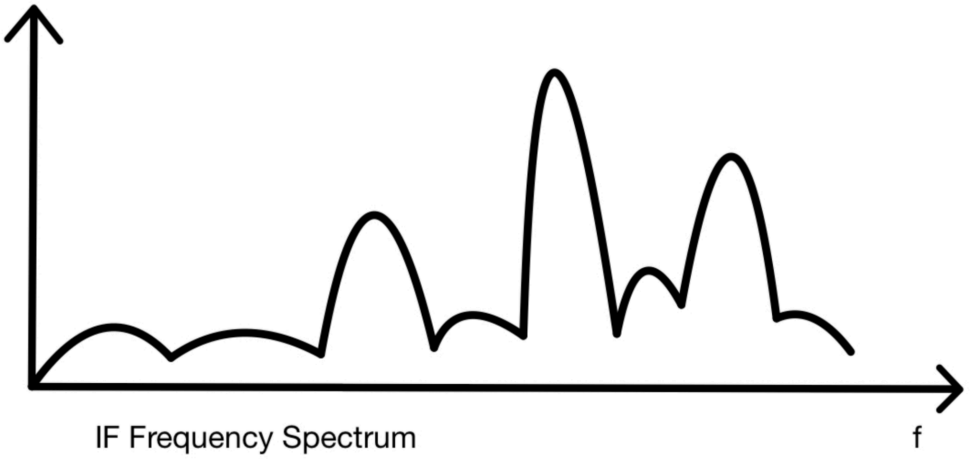


Figure 3.2.4b: Range-FFT Showing 3 Peaks for 3 Detected Objects

If there are multiple objects with the same range from the radar but with different velocities, the Range-FFT will show only one peak, corresponding to the distance.

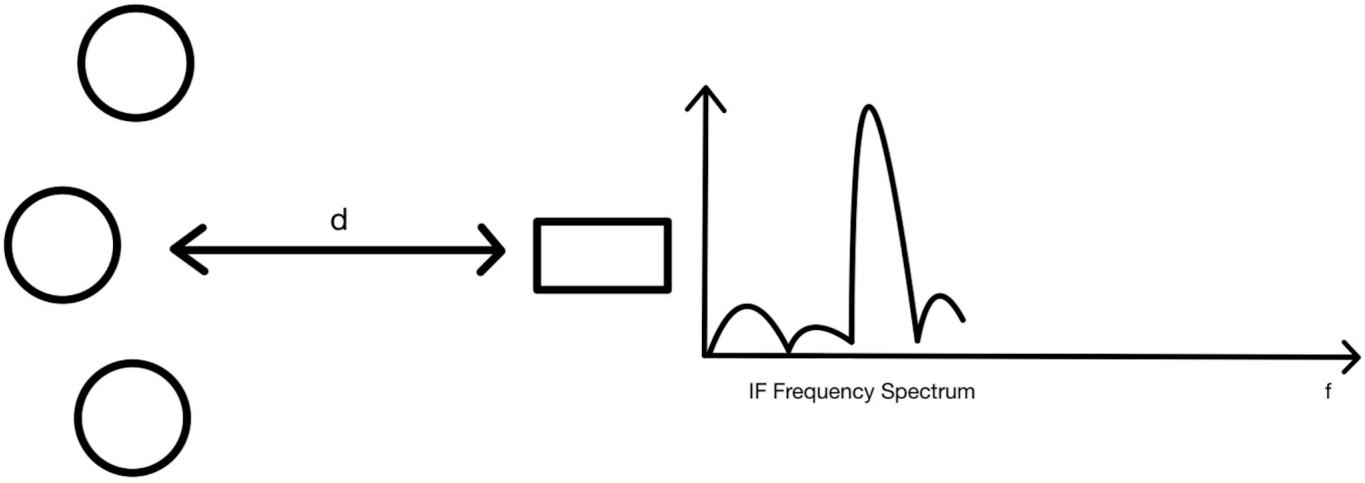


Figure 3.2.4c: Range-FFT Showing One Peak for Multiple Objects

To measure the velocity of an object, two chirps are transmitted, and the corresponding Range-FFTs will have peaks in the same location but with different phases. The phase difference can be used to estimate the object's velocity, by using a Doppler-FFT. By analyzing the N-phase differences from multiple detected objects, the Doppler-FFT can produce distinct peaks that correspond to the different velocities of the objects, therefore distinguishing between multiple objects that have the same range from the radar. The combination of the Range-FFT and Doppler-FFT creates a 2D FFT of the detected objects.

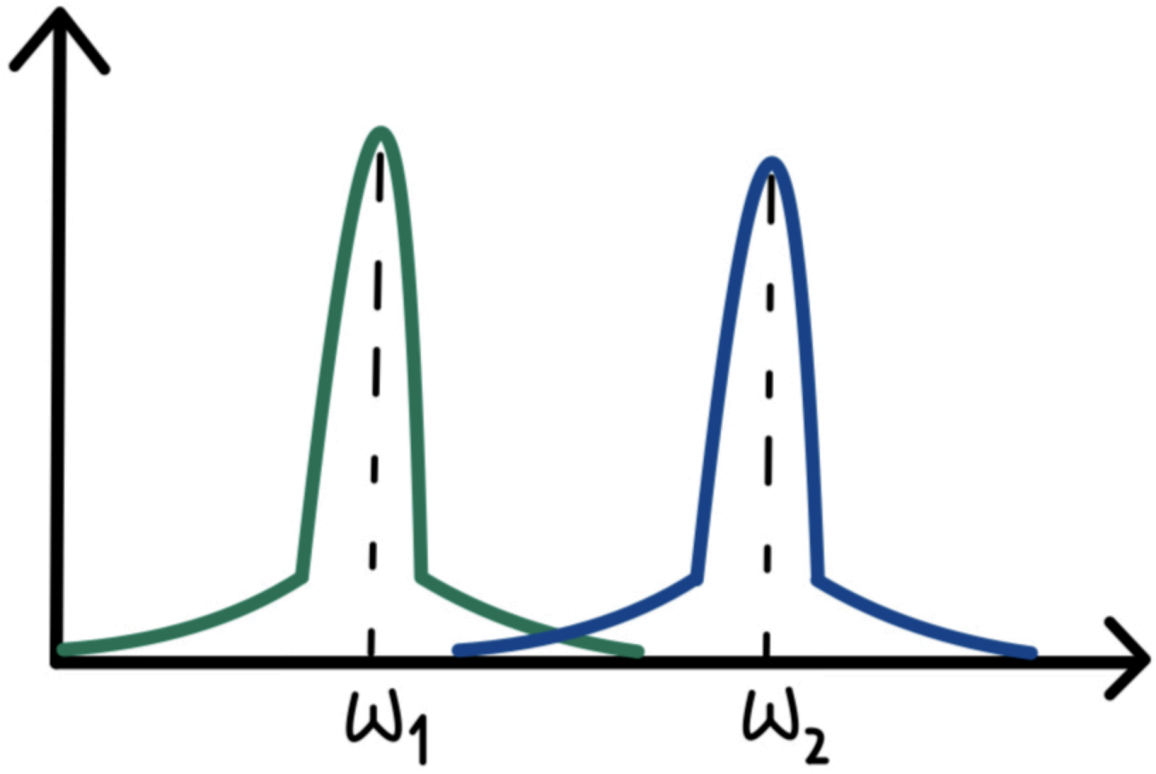


Figure 3.2.4d: Two Peaks on the Doppler-FFT Indicating the Presence of Two Detected Objects at Different Velocities

If there are multiple objects with the same range and velocity but at different angles to the radar, the 2D-FFT will have peaks in the same location. In this case, the angle of arrival needs to be determined. If two RX antennas are used, they will have peaks in the same location but with different phases. The phase difference helps estimate the angle of arrival of the detected object using the following equation:

$$\omega = \frac{2\pi d sin(\theta)}{\lambda} \Rightarrow \theta = sin^{-1}(\frac{\lambda\omega}{2\pi d})$$

where λ is the wavelength of the IF signal, d is the distance between the object and the radar,c ω is the measured phase difference and θ is the angle of arrival.

Using this mechanism for N antennas and multiple objects, Angle-FFT can be executed, which consists of performing an additional FFT on the phase differences corresponding to the 2D-FFT peaks, thus resolving between multiple objects with the same range and velocity but at different angles to the radar.

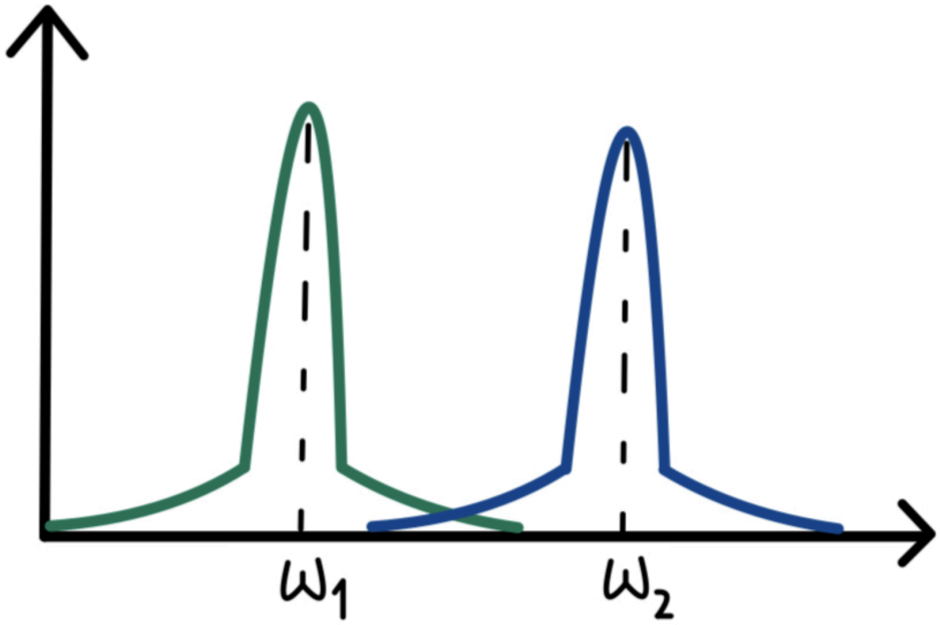


Figure 3.2.4e: Two Peaks on the Angle-FFT Indicating the Presence of Two Detected Objects at Different Angles of Arrival

3.2.5 Constant False-Alarm Rate (CFAR) Algorithm

The Constant False-Alarm (CFAR) Algorithm is used to extract point clouds from the FFT data. It uses a dynamic algorithm that modifies detection thresholds based on the ambient noise level, which helps distinguish the detected objects from complex background noise and enables stable detection performance by maintaining a constant false alarm rate despite changing signal strengths or noise levels (Wei et al., 2023; Zande, I., 2023).

3.2.6 mmWave Demo Visualizer & UniFlash

Both softwares were installed from the Texas Instruments website, following the Texas Instruments mmWave radar User Guide. mmWave Demo Visualizer is required for operating the AWR6843ISK mmWave sensor evaluation kit, and UniFlash is needed for operations such as flashing and factory resetting the aforementioned mmWave sensor.

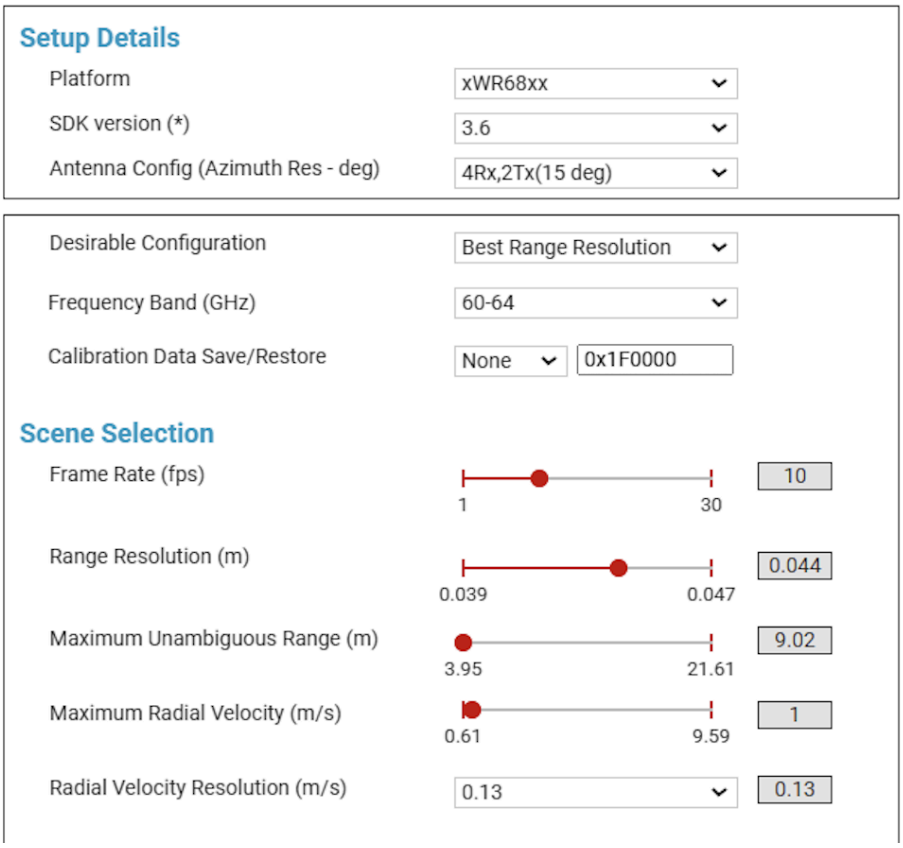


Figure 3.2.6: mmWave Demo Visualizer Settings

The settings were configured for the sensor I used for the project, the TI AWR6843ISK. A frame rate of 10fps was chosen to capture around 40 frames in total, during a 4 second recording period for each instance of recording.

3.3 Point Cloud Data

Raw data obtained from the radar will be in the form of data or .dat files, which will then be converted to .csv files suitable for training the RNN Model using the following MATLAB code, which was provided by Texas Instruments (Texas Instruments, n.d.):

```
import os
import sys
# import the parser function
from parser_mmw_demo import parser_one_mmw_demo_output_packet


if (len(sys.argv) > 1):
    capturedFileName=sys.argv[1]
else:
    print ("Error: provide file name of the saved stream from Visualizer for OOB demo")
    exit()
# Read the entire file
fp = open(capturedFileName,'rb')
readNumBytes = os.path.getsize(capturedFileName)
print("readNumBytes: ", readNumBytes)
allBinData = fp.read()
print("allBinData: ", allBinData[0], allBinData[1], allBinData[2], allBinData[3])
fp.close()

# init local variables
totalBytesParsed = 0;
numFramesParsed = 0;

while (totalBytesParsed < readNumBytes):

    parser_result, \
    headerStartIndex,  \
    totalPacketNumBytes, \
    numDetObj,  \
    numTlv,  \
    subFrameNumber,  \
    detectedX_array,  \
    detectedY_array,  \
    detectedZ_array,  \
    detectedV_array,  \
    detectedRange_array,  \
    detectedAzimuth_array,  \
    detectedElevation_array,  \
    detectedSNR_array,  \
    detectedNoise_array = parser_one_mmw_demo_output_packet(allBinData[totalBytesParsed::1], readNumBytes-totalBytesParsed)

    # Check the parser result
    print ("Parser result: ", parser_result)
    if (parser_result == 0):
        totalBytesParsed += (headerStartIndex+totalPacketNumBytes)
        numFramesParsed+=1
        print("totalBytesParsed: ", totalBytesParsed)
        import csv
        if (numFramesParsed == 1):
            democsvfile = open('mmw_demo_output.csv', 'w', newline='')
            demoOutputWriter = csv.writer(democsvfile, delimiter=',',
                                          quotechar='', quoting=csv.QUOTE_NONE)
            demoOutputWriter.writerow(["frame","DetObj#","x","y","z","v","snr","noise"])

        for obj in range(numDetObj):
            demoOutputWriter.writerow([numFramesParsed-1, obj, detectedX_array[obj],\
                                       detectedY_array[obj],\
                                       detectedZ_array[obj],\
                                       detectedV_array[obj],\
                                       detectedSNR_array[obj],\
                                       detectedNoise_array[obj]])


    else:
        # error in parsing; exit the loop
        break

# All processing done; Exit
print("numFramesParsed: ", numFramesParsed)
```

Figure 3.3a: Texas Instruments Code For Converting .dat files into .csv files (Texas Instruments, n.d.)

The .csv files will contain various data, including:

Spatial Coordinates: The x,y and z coordinates of the objects in the three-dimensional space, which represent their position.

Velocity: The velocity indicates the speed or rate at which each point changes its position, enhancing the understanding of the dynamic scene.

Signal-to-Noise Ratio (SNR): SNR quantifies the strength of a signal relative to background noise, which is vital for precise analysis in radar-based motion detection systems.

Noise: The noise attribute indicates the level of unwanted or random signals, which is crucial for eliminating undesirable components during data processing.

Label: Labels will be assigned to every .csv file during data processing. Motion datasets will be marked with the following labels: 1 for bowing, 2 for waving, 3 for jumping, 4 for body movement and 5 for walking, while datasets indicating no motion will be labelled 0. This labelling will aid in training the RNN Model.

```
for file_id in df['source_file_id'].unique():
    if 0 <= file_id <= 69:
        label_mapping[file_id] = 1  # Bowing

    elif 70 <= file_id <= 278:
        label_mapping[file_id] = 2  # Waving

    elif 279 <= file_id <= 359:
        label_mapping[file_id] = 3  # Jumping

    elif 360 <= file_id <= 442:
        label_mapping[file_id] = 4  # Moving Body

    elif 443 <= file_id <= 630:
        label_mapping[file_id] = 5  # Walking

    elif 631 <= file_id <= 731:
        label_mapping[file_id] = 0  # No Motion
```

Figure 3.3b: Assigning Labels

3.4 Data Collection and Evaluation Planning

3.4.1 Dataset

An extensive and comprehensive dataset comprising of five motion activities, bowing, waving, jumping, body movement and walking, as well as no motion, will be developed with the help of a diverse range of motion, including a subject passing the field of view of the mmWave radar, initially remaining stationary within the radar's field of view and then performing lateral, forward and backward movements. The subject will move both slowly and quickly, and carry out all the aforementioned activities, as well as remaining stationary. Each motion will be executed at least 70 times. All of these actions will be done from different distances and angles to the radar, as well as at different speeds. This will help create a richly detailed dataset that can effectively train the RNN model, which is the main model that will be used in this project.

3.4.2 Evaluation Metrics

The results of the confusion matrix can be categorized into true positives(TP), false positives (FP), true negatives (TN), and false negatives (FN). The following metrics can be used to evaluate the system's performance:

Accuracy: A = $\frac{TP + TN}{TP + TN + FP + FN}$

Precision: P = $\frac{TP}{TP + FP}$

Recall: R = $\frac{TP}{TP + FN}$

F1 Score: F1 = $\frac{2*p*r}{p + r}$

Furthermore, the dataset will be split into training and testing sets, with 70% of the data allocated for training and the remaining 30% reserved for testing, inspired by Li et al. (2022).

3.5 Recurrent Neural Network (RNN)

A Recurrent Neural Network (RNN) Classifier will be implemented in this project for motion detection. An RNN is a neural network that detects patterns in a sequence of data (Schmidt, 2019) and can be trained on sequential data to create a machine learning model that can then make predictions based on inputs. For this project, an RNN model will be trained on motion and no-motion datasets so that it can then determine whether an input received from the radar indicates motion or not.
RNNs are proficient at recognizing patterns in sequences of data. They learn by using training data. RNNs contain internal loops, which allow them to remember previous inputs. Using information gained from prior inputs, they can make predictions.

According to Zargar (2021), RNNs learn the behaviour of sequences through cycles within the network of nodes, thus allowing them to learn from every input and make decisions based on that.

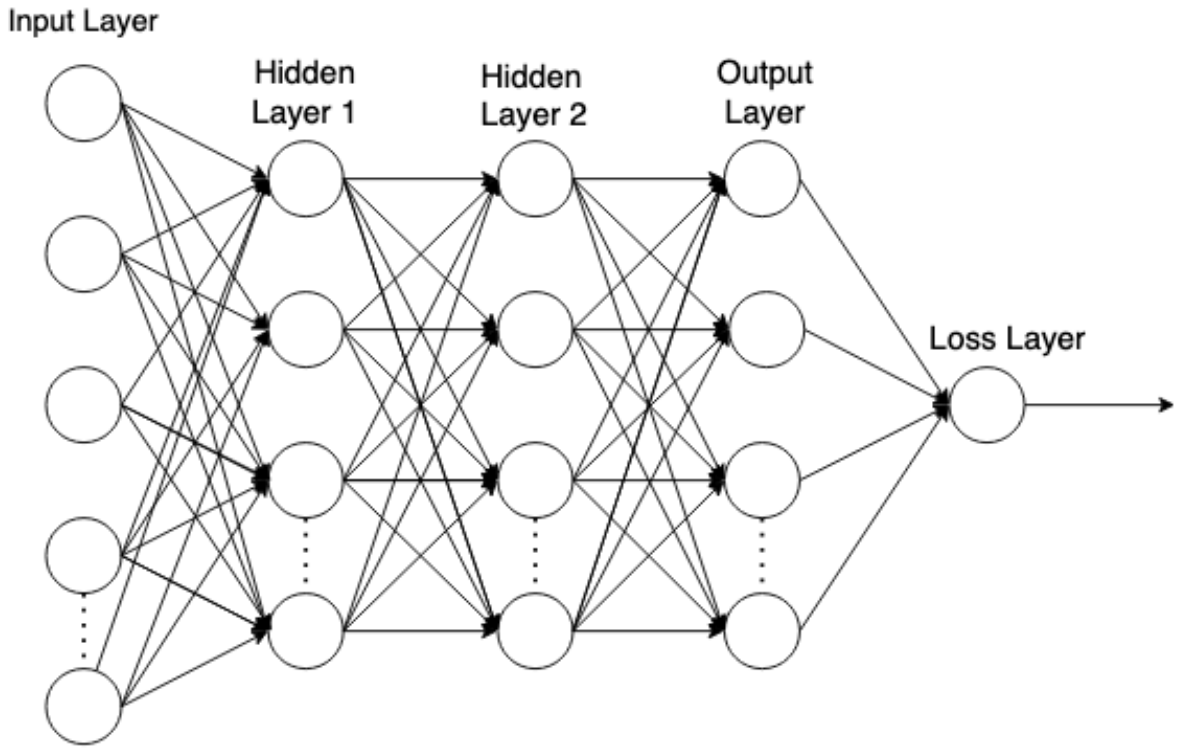


Figure 3.5a: RNN Diagram

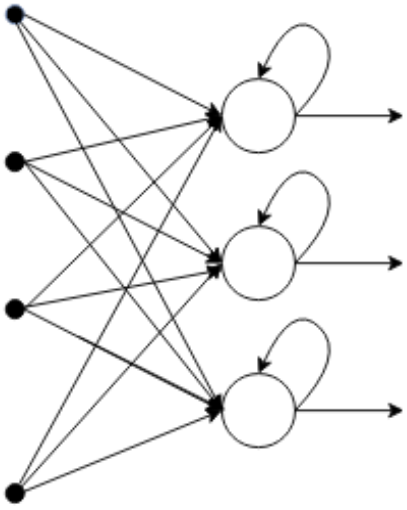


Figure 3.5b: Information Flow in an RNN

Information traverses in a loop in an RNN. The RNN model makes decisions based on the input it receives as well as the information gained from previous inputs and training inputs, making its predictions very accurate and reliable. The RNN trains on data using the backpropagation algorithm in the following way (Werbos, 1990):

Forward Pass: The RNN processes the input sequence, generating predictions at each time step

Backward Pass: The backpropagation algorithm computes the difference or the error between the predicted output and the actual target values, and then propagates this error backward through the network using a technique called Backpropagation Through Time (BPTT). Gradients of the loss with respect to the weights are calculated, allowing for weight updates to minimize the error.

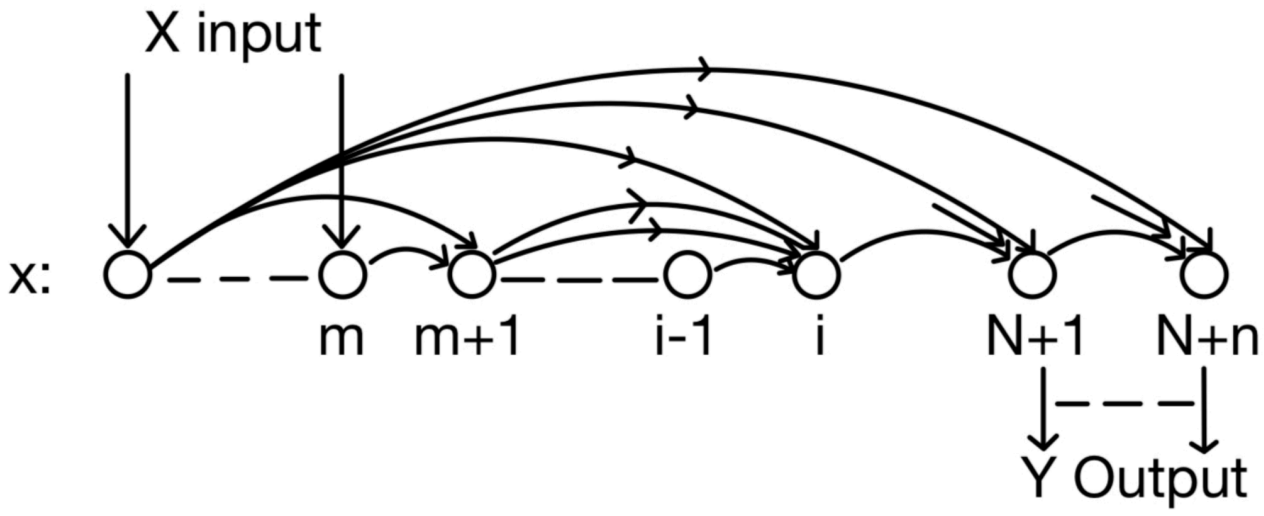


Figure 3.5c: Backpropagation

However, as discussed by Manaswi (2018), a traditional RNN model could suffer from the vanishing gradient problem, or the exploding gradient problem. The vanishing

gradient problem leads to difficulties in learning long-range dependencies in sequential data because the gradients become too small, while the exploding gradient problem results in unstable training and erratic predictions occur due to gradients becoming excessively large.

To address these challenges, various RNN variants have been developed, one of the most effective being the Long Short-Term Memory (LSTM) model. LSTMs are specifically designed to retain long-term dependencies and mitigate the issues associated with gradient instability. A detailed discussion of the RNN as well as the LSTM will be provided in a later section.

3.6 Long Short-Term Memory (LSTM) Classification Model

For preliminary testing and analysis, the Long Short-Term Memory (LSTM) model of the RNN was used. LSTMs are ideal for handling the challenges faced by traditional RNN models when learning from long sequential data. According to Manaswi (2018), LSTM contains a set of gates to control when information enters memory, thus solving both the vanishing and exploding gradient problems faced by traditional RNNs, which were discussed in Section 3.4.

LSTM consists of a memory cell that has linear dependence of its current and past activities (Yao et al., 2015). This memory cell can retain information in memory for long time periods. Information flow in and out of the cell are controlled by three gates:

Forget Gate: Used to modulate information flow between the past and current activities. Helps the network decide what information to discard from the cell state

Input Gate: Used to modulate input. Controls the extent to which new information is added to the cell state

Output Gate: Used to modulate output. Determines what information from the cell state is used to compute the output of the LSTM

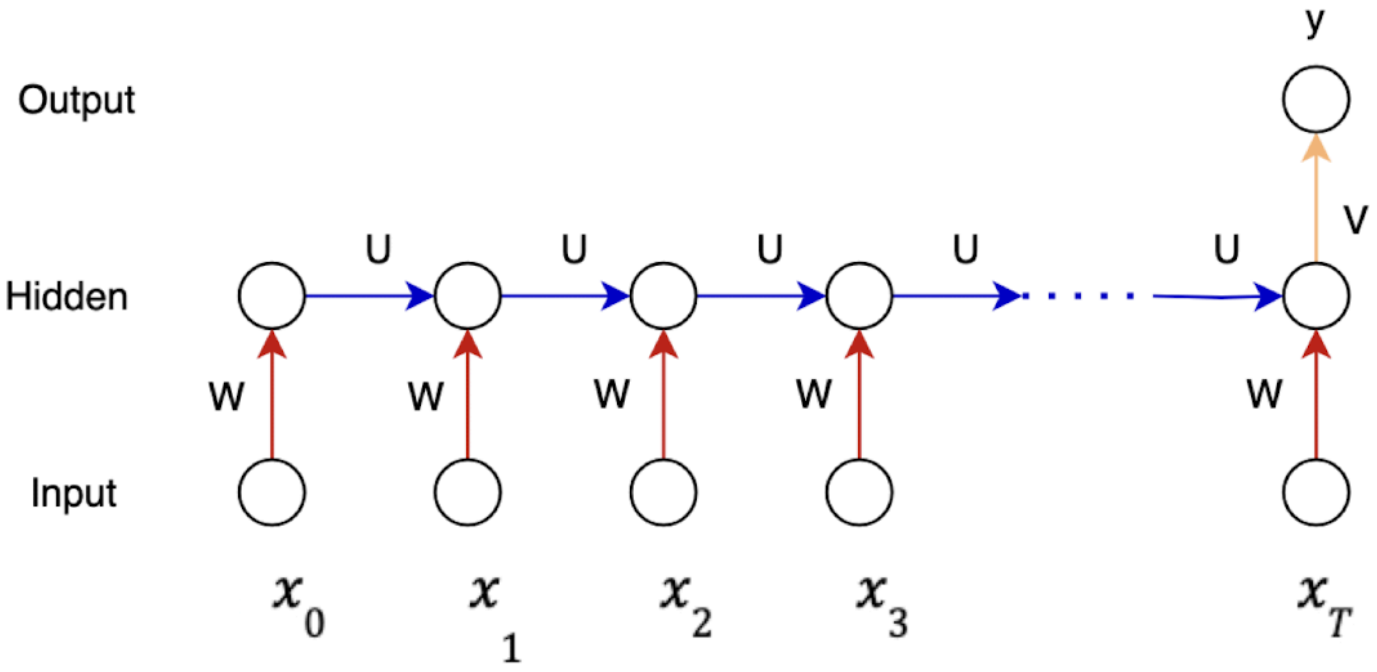


Figure 3.6a: Long Short-Term Memory RNN

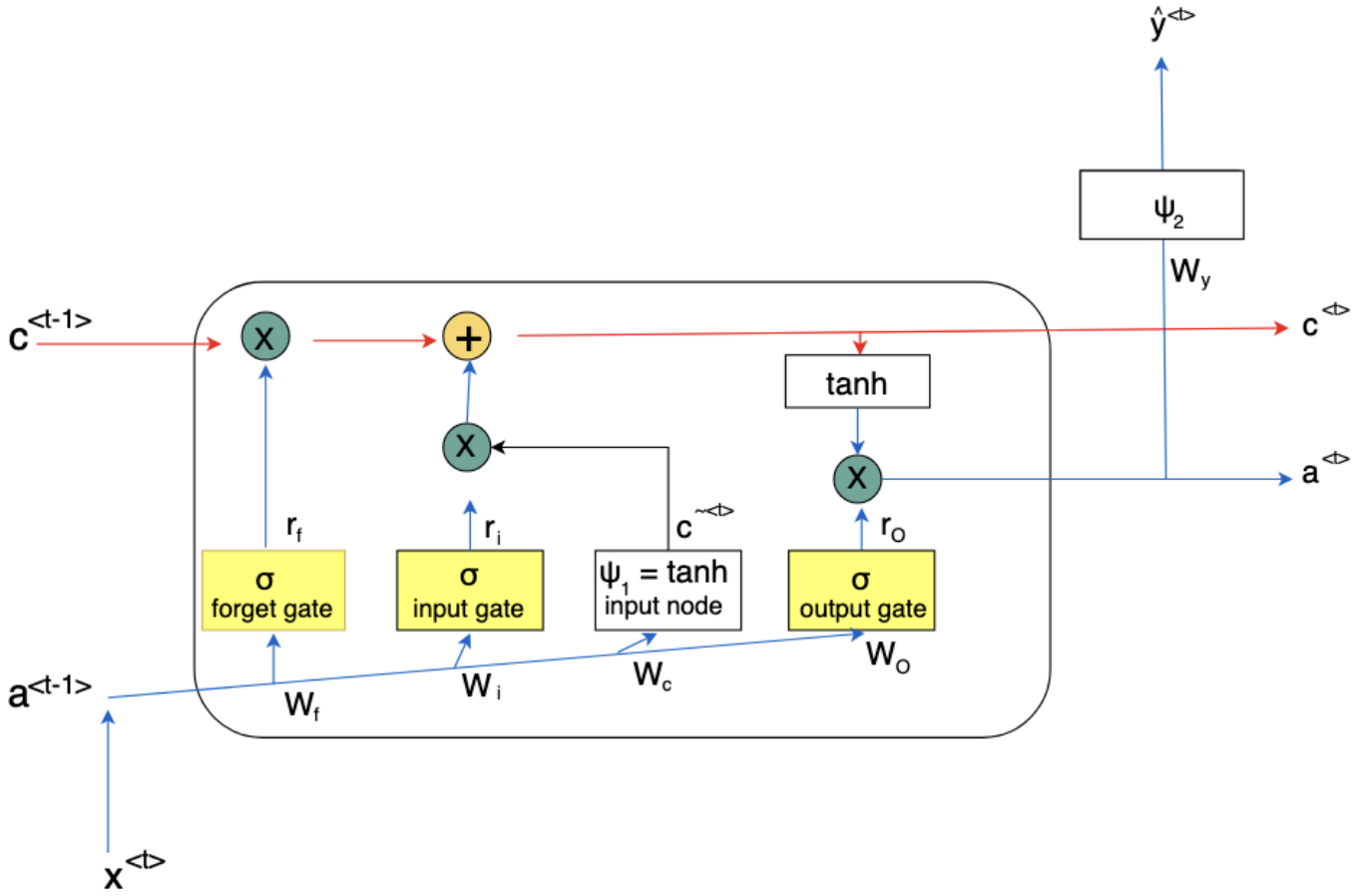


Figure 3.6b: LSTM Unit Cell

3.7 Gated Recurrent Units (GRU)

Gated Recurrent Units (GRUs) are a variation of Recurrent Neural Networks (RNNs) designed to address the long-term dependency issues in sequence data processing. The GRU's specialized architecture helps it learn long-term dependencies, thereby tackling the vanishing gradient problem faced by traditional RNNs (Chandra et al., 2021).

Similar to the LSTM architecture, the GRU architecture also implements a gating mechanism to control information flow, allowing the network to adaptively remember or forget previous information (Tjandra et al., 2016). The GRU is ideal for capturing complex relationships between inputs and hidden layers. The gates controlling information flow are as follows:

Update Gate: This gate controls the amount of past information that needs to be relayed to the future, thus allowing the model to determine how much the previous state influences the current state, and giving the model the ability to learn long-term dependencies.

Reset Gate: This gate controls the amount of past information that needs to be forgotten, giving the model the ability to focus on important information by discarding information that is irrelevant for future predictions.

The GRU's update and reset gates have similar mechanisms to the input and forget gates of the LSTM model, respectively. However, the GRU lacks a gate with a similar

functionality to the LSTM's output gate, which contributes to its efficiency and faster training times. Nonetheless, this trade-off comes at the expense of being unable to match the LSTM's excellent performance on tasks requiring highly complex dependencies.

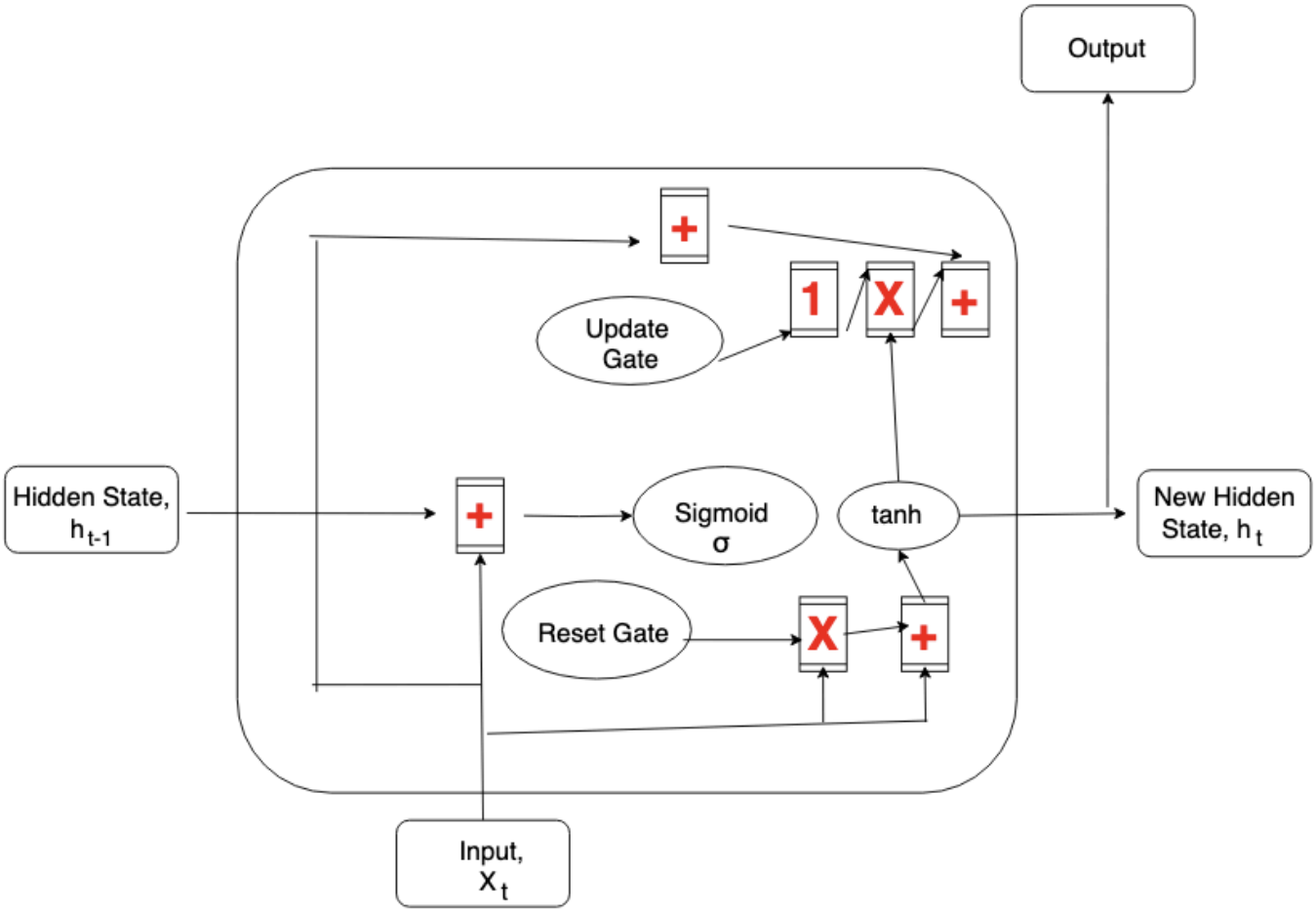


Figure 3.7: GRU Architecture and Functions

4. Detailed Methodology and Implementation

4.1 Dataset Preprocessing

4.1.1 DataFrame Creation

Each .csv file is converted into a DataFrame df, which is a two-dimensional data structure, as this facilitates data manipulation, such as adding the DataFrames to the motion_df and not_df arrays, adding source_file_ids, and combining all the DataFrames into a combined DataFrame combined_df. The DataFrames are assigned with IDs (source_file_id) to keep track of the data since there is a large amount of data. Each .csv

Each .csv file is first added to a dataframe corresponding to the type of motion it contains data for. Files containing data for bowing were added to bowing_df, files containing data for walking were added to walking_df, and so on. All the individual motion dataframes were then added to motion_df, which was combhined with not_df after all no motion files were added to not_df, to form a combined_df.

```python
# Initialize DataFrames and counter
motion_df = pd.DataFrame()
counter = 0

# Function to read files and append to DataFrame
def read_files(file_list, counter):
    df_list = []

    for file in file_list:
        try:
            df = pd.read_csv(file)
            df['source_file_id'] = counter  # Use the counter as the source_file_id
            df_list.append(df)  # Add DataFrame to the list
            counter += 1  # Increment the counter
        except Exception as e:
            print(f"Error reading {file}: {e}")

    if df_list:  # Check if any DataFrames were added
        combined_df = pd.concat(df_list, ignore_index=True)
        print(f"Number of files added: {len(df_list)}")
        return combined_df, counter
    else:
        return pd.DataFrame(), counter  # Return empty DataFrame if no files added
```

Figure 4.1.1a: DataFrame Creation

```python
# Read bowing files
bowing_df, counter = read_files(bowing_files, counter)

bowing_df
```

```
Number of files added: 70
```

	frame	DetObj#	x	y	z	v	snr	noise	source_file_id
0	0	0	-0.082370	0.030813	0.0	0.000000	225	642	0
1	0	1	1.009692	0.470561	0.0	0.000000	159	628	0
2	0	2	-1.820103	3.580003	0.0	0.000000	152	662	0
3	1	0	-0.082370	0.030813	0.0	0.000000	218	648	0
4	1	1	1.009692	0.470561	0.0	0.000000	165	620	0
...	...	...	...	...	...	...	...	...	...
7280	22	3	-0.265708	1.441451	0.0	-0.120853	169	646	69
7281	23	0	-0.082370	0.030813	0.0	0.000000	222	646	69
7282	23	1	1.009692	0.470561	0.0	0.000000	157	632	69
7283	23	2	-1.820103	3.580003	0.0	0.000000	155	673	69
7284	23	3	-0.255080	1.383793	0.0	-0.362560	336	462	69

7285 rows × 9 columns

Figure 4.1.1b: Individual DataFrames

```
# Combine all DataFrames into motion_df
motion_df = pd.concat([bowing_df, waving_df, jumping_df, moving_df, walking_df], ignore_index=True)
print(f"Total number of files added to motion_df: {counter}")
motion_df
```

```
Total number of files added to motion_df: 631
```

	frame	DetObj#	x	y	z	v	snr	noise	source_file_id
0	0	0	-0.082370	0.030813	0.0	0.000000	225	642	0
1	0	1	1.009692	0.470561	0.0	0.000000	159	628	0
2	0	2	-1.820103	3.580003	0.0	0.000000	152	662	0
3	1	0	-0.082370	0.030813	0.0	0.000000	218	648	0
4	1	1	1.009692	0.470561	0.0	0.000000	165	620	0
...	...	...	...	...	...	...	...	...	...
83184	24	3	2.168720	2.060631	0.0	0.362471	215	621	630
83185	24	4	-0.438396	2.026494	0.0	0.604118	413	525	630
83186	24	5	-1.878842	1.631051	0.0	0.604118	182	712	630
83187	24	6	-0.901844	2.318847	0.0	0.604118	182	712	630
83188	24	7	-0.738116	2.095260	0.0	0.724941	209	680	630

83189 rows × 9 columns

Figure 4.1.1c: Combining All Motion Files

```python
# Initialize counter and list to store non-motion DataFrames
not_counter = 0
not_df = []


# Loop through each file in motion_files
for file in not_files:
    df = pd.read_csv(file)

    df['source_file_id'] = counter  # Use the counter as the source_file_id


    not_df.append(df)  # Add DataFrame to the list
    counter += 1  # Increment the counter
    not_counter += 1
# Concatenate the list of DataFrames into a single DataFrame
not_df = pd.concat(not_df, ignore_index=True)

# Output the combined DataFrame
not_df
```

```
/var/folders/pj/p8wrytyd0hl22vdn7hm2sw1r0000gn/T/ipykernel_78591/714953276.py:17:
  not_df = pd.concat(not_df, ignore_index=True)
```

	frame	DetObj#	x	y	z	v	snr	noise	source_file_id
0	0	0	-0.109826	0.041084	0.0	0.0	193	686	631
1	0	1	0.283422	0.373719	0.0	0.0	158	699	631
2	0	2	0.361363	0.603659	0.0	0.0	308	738	631
3	0	3	0.375535	1.507613	0.0	0.0	211	734	631
4	0	4	0.318850	1.729741	0.0	0.0	297	737	631
...	...	...	...	...	...	...	...	...	...
22822	24	5	0.336564	2.202351	0.0	0.0	310	734	731
22823	24	6	0.000000	2.491752	0.0	0.0	256	824	731
22824	24	7	-3.039704	2.394486	0.0	0.0	215	743	731
22825	24	8	0.000000	4.924874	0.0	0.0	223	693	731
22826	24	9	0.000000	7.328681	0.0	0.0	197	538	731

22827 rows × 9 columns

Figure 4.1.1d: No Motion DataFrames

```
combined_df = pd.concat([motion_df, not_df], ignore_index=True)
combined_df.fillna(0, inplace=True)
combined_df
```

	frame	DetObj#	x	y	z	v	snr	noise	source_file_id
0	0	0	-0.082370	0.030813	0.0	0.0	225	642	0
1	0	1	1.009692	0.470561	0.0	0.0	159	628	0
2	0	2	-1.820103	3.580003	0.0	0.0	152	662	0
3	1	0	-0.082370	0.030813	0.0	0.0	218	648	0
4	1	1	1.009692	0.470561	0.0	0.0	165	620	0
...	...	...	...	...	...	...	...	...	...
106011	24	5	0.336564	2.202351	0.0	0.0	310	734	731
106012	24	6	0.000000	2.491752	0.0	0.0	256	824	731
106013	24	7	-3.039704	2.394486	0.0	0.0	215	743	731
106014	24	8	0.000000	4.924874	0.0	0.0	223	693	731
106015	24	9	0.000000	7.328681	0.0	0.0	197	538	731

106016 rows × 9 columns

Figure 4.1.1e: Combined DataFrames Creation

4.1.2 Filling and Adding Frames

RNNs models require input sequences of constant length. We first check if all files contain the same number of frames using the check_equal_frames function. The combined DataFrame combined_df obtained at the end of DataFrame creation is first sorted by source_file_id, and then the nunique() function is used to count the number of unique frames for each file. The all_equal boolean then determines whether all files in the DataFrame have the same number of frames.

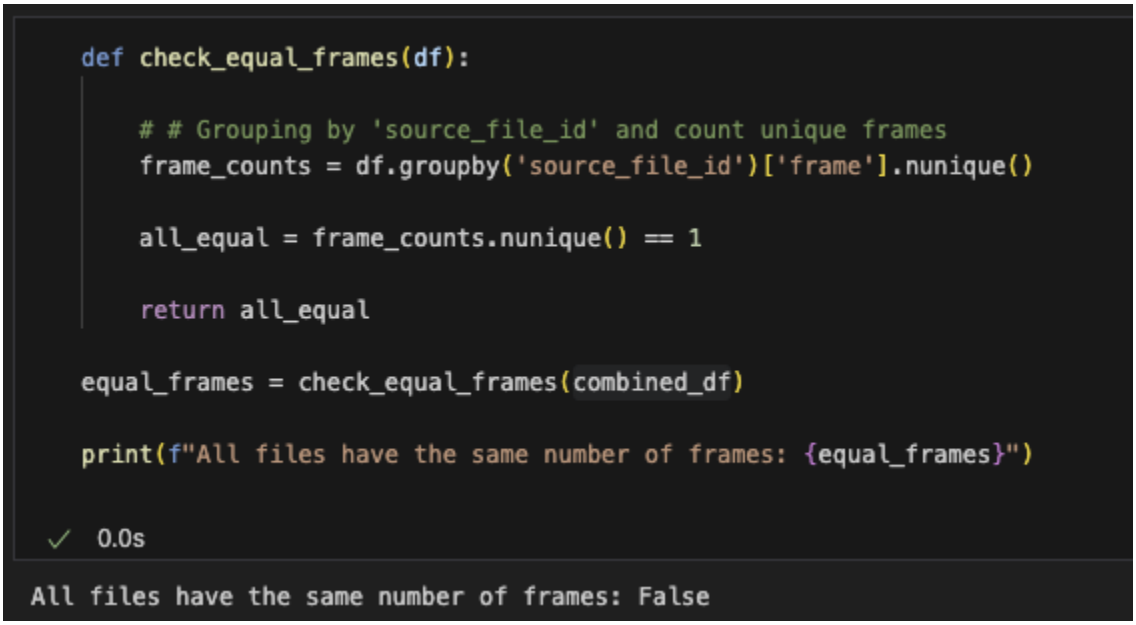


Figure 4.1.2a: Checking if Number of Frames is Equal for All Files

The fill_frames() function is then used to add missing frames to each file in the DataFrame. The insert_frame() function creates empty frames with zeros, which are then added to each file. All source_file_ids are iterated over, to ensure that the fill_frames() function is applied to every file.

```python
def fill_frames(frame_num, file_id, df):
    # Handle NaN in 'frame' column
    max_frame = int(df['frame'].max()) if not pd.isna(df['frame'].max()) else 0

    # Create a set of existing frames
    existing_frames = set(df['frame'].dropna().astype(int))
    all_frames = set(range(max_frame + 1))
    missing_frames = sorted(all_frames - existing_frames)

    # Function to insert a new frame into the DataFrame
    def insert_frame(frame_value):
        new_row = pd.DataFrame([{
            'frame': frame_value,'DetObj#': 0,'x': 0,'y': 0,'z': 0,'v': 0,'source_file_id': file_id
        }])
        return new_row

    # Fill missing frames
    i = 0
    while i < len(missing_frames):
        frame_to_fill = missing_frames[i]
        if frame_to_fill == 0 or (frame_to_fill - 1 in existing_frames):
            index_tochange = df[df['frame'] == frame_to_fill - 1].index[-1] if frame_to_fill > 0 else -1
            new_df = insert_frame(frame_to_fill)
            df = pd.concat([df.iloc[:index_tochange + 1], new_df, df.iloc[index_tochange + 1:]]).reset_index(drop=True)
            existing_frames.add(frame_to_fill)  # Update existing frames
        i += 1

    # Fill frames up to frame_num
    current_frame = max_frame + 1
    while current_frame <= frame_num:
        new_df = insert_frame(current_frame)
        df = pd.concat([df, new_df], ignore_index=True).reset_index(drop=True)
        current_frame += 1

    return df
```

Figure 4.1.2b: insert_frame() and fill_frames() Functions

```python
max_frames = combined_df.groupby('source_file_id')['frame'].max().max()
modified_df = pd.DataFrame()
i = 0

while i <= combined_df['source_file_id'].max():
    temp_df = fill_frames(max_frames, i, combined_df[combined_df['source_file_id'] == i])
    modified_df = pd.concat([modified_df, temp_df], ignore_index=True)
    i += 1

modified_df.fillna(0, inplace=True)
modified_df
```

Figure 4.1.2c: Iterating Through Files and Adding Frames

```
def check_equal_frames(df):

    # # Grouping by 'source_file_id' and count unique frames
    frame_counts = df.groupby('source_file_id')['frame'].nunique()

    all_equal = frame_counts.nunique() == 1

    return all_equal

equal_frames = check_equal_frames(modified_df)

print(f"All files have the same number of frames: {equal_frames}")
```

✓ 0.0s

```
All files have the same number of frames: True
```

Figure 4.1.2d: Checking if Number of Frames is Equal for All Files After Applying fill_frames() Function

4.1.3 Filling and Adding Points

Each frame must also contain the same number of detected objects, or points. Similar to checking and adding frames, to ensure this, we first check if that is the case, using the check_equal_points() function, and then add missing points to each frame using the fill_DetObj() function. Similar to adding frames, missing points were also added by creating points with zeros using the create_empty_point() function.

```
    def check_equal_points(df):

        # Grouping by 'source_file_id' and 'frame', then counting unique detected objects
        points_counts = df.groupby(['source_file_id', 'frame'])['DetObj#'].nunique()

        # Checking if all counts are the same
        all_equal = points_counts.nunique() == 1

        return all_equal

    equal_points = check_equal_points(modified_df)

    print(f"All frames have the same number of detected objects: {equal_points}")
]
```

```
All frames have the same number of detected objects: False
```

Figure 4.1.3a: Checking if All Frames Have the Same Number of Points

```
def insert_frame_rows(frame_value, file_id, start_index, num_missing):
    new_rows = pd.DataFrame({
        'frame': frame_value,
        'DetObj#': range(start_index + 1, start_index + num_missing + 1),'x': 0,'y': 0,'z': 0,'v': 0,
        'source_file_id': file_id
    })
    return new_rows

def fill_DetObj(maxpoints, file_id, df):
    for i in range(df['frame'].max() + 1):
        df_thisframe = df[df['frame'] == i]
        thisframe_maxpoints = df_thisframe['DetObj#'].max()

        if thisframe_maxpoints < maxpoints:
            index_tochange = df_thisframe.index[-1] if not df_thisframe.empty else -1
            num_missing = maxpoints - thisframe_maxpoints

            new_rows = insert_frame_rows(i, file_id, thisframe_maxpoints, num_missing)
            df = pd.concat([df.iloc[:index_tochange + 1], new_rows, df.iloc[index_tochange + 1:]]).reset_index(drop=True)

    return df
```

Figure 4.1.3b: create_empty_point and fill_DetObj() Functions

```python
max_detobj = modified_df.groupby(['source_file_id', 'frame']).agg({'DetObj#': 'max'}).values.max()
final_df = pd.DataFrame()
i = 0

while i <= modified_df['source_file_id'].max():
    temp_df = fill_DetObj(max_detobj, i, modified_df[modified_df['source_file_id'] == i])
    final_df = pd.concat([final_df, temp_df], ignore_index=True)
    i += 1

final_df.fillna(0, inplace=True)
final_df
```

✓ 13.6s

	frame	DetObj#	x	y	z	v	snr	noise	source_file_id
0	0	0	-0.082370	0.030813	0.0	0.0	225.0	642.0	0
1	0	1	1.009692	0.470561	0.0	0.0	159.0	628.0	0
2	0	2	-1.820103	3.580003	0.0	0.0	152.0	662.0	0
3	0	3	0.000000	0.000000	0.0	0.0	0.0	0.0	0
4	0	4	0.000000	0.000000	0.0	0.0	0.0	0.0	0
...	...	...	...	...	...	...	...	...	...
483115	0	17	0.000000	0.000000	0.0	0.0	0.0	0.0	731
483116	0	18	0.000000	0.000000	0.0	0.0	0.0	0.0	731
483117	0	19	0.000000	0.000000	0.0	0.0	0.0	0.0	731
483118	0	20	0.000000	0.000000	0.0	0.0	0.0	0.0	731
483119	0	21	0.000000	0.000000	0.0	0.0	0.0	0.0	731

483120 rows × 9 columns

Figure 4.1.3c: Iterating Through Frames and Adding Points

```python
equal_points = check_equal_points(final_df)
print(f"All frames have the same number of detected objects: {equal_points}")
final_df.groupby(['source_file_id','frame']).agg({'DetObj#':'max'}).values.min()
```

```
All frames have the same number of detected objects: True

np.int64(21)
```

Figure 4.1.3d: Checking if Number of Points is Equal for All Frames After Applying fill_DetObj() Function

The resulting DataFrame final_df is converted to a .csv file, and saved for further analysis and use in the RNN model.

```
final_df.to_csv('Matrix.csv', index=False)
```

✓ 0.8s

```
final_df = pd.read_csv('Matrix.csv')
final_df
```

✓ 0.1s

	frame	DetObj#	x	y	z	v	snr	noise	source_file_id
0	0	0	-0.082370	0.030813	0.0	0.0	225.0	642.0	0
1	0	1	1.009692	0.470561	0.0	0.0	159.0	628.0	0
2	0	2	-1.820103	3.580003	0.0	0.0	152.0	662.0	0
3	0	3	0.000000	0.000000	0.0	0.0	0.0	0.0	0
4	0	4	0.000000	0.000000	0.0	0.0	0.0	0.0	0
...	...	...	...	...	...	...	...	...	...
483115	29	17	0.000000	0.000000	0.0	0.0	0.0	0.0	731
483116	29	18	0.000000	0.000000	0.0	0.0	0.0	0.0	731
483117	29	19	0.000000	0.000000	0.0	0.0	0.0	0.0	731
483118	29	20	0.000000	0.000000	0.0	0.0	0.0	0.0	731
483119	29	21	0.000000	0.000000	0.0	0.0	0.0	0.0	731

483120 rows × 9 columns

Figure 4.1.3e: Storing the Final DataFrame in a .csv File

4.1.4 RNN Model Input Matrix

To create labels to assign to each type of motion, an assign_labels() function is used, which maps labels 0-5 to each unique source_file_id in final_df based on the type of motion/no motion they represent. This mapping of source_file_id to labels is stored in a new DataFrame called final_labels_df, so that the point cloud of each data group as a whole is used for training the model rather than each individual point as the former is significantly less useful and yields highly inaccurate results, as seen in Section 5.

```
def assign_labels(df):
    label_mapping = {}

    for file_id in df['source_file_id'].unique():
        if 0 <= file_id <= 69:
            label_mapping[file_id] = 1  # Bowing

        elif 70 <= file_id <= 278:
            label_mapping[file_id] = 2  # Waving

        elif 279 <= file_id <= 359:
            label_mapping[file_id] = 3  # Jumping

        elif 360 <= file_id <= 442:
            label_mapping[file_id] = 4  # Moving Body

        elif 443 <= file_id <= 630:
            label_mapping[file_id] = 5  # Walking

        elif 631 <= file_id <= 731:
            label_mapping[file_id] = 0  # No Motion

        else:
            label_mapping[file_id] = -1  # Unknown or out of range

    label_df = pd.DataFrame({
        'source_file_id': label_mapping.keys(),
        'label': label_mapping.values()
    })

    unique_df = df[['source_file_id']].drop_duplicates().merge(label_df, on='source_file_id', how='left')

    return unique_df

final_labels_df = assign_labels(final_df)

final_labels_df
```

Figure 4.1.4a: final_labels_df

The two DataFrames final_df and final_labels_df are then merged to create a dictionary that contains the source_file_id of each file, all its data, and one label assigned to the entire point cloud. This structure is suitable for training the RNN Model. This structured data is saved as a pickle file samples.pkl, allowing for efficient storage and retrieval for further processing or model training.

```
grouped_frames = final_df.groupby('source_file_id').apply(lambda x: x.to_dict(orient='records')).reset_index()

merged_df = grouped_frames.merge(final_labels_df, on='source_file_id', how='left')

samples = []
for _, row in merged_df.iterrows():
    source_file_id = row['source_file_id']
    frames = row[0]
    label_value = row['label']

    action_classes = {
        1: 'Bowing',
        2: 'Waving',
        3: 'Jumping',
        4: 'Moving Body',
        5: 'Walking',
        0: 'No Motion'
    }

    action_class = action_classes.get(label_value, 'Unknown')

    sample = {
        'source_file_id': source_file_id,
        'label': {
            'value': label_value,
            'action': action_class
        },
        'frames': frames
    }

    samples.append(sample)

samples
```

Figure 4.1.4b: Creation of samples.pkl

The data is then further processed using the preprocess_samples() function, which calculates two derived features: relative distance and change in velocity for each frame.

The function computes the relative distance traveled from the previous frame using the Euclidean distance formula applied to the x, y, and z coordinates. Additionally, it calculates the change in velocity by subtracting the previous frame's velocity from the current frame's velocity. These derived features are then added to each frame's dictionary, enhancing the dataset with critical motion dynamics, which can significantly aid in training the model for tasks such as action recognition.

```python
def preprocess_samples(samples, target_sequence_length=100):
    """Preprocess the samples to calculate derived features."""
    processed_samples = []

    for sample in samples:
        frames = sample['frames']

        if len(frames) > target_sequence_length:
            step = len(frames) // target_sequence_length
            frames = frames[::step]
        elif len(frames) < target_sequence_length:
            frames += [{'x': 0, 'y': 0, 'z': 0, 'v': 0, 'snr': 0, 'noise': 0} for _ in range(target_sequence_length - len(frames))]

        for i in range(len(frames)):
            if i > 0:
                dx = frames[i]['x'] - frames[i-1]['x']
                dy = frames[i]['y'] - frames[i-1]['y']
                dz = frames[i]['z'] - frames[i-1]['z']
                frames[i]['rel_distance'] = np.sqrt(dx**2 + dy**2 + dz**2)
                frames[i]['v_change'] = frames[i]['v'] - frames[i-1]['v']
            else:
                frames[i]['rel_distance'] = 0
                frames[i]['v_change'] = 0

        processed_samples.append({
            'source_file_id': sample['source_file_id'],
            'label': sample['label'],
            'frames': frames
        })

    return processed_samples
```

Figure 4.1.4c: Calculating relative distance and change in velocity

Finally, a three-dimensional matrix called input_matrix is constructed, where the dimensions represent the total number of samples, the fixed number of frames per sample, and the total number of features per frame, calculated as the number of detected objects multiplied by the number of relevant features. This matrix will be used to train the RNN model.

```
# Creating input feature matrix
input_matrix = np.zeros((num_samples, sequence_length, input_size), dtype=np.float32)
labelsArr = []

# Processing samples
source_file_ids = sorted(set(sample["source_file_id"] for sample in samples))  # Unique file IDs
file_id_to_index = {file_id: idx for idx, file_id in enumerate(source_file_ids)}

for sample in samples:
    file_index = file_id_to_index[sample["source_file_id"]]
    labelsArr.append(sample["label"]["value"])

    frame_data = np.zeros((sequence_length, input_size), dtype=np.float32)

    for frame in sample["frames"]:
        if frame["frame"] < sequence_length and frame["DetObj#"] < num_detobj:
            det_idx = frame["DetObj#"] * num_features  # Determine feature position
            frame_data[frame["frame"], det_idx:det_idx+num_features] = [frame["x"], frame["y"], frame["z"], frame["v"]]

    input_matrix[file_index] = frame_data
```

✓ 0.3s

```
# Converting to PyTorch tensors
input_matrix = torch.tensor(input_matrix, dtype=torch.float32)
labelsArr = torch.tensor(labelsArr, dtype=torch.long)

# Splitting into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(input_matrix, labelsArr, test_size=0.3, random_state=30)

# Creating DataLoaders
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

test_dataset = TensorDataset(X_test, y_test)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

print(f"Input tensor shape: {X_train.shape}")
print(f"Target tensor shape: {y_train.shape}")
```

✓ 0.1s

```
Input tensor shape: torch.Size([512, 29, 88])
Target tensor shape: torch.Size([512])
```

Figure 4.1.4d: Matrix Construction

4.2 Training the Model

The hyperparameters that were used are as follows:

```
num_samples = 732  # Unique source file IDs / (max source_file_id + 1)
sequence_length = 29  # Highest number of frames
num_features = 4  # x, y, z, v
num_detobj = 21  # Highest DetObj#
input_size = 88  # num_detobj+1 * 4(x,y,z,v)
num_classes = 6  # Number of motion classes
hidden_size = 128  # LSTM hidden size
num_layers = 3  # LSTM layers
```

Figure 4.2a: Hyperparameters

The labels were stored in a separate array labels_arr, which will be used for training and testing the RNN model. The dataset was split into a training set (70%) and a test set (30%), and random_state was set to 30 for shuffling the data before splitting it into training and test sets.

```
# Splitting into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(input_matrix, labelsArr, test_size=0.3, random_state=30)

# Creating DataLoaders
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

test_dataset = TensorDataset(X_test, y_test)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

print(f"Input tensor shape: {X_train.shape}")
print(f"Target tensor shape: {y_train.shape}")
```

Figure 4.2b: Splitting the Data and Creating Training and Test Datasets and Loaders

The feature tensors x_train and x_test were created from the matrix, while label tensors y_train and y_test were created from labels_arr. train_dataset was created from x_train and y_train for training the RNN model, while test_dataset was created from x_test and y_test for testing.

```python
class RNN_LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes, dropout_rate=0.2):
        super(RNN_LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.dropout = nn.Dropout(dropout_rate)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)

        out, _ = self.lstm(x, (h0, c0))  # LSTM forward pass
        out = self.dropout(out[:, -1, :])
        out = self.fc(out)
```

Figure 4.2c: LSTM Model

```python
# Training Loop
num_epochs = 70
trainloss_list = []
testloss_list = []
trainaccuracy_list = []
testaccuracy_list = []

for epoch in range(num_epochs):
    # Train Network
    model.train()
    correct = 0
    total = 0
    running_train_loss = 0.0

    for batch_idx, (data, targets) in enumerate(tqdm(train_loader)):
        data, targets = data.to(device), targets.to(device)

        optimizer.zero_grad()
        output = model(data)
        trainloss = criterion(output, targets)
        trainloss.backward()
        optimizer.step()

        running_train_loss += trainloss.item()
        total += targets.size(0)
        _, predicted = torch.max(output, 1)
        correct += (predicted == targets).sum().item()

    train_loss_epoch = running_train_loss / len(train_loader)
    train_accuracy_epoch = correct / total
    trainloss_list.append(train_loss_epoch)
    trainaccuracy_list.append(train_accuracy_epoch)

    # Test Network
    model.eval()
    num_correct = 0
    num_samples = 0
    running_test_loss = 0.0

    with torch.no_grad():
        for x, y in test_loader:
            x, y = x.to(device), y.to(device)
            output = model(x)
            testloss = criterion(output, y)
            running_test_loss += testloss.item()

            _, predicted = torch.max(output, 1)
            num_correct += (predicted == y).sum().item()
            num_samples += y.size(0)

    test_loss_epoch = running_test_loss / len(test_loader)
    test_accuracy_epoch = num_correct / num_samples
    testloss_list.append(test_loss_epoch)
    testaccuracy_list.append(test_accuracy_epoch)

    print(f'Epoch {epoch + 1}/{num_epochs}, Train Loss: {train_loss_epoch:.4f}, '
          f'Test Loss: {test_loss_epoch:.4f}, Train Acc: {train_accuracy_epoch:.4f}, '
          f'Test Acc: {test_accuracy_epoch:.4f}')

# Print Final Accuracy
final_train_acc = trainaccuracy_list[-1]
final_test_acc = testaccuracy_list[-1]

print("\n==== Final Accuracy ====")
print(f"Train Accuracy: {final_train_acc:.4f}")
print(f"Test Accuracy: {final_test_acc:.4f}")
```

Figure 4.2d: Training and Testing the Model

5. Preliminary Results and Future Improvements

As previously mentioned, due to receiving the mmWave radar kit just three days before the report was due, it was not feasible for me to collect my own data. Consequently, I utilized datasets obtained online for the preliminary analysis.

Preliminary analysis has been conducted using the LSTM model. The LSTM model has been used in preference to the traditional RNN Model due to the advantages of the former as discussed in Section 3.4.

As indicated by the initial training and testing results, as well as the accuracy and F1 score, the LSTM model is currently not reliable enough to be used for motion detection. However, it is a promising start.

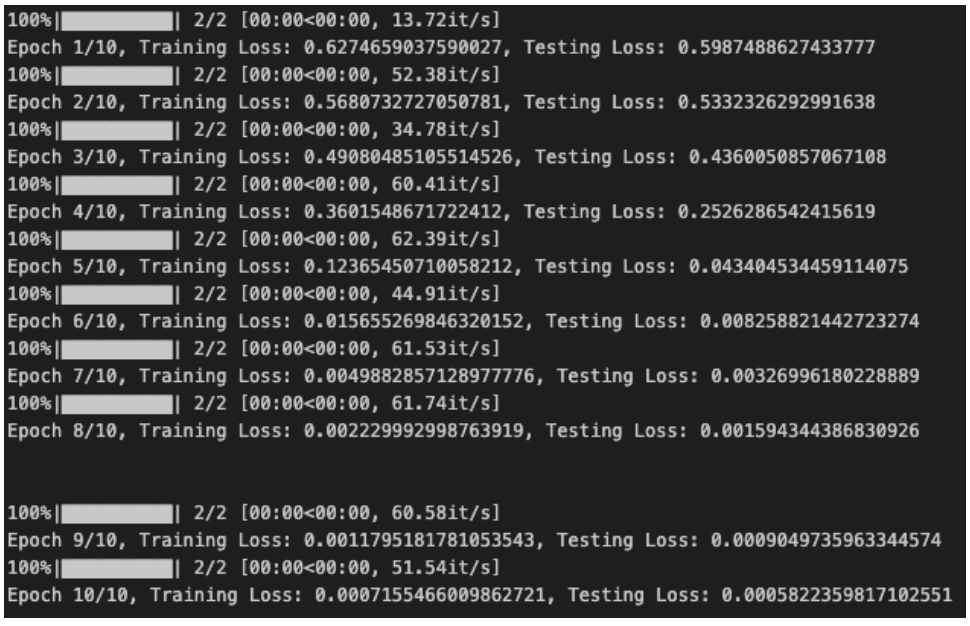


Figure 5a: Initial Training and Testing Results

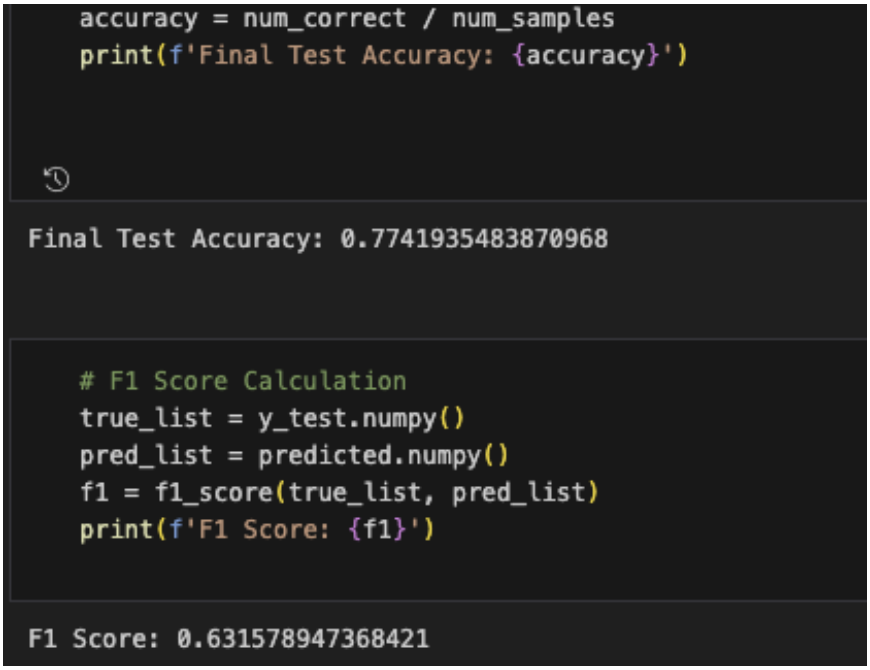


Figure 5b: Accuracy and F1 Score

Further analysis will be conducted over the coming months, additional datasets that encompass a wide variety of motion and no motion scenarios. With the radar and evaluation module now in my possession, I will be able to gather my own data, which I believe will be imperative for the development of this model, and will significantly increase its accuracy.

Furthermore, additional analysis will also be done on other RNN variants, such as the traditional or Vanilla RNN Model, as well as the Gated Recurrent Units (GRU) Model, to compare their performance.

5.1 Analysis of Preliminary Results

The main reason behind the low accuracy during initial testing was due to improper assignment of labels. Previously, labels were assigned to each row in the final DataFrame final_df, which was then being used to train the model. Since each row represents a singular spatial point, assigning labels to each spatial point adversely affected the model's training and testing accuracies, leading to a high number of incorrect predictions during testing (Mr. Danei Gong, pc). This was fixed, and labels were assigned to entire point clouds, as seen in Section 4.1.4.

	frame	DetObj#	x	y	z	v	source_file_id	label
0	0	0	-0.082370	0.030813	0.0	0.0	0	1
1	0	1	1.009692	0.470561	0.0	0.0	0	1
2	0	2	-1.820103	3.580003	0.0	0.0	0	1
3	0	3	0.000000	0.000000	0.0	0.0	0	1
4	0	4	0.000000	0.000000	0.0	0.0	0	1
...	...	...	...	...	...	...	...	...
336331	25	17	0.000000	0.000000	0.0	0.0	587	0
336332	25	18	0.000000	0.000000	0.0	0.0	587	0
336333	25	19	0.000000	0.000000	0.0	0.0	587	0
336334	25	20	0.000000	0.000000	0.0	0.0	587	0
336335	25	21	0.000000	0.000000	0.0	0.0	587	0

336336 rows × 8 columns

Figure 5.1a: Improper Assignment of Labels to Individual Points

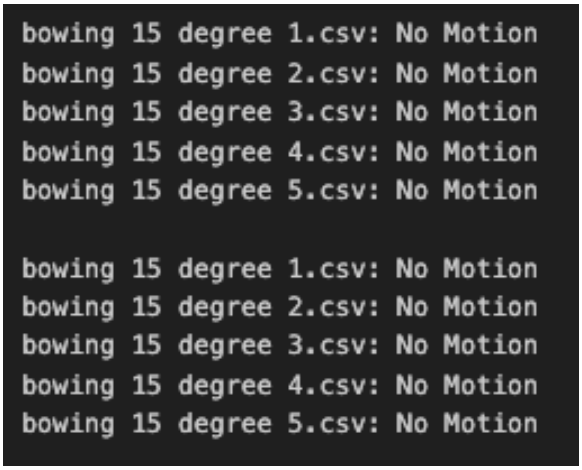


Figure 5.1b: Incorrect Predictions

```
[{'source_file_id': 0,
  'label': {'value': 1, 'action': 'Bowing'},
  'frames': [{'frame': 0,
    'DetObj#': 0,
    'x': -0.0823696106672287,
    'y': 0.0308127366006374,
    'z': 0.0,
    'v': 0.0,
    'snr': 225.0,
    'noise': 642.0,
    'source_file_id': 0},
   {'frame': 0,
    'DetObj#': 1,
    'x': 1.0096920728683472,
    'y': 0.470561146736145,
    'z': 0.0,
    'v': 0.0,
    'snr': 159.0,
    'noise': 628.0,
    'source_file_id': 0},
   {'frame': 0,
    'DetObj#': 2,
    'x': -1.8201026916503904,
    'y': 3.580003023147583,
    'z': 0.0,
...
    'z': 0.0,
    'v': 0.0,
    'snr': 0.0,
    'noise': 0.0,
    'source_file_id': 731}]}]
```


Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Figure 5.1c: Proper Assignment Labels to Point Clouds

```
File: 128_hand_waving_6.csv -> Predicted Motion: Waving (Label - 2)
File: 125_hand_waving_3.csv -> Predicted Motion: Waving (Label - 2)
File: 124_hand_waving_2.csv -> Predicted Motion: Waving (Label - 2)
File: 126_hand_waving_4.csv -> Predicted Motion: Waving (Label - 2)
File: 127_hand_waving_5.csv -> Predicted Motion: Waving (Label - 2)
File: 123_hand_waving_1.csv -> Predicted Motion: Waving (Label - 2)

Prediction Fractions:
Waving: 100.00% (6/6)
```

Figure 5.1d: Accurate Predictions

6. Final Results and Evaluation

Extensive training and testing were conducted using three variants of recurrent neural networks: the traditional Vanilla RNN, Gated Recurrent Unit (GRU), and Long Short-Term Memory (LSTM). To ensure fairness and consistency across all models, identical hyperparameters were employed, including a uniform learning rate for the Adam optimizer and a fixed number of training epochs set to 20. Additionally, the dataset was split into training and testing sets in a 70-30 ratio, with a fixed random seed to ensure reproducibility across experiments.

Parameter/ Hyperparameter	Value
num_samples	732
sequence_length	29
num_features	4
num_detobj	21
input_size	88
num_classes	6
hidden_size	128
num_epochs	20
test_size	0.3
random_state	30
lr	0.001

Table 6: Chosen List of Parameters and Hyperparameters for Comparing the Models

6.1 Comparison

Performance Metric	Vanilla RNN	GRU	LSTM
Accuracy (A)	88.64%	92.27%	95.00%
Precision (P)	0.8910	0.9262	0.9519
Recall (R)	0.8864	0.9227	0.9500
F1 Score	0.8857	0.9241	0.9499

Table 6.1: Performance Metrics of the Three Models

The formulas for calculating these scores have been mentioned in Section 3.4.2. The accuracy A is the proportion of correct predictions made by the model out of all predictions, the precision P is the ratio of true positive predictions to the total predicted positives, indicating the accuracy of positive predictions, the recall R is the ratio of true positive predictions to the total actual positives, measuring the model's ability to identify relevant instances, and the F1 Score is the harmonic mean of precision and recall, providing a balance between the two metrics.

6.1.1 Vanilla RNN

The performance metrics indicate decent performance, but they are not sufficient in scenarios where false negatives carry significant consequences, hence rendering the Vanilla RNN unusable in this project.

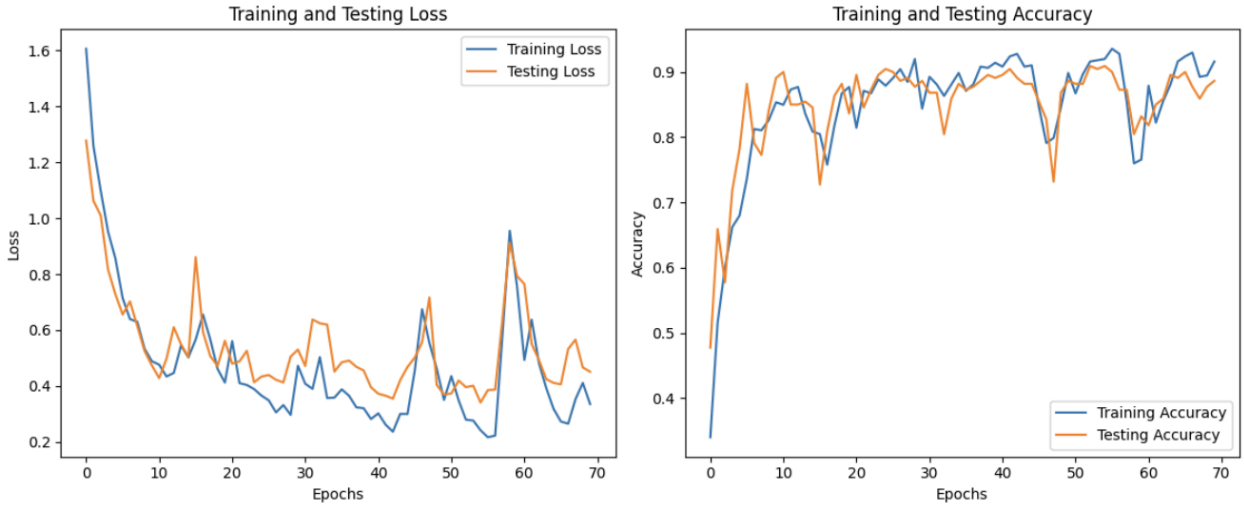


Figure 6.1.1a: Vanilla RNN Training and Testing Curves

The training and testing loss curves indicate that the model is learning adequately over the epochs as there is a steady decline in training and testing losses. However, there are considerable spikes in testing loss around epochs 45 and 60, indicating sharp sudden increases in error, and the testing loss remains slightly higher than the training loss almost entirely throughout the training process. This could be because of a potential issue with generalization, suggesting that the model might not be reliably capturing the underlying patterns in the data. Additionally, there are many sudden drops in the testing and training accuracy curves, particularly around epochs 15, 47 and 58,

which leads me to the conclusion that the model's performance is very poor and the model is unstable.

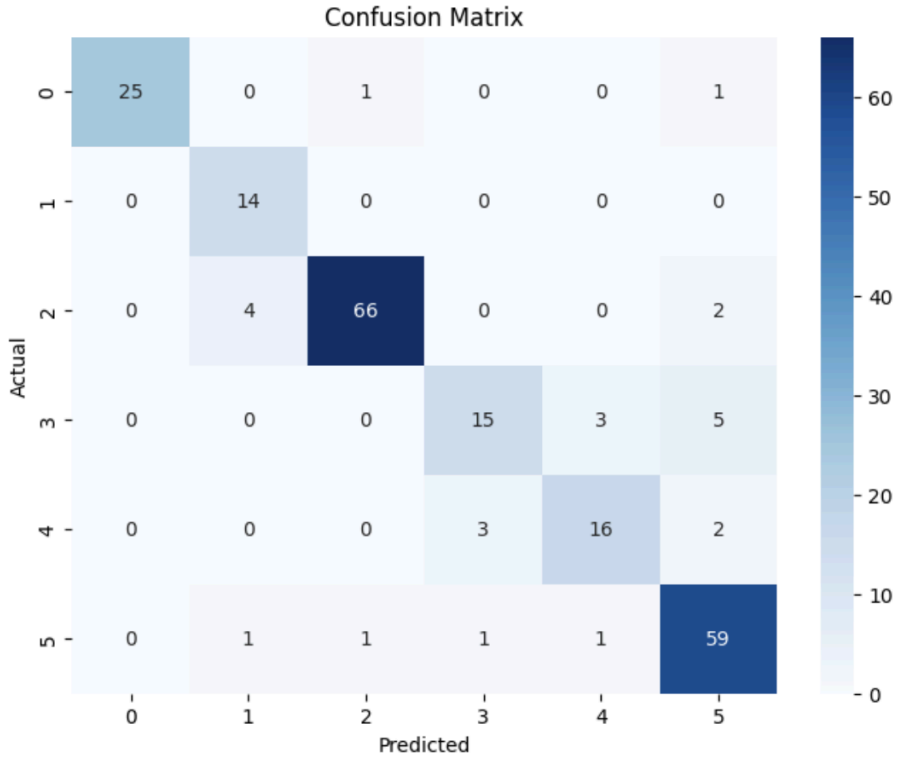


Figure 6.1.1b: Vanilla RNN Confusion Matrix

Labels 2,3 and 4 (i.e. waving, jumping and body movement) exhibit significant misclassifications in the model's predictions. Specifically, waving is frequently misclassified as bowing, jumping is often mistaken for body movement and walking, and there are a few other misclassifications as well. These patterns indicate that the model struggles to distinguish between these motion categories.

6.1.2 GRU

The scores indicate better performance than the Vanilla RNN, but still fall short of my desired scores of more than 94% in all metrics. As a result, I had to dismiss this model as well.

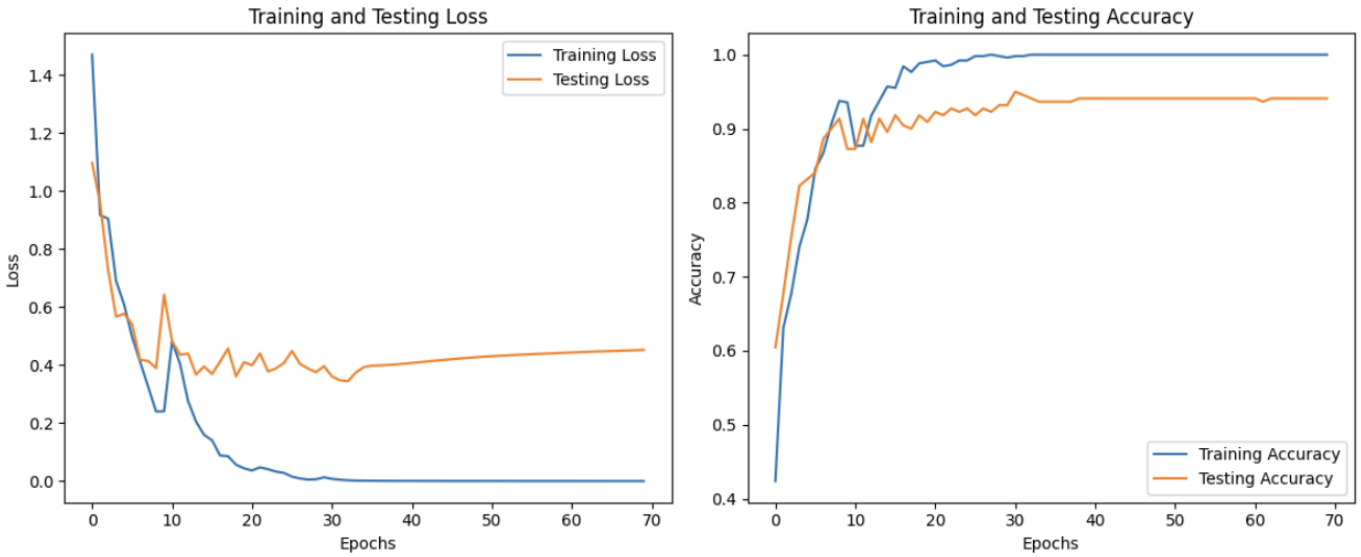


Figure 6.1.2a: GRU Training and Testing Curves

Similar to the Vanilla RNN, the training and testing loss curves suggest that the model is learning well over the epochs, evidenced by a steady decline in both training and testing losses. However, there is a large spike in testing loss around 10 epochs, indicating sharp sudden increases in error. Furthermore, the testing loss is initially lower than the training loss but surpasses it at around 5 epochs, and then remains higher than the training loss until the end of the training process, highlighting the same generalization issue that was seen with the Vanilla RNN and suggesting that the model may not properly capture the intrinsic patterns in the data. The training and testing accuracy

curves level off towards very early, and at a below satisfactory level of around 92%, which makes the model unreliable.

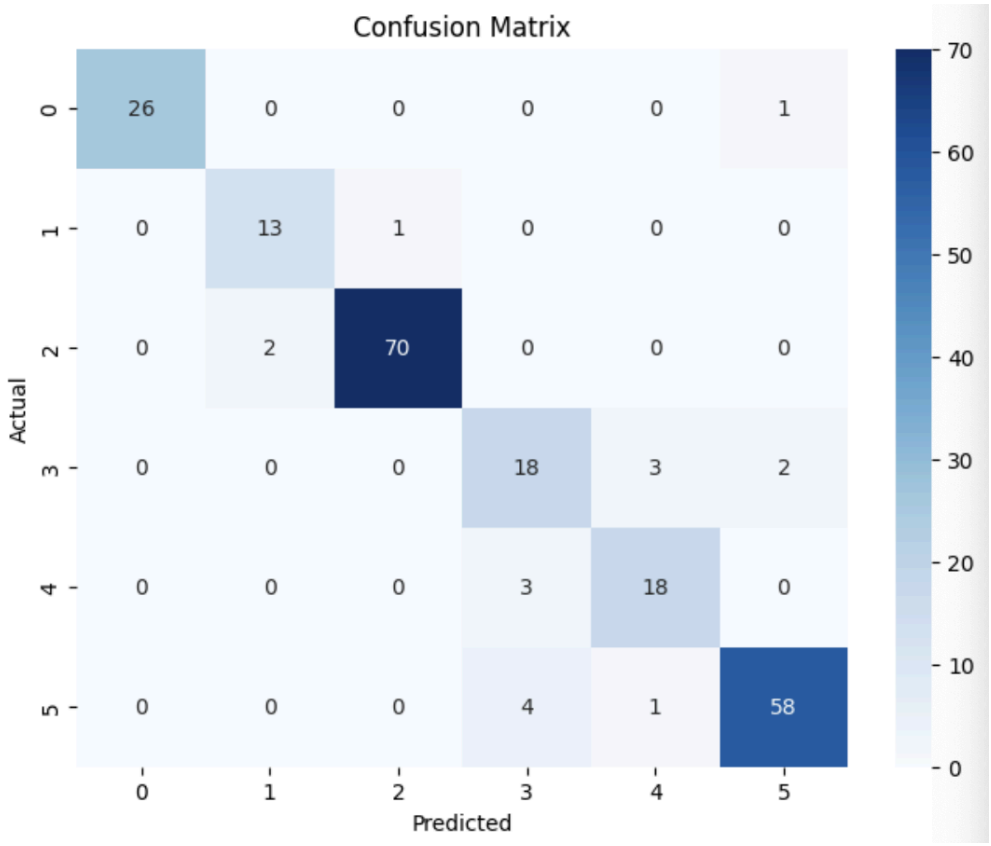


Figure 6.1.2b: GRU Confusion Matrix

Similar to Vanilla RNN, there are quite a few notable misclassifications. Waving (Label 2) is regularly incorrectly identified as bowing (Label 1), while jumping (Label 3) is often misclassified as body movement (Label 4) and walking (Label 5), and body movement (Label 4) and walking (Label 5) are frequently incorrectly identified as jumping (Label 3).

6.1.3 LSTM

The LSTM outperforms the other models across all performance metrics, demonstrating its robust ability to capture the complexities of the data and affirming its suitability for this project.

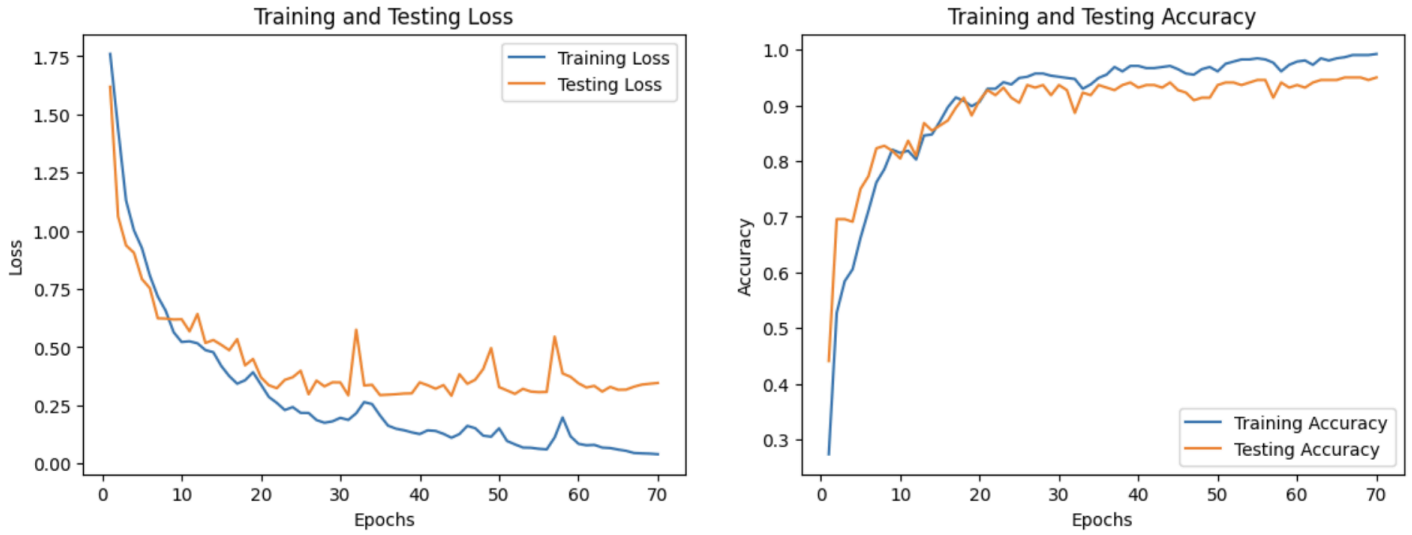


Figure 6.1.3a: LSTM Training and Testing Curves

The training and testing loss curves consistently decrease as the number of epochs increases, indicating that the model is learning effectively and minimizing errors over time. The training and testing accuracy curves rise to a high level and then stabilize, reflecting the model's ability to make accurate predictions and signifying successful training. This combination of decreasing loss and increasing accuracy highlight the model's prowess in capturing the underlying patterns in the data.

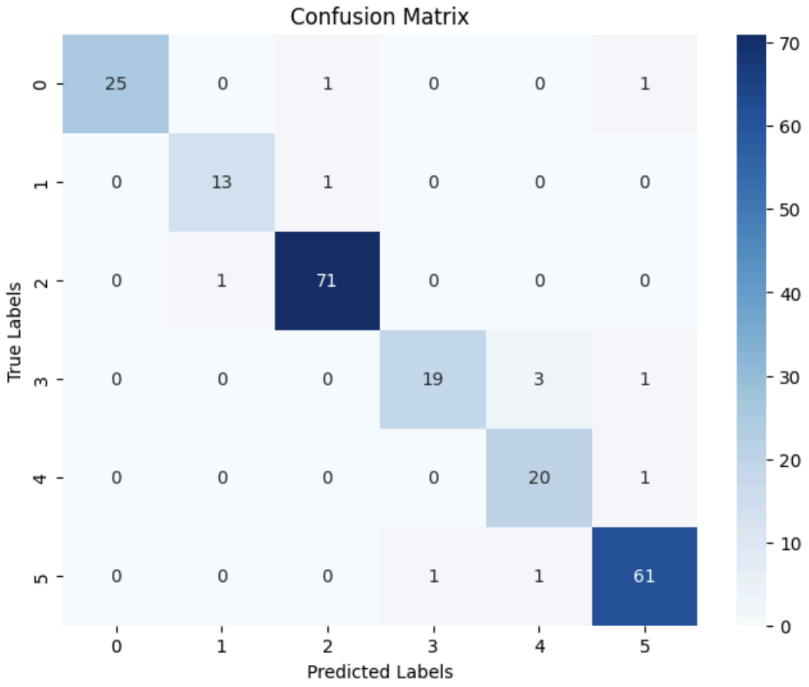


Figure 6.1.3b: LSTM Confusion Matrix

The confusion matrix further illustrates the efficacy of the model's training. There are no significant misclassifications, with the highest number of misclassifications only being a total of 4 for body movement (Label 4), indicating that the model reliably captures the nuances of the different motion categories.

Therefore, the LSTM emerges as the best option among all the variants. Its strength in retaining information over extended sequences makes it well-suited for complex tasks where context and dependencies between data points span long sequences.

7. Conclusion

7.1 Final Observations

From the testing conducted in Sections 6.1.1-6.1.3, it can be observed that the Vanilla RNN consistently exhibits the poorest performance across all metrics overall. The GRU performs significantly better than the Vanilla RNN, but is outperformed by the LSTM. This comparison highlights the importance of architectural design in neural networks, as the presence of gating mechanisms plays a crucial role in enhancing model performance.

7.1.1 Analysis

Based on my learning from the research papers and other resources I have extensively studied for this project, I can confidently make the following final concluding analysis:

The Vanilla RNN achieved the worst results due to its lack of any gates for properly managing the flow of information, leading to sudden bursts of training and testing loss, as well as very high losses overall. This instability results in poor generalization and as a result, the Vanilla RNN fails to adapt to the complexities of the data. Ultimately, the model fails to achieve satisfactory training and testing accuracies, and it also exhibits high training and testing loss values throughout the training process. The confusion matrix further highlights the Vanilla RNN’s inability to accurately distinguish between the different types of motion, as it makes a large number of misclassifications.

The GRU performs significantly better than the Vanilla RNN by tackling many of the issues faced by the Vanilla RNN with its Update and Reset Gates, but is unable to reach the training and testing accuracy levels of the LSTM due to the lack of a gate similar to the latter's Output Gate. As a result, while the GRU demonstrates significantly improved performance from the Vanilla RNN, it still struggles with tasks that require capturing intricate dependencies over time, as portrayed by notable misclassifications in its confusion matrix.

In contrast, the LSTM demonstrates excellent, consistent and robust performance across all metrics, with high training and testing accuracies and minimal training and testing losses. This superior performance underscores the LSTM's capability to retain relevant information and efficiently manage dependencies over long sequences, making it well-suited for tasks that require precise classification, and a perfect fit for this project.

Owing to its Input, Output, and Forget Gates, the LSTM can not only retain information over long sequences, but also forget undesired information. This ability allows the LSTM to maintain relevant context while discarding noise, which is crucial for accurate predictions in dynamic environments.

Therefore, the LSTM emerges as the best option among all the variants. Its strength in retaining information over extended sequences makes it well-suited for complex tasks where context and dependencies between data points span long sequences. Overall,

the LSTM’s robust architecture enables it to excel in challenging scenarios that require high accuracy and reliability, making it the preferred choice for my project.

7.1.2 Parameters Selection

In order to select the best set of hyperparameters, an extensive analysis was conducted where each hyperparameter was tested independently, i.e. when one hyperparameter was being tested, the others were kept constant, so that I could visualize and understand the effect of that specific hyperparameter on the LSTM’s accuracy and performance. The following hyperparameters were tested:

Number of Layers (num_layers): Increasing the number of layers can enhance the model's capacity to learn complex patterns, but excess layers could lead to overfitting. I regulated between 1,2 and 3 layers to find the right balance between the two factors.

Number of Features (num_features): Adjusting the number of features affects the model's ability to capture relevant information from the input data. I tested between 4 (x, y, z, v) and 6 features (x, y, z, v, snr, noise) to determine the optimal configuration that maximizes accuracy.

Input Size (input_size): Changing the input size influences the model’s ability to learn temporal dependencies. The input size changed with the number of features: for num_features=4, the input size was 88, and for num_features=6, it was 132.

Hidden Size (hidden_size): The hidden size determines the number of units in the LSTM layers, impacting the model's capacity to retain information. I tested between 64, 128 and 256 to find the best value for determining the model’s capability in pattern recognition.

Number of Epochs (num_epochs): Varying the number of epochs helps find the right balance between convergence and overfitting. I tested different epoch counts (10, 20, 30, 50, 70, 100) to ensure the model learned adequately without overfitting.

Training-Testing Split (test_size): Adjusting the training-testing split impact can impact the model's generalization ability. I tested between the ratios 50:50, 60:40 and 70:30 (i.e., test_size=0.5, 0.4 and 0.3, respectively) to determine the ideal test size and data allocation ratio.

Random State (random_state): The random state affects the initialization of weights and the shuffling of data, which can influence model performance and reproducibility. I experimented with different random states (30, 40, 50) to assess their impact on the stability and consistency of model results.

Learning Rate (lr): Testing different learning rates helps find the optimal speed for the model to train, balancing stable learning and avoiding overshooting. I adjusted the

learning rate between the values 0.1, 0.01, 0.001 and 0.0001 to identify the most suitable value for training stability.

Batch Size (batch_size): The batch size influences the model's training dynamics and convergence speed. I tested batch sizes 16, 32 and 64 to determine the right balance between training time and model performance.

Dropout Rate (dropout_rate): Dropout is used to prevent overfitting by randomly setting a fraction of the input units to zero during training. I tested various dropout rates (0.2, 0.4, 0.5) to find the optimal balance between model generalization and retention of meaningful patterns.

Initial Settings: num_layers = 1, num_features = 4, input_size = 88, hidden_size = 128, num_epochs = 20, test_size = 0.3, random state = 30, lr=0.001, batch_size = 64, dropout_rate = 0.2.

The results were as follows:

num_layers	num_features	input_size	hidden_size	num_epochs	test_size	random_state	lr	batch_size	dropout_rate	Accuracy (%)	Precision	F1 Score
3	4	88	128	10	0.3	30	0.001	64	0.2	91.16	0.9193	0.9143
3	4	88	128	20	0.3	30	0.001	64	0.2	91.36	0.9213	0.9153
3	4	88	128	30	0.3	30	0.001	64	0.2	92.27	0.93	0.9243
3	4	88	128	50	0.3	30	0.001	64	0.2	92.73	0.9324	0.9288
3	4	88	128	70	0.3	30	0.001	64	0.2	95	0.9519	0.9499
3	4	88	128	100	0.3	30	0.001	64	0.2	91.36	0.921	0.915
3	4	88	128	20	0.3	30	0.001	64	0.2	91.36	0.9213	0.9153
3	4	88	128	20	0.3	30	0.001	64	0.4	90.91	0.913	0.9094
3	4	88	128	20	0.3	30	0.001	64	0.5	88.18	0.909	0.8865
3	4	88	128	20	0.3	30	0.1	64	0.2	91.36	0.9213	0.9153
3	4	88	128	20	0.3	30	0.01	64	0.2	6.36	0.004	0.0076
3	4	88	128	20	0.3	30	0.001	64	0.2	92.73	0.9324	0.9288
3	4	88	128	20	0.3	30	0.0001	64	0.2	75	0.6091	0.6692
3	4	88	128	20	0.3	30	0.001	16	0.2	88.75	0.8985	0.8956
3	4	88	128	20	0.3	30	0.001	32	0.2	90.45	0.9127	0.912
3	4	88	128	20	0.3	30	0.001	64	0.2	91.36	0.9213	0.9153
3	4	88	128	20	0.3	30	0.001	64	0.2	91.36	0.9213	0.9153
3	4	88	128	20	0.3	40	0.001	64	0.2	89.5	0.9024	0.898
3	4	88	128	20	0.3	50	0.001	64	0.2	90.75	0.9148	0.9105
3	4	88	128	20	0.3	30	0.001	64	0.2	91.36	0.9213	0.9153
3	4	88	128	20	0.4	30	0.001	64	0.2	88.85	0.8952	0.892
3	4	88	128	20	0.5	30	0.001	64	0.2	87.6	0.8881	0.8845
3	4	88	128	20	0.3	30	0.001	64	0.2	91.36	0.9213	0.9153
3	4	88	256	20	0.3	30	0.001	64	0.2	89.2	0.9035	0.9015
1	4	88	128	20	0.3	30	0.001	64	0.2	85.9	0.8705	0.865
2	4	88	128	20	0.3	30	0.001	64	0.2	89.65	0.895	0.8915
3	4	88	128	20	0.3	30	0.001	64	0.2	91.36	0.9213	0.9153
3	4	88	128	20	0.3	30	0.001	64	0.2	91.36	0.9213	0.9153
3	6	132	128	20	0.3	30	0.001	64	0.2	89.75	0.9032	0.898

Table 7.1.2: Excel Table Showing Hyperparameter Adjustment Testing Results

The highlighted Accuracy, Precision and F1 Score rows represent the best results for each hyperparameter, and these rows are colour-coordinated to match the hyperparameter that was being tested.

Based on the results obtained, the following hyperparameters were determined to be the best possible set for training the LSTM model: num_layers = 3, num_features = 4, input_size = 88, hidden_size = 128, num_epochs = 70, test_size = 0.3, random state = 30, lr=0.001, batch_size = 64, dropout_rate = 0.2.

These hyperparameters are a well-balanced set, ensuring that the model is not too simple, but at the same time, does not suffer from overfitting. Using 70 epochs ensures that the model has sufficient time to converge without risking unfinished learning or overfitting. Additionally, the dropout rate helps mitigate overfitting by promoting generalization, while the learning rate and batch size are set to values that support efficient and stable training. Overall, these hyperparameters create an optimal environment for the model to achieve robust performance.

All the previously chosen hyperparameters seem to be the best options, with the exception of num_layers and num_epochs.

Number of Layers (num_layers): num_layers = 3 allows the model to learn hierarchical representations effectively by capturing intricate patterns in the data. Lower values show inferior results due to insufficient capacity

Number of Epochs (num_epochs): num_epochs = 70 strikes a balance between learning and avoiding overfitting. At lower epochs, the model does not have sufficient iterations to learn the underlying patterns in data while at higher epochs, the model suffers from overfitting.

Number of Features (num_features) and Input Size (input_size): num_features = 4 and input_size = 88 is the right balance for learning from the data. num_features = 6 and

input_size = 132 introduces unnecessary complexity and noise, making it harder for the model to generalize proficiently.

Hidden Size (hidden_size): hidden_size=128 provides adequate capacity to capture complex patterns without overfitting. A larger hidden size leads to excessive complexity, resulting in memorization of training data rather than generalization, and worsening performance.

Training-Testing Split (test_size): test_size = 0.3 allows an adequate amount of data for training while maintaining a robust set for evaluation. Larger test sizes reduce the training data too much, adversely affecting the model's ability to learn efficiently.

Random State (random_state): random_state = 30 yields consistent and reliable results. Higher random states reduce all performance metrics due to excess variability which hampers the model's ability to generalize.

Learning Rate (lr): The optimal learning rate is 0.001. Higher and lower rates lead to lower performance scores due to overshooting optimal weights and slow convergence, respectively. A learning rate of 0.001 strikes a balance between stable convergence and efficient learning.

Batch Size (batch_size): batch_size = 64 is ideal as it provides a good balance between training stability and memory efficiency. Higher batch sizes provide unsatisfactory results due to noisy gradient estimates.

Dropout Rate (dropout_rate): dropout_rate = 0.2 is best suited to allow the model to learn properly, while mitigating overfitting by preventing the model from becoming too reliant on specific neurons. Higher dropout rates remove too many neurons, hindering learning and lowering performance.

7.1.3 Testing LSTM Model on New Unseen Data

Upon testing the LSTM Model on new files that were not included in the training dataset, and thus not used for training the model, the following results were obtained:

```
# Folder containing CSV files
folder_path = '/Users/zuhayrrahman/Documents/Semester B 24 25/CS4514 Project/Project/My Data/Final (Motion & N
# motion prediction
predict_motion_in_folder(folder_path, model)
print(' ')
```

Python

```
File: xwr68xx_processed_stream_2025_03_16T20_03_51_953.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_04_31_159.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_06_16_314.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_07_21_805.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_06_37_582.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_06_44_790.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_03_59_207.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_06_30_516.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_06_57_282.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_03_21_477.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_04_54_068.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_03_28_605.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_05_56_990.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_03_36_109.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_04_25_052.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_05_26_276.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_06_51_248.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_03_45_730.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_03_14_270.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_06_23_662.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_04_12_843.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_05_18_454.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_07_07_893.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_05_12_051.csv -> Predicted Motion: No Motion (Label - 0)
File: xwr68xx_processed_stream_2025_03_16T20_05_32_017.csv -> Predicted Motion: No Motion (Label - 0)
...
No Motion: 100.00%

No files were skipped.

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Figure 7.1.3a: Testing On New No Motion Files

```
# Folder containing CSV files
folder_path = '/Users/zuhayrrahman/Documents/Semester B 24 25/CS4514 Project/Project/My Data/Final (Motion & N
# motion prediction
predict_motion_in_folder(folder_path, model)
print(' ')
```

✓ 2.2s Python

```
File: 46_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 53_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 7_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 22_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 37_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 8_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 38_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 49_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 54_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 41_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 30_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 25_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 40_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 55_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 1_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 24_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 31_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 39_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 9_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 48_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 52_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 47_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 36_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 23_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
File: 6_bowing csv.csv -> Predicted Motion: Bowing (Label - 1)
...
Bowing: 100.00% (71/71)

No files were skipped.
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Figure 7.1.3b: Testing On New Bowing Files

```python
# Folder containing CSV files
folder_path = '/Users/zuhayrrahman/Documents/Semester B 24 25/CS4514 Project/Project/My Data/Final (Motion & N
# motion prediction
predict_motion_in_folder(folder_path, model)
print(' ')
```

✓ 1.1s Python

```
File: xwr68xx_processed_stream_2025_03_16T19_15_00_939.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_14_23_557.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_38_08_494.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_42_28_068.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_13_13_364.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_14_01_979.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_38_13_962.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_13_53_797.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_38_01_279.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_37_48_615.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_41_29_882.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_13_30_581.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_14_54_945.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_37_38_891.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_15_33_702.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_14_37_014.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_13_38_396.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_41_38_274.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_41_05_289.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_13_46_128.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_15_42_881.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_13_07_132.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_14_17_579.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T19_13_26_404.csv -> Predicted Motion: Waving (Label - 2)
File: xwr68xx_processed_stream_2025_03_16T18_37_54_331.csv -> Predicted Motion: Waving (Label - 2)
...
Waving: 100.00%

No files were skipped.
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Figure 7.1.3c: Testing On New Waving Files

```python
# Folder containing CSV files
folder_path = '/Users/zuhayrrahman/Documents/Semester B 24 25/CS4514 Project/Project/My Data/Final (Motion &
# motion prediction
predict_motion_in_folder(folder_path, model)
print(' ')
```

✓ 1.8s Python

```
File: 240_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 207_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 223_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 184_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 211_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 256_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 192_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 208_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 232_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 248_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 195_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 251_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 216_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 183_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 224_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 200_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 247_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 249_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 194_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 233_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 217_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 250_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 225_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 218_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
File: 182_jumping csv.csv -> Predicted Motion: Jumping (Label - 3)
...
Jumping: 100.00% (75/75)
```

Figure 7.1.3d: Testing On New Jumping Files

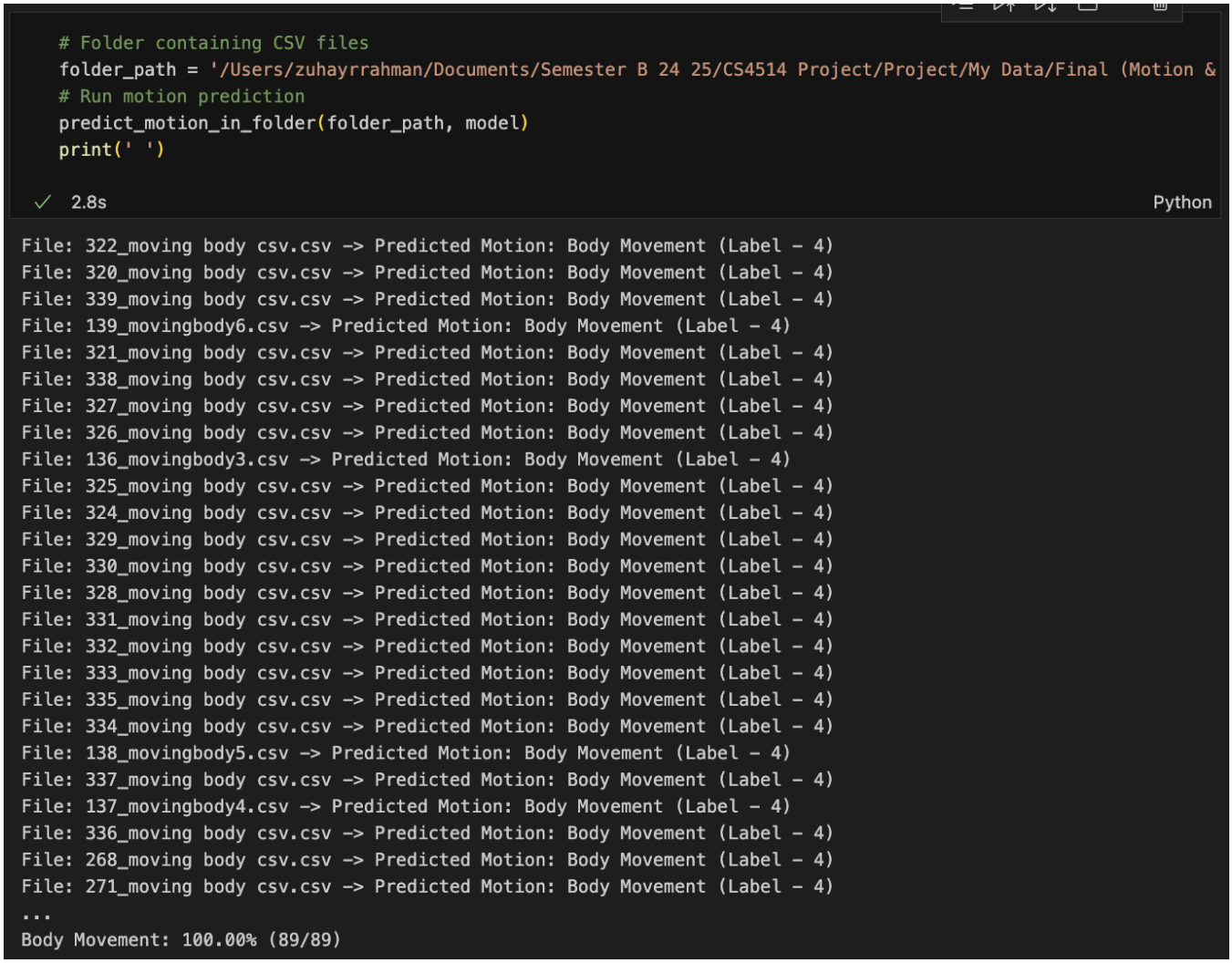


Figure 7.1.3e: Testing On New Body Movement Files

```
# Folder containing CSV files
folder_path = '/Users/zuhayrrahman/Documents/Semester B 24 25/CS4514 Project/Project/My Data/Final (Motion &
# Run motion prediction
predict_motion_in_folder(folder_path, model)
print(' ')
```

✓ 1.7s Python

```
File: 150_walking7.csv -> Predicted Motion: Walking (Label - 5)
File: 144_walking1.csv -> Predicted Motion: Walking (Label - 5)
File: 23_walking.csv -> Predicted Motion: Walking (Label - 5)
File: 152_walking9.csv -> Predicted Motion: Walking (Label - 5)
File: 169_walking26.csv -> Predicted Motion: Walking (Label - 5)
File: 179_walking36.csv -> Predicted Motion: Walking (Label - 5)
File: 184_walking41.csv -> Predicted Motion: Walking (Label - 5)
File: 187_walking44.csv -> Predicted Motion: Walking (Label - 5)
File: 156_walking13.csv -> Predicted Motion: Walking (Label - 5)
File: 18_walking.csv -> Predicted Motion: Walking (Label - 5)
File: 36_walking.csv -> Predicted Motion: Walking (Label - 5)
File: 155_walking12.csv -> Predicted Motion: Walking (Label - 5)
File: 183_walking40.csv -> Predicted Motion: Walking (Label - 5)
File: 33_walking.csv -> Predicted Motion: Walking (Label - 5)
File: 26_walking.csv -> Predicted Motion: Walking (Label - 5)
File: 145_walking2.csv -> Predicted Motion: Walking (Label - 5)
File: 14_walking.csv -> Predicted Motion: Walking (Label - 5)
File: 17_walking.csv -> Predicted Motion: Walking (Label - 5)
File: 172_walking29.csv -> Predicted Motion: Walking (Label - 5)
File: 158_walking15.csv -> Predicted Motion: Walking (Label - 5)
File: 25_walking.csv -> Predicted Motion: Walking (Label - 5)
File: 163_walking20.csv -> Predicted Motion: Walking (Label - 5)
File: 173_walking30.csv -> Predicted Motion: Walking (Label - 5)
File: 180_walking37.csv -> Predicted Motion: Walking (Label - 5)
File: 30_walking.csv -> Predicted Motion: Walking (Label - 5)
...
Walking: 100.00% (77/77)
```

Figure 7.1.3f: Testing On New Walking Files

The results highlight the trained LSTM Model's impressive capability in accurately classifying the 5 types of motion, as well as no motion. This performance underscores the model's robustness and effectiveness in generalizing unseen data, highlighting its potential for real-world applications in motion classification tasks.

7.2 Challenges

7.2.1 Sensor Installation and Setup

Setting up the AWR6843ISK mmWave Radar Sensor Evaluation Kit and DCA1000EVM Evaluation Module took significantly longer than anticipated. The official documentation provided by Texas Instruments is neither concise nor well-organized. A lot of trial and error led me to discover that my laptop is incompatible for using mmWave Studio due to its lack of an Ethernet port. The DCA1000EVM must be connected directly via an ethernet port rather than through an adapter, as the latter fails to establish a secure connection. Even with a secure connection between the mmWave Radar Sensor and my laptop, though, I struggled more often than not to establish a connection.

I then tried capturing data using another laptop equipped with an Ethernet port; however, I faced various issues with this option as well, mainly due to the fact that it is a very old laptop and is extremely slow, taking around 10 minutes to even power on, with basic operations such as clicking buttons or searching for files taking an excessively long time. Consequently, this option proved to be unfeasible as well. This entire failed setup process cost me two to three weeks of the limited time I had the mmWave Radar in my possession, resulting in significantly less time available for data collection. I then spent another week or two finding and setting up the alternate solution, mmWave Demo Visualizer, which requires just the mmWave Radar Sensor without the DCA1000EVM Evaluation Module.

7.2.2 Dataset Collection

I managed to collect around 1,000 individual datasets; however, I believe this is still insufficient for optimally training the LSTM model. Due to time constraints and the tedious nature of data collection—clicking the “Record Start” button each time, executing the same motion repeatedly (such as jumping 4-5 times for each instance of motion collection), and repeating this process 80-100 times—along with saving the .dat and .cfg files and converting all the .dat files into .csv files, I was unable to collect more data.

Additionally, the dataset capture was limited to a single subject, myself, and a single environment, my dorm room. Setting up the laptop and mmWave Radar Sensor in an outdoor environment was not practical as my laptop does not last very long on battery power, and I also did not have any mounting stand to securely place the mmWave Radar Sensor in, so there was an additional risk of damaging the radar if I took it outside, and that prevented me from expanding the dataset further.

7.3 Future Plans and Development

7.3.1 Enriched Dataset

As noted in Section 7.2.2, I found my initial dataset to be unsatisfactory. Consequently, I plan to create a comprehensive dataset that encompasses a greater variety of subjects, a wide variety of both interior and exterior environments, and an expanded range of motion and no motion types. This will greatly enhance the quality and robustness of the data for training the LSTM model, and make the LSTM model much more capable of detecting and classifying a wide array of motion and non-motion scenarios.

7.3.2 Real-Time Detection

By enhancing the automation script, I plan on automating the entire process—from recording motion to outputting the predictions made by the LSTM model—so that all of it can be executed with just one click. This improvement will eliminate the need to manually save the .dat and .cfg files every time, significantly increasing the system's efficiency and making it much more adept at providing real-time motion detection.

7.3.3 Additional Derived Features

The LSTM could be trained more effectively to further enhance its performance by incorporating a greater number of complex derived features, such as the relative distance and change in velocity that I used to train the LSTM Model in this project.
By deriving intricate features such as acceleration and angular velocity, I plan on training the LSTM model more effectively. This approach will enable the model to learn

significantly more complex and intricate patterns in the data, and greatly improve its performance.

8. Project Schedule

8.1 Milestones

- Project Planning and Setup
- Data Exploration
- Development of RNN Model
- RNN Model Optimization
- System Integration and Testing
- Performance Evaluation: Reporting and Documentation
- Prepare and Submit Final Report and Presentation

8.2 Gantt Chart

9. Monthly Logs

October: I dedicated this month to deepening my understanding of the mechanism of the mmWave motion detection system, including how the radar utilizes chirps, the role of Fast Fourier Transform, and how the sensor differentiates between objects. This has helped me understand the sensor's mechanism, and will be crucial in the upcoming stages of my project.

November: I started researching into the various machine learning algorithms, such as the Random Forest (RF) Model and Recurrent Neural Network (RNNs). Initially, I had decided to use the RF Model for my project, and wrote about my findings regarding said model in Interim Report I.

December: Further research took me to the conclusion that the Recurrent Neural Network would be more suitable for my project, so I began looking into how to create a motion detection system based on that.

January: Finding datasets online due to not being able to get the radar before February, processing the data to make it suitable for training the RNN Model.

February: Initial training and testing on RNN Model, preparation of Interim Report II based on initial findings.

March: mmWave Radar Sensor installation and setup, data collection using the mmWave Radar Sensor, data processing, preparation of input matrix for RNN Model, testing and comparison on Vanilla RNN, GRU and LSTM, further testing on LSTM, preparation of the Final Report.

April: Fine-tuning the Final Report

References

Abdu, F.J.; Zhang, Y.; Fu, M.; Li, Y.; Deng, Z. (2021). Application of Deep Learning on Millimeter-Wave Radar Signals: A Review. *Sensors* 2021, *21*(6), 1951. https://doi.org/10.3390/s21061951

Airat, Z., Sakhabutdinov, E. P., Denisenko, P. E., Denisenko, A. A., Lustina, V. D., & Andreev, A. (2022). Radiophotonic Method for Doppler Frequency Shift Measurement of a Reflected Radar Signal Based on Tandem Amplitude-Phase Modulation. *Proceedings of the IEEE Conference*, 1-5. https://doi.org/10.1109/IEEECONF53456.2022.9744338

Ameen, A., M., Liu, J., & Kwak, K. (2010). Security and Privacy Issues in Wireless Sensor Networks for Healthcare Applications. *Journal of Medical Systems, 36*(1), 93-101. https://doi.org/10.1007/s10916-010-9449-4

Andrews, J., Kowsika, M., Vakil, A., & Li, J. (2020, April). A motion induced passive infrared (PIR) sensor for stationary human occupancy detection. In *2020 IEEE/ION position, location and navigation symposium (PLANS)* (pp. 1295-1304). IEEE.

Aydin, I., & Othman, N. A. (2017). A New IoT Combined Face Detection of People by Using Computer Vision for Security Application. In *2017 International Artificial*

Intelligence and Data Processing Symposium (IDAP) (pp. 1-6). IEEE. https://doi.org/10.1109/IDAP.2017.8090171

Chandra, N., Ahuja, L., Khatri, S. K., & Monga, H. (2021). Utilizing gated recurrent units to retain long term dependencies with recurrent neural network in text classification. *Journal of Information Systems and Telecommunication*, *9*(34), 89-102. https://doi.org/10.52547/jist.9.34.89

Chen, X., Zhang, X., Xia, Q., Fang, X., Lu, C. X., & Li, Z. (2023). Differentiable Radio Frequency Ray Tracing for Millimeter-Wave Sensing. *arXiv preprint arXiv:2311.13182*.https://doi.org/10.48550/arxiv.2311.13182

Choubisa, T., Upadrashta, R., Panchal, S., Praneeth, A., Ranjitha, H. V., Senthoor, K., Bhattacharya, A., Anand, S. V. R., Hegde, M., Kumar, A., Kumar, P. V., Iyer, M. S., Sampath, A., Prabhakar, T. V., Kuri, J., & Singh, A. N. (2016). Challenges in Developing and Deploying a PIR Sensor-Based Intrusion Classification System for an Outdoor Environment. In *2016 IEEE 41st Conference on Local Computer Networks Workshops (LCN Workshops)* (pp. 148-155). IEEE. https://doi.org/10.1109/LCN.2016.041

Damarla, T., & Mehmood, A. (2013). Detection of targets using distributed multi-modal sensors with correlated observations. *2013 IEEE SENSORS*, 1-4. https://doi.org/10.1109/ICSENS.2013.6688471.

Danei Gong, pc

Doctor, A. P. (1994). Passive Infrared Motion Sensing Technology. *Proceedings of SPIE, 2217*, Aerial Surveillance Sensing Including Obscured and Underground Object Detection. https://doi.org/10.1117/12.179959

Fanti, M. P., Roccotelli, M., Faraut, G., & Lesage, J. J. (2017). Smart Placement of Motion Sensors in a Home Environment. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)* (pp. 894-899). IEEE. https://doi.org/10.1109/SMC.2017.8122723

Frankiewicz, A., & Cupek, R. (2013). Smart passive infrared sensor - Hardware platform. *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, 7543-7547. https://doi.org/10.1109/IECON.2013.6700389.

Hong, S., Kim, N., & Kim, W. (2013). Reduction of False Alarm Signals for PIR Sensor in Realistic Outdoor Surveillance. *ETRI Journal*, 35. https://doi.org/10.4218/etrij.13.0112.0219.

Iovescu, C., & Rao, S. (2017). The fundamentals of millimeter wave sensors. *Texas Instruments*, 1-8. https://www.ti.com/lit/wp/spyy005a/spyy005a.pdf?ts=1731125694089

Jardak, S., Alouini, M., Kiuru, T., Metso, M., & Ahmed, S. (2019). Compact mmWave FMCW radar: Implementation and Performance analysis. *IEEE Aerospace and*

Electronic Systems Magazine, 34, 36-44. https://doi.org/10.1109/MAES.2019.180130.

Kavuncuoğlu, E., Uzunhisarcıklı, E., Barshan, B., & Özdemir, A. T. (2021). Investigating the Performance of Wearable Motion Sensors on recognizing falls and daily activities via machine learning. *Digital Signal Processing, 126,* 103365. https://doi.org/10.1016/j.dsp.2021.103365

Li, W., Zhang, D., Li, Y., Wu, Z., Chen, J., Zhang, D., Hu, Y., Sun, Q. & Chen, Y. (2022). Real-Time Fall Detection Using MmWave Radar. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 16-20). IEEE.* https://doi.org/10.1109/ICASSP43922.2022.9747153

Liddiard, K. (2007). Technology for low cost PIR security sensors. , 6835. https://doi.org/10.1117/12.758499.

Likhitha, K., Malineni, S., Jampani, N., & Prasanna, N. (2019). Home Security System Using PIR Sensor-IoT. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*. https://doi.org/10.32628/CSEIT195272.

Manaswi, N. K. (2018). RNN and LSTM. In *Deep Learning with Applications Using Python* (pp. 115–126). Apress L. P. https://doi.org/10.1007/978-1-4842-3516-4_9

Maiwald, T., Gabsteiger, J., Weigel, R., & Lurz, F. (2024). Gesture Recognition to Control a Moving Robot With FMCW Radar. In *2024 IEEE Radio and Wireless Symposium (RWS)* (pp. 105-108). IEEE. https://doi.org/10.1109/RWS56914.2024.10438564

Mukhopadhyay, B., Srirangarajan, S., & Kar, S. (2018). Modeling the analog response of passive infrared sensor. *Sensors and Actuators A: Physical*. https://doi.org/10.1016/J.SNA.2018.05.002.

Naaraju, P., & Sadanandam, M. (2024). Advancements in Motion Detection Within Video Streams Through the Integration of Optical Flow Estimation and 3D-Convolutional Neural Network Architectures. *Journal of Electrical Systems*, *20*(6s), 2502-2517. https://doi.org/10.52783/jes.3238.

Narayana, S., Prasad, R. V., Rao, V. S., Prabhakar, T. V., Kowshik, S. S., & Iyer, M. S. (2015). PIR sensors: Characterization and novel localization technique. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN '15)* (pp. 142–153). Association for Computing Machinery. https://doi.org/10.1145/2737095.2742561.

Palipana, S., Salami, D., Leiva, L. A., & Sigg, S. (2021). Pantomime: Mid-air gesture recognition with sparse millimeter-wave radar point clouds. *Proceedings of the ACM on interactive, mobile, wearable and ubiquitous technologies*, *5*(1), 1-27. https://doi.org/10.1145/3448110

Piotrowsky, L., Kueppers, S., Jaeschke, T., & Pohl, N. (2022). Distance Measurement Using mmWave Radar: Micron Accuracy at Medium Range. *IEEE Transactions on Microwave Theory and Techniques*, 70, 5259-5270. https://doi.org/10.1109/TMTT.2022.3195235.

Rao, S. (2017). Introduction to mmWave sensing: FMCW radars. *Texas Instruments (TI) mmWave Training Series*, 1-11.https://www.ti.com/content/dam/videos/external-videos/zh-tw/2/3816841626001/5415203482001.mp4/subassets/mmwaveSensing-FMCW-offlineviewing_0.pdf

Schmidt, R. M. (2019). Recurrent neural networks (rnns): A gentle Introduction and Overview. *arXiv preprint arXiv:1912.05911.* https://doi.org/10.48550/arXiv.1912.05911

Sharanbasappa, A., Kumar, S., B., Naveen, A., Praveen, M., K., & Raj, B., T. (2023). Automatic Control of LED Lamp Using PIR Motion Sensor. *International Journal for Research in Applied Science and Engineering Technology*. https://doi.org/10.22214/ijraset.2023.51379.

Song, B., Choi, H., & Lee, H. S. (2008). Surveillance Tracking System Using Passive Infrared Motion Sensors in Wireless Sensor Network. In *2008 International Conference on Information Networking* (pp. 1-5). IEEE. https://doi.org/10.1109/ICOIN.2008.4472790.

Soumya, A., Krishna Mohan, C., & Cenkeramaddi, L. R. (2023). Recent Advances in mmWave-Radar-Based Sensing, Its Applications, and Machine Learning Techniques: A Review. *Sensors, 23*(21), 8901. https://doi.org/10.3390/s23218901

Tjandra, A., Sakti, S., Manurung, R., Adriani, M., & Nakamura, S. (2016). Gated recurrent neural tensor network. In *2016 International Joint Conference on Neural Networks (IJCNN)* (pp. 448-455). IEEE. https://doi.org/10.1109/IJCNN.2016.7727233

Venon, A., Dupuis, Y., Vasseur, P., & Merriaux, P. (2022). Millimeter Wave FMCW RADARs for Perception, Recognition and Localization in Automotive Applications: A Survey. *IEEE Transactions on Intelligent Vehicles*, 7, 533-555. https://doi.org/10.1109/TIV.2022.3167733.

Werbos, P. (1990). Backpropagation Through Time: What It Does and How to Do It. *Proc. IEEE*, 78, 1550-1560. https://doi.org/10.1109/5.58337.

Yao, K., Cohn, T., Vylomova, K., Duh, K., & Dyer, C. (2015). Depth-gated LSTM. *arXiv preprint arXiv:1508.03790*. https://doi.org/10.48550/arXiv.1508.03790

Yong, C. Y., Sudirman, R., & Chew, K. M. (2011). Motion Detection and Analysis with Four Different Detectors. In *2011 Third International Conference on Computational Intelligence, Modelling & Simulation* (pp. 46-50). IEEE. https://doi.org/10.1109/CIMSim.2011.18

Zargar, S.A. (2021). Introduction to sequence learning models: RNN, LSTM, GRU. *Department of Mechanical and Aerospace Engineering, North Carolina State University*. https://doi.org/10.13140/RG.2.2.36370.99522