CG4002 Capstone Project

January 2020 semester

"Dance Dance"
Final Report

Group 6	Name	Student Number	Sub-Team	Specific Contributions
Member #1	Lim Yi Sheng	A0168536U	HW1: Sensors	Section 1 Section 2.1, 2.2.5 Section 3 Section 10
Member #2	Soh Boon Jun	A0168082B	HW2: FPGA	Section 1 Section 2.2.6 Section 4 Section 10
Member #3	Melvin Tan	A0164512M	Comm1: Internal	Section 2.2.2 - 2.2.4 Section 5 Section 10
Member #4	Jin Shuyuan	A0162475B	Comm2: External	Section 2.2.1- 2.2.4 Section 6 Section 10
Member #5	Davindran S/O Sukumaran	A0167009E	SW1: Machine Learning	Section 7 Section 10
Member #6	Gerald Chua Deng Xiang	A0167371B	SW2: Dashboard	Section 8 Section 9 Section 10

Table of Contents

Table of Contents 3

Glossary 8

Section 1 System Functionalities 9

Section 1.1 Use Cases 9

Section 1.2 Feature List 10

Section 2 Overall System Architecture 10

Section 2.1 Design of the System 10

Section 2.1.1 Initial Form of our System 10

Section 2.1.2 Final Form of our System 11

Section 2.1.3 Reasons behind the design changes 14

Section 2.2 High-Level System Implementation 16

Section 2.2.1 High-Level System Architecture 16

Section 2.2.2 Communication Processes 17

Section 2.2.3 Implementation on Beetle and Ultra96 18

Section 2.2.4 Communication Protocols 20

Section 2.2.5 Hardware Components 21

Section 2.2.6 Machine Learning High Level Algorithm 22

Section 3 Hardware Details: Hardware 1 - Sensors 24

Section 3.1 Introduction 24

Section 3.2 Brief Overview 24

Section 3.3 Hardware Components/Devices 25

Section 3.3.1 Beetle V1.1 25

Section 3.3.2 LM7805 Voltage Regulator 26

Section 3.3.3 IMU 26

Section 3.3.4 MyoWare Muscle Sensor 26

Section 3.3.5 Ultra96-V2 27

Section 3.3.6 XL4015 DC/DC Step-down Converter 27

Section 3.4 Pin Table 27

Section 3.4.1 Beetle to IMU 28

Section 3.4.2 Beetle to MyoWare Muscle Sensor 29

Section 3.4.3 Beetle to LM7805 Voltage Regulator 30

Section 3.5 Schematics 30
Section 3.5.1 Beetle to IMU 31
Section 3.5.2 Beetle to MyoWare Muscle Sensor 32
Section 3.5.3 LM7805 Voltage Regulator Circuitry 33
Section 3.5.4 XL4015 Circuitry 34
Section 3.6 Operational Voltage and Current 34
Section 3.6.1 Beetle 34
Section 3.6.2 IMU 34
Section 3.6.3 MyoWare Muscle Sensor 35
Section 3.6.4 Ultra96-V2 Development Board 35
Section 3.7 Algorithms and Libraries 35
Section 3.7.1 IMU 35
Section 3.7.2 Self-written Thresholding Algorithm 36
Section 3.7.3 EMG Time Features Extraction 38
Section 3.7.4 EMG Frequency Features Extraction 41
Section 3.8 Changes made 43
Section 3.8.1 Report changes 43
Section 3.8.2 Design changes 44
Section 3.8.3 Sensors algorithm changes 48
Section 3.9 Research: Extension of Hardware 1 Sensors 48
Section 3.9.1 Further improvements on hardware components 49
Section 3.9.2 Realizing hardware components (Physiotherapy exercises) 50

Section 4 Hardware Details: Hardware 2 - FPGA 51
Section 4.1 Introduction 51
Section 4.2 Hardware Acceleration with hls4ml tool (Before week 7) 51
Section 4.3 Hardware Acceleration of Quantized Neural Network with FINN 57
Section 4.4 Hardware Acceleration Evaluation Metrics 66
Section 4.5 Limitations of FINN Compiler and future changes 66

Section 5 Firmware & Communication Details: Communication 1 - Internal 67
Section 5.1 Introduction 67
Section 5.1.1 Architecture diagram 67
Section 5.2 Brief Overview 68
Section 5.2.1 Job scope 68
Section 5.2.2 Difficulties encountered and alternative solutions 68
Section 5.3 Design of internal communications 69
Section 5.3.1 Original design (before Week 7) 69

Section 5.3.2 New design (after Week 7) 73
Section 5.3.3 Ensuring stability of Beetles 73
Section 5.3.4 Transmission of real sensor data 74
Section 5.3.5 Transmit data only when needed 75
Section 5.3.6 New packet types and codes 77
Section 5.3.7 Greater robustness in internal communications architecture 78
Section 5.3.8 Machine learning, evaluation and dashboard server integration 80
Section 5.3.9 Collecting EMG data 81
Section 5.4 Protocol to coordinate Beetles and Ultra96 83
Section 5.5 Improvements to existing design 84
Section 5.6 Extension of internal communications 85
Section 5.6.1 Rain-Alert-O-Meter 85
Section 5.6.2 Change in project specifications 85
Section 5.6.3 Utilizing machine learning to predict weather patterns 86
Section 5.6.4 Rationale for this extension proposal 87
Section 6 Firmware & Communication Details: Communication 2 - External 87
Section 6.1 Overview and Contributions 87
Section 6.2 Communication between Ultra96 and Evaluation Server 89
Section 6.2.1 Process Overview 89
Section 6.2.2 Rationale and Benefits for the System 89
Section 6.2.3 Message Format 90
Section 6.2.4 Socket Implementation 90
Section 6.3 Communication between Ultra96 and dashboard server 91
Section 6.3.1 Process Overview 91
Section 6.3.2 Rationale and Benefit for the System 92
Section 6.3.3 Socket Implementation 92
Section 6.3.4 Sending Data to Dashboard 93
Section 6.4 Secure Communication 93
Section 6.4.1 Main Idea, Rationale and Benefits for the System 93
Section 6.4.2 AES Encryption and Decryption Scheme 93
Section 6.4.3 Encryption Implementation 95
Section 6.5 Detection of Start of a Dance Move and Splitting Data 96
Section 6.5.1 Main Idea, Rationale and Benefits for the System 96
Section 6.5.2 Thresholding Implementation 96
Section 6.5.3 Splitting Data with Thresholding 97
Section 6.5.4 Passing Split Datasets to Machine Learning 98
Section 6.6 Clock Synchronization Protocol 99

Section 6.6.1 Main Idea, Rationale and Benefits for the System 99
Section 6.6.2 Protocol Implementation 99
Section 6.6.3 Packet Format 101
Section 6.7 Time Calibration 101
Section 6.7.1 Clock Offset Calculation 101
Section 6.7.2 Frequency of Time Calibration 102
Section 6.8 Synchronization Delay Calculation 102
Section 6.8.1 Timestamp of Start of a Dance for Beetles 102
Section 6.8.2 Asynchrony among the Dancers 103
Section 6.9 Further Development of Communication External Components 104
Section 6.9.1 Better Design of Clock Synchronization Protocol 104
Section 6.9.2 Applying Secure Communication to Email Services 104

Section 7 Software Details 105
Section 7.1 Segmenting the sensed data 105
Section 7.2 Features to be explored 106
Section 7.3 Machine learning models under consideration, and optimizing chosen machine learning algorithm 107
Section 7.4 Training and testing the model 110
Section 7.5 Storing the incoming sensor data 113
Section 7.6 Real Time Streaming 120
Section 7.7 Transition and dance prediction 121
Section 7.8 Extensions to machine learning model 125

Section 8 Software Details: Software 2 - Dashboard and Client-Server 132
Section 8.1 Architecture Design 132
Section 8.1.1 Architecture Diagram 132
Section 8.1.2 Code Contributions to Project 132
Section 8.2 Front-End: Dashboard Design 133
Section 8.2.1 Screenshots of Design Process 133
Section 8.2.2 Changes from Initial Dashboard Design 134
Section 8.2.3 Final Dashboard Design 135
Section 8.2.4 Approach to Real-Time Streaming Data 136
Section 8.2.5 Explanation of Front-End Code 136
Section 8.2.6 Explanation of Performance Enhancing Front-End Code 144
Section 8.3 Back-End Design 144
Section 8.3.1 Choice of Database 145
Section 8.3.2 API Implementation 145

Section 8.3.3 Client-Server Implementation 147
Section 8.4 Data Analytics 151
Section 8.5 Software 2 Project Extensions 153
Section 9 Societal and Ethical Impact 154
Applications of Human Activity Detection: 154
Performance of Athletes 154
Societal and Ethical Impacts 155
Early Intervention of Elderly in Emergencies 155
Societal and Ethical Impacts 156
Hand Gesture Recognition to Operate Machinery 156
Societal and Ethical Impacts 157
Concluding Thoughts 157
Section 10 Project Management 158
Section 10.1 Team Structure 158
Section 10.2 Progress tracking 158
Section 11 References 163

Glossary

Notation	Definition
AXI	Short hand for Advanced eXtensible Interface. It is part of the ARM Advanced Microcontroller Bus Architecture 3 and 4 specifications, is a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface, mainly designed for on-chip communication
Beetle/Beetles	Shorthand for Bluno Beetles hardware component
BLE	Shorthand for bluetooth low energy
Dancing Phase	The phase that dancers are performing the dance move
EMG	Shorthand for Electromyography
FINN Compiler	An experimental framework from Xilinx Research Labs to explore deep neural network inference on FPGAs. Generates bitstream from ONNX model.
GUI	Shorthand for Graphic User Interface
HCI	Shorthand for Host Controller Interface
HLS	Shorthand for High Level Synthesis. Vivado HLS transforms a C, C++, or SystemC design specification into Register Transfer Level (RTL) code for synthesis and implementation by the Vivado tools
IMU	Shorthand for GY-521 (MPU 6050) Inertial Measurement Unit
RTL	Shorthand for Register Transfer Level. It is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals between hardware registers, and the logical operations performed on those signals
SRAM	Shorthand for static random access memory
ONNX	Shorthand for Open Neural Network Exchange. It is an open format built to represent machine learning models
UART	Shorthand for Universal Asynchronous Receiver-Transmitter
Walking Phase	The phase that dancers walk to their relative position

Section 1 System Functionalities

Section 1.1 Use Cases

For all the use cases below, unless specified otherwise:

1) The computer used to give the dancers instructions is the Evaluation Server
2) The computer used to show the analytics data is called the Dashboard
3) The dancers are the Users
4) The Ultra96 and Beetle setup, along with the connected sensors and algorithms in place, is the System.

- **Use Case: UC01 - Correct position and dance move**
    - **MSS:**
        1. Evaluation Server displays positions the users are supposed to be in alongside a random dance move on screen.
        2. Users move to their destined positions and perform the dance move.
        3. System detects and predicts the user's position, and dance moves performed by the users and estimates the synchronization delay between the users, and sends the information to the Dashboard and Evaluation Server.
        4. Evaluation server checks and stores the correctness of the position, and dance move prediction and records synchronization delay of the users alongside the correctness of position and dance move in a log file and shows the next set of positions and dance move on screen. Dashboard stores and analyses the sensor data received.
           Use Case ends

- **Use Case: UC02 - Final logout dance action**
    - **MSS:**
        1. Evaluation Server displays "Logout" and the positions the users are supposed to be in on screen.
        2. Users move to their destined positions and perform the final dance move.
        3. System detects and predicts the users' position and dance moves performed by each user.
        4. The system closes the connection with the Evaluation Server.
           Use Case ends

Section 1.2 Feature List

1. Lightweight and comfortable wearable.
2. One size fit all, adjustable size for the wearable.
3. Ease of use.
4. Fast in the sensing of movements and performs accurate predictions.
5. Ability to identify whether the user is facing muscle fatigue.
6. Ease of visualisation of sensor datas, dance move predictions and muscle fatigues
7. Lowered and efficient power consumptions.
8. Reliable and low maintenance rate, the system can be used several times prior to requiring maintenance works.

Section 2 Overall System Architecture

Section 2.1 Design of the System

In this section, we will first present the initial design of our system that was as of week 4 and following that we will showcase the final design of our system. After showcasing the initial and final form of our system, we will explain the challenges faced and the changes made to the design to overcome the challenges and still suit the needs of this project.

Section 2.1.1 Initial Form of our System

In the initial design report submitted on week 4, we designed the initial form of our system as shown in Figure 1. The initial design includes having two IMUs, one on each wrist of the dancer, one MyoWare Muscle Sensor on the biceps of the main dancer alongside the Ultra96 mounted on the chest of that main dancer. This initial design has been changed accordingly to counter the challenges faced when developing the hardware as well as taking into account the comments and feedback from the professors and teaching assistants. The new design for the final form of our system will be discussed in Section 2.1.2.

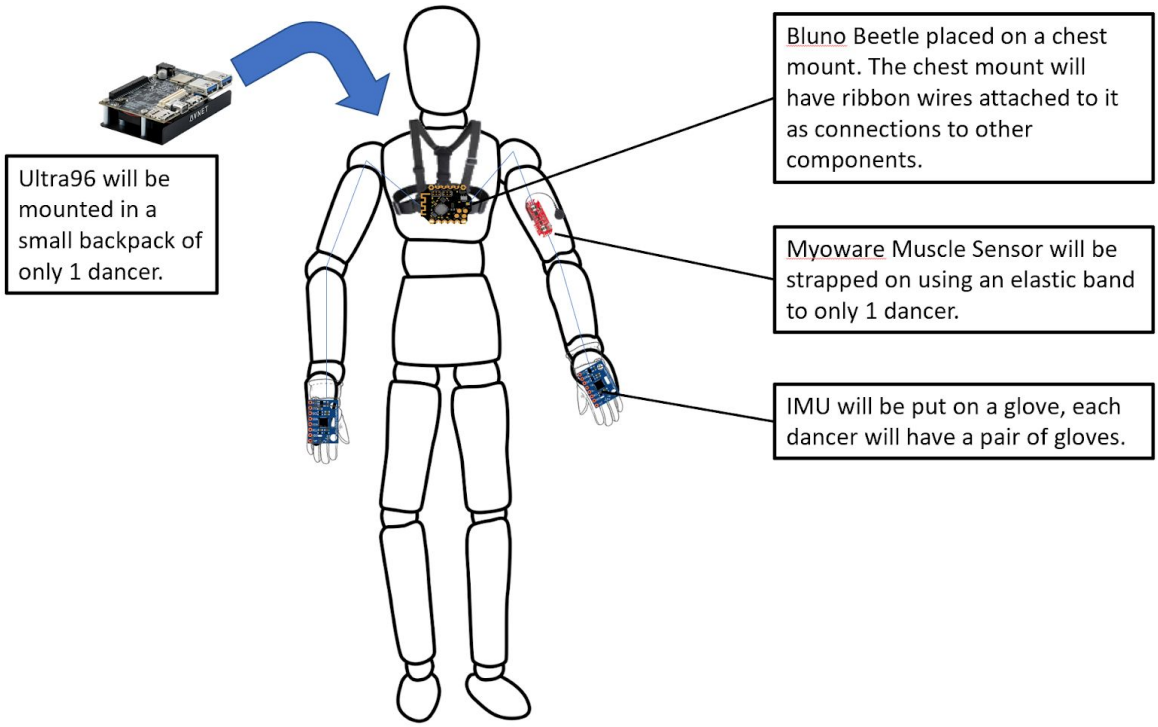


Figure 1: System's design initial form (as of week 4)

Section 2.1.2 Final Form of our System

After taking into account the comments and feedback from the professors and teaching assistants, we took those feedback to improve on our design. The final design for our system is as shown across Figure 2. It includes decreasing the number of IMUs, connected to Beetle, from 2 to 1 making the circuitry compact enough to be placed within a pouch. This pouch is then tied to the hand using a Velcro strap as seen across Figure 2. The contents of the pouch are the IMU sensor alongside Beetle and its circuitry can be seen in Figure 3. The position of the MyoWare Muscle Sensor had no changes from the initial design but there is an additional pouch with a Velcro strap placed above the biceps to secure the Beetle and its circuitry that is connected to the MyoWare Muscle Sensor. The pouch that stores the Beetle and its circuitry connected to MyoWare Muscle Sensor can be seen across Figure 4. Additionally, instead of using a chest mount to hold the Ultra96 and its relevant circuitry in place, we fitted it into a container and placed the container into a sling bag as shown in Figure 5. The fitting of the Ultra96 and its circuitry into the container can be seen across Figure 6. The sling bag is also filled with newspapers such that the container, with the Ultra96 and its circuitry, will be kept in place instead of moving around when the dancer wearing the sling bag is dancing or moving.

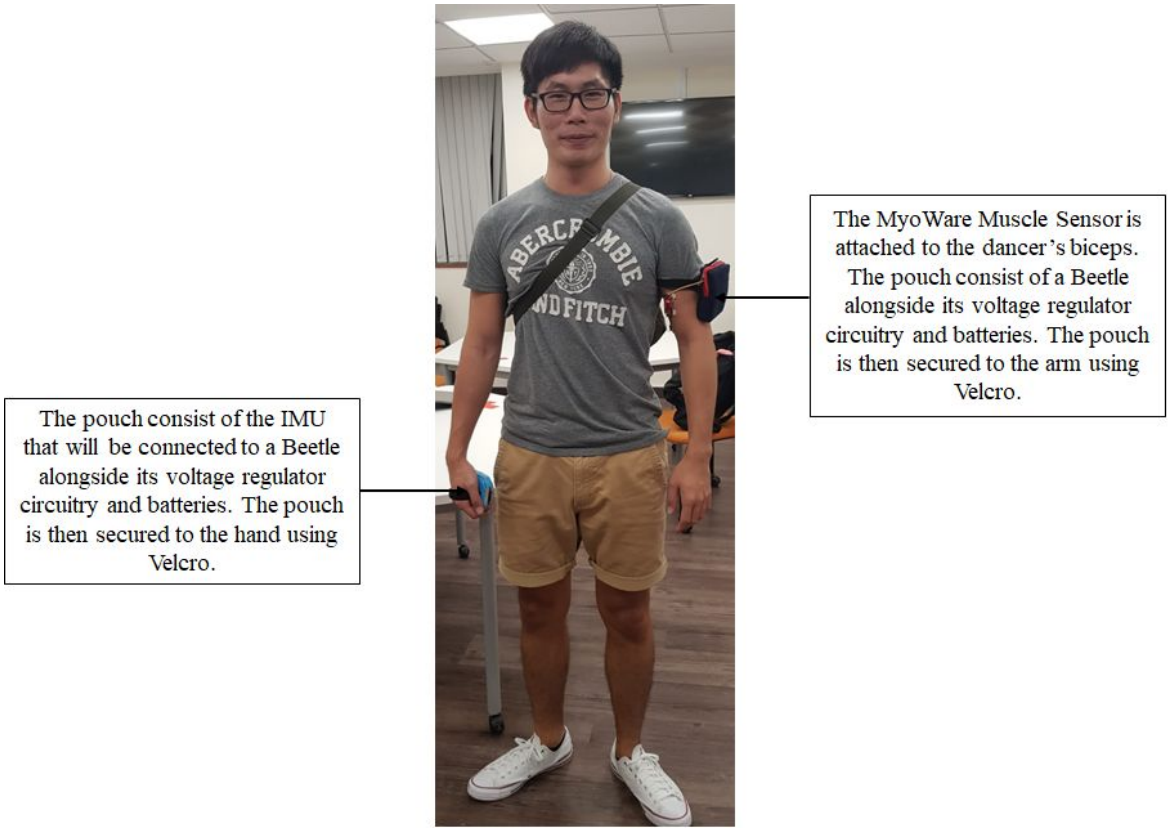


Figure 2: System's design final form (front view)

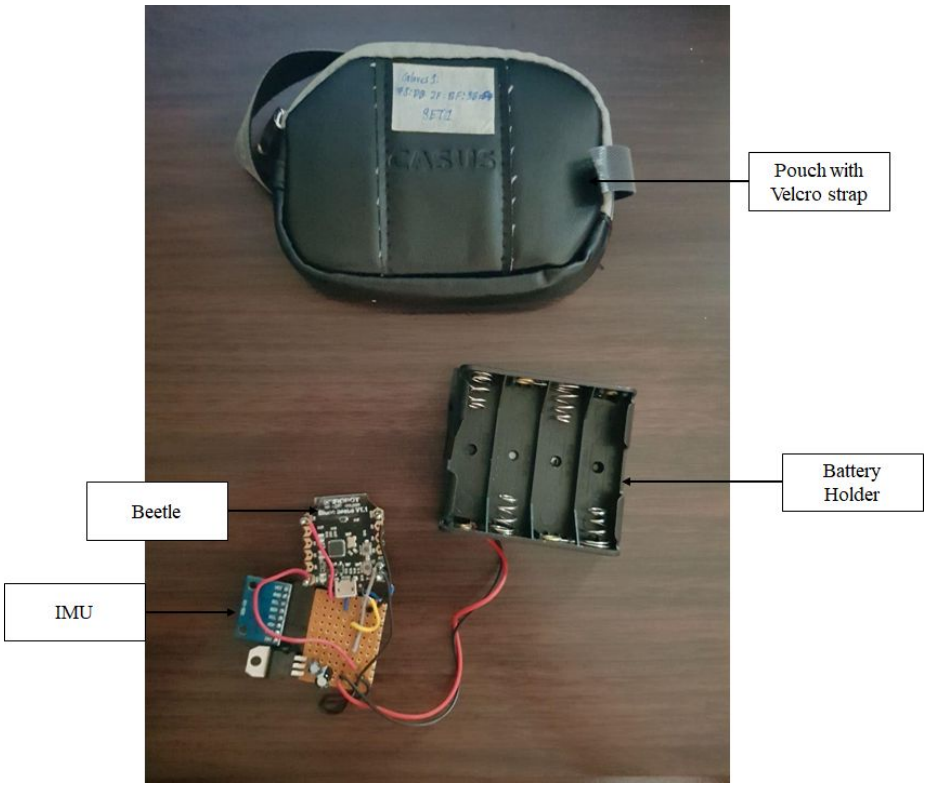


Figure 3: IMU connected with Beetle and its circuitry

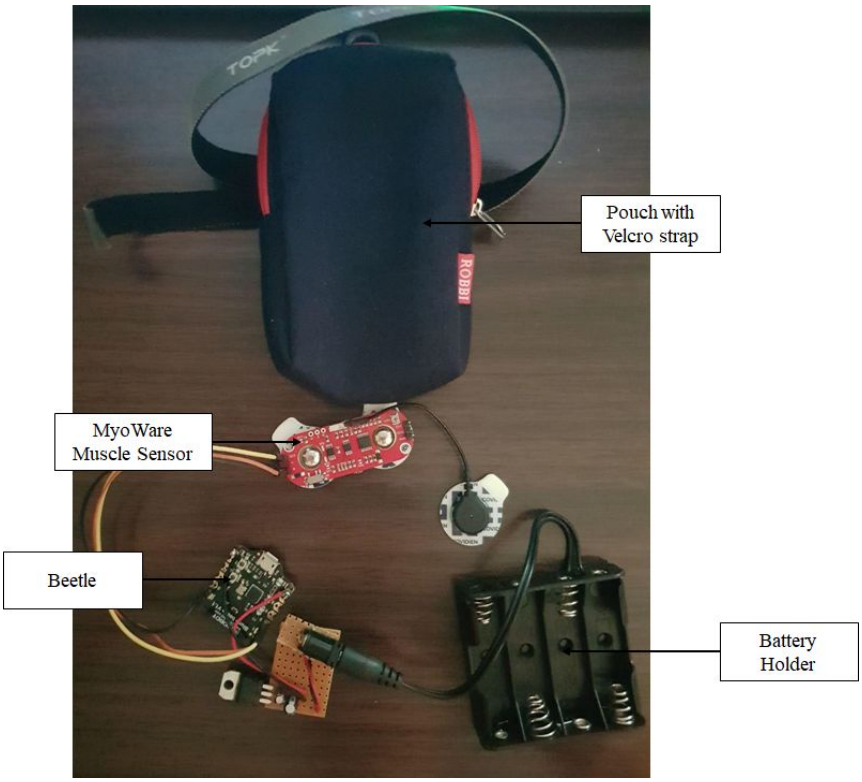


Figure 4: MyoWare Muscle Sensor connected with Beetle and its circuitry

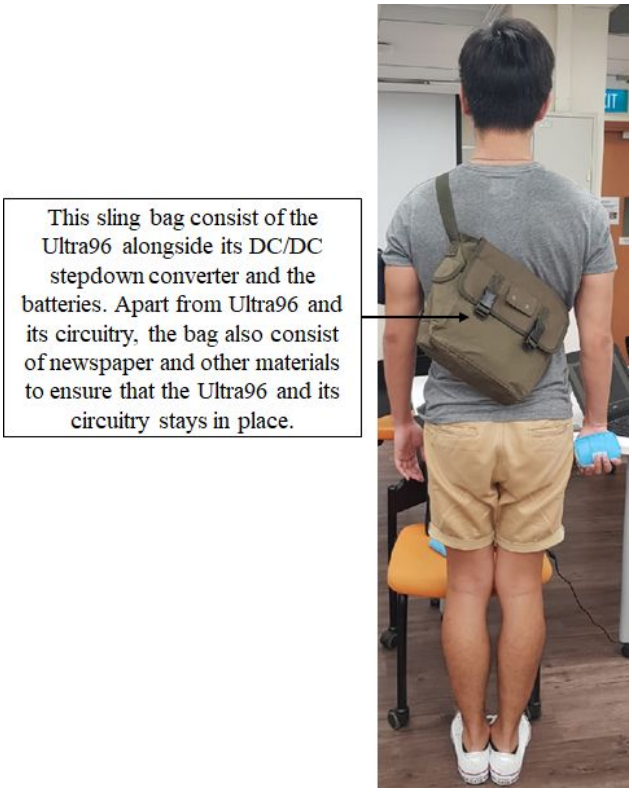


Figure 5: System's design final form (back view)

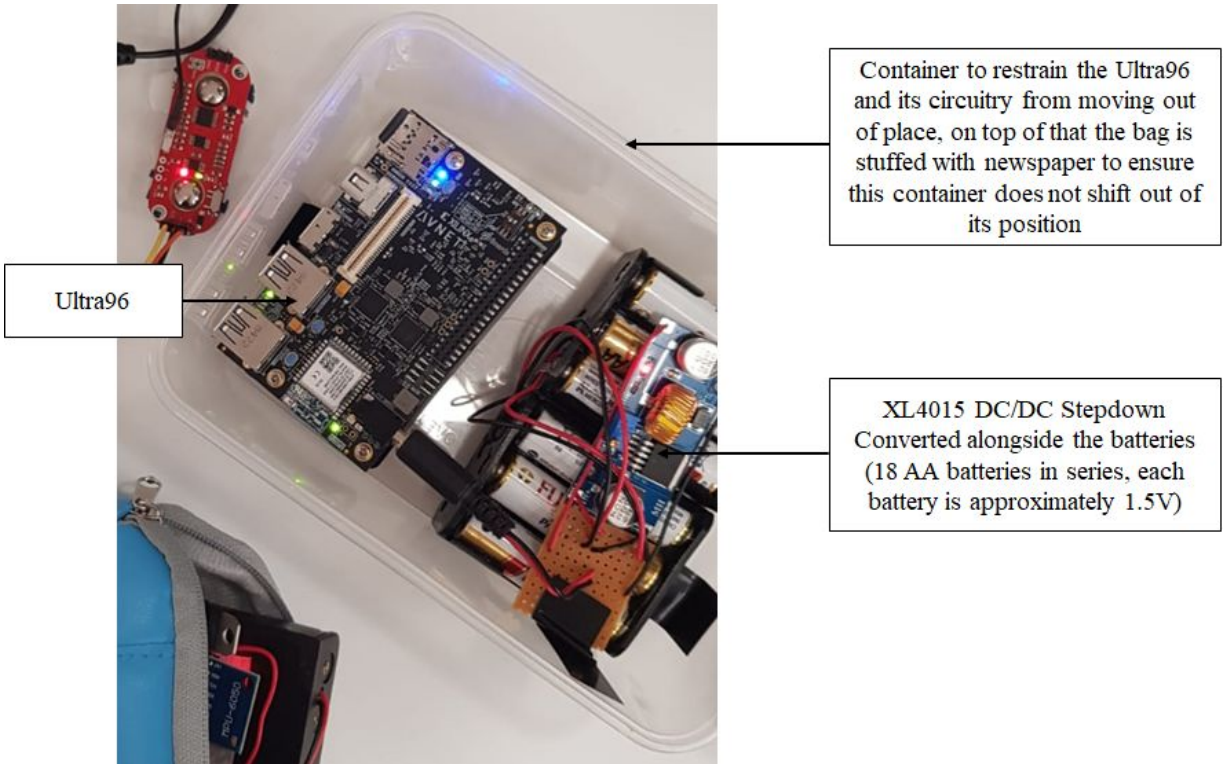


Figure 6: Ultra96 that is placed within the sling bag

Section 2.1.3 Reasons behind the design changes

In the initial design, we intended to use one IMU on each wrist. However, after further consideration and taking into account comments and feedback from our teaching assistant, we realised that it is not practical and efficient to place both IMUs on the wrists as the dance moves that we are supposed to perform are symmetrical which also means that the data collected from both the IMUs will be similar. Therefore, we decided to place one of the two IMUs on the ankle instead of the wrist such that we will be able to get more data that could potentially allow us to better differentiate between the movements and dance moves by the dancers. In addition to having one IMU on the wrist and one on the ankle, the hardware personnel decided to do up a compact circuit such that we do not have dangling wires all around the dancer and therefore we had six Beetles instead of the original three. However, when we were integrating our individual components together into a full system, we encountered problems trying to get concurrent data transfer from six Beetles to the Ultra96. Instead of getting concurrent sensor datas from the Beetles, we had times of sequential data collection and this is a call for concern as we needed the Beetles on the same dancer to transfer data concurrently such that the data are of the same timestamp. Therefore, we dropped the idea of having an IMU on the ankle as we realised that most dance moves do not involve movement on the legs and data collection on the leg movements would be less crucial as compared to the data collection on the hand movements. As

such, we reduced the amount of Beetles usage back to three and each Beetle is connected to a single IMU where the whole circuitry is placed into a pouch and attached to a dancer's wrist via a Velcro strap.

Another change we made was to use a sling bag to store the Ultra96 with its DC/DC step-down converter circuitry instead of mounting it on a chest mount. The reason behind this change was that the Ultra96 requires a 12V supply to power it up and get it functioning and that will require at least 9 batteries of 1.5V each in series, as such using a chest mount is not ideal as adhesives to hold the weight of 9 batteries and Ultra96 will wear off overtime and it will eventually lead to the equipment dropping off the mount. Additionally, as voltages of the batteries will drop overtime, it is crucial that we have a higher capacity than what is required and the hardware personnel decided to go with 12 batteries in series which adds up to having 18V in total and this increases the amount of weight required to be held on the chest mount. The usage of 12 batteries in series is crucial as it allowed Ultra96 to last for at least 2 hours and this is definitely more than enough for the dancers to complete their dance activity of 12 dance moves. Therefore, we decided to place the batteries alongside the Ultra96 and its circuitry in a container that would protect it prior to placing it into a sling bag. As an additional security measure, we made sure the sling bag was filled with newspaper so that the container will not bounce rigorously inside the sling bag when the dancer wearing it is moving or dancing.

Section 2.2 High-Level System Implementation

Section 2.2.1 High-Level System Architecture

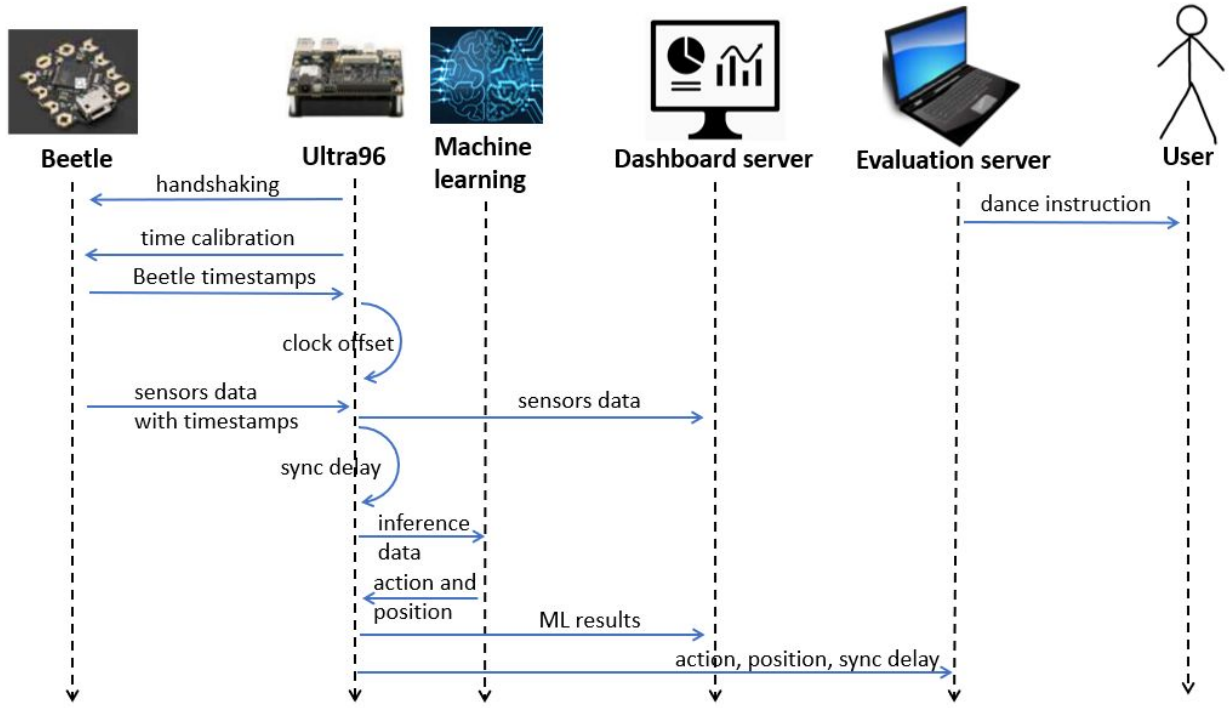


Figure 7: Overall architecture

The main algorithm of our system is as follows:

1. Ultra96 connects with the Beetle.
2. Ultra96 calibrates time with the Beetle.
3. Beetles collects sensors data and separates the data to walking phase data and dancing phase data based on thresholding values
4. Ultra96 collects walking phase data and dancing phase data from the Beetles.
5. Ultra96 calculates synchronization delay using timestamps in the dancing phase data.
6. Ultra96 passes walking phase data to the position prediction machine learning model and dancing phase data to the dance action prediction machine learning model. The models return relative position and dance action results.
7. Ultra96 sends the sensor data and machine learning results to the dashboard server to be displayed on it's GUI.
8. Ultra96 sends dance action, position alongside the synchronization delay to the evaluation server.
9. Evaluation server receives the results and starts the next dance instruction. The algorithm repeats from 2 to 9.

Section 2.2.2 Communication Processes

Communication Processes on Ultra96			
Number	Name	Purpose	Priority
1	Connecting to Beetles, handshaking	To establish a stable connection between Ultra96 and calibrate time on beetles to synchronize with Ultra96	Highest
2	Time calibration	Calibrates time between Ultra96 and Beetles using clock synchronization protocol	High
3	Receive data from 3 Beetles	Creating three worker threads for concurrent data collection from 3 Beetles.	High
4	Use data for machine learning algorithms	To feed all relevant data into machine learning models and compute the most likely dance move and relative positions.	Medium
5	Pass data to the evaluation and dashboard server	Send predictions and synchronization delay to the evaluation server for evaluation. Send data to the dashboard server after every fixed interval of datasets received for real-time visualization of dashboard analytics. Also sends predictions to the dashboard server.	Low
6	Receive EMG data	Collects EMG datasets from Beetle with EMG sensor	Low

The process of connecting to the Beetle and conducting handshakes is given the highest priority as the processes further downstream would be unable to proceed without first waiting for connection to be established. Following the handshake, is the process responsible for collecting timestamp data from the Beetles. This process has a high priority as we need the Beetle's Arduino timestamp in order to calculate the clock offset for that particular Beetle, so that we can calculate the synchronization delay between the dancers. The next process which is receiving sensor data from the Beetles is given a higher priority than the rest of the downstream processes as there needs to be collected datasets before we can pass them to the machine learning algorithm. The process for feeding datasets into the machine learning models has a higher priority than the remaining two processes below it as we are unable to transmit anything to the evaluation and dashboard server without first waiting for the machine learning to be done with its task. The last process which is receiving EMG data is given a low priority as apart from the

dashboard GUI, no other process is blocked waiting for it to collect EMG data. It would be okay if the process is delayed in-lieu of other processes to be completed first as it is just an additional feature in our overall system.

Communication Processes on Beetles			
Number	Name	Purpose	Priority
1	Handshake and clock synchronization	To create a connection between the Beetle and Ultra96 and calibrate time on beetles to synchronize with Ultra96	Highest
2	Read, process and transmit sensor data to Ultra96	To read data produced by the sensors whenever a dance move has started. To ensure correct formatting of packet structure into a standard that can be replicated in Ultra96's side for a smooth transfer of data	High

The process of doing handshaking and time calibration is given the highest priority, as it is necessary to establish connection with Ultra96 first before any data can be transmitted.

Section 2.2.3 Implementation on Beetle and Ultra96

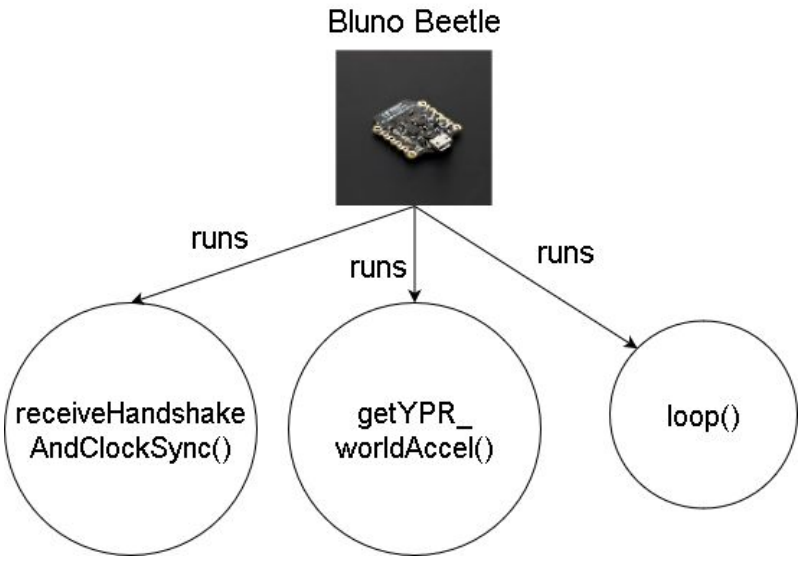


Figure 8: Functions defined in Beetle

The Beetle has three important functions that ensure smooth data transmission to the Ultra96. The first function, receiveHandshakeAndClockSync(), performs handshaking with the Beetles during the connection stage and calibrates time during the evaluation stage. The second function,

getYPR_worldAccel(), collects actual data from the IMU sensors. The third function, loop(), is a standard Arduino default function that runs forever until the Beetle is reset. Inside loop(), is where the getYPR_worldAccel() function is called, and the sensor data is processed before being transmitted to the Ultra96.

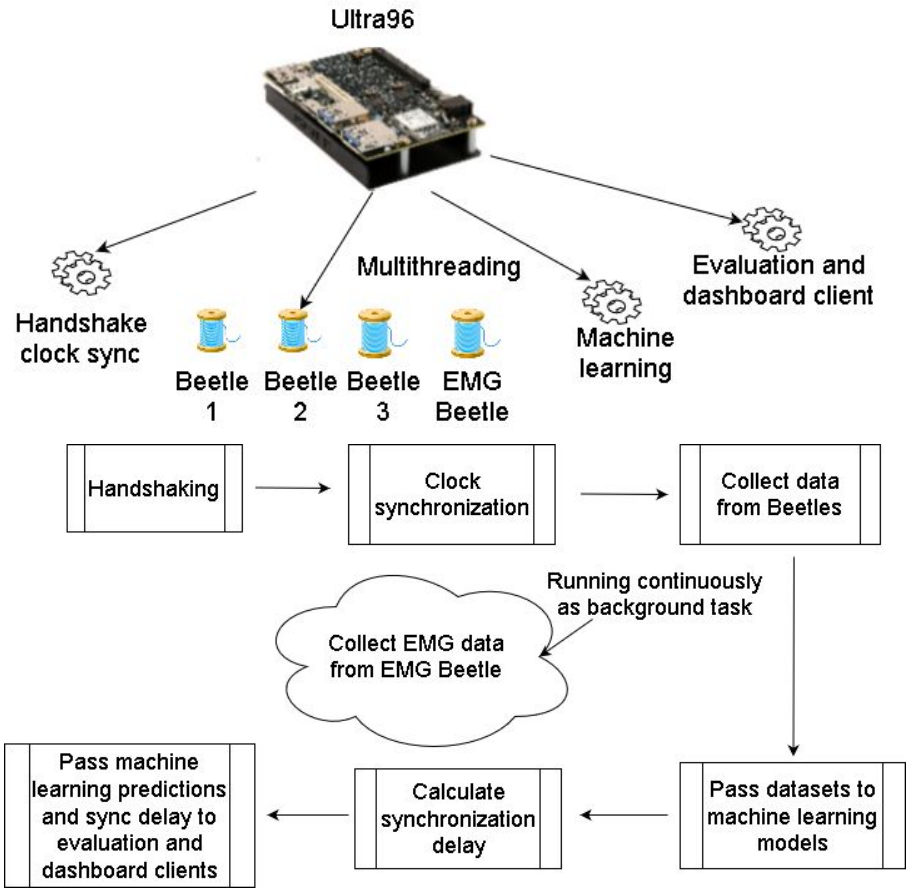


Figure 9: Order of process flow in Ultra96

Section 2.2.4 Communication Protocols

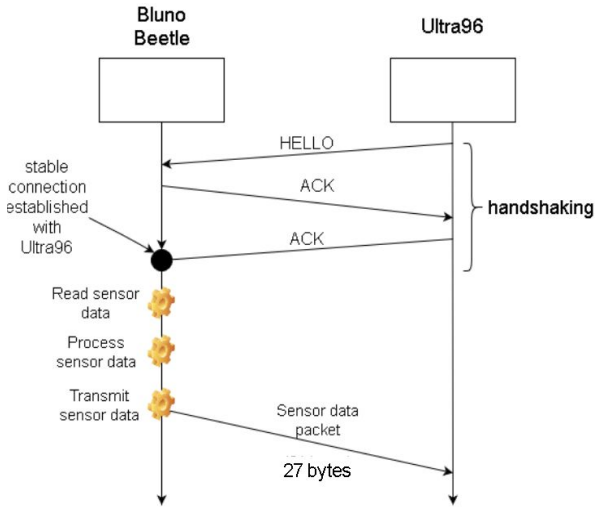


Figure 10: Establishing connection and sending data

1. **Establishing connection**

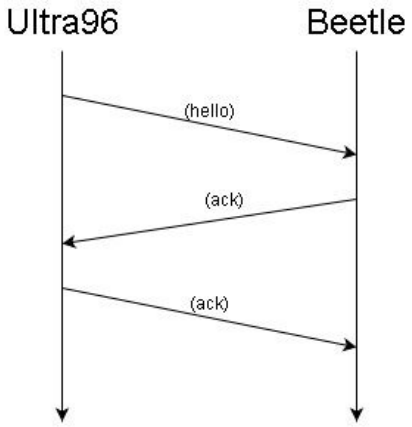


Figure 11: 3-way handshake

During the connection stage, the Ultra96 will initialize the handshaking process by sending a hello packet to the Beetle, followed by the Beetle sending a ack packet back to Ultra96 to reciprocate the hello packet and to tell Ultra96 that it is ready to transmit data, then Ultra96 will send ack packet in return to tell the Beetle that it is ready to receive data.

2. **Time calibration**

We design the following clock synchronization protocol to calibrate and synchronize time between Ultra96 and the Beetles. Details of the protocol are in Section 6.6 Clock Synchronization Protocol.

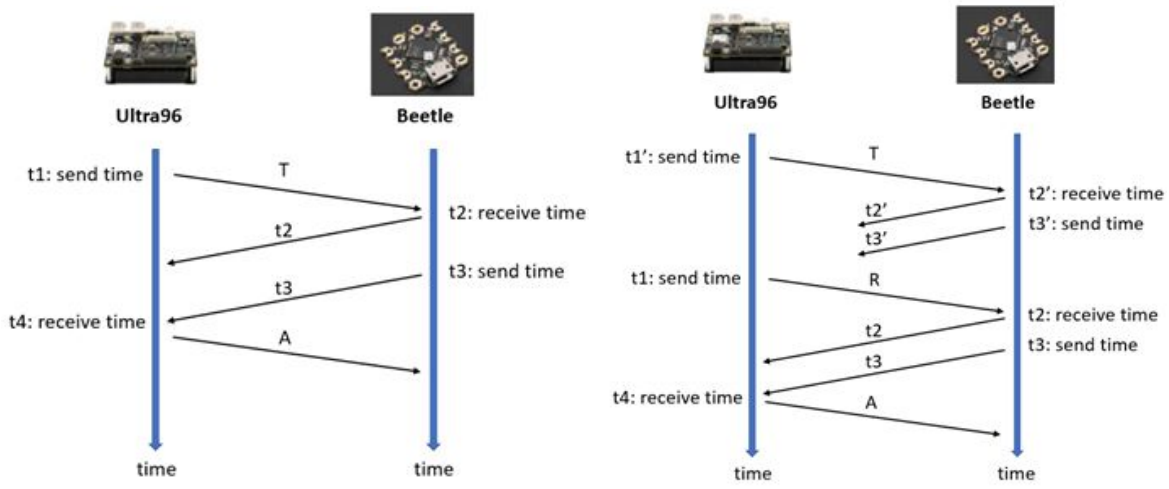


Figure 12: Clock synchronization protocol

3. **Communication with evaluation and dashboard servers**

We use TCP as the communication protocol and utilizes socket programming to establish connections. The Ultra96 keeps the evaluation client and dashboard client running while the evaluation server and dashboard server runs separately on 2 laptops. The explanation and implementations of this communication will be explained in Section 6.2 and 6.3.

Section 2.2.5 Hardware Components

The main hardware components involved in our final design did not have any changes from our initial design and they constitute of the following components:

- Beetle
- IMU
- MyoWare Muscle Sensor
- Ultra96

The communications between the various individual components can be seen across Figure 13:

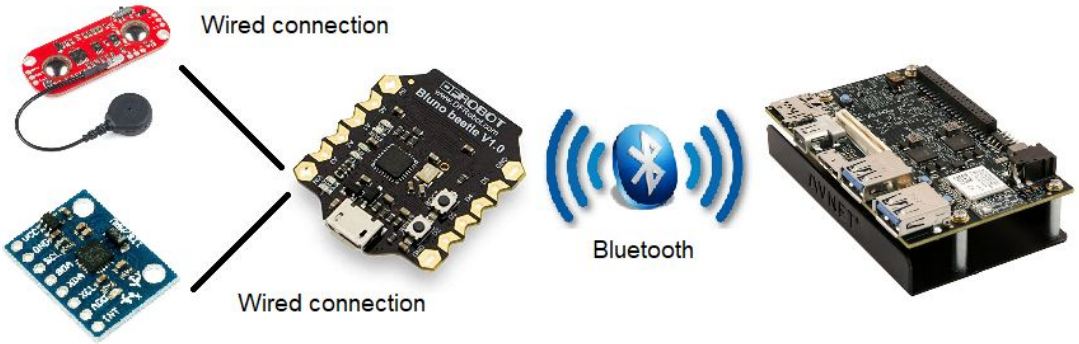


Figure 13: Hardware connection and interfacing

As seen in Figure 13, we will be using wired connections from the IMU and MyoWare Muscle Sensor to the Beetle for transferring of the sensor data received from the respective sensors to the Beetle. The Beetle will proceed to process the data received and transfer these processed data to the Ultra96 via Bluetooth. By processing the data, we meant to segment the data into walking phase data and dancing phase data. This segmentation is achieved through means of a thresholding algorithm that will be explained in detail in Section 3.7 Algorithms and Libraries.

Section 2.2.6 Machine Learning High Level Algorithm

On our system, we have two different machine learning models: one for walking phase prediction and another for dancing phase prediction. Both models are implemented with neural networks, and the details regarding its architecture will be explained in future sections. Each set of IMU readings contains the acceleration along the 3 axes and yaw, pitch and roll. After the IMU readings are collected, they will be passed into the relevant neural networks as features to perform the prediction.

For each dancer, the dancing phase prediction model takes in a tumbling time window of 5 sets of back to back IMU readings as its inputs, and predicts one of the following dancing actions: shout out, muscle and weightlifting. There is more than 1 time window within the set of all collected IMU readings and each time window when passed into the dancing action prediction model will return 1 dance prediction. Our final dance phase prediction result for each dancer will depend on which dance action has the highest number of predictions given the set of all collected IMU readings. When we have 3 dance predictions for each of the 3 dancers, the overall dance prediction that will be sent to the evaluation server will depend on which dance action is predicted the most number of times. In the case where all 3 possible dance actions are predicted once for the 3 dancers, one of the dance actions will be chosen randomly and sent over to the evaluation server.

The walking phase prediction model takes in 1 set of IMU readings at a time. With each set of IMU reading, it will predict one of the following walking actions, moving left, moving right or standing still. Similar to dance action prediction, the final walking phase prediction result will depend on which walking action has the highest number of predictions. Once we have the movement action predictions for each of the 3 dancers and along with the current position, we will use that to predict the new positions of the 3 dancers. However one thing to take note is that the combination of movement actions and current position can return an invalid position (e.g all 3 dancers moving right). In that case, then we will return a random position.

The following depicts the pseudo-code describing the flow from collection of data to activity detection on the Ultra96:

1. Finished receiving data from Bluno Beetles
2. Segment data into dancing phase data and walking phase data depending on the timestamp that is received within the received IMU readings. Walking phase data has a timestamp of 0.

Dancing phase prediction

3. For each dancer:
   - 3.1 Initialize list of counters, one counter for each dance action, shoutout, muscle and weightlifting.
   - 3.2 Split dancing data into tumbling time window of 5 back to back IMU readings
   - 3.3 For each time window make a prediction.
     - 3.3.1 For the predicted dance action, increase its counter by 1.
   - 3.4 Return dance action with the highest counter. In the case of a tie, randomly return 1 of the dance action that ties.
4. If each dance action is predicted exactly once
   - 4.1 Send one of the dance actions to the evaluation server.
   - 4.2 Else return the dance action that is predicted the most number of times.

Walking phase prediction

5. Initialize the array representing the initial position. (Initialized to 1,2,3 before the first prediction and received from the evaluation server after the first prediction.)
6. For each dancer:
   - 6.1 Initialize list of counters, one counter for each walking action, moving left, moving right, standing still
   - 6.2 For each set of IMU reading in walking data make a prediction.
     - 6.2.1 For the predicted walking action, increase its counter by 1.
   - 6.3 Return walking action with the highest counter. In the case of a tie, randomly return 1 of the walking action that ties.
7. Given the current position, and list of walking action for each dancer
   - 7.1 If the combination of position and walking action can derive a valid new position, return the new position
   - 7.2 Else return a random position.

Communication with evaluation server

8. Send final dance phase prediction and position prediction to the evaluation server.
9. If dance phase prediction is not logout move, wait until the actual correct position from the evaluation server, else exit from the script
10. Once received the actual correct position from the evaluation server, signal the Beetles to

start collecting data again.

Section 3 Hardware Details: Hardware 1 - Sensors

Section 3.1 Introduction

In Section 3 about Hardware Sensors, a brief overview on hardware sensors' role will be first described alongside the main steps that were carried out to achieve the requirements. The following sections after the brief overview will focus on the hardware components used in the project which includes the pin table alongside the schematics of the respective circuitries. The respective operational voltages and current of these relevant hardware components will then be presented after the schematics. After which, the algorithms and libraries used for the sensor data collection for machine learning predictions and EMG data for the dashboard analysis will be discussed. This section on hardware sensors will then end off with discussing the changes made across the course of the project, challenges faced and the possible extension(s) for this role in this project. Apart from discussing the possible extension(s) for this role in this project, other possible applications that could potentially realise the hardware components better are brought into discussion as well.

Section 3.2 Brief Overview

In a nutshell, the role of the hardware sensors personnel was to develop a full wearable hardware system that constitutes the following components:

- Sensors that can provide data that is capable of showing differences in value when dancers are moving to their relative positions or performing their dance moves

- Voltage regulator circuitries that controls the voltage output supplied from batteries to the hardware components, namely Beetle and Ultra96

- Processing of EMG data collected from MyoWare Muscle Sensor to extract features that are capable of identifying muscle fatigue in a dancer

On top of putting together the hardware components, another important requirement is to design the system in a way such that it is fully wearable on the dancer.

To achieve the requirements required, researches on the sensors provided, namely IMU and MyoWare Muscle Sensor, alongside searching for other sensors that could potentially provide data useful for the machine learning predictions on relative positions and dance moves were

carried out. Apart from researching on the hardware components, the various algorithms and libraries available for use in these sensors to process and provide data that could potentially assist in the respective predictions were looked into. The hardware components used in the project will be explained in detail in Section 3.3 and the circuitries required of these components are described across Section 3.4 and 3.5. Additionally, the operational voltage and current of these hardware components will be presented in Section 3.6, and the algorithms and libraries used in our project will be explained in detail in Section 3.7.

Apart from handling the required sensors and the algorithms for the data collection, voltage regulator and DC/DC step-down converter circuitries that regulate voltage output from a power source to a suitable voltage level for the hardware components used in our project were also looked into. Practical trials were carried out on these circuitries to ensure that the voltage output from these circuitries are at an acceptable level for the hardware components.

Despite carrying out extensive research and trials, the hardware design and its circuitries still has room for improvements. However, the timeline for the project was limited in addition to the COVID-19 virus ongoing, further improvements or changes to the hardware components and/or algorithms came to a halt. Therefore, the ideas for possible improvements of the hardware components and/or algorithms alongside how the functionality of these components can be brought to other applications or work will be presented in theory under Section 3.9.

Section 3.3 Hardware Components/Devices

In this section, the hardware components/devices used in our wearable system will be listed accordingly. The supporting components required of each individual hardware component/device will also be listed alongside providing the datasheet of the hardware component/device.

Section 3.3.1 Beetle V1.1

The Beetle is an Arduino Uno based board with bluetooth functionalities (Dfrobot). The purpose of the Beetle is to gather the output from the connected sensors as input and process these input into output required of our machine learning models to classify the information and make predictions on the relative positions and dance moves. The bluetooth functionality will be used for communication between Beetle and Ultra96-V2.

Supporting Components:

- LM7805 Voltage Regulator and its circuitry

Datasheet:

- https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339

- https://datasheet4u.com/datasheet-parts/ATmega328P-datasheet.php?id=1057332

Section 3.3.2 LM7805 Voltage Regulator

The LM7805 Voltage Regulator as the name suggests is a voltage regulator that outputs +5 volts which is the operational voltage for the Beetle (Learning about Electronics). Therefore, this component has been chosen to regulate the output from 4x AA Batteries at 1.5V each which provides 6V output. This regulator is used to ensure that the Beetle does not burn out and that operational voltage is supplied to the Beetle..

Supporting Components:

- 10 µF Capacitor
- 1 µF Capacitor
- Power Supply - 4x AA batteries in series at 1.5V each

Datasheet:

- https://pdf1.alldatasheet.com/datasheet-pdf/view/82833/FAIRCHILD/LM7805.html

Section 3.3.3 IMU

The IMU enables us to gather data on the amount of rotation and acceleration of the object attached to it from its initial position to its final position, which in our design the object would be the dancer's hand, more specifically the wrist. The IMU uses an accelerometer and a gyroscope to gather such data in the 3-dimensional space. The changes of the rotation and acceleration in the 3-dimensional space will enable us to determine whether any movements have been made by the dancer. These data can then be passed to machine learning models to determine through classification as to what movement has been made by the dancer, it can predict the relative positions and dance moves performed by the dancer using the data from this sensor.

Supporting Components:

- Beetle V1.1

Datasheet:

- https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf
- https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf

Section 3.3.4 MyoWare Muscle Sensor

The MyoWare Muscle Sensor enables us to measure muscle activation via EMG data (Theorycircuit). It allows us to gather EMG data where features can be extracted to examine an individual's muscle activity and we will then be able to tell the fatigue level of the individual's muscle alongside the improvement, if any, across the weeks.

Supporting Components:

- Beetle V1.1

Datasheet:

- https://cdn.sparkfun.com/datasheets/Sensors/Biometric/MyowareUserManualAT-04-001.pdf

Section 3.3.5 Ultra96-V2

The Ultra96 contains the Xilinz Zynq UltraScale + MPSoC ZU3EG. This FPGA is used to perform machine learning algorithms on sensor data. The bluetooth feature of the Ultra96 is used to interact with the Beetles to obtain sensor data to be used in the FPGA. The Linux environment in the Ultra96 will enable information to be sent through wifi from the Ultra96 to a remote dashboard for data analytics

Supporting Components:

- XL4015 DC/DC Step-down Converter

Datasheet:

- [Product Brief] http://zedboard.org/sites/default/files/product_briefs/5354-pb-ultra96-v3b.pdf

Section 3.3.6 XL4015 DC/DC Step-down Converter

The XL4015 is a DC/DC Step-down Converter that can step down voltages from 4 to 38V to an adjustable range of 1.25 to 35V and it has an output rating of up to 5A which is suitable for Ultra96-V2 as it requires a 12V input and 4A current input to function (Hareendran). Therefore, this component was chosen to step down the voltage from 12x AA batteries at 1.5V each which provides 18V output. This converter is used to ensure that the Ultra96-V2 does not burn out and ensures that it has a stable voltage input and current rating to keep it functioning.

Supporting Components:

- Power Supply - 12x AA batteries in series at 1.5V each

Datasheet:

- https://pdf1.alldatasheet.com/datasheet-pdf/view/763183/ETC2/XL4015.html

Section 3.4 Pin Table

In this section, the pinout diagram for each individual hardware component/device used in the project will be presented. The connections of the pins between the individual hardware components will also be listed in this section. Each wearable set consists of one IMU alongside a

Beetle and a special wearable set will come with the MyoWare Muscle Sensor to gather EMG data of the individual. The special wearable set will also come with the sling bag that has the Ultra96 placed inside.

Section 3.4.1 Beetle to IMU

The following are screenshots to the pinout diagrams of the Beetle, shown in Figure 14, and IMU, shown in Figure 15, and a table to show the connections from Beetle to IMU. Instead of the original idea of connecting two IMUs to one Beetle, the final design to get a compact circuit consists of only one IMU per Beetle. The connections will be shown across the pin table below.

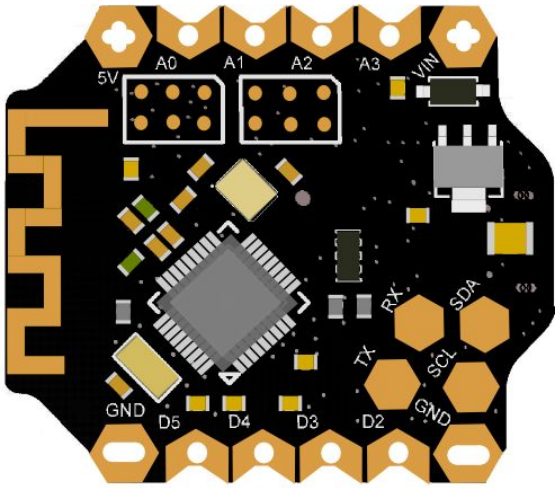


Figure 14: Beetle Pinout diagram (Adapted from https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339)

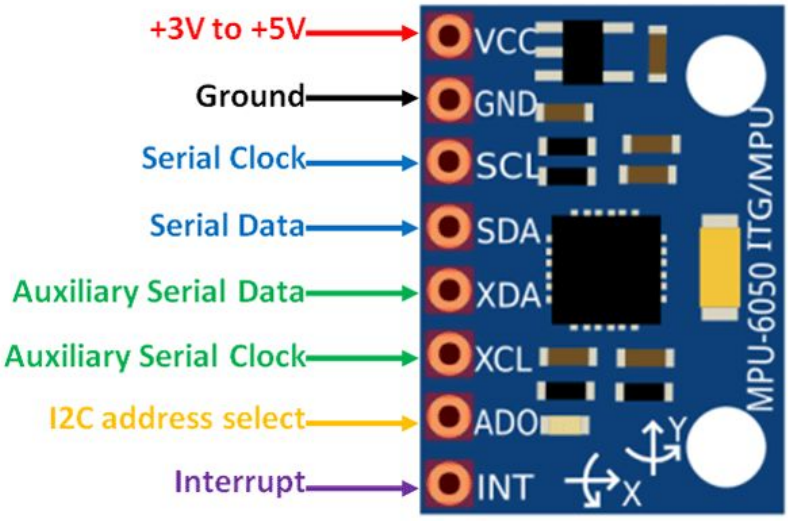


Figure 15: IMU Pinout diagram (Adapted from https://components101.com/sensors/mpu6050-module)

Beetle	IMU
5V	VCC
GND	GND
D2	INT
SCL	SCL
SDA	SDA

Pin table for connections between Beetle and one IMU, this connection will be consistent across the three sets of wearables

Section 3.4.2 Beetle to MyoWare Muscle Sensor

The pinout diagram of the MyoWare Muscle Sensor is as shown across Figure 16. In addition to that, a table to show the connection between the Beetle and the MyoWare Muscle Sensor will be included below as well. The MyoWare Muscle Sensor will only be attached to one of the three sets of wearables, which is known as the special or main wearable set.

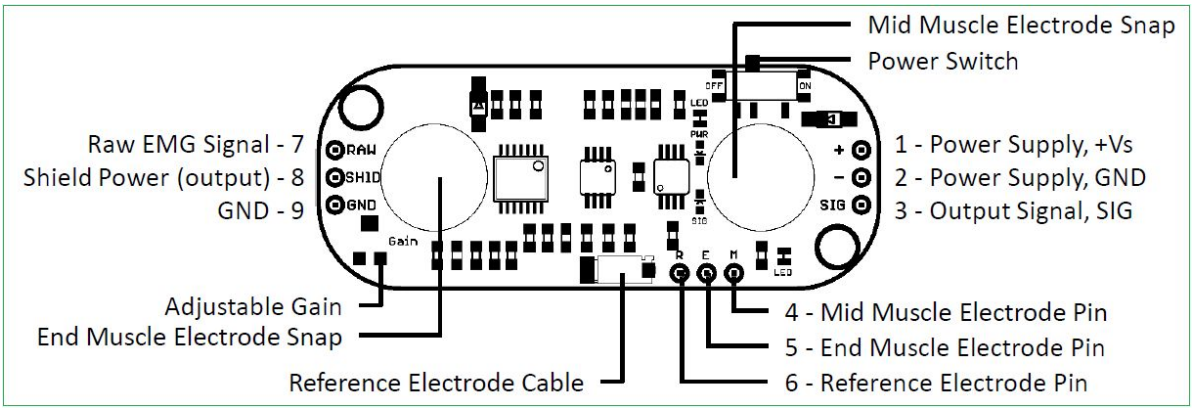


Figure 16: MyoWare Muscle Sensor Pinout diagram (Adapted from http://www.theorycircuit.com/myoware-muscle-sensor-interfacing-arduino/)

Beetle	MyoWare Muscle Sensor
5V	1 - Power Supply, +Vs
GND	2 - Power Supply, GND
A3	3 - Output Signal, SIG

Pin table for connections between Beetle and MyoWare Muscle Sensor, this connection will only be applicable to one of the three sets of wearables

Section 3.4.3 Beetle to LM7805 Voltage Regulator

The pinout diagram of the Beetle has been previously shown in Section 3.4.1 and the pinout diagram for LM7805 Voltage Regulator can be seen across Figure 17. A table showing the connection between the Beetle and LM7805 Voltage Regulator is shown below as well.

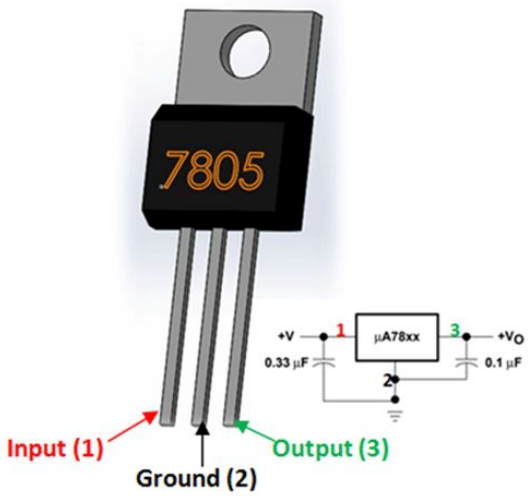


Figure 17: LM7805 Voltage Regulator Pinout diagram (Adapted from https://components101.com/7805-voltage-regulator-ic-pinout-datasheet)

Beetle	LM7805 Voltage Regulator
GND	Ground (2)
VIN	Output (3)

The Input (1) pin of the LM7805 Voltage Regulator is connected to the power supply which is 4x AA batteries in series with each battery at 1.5V.

Section 3.5 Schematics

In this section, the schematics of the various circuitry will be presented. The schematics included in this section are the following:

- Beetle to IMU
- Beetle to MyoWare Muscle Sensor
- LM7805 Voltage Regulator Circuitry

- XL4015 Circuitry

The special wearable set will consist of all the above connections and circuitries whereas the other wearable sets will only include the Beetle to IMU circuitry. In the following subsections, the schematics will be shown.

Section 3.5.1 Beetle to IMU

The following schematic shown in Figure 18 is for the connection between Beetle and IMU. In the schematics, an Arduino Uno schematic was used instead of the Beetle as the Beetle schematic is not available in the software used and since Beetle is an Arduino Uno based board, we will use the schematics of an Arduino Uno. A table is provided after Figure 18 to show the corresponding pin of the Beetle to the Arduino Uno used in Figure 18.

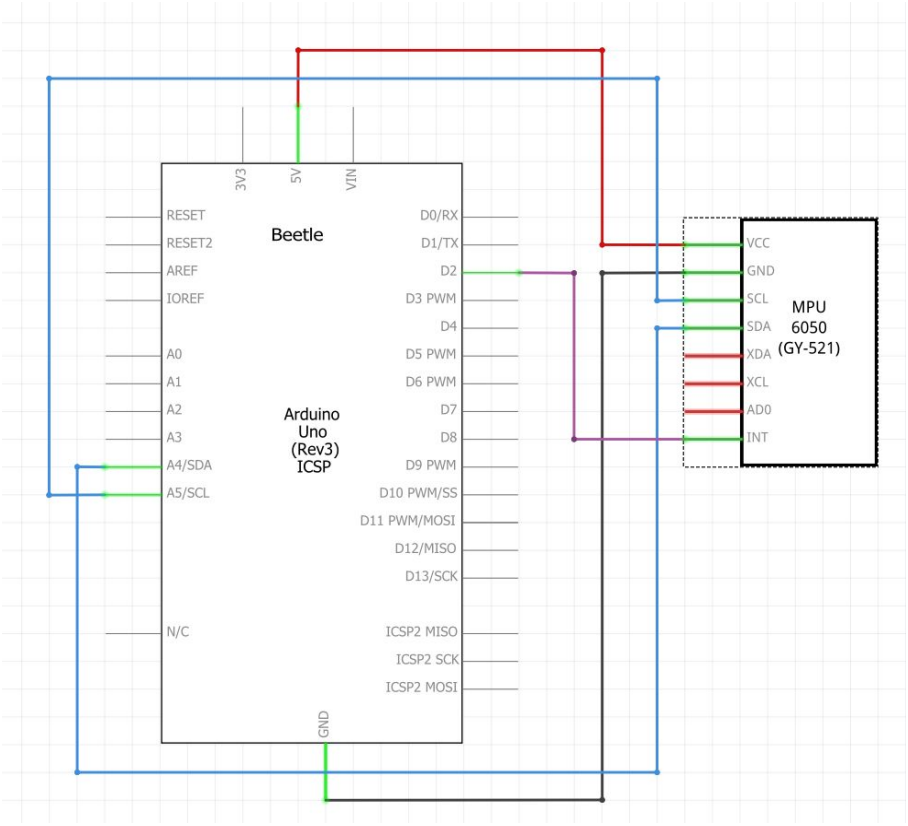


Figure 18: Beetle to IMU Schematic

Arduino Uno	Beetle
5V	5V
GND	GND
A4/SDA	SDA

A5/SCL	SCL
D2	D2

Table: Corresponding pins from Arduino Uno to Beetle

Section 3.5.2 Beetle to MyoWare Muscle Sensor

The following schematic shown in Figure 19 is for the connection between Beetle and MyoWare Muscle Sensor. In the schematics, an Arduino Uno schematic was used instead of the Beetle as the Beetle schematic is not available in the software used and since Beetle is an Arduino Uno based board, we will use the schematics of an Arduino Uno. A table is provided after Figure 19 to show the corresponding pin of the Beetle to the Arduino Uno used in Figure 19.

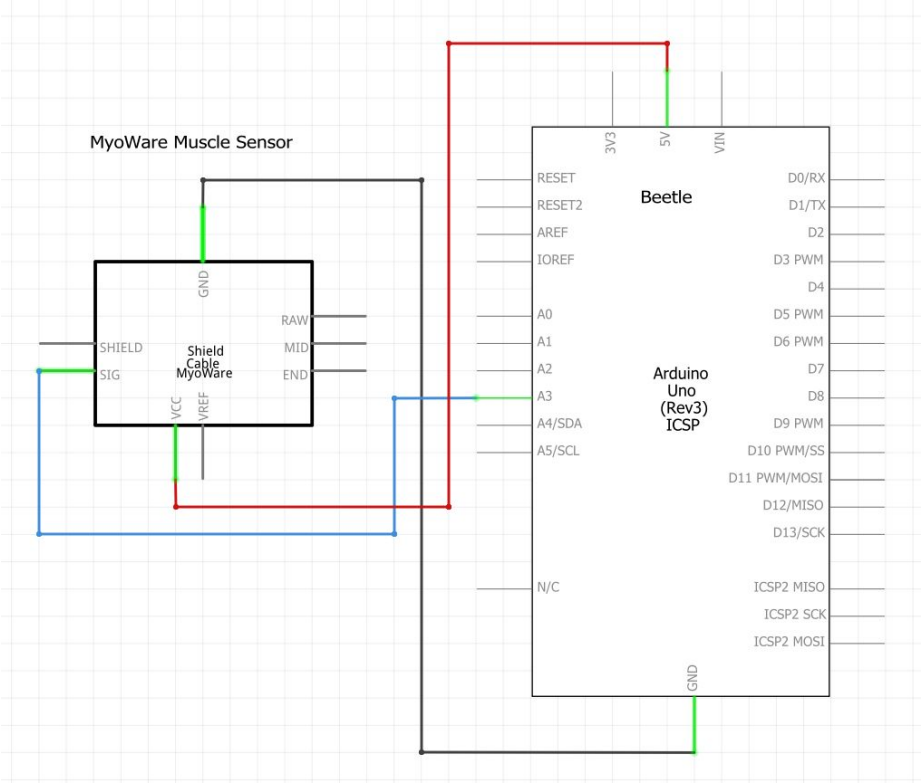


Figure 19: Beetle to MyoWare Muscle Sensor Schematic

Arduino Uno	Beetle
5V	5V
GND	GND
A3	A3

Table: Corresponding pins from Arduino Uno to Beetle

Section 3.5.3 LM7805 Voltage Regulator Circuitry

The following schematic shown in Figure 20 is for the LM7805 Voltage Regulator Circuitry that is for powering up the Beetle and ensuring that voltage is kept at +5V. In the schematics, an Arduino Uno schematic was used instead of the Beetle as the Beetle schematic is not available in the software used and since Beetle is an Arduino Uno based board, we will use the schematics of an Arduino Uno. A table is provided after Figure 20 to show the corresponding pin of the Beetle to the Arduino Uno used in Figure 20.

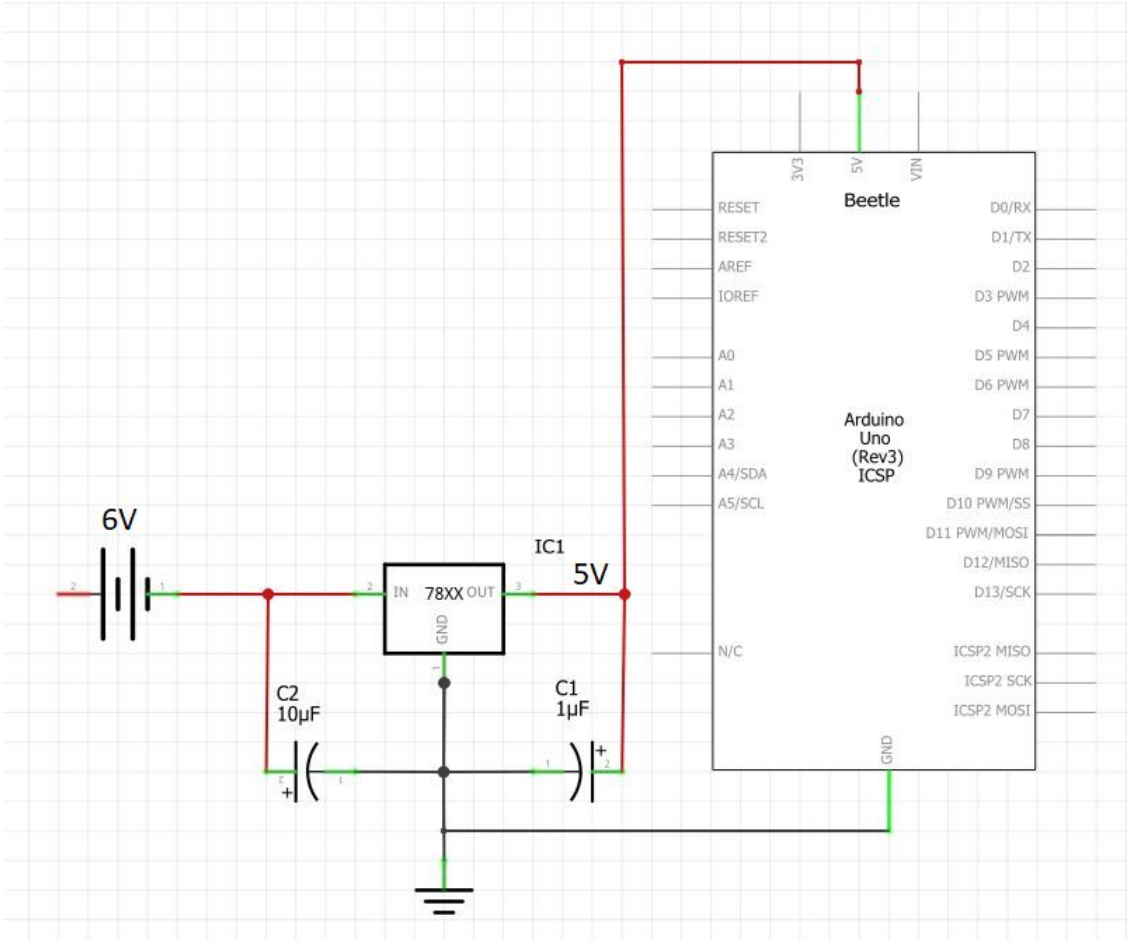


Figure 20: LM7805 Voltage Regulator Schematic

Arduino Uno	Beetle
5V	5V
GND	GND

Table: Corresponding pins from Arduino Uno to Beetle

Section 3.5.4 XL4015 Circuitry

The following schematic shown in Figure 21 is for the XL4015 Circuitry that is for powering up the Ultra96 and ensuring that voltage is kept at +12V. In the schematics, LM2596 was used instead of the XL4015, used in our project, as the schematic of XL4015 is not available in the software used. However, the schematics of XL4015 is highly similar to that of LM2596. Therefore, the circuitry in Figure 21 still holds true to XL4015.

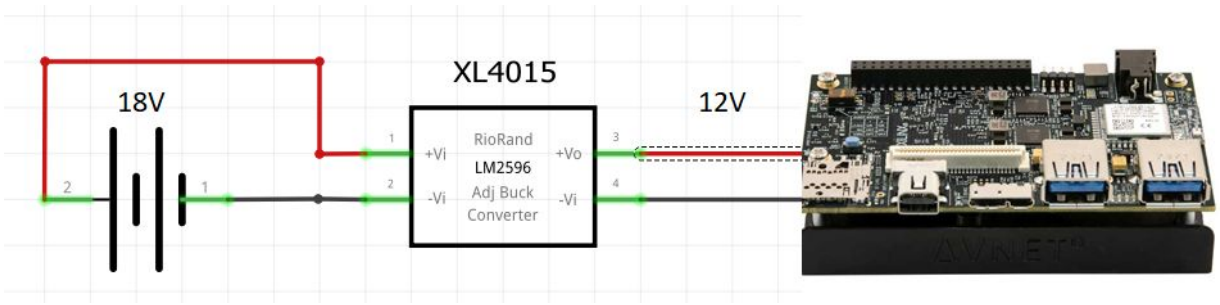


Figure 21: XL4015 Circuitry Schematic

Section 3.6 Operational Voltage and Current

In this section, the operational voltage and current of each individual hardware component/device will be listed in their individual subsections.

Section 3.6.1 Beetle

The operational voltage and current for Beetle is as follows:

- Operational Voltage: 5V
- Operational Current: 0.2mA

The usage of LM7805 alongside the respective capacitors serves the purpose of ensuring that voltage provided to Beetle is at 5V to keep it operational and at the same time ensuring that the voltage provided to Beetle does not exceed 5V which may in turn damage the board.

Section 3.6.2 IMU

The operational voltage and current for IMU is as follows:

- Operational Voltage: 2.375V to 3.46V
- Operational Current: 3.8mA

The IMU sensor is connected to the 5V output of the Beetle ensuring that it is kept at its operational voltage to stay functional and capable of data collection.

Section 3.6.3 MyoWare Muscle Sensor

The operational voltage and current for MyoWare Muscle Sensor is as follows:

- Operational Voltage: 3.3V or 5V
- Operational Current: 9mA

The MyoWare Muscle Sensor is connected to the 5V output of the Beetle ensuring that it is kept at its operational voltage to stay functional and be capable of collecting EMG data from the dancer.

Section 3.6.4 Ultra96-V2 Development Board

The operational voltage and current for Ultra96-V2 Development Board is as follows:

- Operational Voltage: 12V
- Operational Current: 2A - 4A

The usage of XL4015 serves the purpose of ensuring the voltage provided to Ultra96 is at 12V to keep it operational and at the same time ensuring that the voltage provided does not exceed 12V which may in turn damage the board.

Section 3.7 Algorithms and Libraries

In this section, the algorithms and/or libraries used on the Beetle to gather the relevant data will be listed and details of each will be explained in their respective subsections.

Section 3.7.1 IMU

The IMU sensor consists of an accelerometer and gyroscope and data obtained has to be processed before it can be used for classification purposes. After researching on the possible algorithms and libraries for use, we decided to use the I2Vdevlib by jrowberg as his library has a is capable of providing us processed sensor data which makes use of both the gyroscope and accelerometer in the IMU. Our team decided to obtain the yaw, pitch, roll data to determine the rotation of the IMU with respect to its initial position and the world-frame acceleration data that has been adjusted to remove gravity and it is rotated based on known orientation. These data are all obtained via the use of jrowberg's library and they are as shown in Figure 22 below.

```
// display Euler angles in degrees
mpu.dmpGetQuaternion(&q, fifoBuffer);
mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

// display initial world-frame acceleration, adjusted to remove gravity
// and rotated based on known orientation from quaternion
mpu.dmpGetAccel(&accel, fifoBuffer);
mpu.dmpGetLinearAccel(&accelReal, &accel, &gravity);
mpu.dmpGetLinearAccelInWorld(&accelWorld, &accelReal, &q);
```

Figure 22: Jrowberg library to obtain yaw, pitch, roll and world acceleration data from IMU

The first part of the code uses the library dmpGetQuaternion, dmpGetGravity and dmpGetYawPitchRoll from I2Cdevlib to obtain the yaw, pitch and roll value of the IMU sensor. The second part of the code uses the library dmpGetQuarternion, dmpGetGravity, dmpGetAccel, dmpGetLinearAccel and dmpGetLinearAccelInWorld from I2Cdevlib to obtain the world acceleration value of the IMU sensor. However, as dmpGetQuarternion and dmpGetGravity have been called earlier to obtain yaw, pitch and roll, they were not called again in the 2nd code segment. The usage of these libraries to obtain the mentioned values of the IMU sensor is learnt from the example MPU6050_DMP6 that is made by jrowberg as well.

In addition to the usage of jrowberg's library, we also used the library MPU6050 by Electronic Cats which can be found in the Library Manager of Arduino INO. The MPU6050 library provided an example code "IMU_Zero" that allowed us to calibrate the respective IMU sensors to gather their gyroscope and accelerometer offset values. The values were obtained via the use of that library example and were placed into our setup code to scale each IMU sensor individually for minimum sensitivity. The offset code can be seen across Figure 23.

```
// supply your own gyro offsets here, scaled for min sensitivity
mpu.setXGyroOffset(275);
mpu.setYGyroOffset(-75);
mpu.setZGyroOffset(53);
mpu.setZAccelOffset(727);
```

Figure 23: Code to offset gyroscope and accelerometer of IMU

Section 3.7.2 Self-written Thresholding Algorithm

The project requires us to predict both relative positions of the dancers alongside the dance move performed by the dancers and in order to do so, our machine learning personnel created two separate models with one classifying for relative position and one classifying for dance moves. Therefore, we needed to segment our sensor data into walking phase and dancing phase. In order

to differentiate between walking phase and dancing phase data, a self-written thresholding algorithm was created to segment the data up. The code is as shown across Figure 24 below.

```
// Compute the sum of differences
volatile float yawDiffSum = 0.0;
volatile float pitchDiffSum = 0.0;
volatile float rollDiffSum = 0.0;
volatile long accelXDiffSum = 0;
volatile long accelYDiffSum = 0;
volatile long accelZDiffSum = 0;

for (int j = 0; j < THRESHOLDING_SAMPLES; j++) {
  yawDiffSum += yprDiff[j][0];
  pitchDiffSum += yprDiff[j][1];
  rollDiffSum += yprDiff[j][2];
  accelXDiffSum += accelDiff[j][0];
  accelYDiffSum += accelDiff[j][1];
  accelZDiffSum += accelDiff[j][2];
}

if ((abs(yawDiffSum) <= 10 || abs(pitchDiffSum) <= 10 || abs(rollDiffSum) <= 10)
    && (abs(accelXDiffSum) <= 2000 || abs(accelYDiffSum) <= 2000 || abs(accelZDiffSum) <= 2000)) {
  stoppedMoving = true;
}

if (stoppedMoving && ((abs(yawDiffSum) >= 15 || abs(pitchDiffSum) >= 15 || abs(rollDiffSum) >= 15) '
    && (abs(accelXDiffSum) >= 5000 || abs(accelYDiffSum) >= 5000 || abs(accelZDiffSum) >= 5000))) {
  stoppedMoving = false;
  Serial.println("EXCEED THRESHOLD");
}
```

Figure 24: Code to segment walking phase and dancing phase data

In this thresholding algorithm, the sum of differences across five datasets are collected and the rationale behind this is because movements by dancers may not be captured within one dataset and we require it across a few datasets. There are two parts to this thresholding algorithm and they are found in the two if conditional statements across Figure 24. The thresholding values used are obtained via trial and error through human testing by the team.

In the first thresholding, the first if conditional statement, it checks whether the dancer has stopped moving. When the dancer stops moving, the IMU sensor values will drop and be below the thresholding value set in the first if conditional statement leading to the stoppedMoving variable to be set to true and any data collected prior this occurring will be classified as the walking phase data.

In the second thresholding, the second if conditional statement, it checks whether the dancer has started dancing. When the dancer starts dancing, the IMU sensor values will go above the thresholding values set in the second if conditional statement leading to the stoppedMoving

variable to be set to false and any data collected after this occurring will be classified as the dancing phase data.

Section 3.7.3 EMG Time Features Extraction

The EMG features extracted are the max amplitude, mean amplitude and root mean square amplitude in the time domain and mean frequency in the frequency domain. The reason behind the choice of these features is because the assessment of these features can represent muscular effort and fatigue depending on the change in the EMG signal (Toro, et al., 2019). Additionally, it is recommended by researchers that a window period of 100ms~300ms should be used to obtain a set of EMG signals prior to extraction of features (Jirapong, 2019). Therefore, there should be about 100~300 data points as the sampling frequency used in our project for EMG is at 1KHz and that would be a period of 1ms. However, there were limitations to Beetle's memory and the cap of an array size is 200 and to perform fast fourier transform using ArduinoFFT, which is essential for us to obtain frequency domain data, the array size has to be in multiples of 2. Therefore, an array size of 128 is chosen and that also means that the window period for the EMG signal is 128ms given a sampling frequency of 1KHz. In this section, the focus will be explaining in detail the computation of time domain features while Section 3.7.4 will cover the explanation for the computation of mean frequency which is in the frequency domain.

Before diving into explaining how the computation of these features are computed, the following explains the preprocessing of the EMG signal, which has been carried out by the MyoWare Muscle Sensor hardware itself. The EMG signal output by the MyoWare Muscle Sensor is an amplified, rectified and integrated signal, which is also known as the EMG's envelope according to the datasheet given in Section 3.3.4. The following Figure 25 shows the graphs of the rectified and integrated EMG signal.

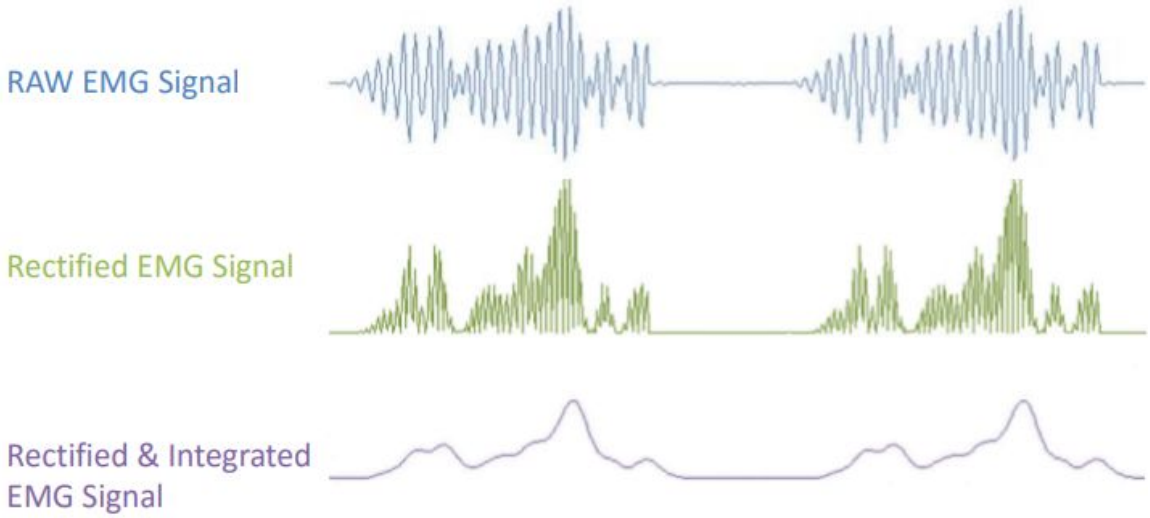


Figure 25: Representative EMG Signal graph from Raw EMG Signal to EMG Envelope (Adapted from https://cdn.sparkfun.com/datasheets/Sensors/Biometric/MyowareUserManualAT-04-001.pdf)

The code on Beetle for the computation of max amplitude, mean amplitude and root mean square amplitude is as shown in Figure 26 and Figure 27 below.

```
double totalSensorValue = 0;
double meanSquaredValue = 0;
float meanAmplitude = 0;
float rmsAmplitude = 0;
float maxAmplitude = -1;

// Collect 128 samples at the frequency of 1kHz, 128ms window period
for (int i = 0; i < SAMPLES; i++) {
  float sensorValue = analogRead(A0);
  float convertedSensorValue = (sensorValue / 1024.0) * 5.0;

  // Compute the sum of all samples collected
  totalSensorValue += convertedSensorValue;

  // Compute the sum of the squared value
  meanSquaredValue += (convertedSensorValue * convertedSensorValue);

  // Gather the largest amplitude of the EMG signal
  if (sensorValue > maxAmplitude) {
    maxAmplitude = convertedSensorValue;
  }

  // Delay for a period of 1ms such that the frequency is 1kHz
  delay(1);
}
```

Figure 26: Code to compute max, mean and root mean square amplitude (Part 1)

```
// Computing the mean amplitude value
meanAmplitude = totalSensorValue / (SAMPLES * 1.0);

// Computing the root mean square of the amplitude
meanSquaredValue = meanSquaredValue / (SAMPLES * 1.0);
rmsAmplitude = sqrt(meanSquaredValue);
```

Figure 27: Code to compute max, mean and root mean square amplitude (Part 2)

Prior to computing the max amplitude, mean amplitude and root mean squared amplitude, the sensor value was first converted from 0~1023 into its voltage equivalent value of 0~5V and this is to prevent integer overflow issues that may arise when huge values are constantly summed together into an integer or long variable. The formula to convert the value of 0~1023 into 0~5 is as shown across Figure 28. The max amplitude is obtained via a check against all values obtained during the window period and the maximum value seen across the window period will be assigned to this variable as shown in Figure 26. The mean amplitude and root mean square amplitude is computed based on formulas shown across Figure 29 below and these formulas are obtained from the research paper by Toro, et al (2019).

$$\frac{1023}{5} = \frac{ADC\ Reading}{Analog\ Voltage\ Measured}$$

Figure 28: Formula for Analog to Digital conversion (Adapted from https://learn.sparkfun.com/tutorials/analog-to-digital-conversion/all)

$$MAV = \frac{1}{N}\sum_{i=1}^{N}|x_i|$$

$$RMS = \sqrt{\frac{1}{N}\sum_{i=1}^{N}x^2}$$

Figure 29: Formula for mean and root mean square amplitude (Toro, et al., 2019)

The code written in Figure 26 and Figure 27 follows the formulas in Figure 29 closely. For mean amplitude, the sum of all amplitudes across the window period is first computed and it is then divided by the amount of samples, which is 128 in this code. For root mean square amplitude, the

sum of all squared amplitudes across the window period is first computed and it is then divided by the amount of samples followed by performing a square root on the outcome.

It is stated in the research paper by Toro, et al. (2019) that an increase in the signal amplitude implies muscle fatigue and therefore, the three time domain features of max, mean and root mean squared amplitude are used. Therefore, it means that when there is an increase in these three features, the dancer with the EMG attached is defined to be experiencing muscle fatigue. On top of using time domain features to determine whether the dancer is experiencing muscle fatigue, our code also includes a frequency domain feature, mean frequency, to determine whether the dancer is experiencing muscle fatigue. The computation of the mean frequency in frequency domain will be explained in detail in the following section 3.7.4 and details of how this feature determines muscle fatigue will also be explained in that section.

Section 3.7.4 EMG Frequency Features Extraction

As mentioned in Section 3.7.3, the EMG signal used is the EMG envelope signal and therefore the preprocessing of the has already been completed by the MyoWare Muscle Sensor. The focus of this section will be on how the computation of mean frequency is completed in Beetle. Prior to obtaining the mean frequency of the EMG signal across its window period, it is required of the data to be in frequency domain. However, as the sensor data collected from the MyoWare Muscle Sensor is in time domain, conversion to the frequency domain has to be carried out and in our code, we made use of the ArduinoFFT library. The guide to installing and usage of ArduinoFFT library is in this link: https://www.norwegiancreations.com/2017/08/what-is-fft-and-how-can-you-implement-it-on-an-arduino/.

The following Figure 30 shows how the ArduinoFFT library is used in the code and how the computation of mean frequency is carried out.

```
double meanFrequency = 0;
long long powerSpectrum = 0;
long long powerSpectrumAndFreq = 0;

// Collect 128 samples at the frequency of 1kHz, 128ms window period
for (int i = 0; i < SAMPLES; i++) {
  float sensorValue = analogRead(A0);
  float convertedSensorValue = (sensorValue / 1024.0) * 5.0;

  // Collection of data for ArduinoFFT
  vReal[i] = sensorValue;
  vImag[i] = 0;
  // Recording the frequency
  if (i < SAMPLES/2) {
    frequency[i] = ((i * 1.0 * SAMPLING_FREQUENCY) / (SAMPLES * 1.0));
  }

  // Delay for a period of 1ms such that the frequency is 1kHz
  delay(1);
}

// Arduino FFT
FFT.Windowing(vReal, SAMPLES, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
FFT.Compute(vReal, vImag, SAMPLES, FFT_FORWARD);
FFT.ComplexToMagnitude(vReal, vImag, SAMPLES);

// Serial.println("FFT");

// Computing the mean frequency based on the formula in the research paper, sum of frequency multipled power spectra over sum of power spectra
// The power spectra is computed by squaring the amplitude
for (int i = 0; i < SAMPLES / 2; i++) {
  // Computing the power spectra value
  // Serial.println(frequency[i]);
  // Serial.print("vReal: ");
  // Serial.println(vReal[i]);
  double powerSpec = vReal[i] * vReal[i];
  powerSpectrum += powerSpec;
  powerSpectrumAndFreq += (powerSpec * frequency[i]);
}

// Computing the mean frequency
// Serial.println(powerSpectrumAndFreq);
// Serial.println(powerSpectrum);
meanFrequency = (powerSpectrumAndFreq * 1.0) / (powerSpectrum * 1.0);
```

Figure 30: Code for ArduinoFFT library and computation of mean frequency

The code first gathers the sensor value into the array of vReal and places the value of 0 into the array of vImag. Apart from placing the sensor values and 0 into their respective array, the code also computes the frequency of each sample based on the guide by Mads Aavik (2017). The ArduinoFFT library is then used to convert the data from the time domain to the frequency domain by calling the functions FFT.Windowing, FFT.Compute and FFT.ComplexToMagnitude. After computing the data into frequency domain data, the mean frequency can then be computed and the computation follows the formula as shown in Figure 31 below.

$$MNF = \frac{\sum_{j=1}^{M} f_j P_j}{\sum_{j=1}^{M} P_j}$$

Figure 31: Formula to compute mean frequency (Toro, et al., 2019)

The code in Figure 30 to compute the mean frequency of the window period follows the formula shown in Figure 31 strictly. As seen in the code, it first computes the power spectrum which is essentially the square of the signal's amplitude in the frequency domain (Wikipedia). Upon computing the power spectrum, the code then computes the summation of the power spectrum which is the denominator of the formula shown in Figure 31 and at the same time, it computes the summation of the frequency multiplied by the power spectrum which is the numerator of the formula shown in Figure 31. The numerator and denominator are then put together in the last line of the code shown in Figure 30 to obtain the mean frequency of the window period.

It is stated in the research paper by Toro, et al. (2019) that a decrease in the mean frequency implies muscle fatigue and therefore, this frequency domain feature is extracted to help determine whether the dancer is experiencing muscle fatigue. As mentioned previously in Section 3.7.3, this feature is used alongside the three time domain features, namely max, mean and root mean squared amplitude, to identify whether the dancer with the EMG attached is facing muscle fatigue. Therefore, when there is an increase in the time domain features value and a decrease in the frequency domain feature value, we classify the dancer to be experiencing muscle fatigue. This information was passed to the dashboard personnel to show whether the dancer with EMG attached is experiencing muscle fatigue or not.

Section 3.8 Changes made

In this section, the changes that were made across the weeks since the submission of the initial design report and the individual subcomponent test will be explained in detail. Apart from covering the changes that were carried out, this section will also talk about the challenges faced across the weeks that invoked these changes made, if any. This section will be split into various subsections to cover the changes made to different components of the project. These subsections include the report changes from the initial design report to the final report for the hardware sensors section, the design changes to the form of the system and the changes made to the hardware components since week 7.

Section 3.8.1 Report changes

There were several changes made from the initial design report to the current final report. In the initial design report, only the hardware components alongside their pin tables, schematics, operational voltage and current, and algorithms and libraries were presented to give a general idea of the direction that the hardware sensors will be moving towards. In this final report, apart from adding more details to those sections from the initial design report, new sections were added. I will be detailing the changes that were made to each section in addition to sharing the new sections that were added.

First and foremost, an introduction section has been added to give an introduction to what the report will cover about the hardware sensors portion. Additionally, a section on brief overview has also been added to cover the requirements of the hardware personnel alongside giving a brief outlook on what has been carried out to achieve these requirements.

Secondly, new hardware components were introduced into this project to regulate the voltage output from our power source such that our hardware components, namely Beetle and Ultra96, get their operational voltage to power up and at the same time control the voltage input to these components such that the components do not get damaged. The information on these hardware components were added into Section 3.3, 3.4 and 3.5 accordingly. Additionally, new schematics drawings were made in Section 3.5 as the initial design has changed during the course of the project.

Thirdly, descriptions have been added to Section 3.6 detailing the components used to maintain the hardware components at their operational voltages to ensure the hardware components are functional. In addition to that, a revamp has been made to Section 3.7 to give greater details on the algorithms and libraries used in this project.

Finally, there were two new sections added to the back of this report as well. Section 3.8 has been added to reflect the changes that have been made across the course of the project. Additionally, as an abrupt end to the project has been made due to COVID-19, Section 3.9 has been added to detail the improvements that could possibly be made to the hardware components alongside how the hardware component can possibly be extended to another application to better realize the component.

Section 3.8.2 Design changes

After the individual subcomponent test in week 7, further improvements were made to the hardware components to ensure its stability for the demonstrations. During the individual subcomponent test, the Beetle and IMU set were mounted on gloves and socks as shown across Figure 32 and Figure 33 below.

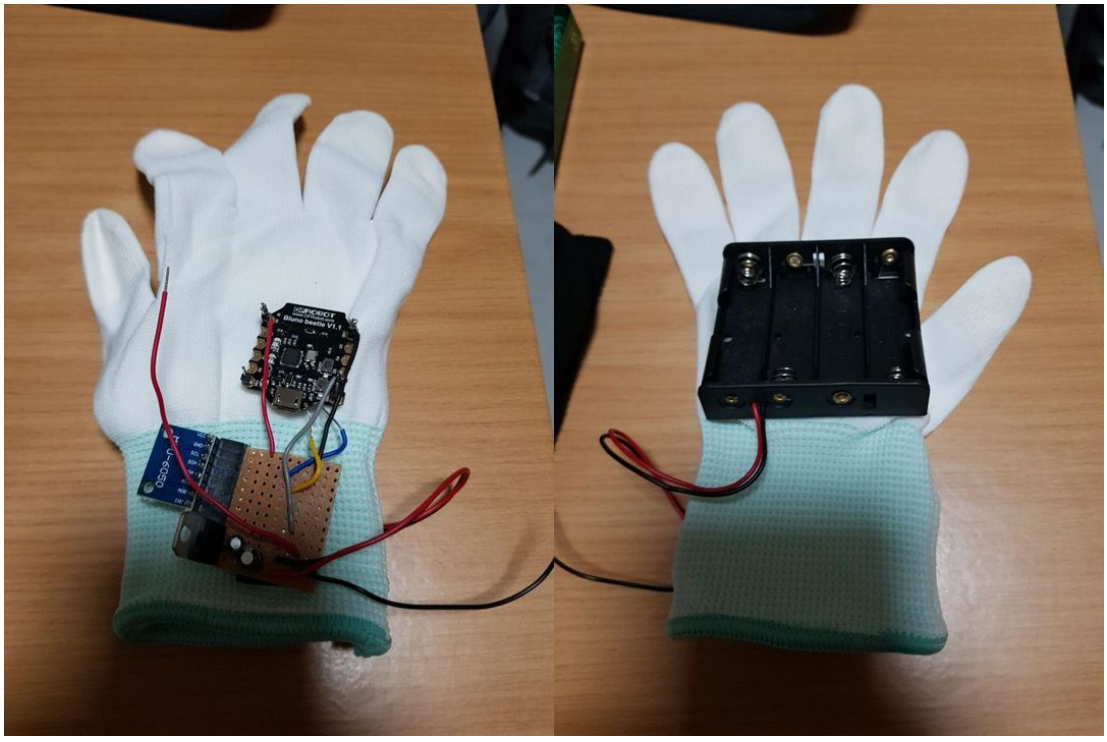


Figure 32: Gloves design as of week 7

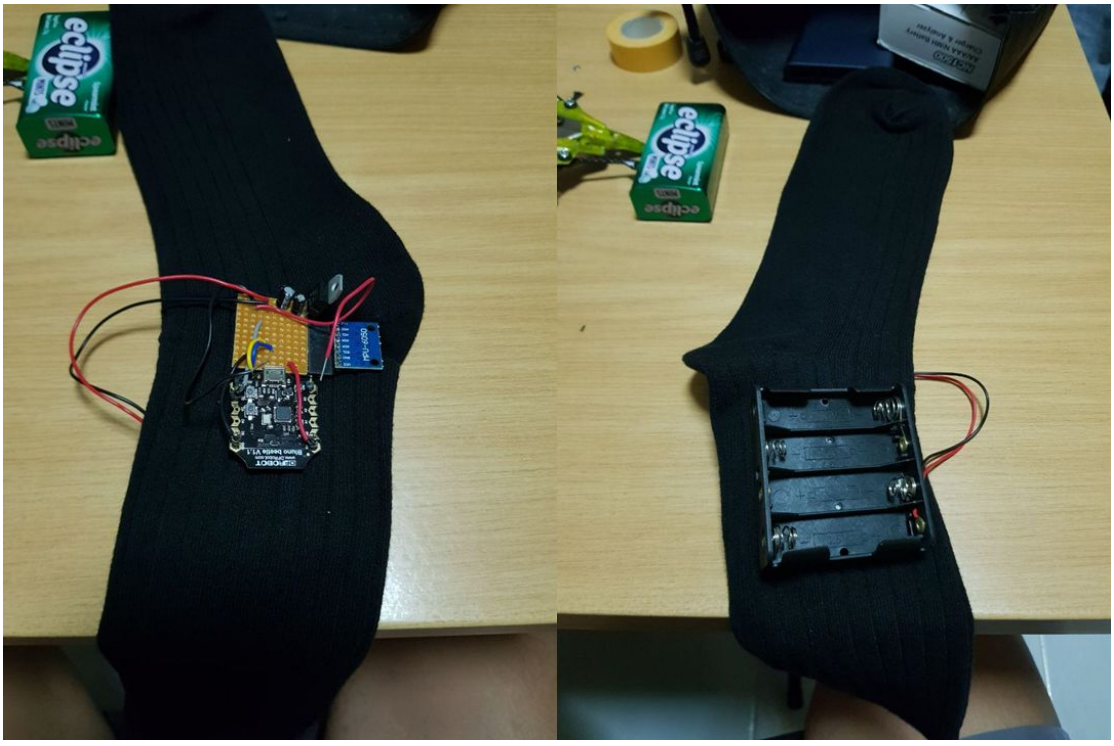


Figure 33: Socks design as of week 7

During the course of intensive dancing and extensive testing after the individual subcomponent tests, the adhesive holding the stripboard circuitry and the batteries started to wear off and the

hardware components started falling off. Therefore, changes were made to this initial design and instead of attaching these components to gloves and/or socks, the circuitries were placed into pouches and these pouches were attached to the dancer via a Velcro strap. The pouches used to store the circuitry can be seen across Figure 34 below.

Figure 34: Pouch with IMU and Beetle circuitry

These pouches kept the hardware circuitry safe from any harm and with the hardware kept in a pouch, there were no chances of it falling from the dancers. Additionally, the Velcro strap attached to the pouch was secured tightly to the pouch and it helped to secure the pouch tightly to the dancer. The Velcro strap was also made long enough such that it can cater for people of different sizes. Additionally, instead of using six Beetles as intended initially, we dropped the three Beetles that were supposed to be attached to the ankles of the dancers. The reason behind this drop is because we were unable to perform concurrent data collection from the six Beetles to the Ultra96. There were many times where data were collected sequentially and this is a call for concern as we needed the Beetles on the same dancer to have concurrent data collection. Therefore, the Beetles that were supposed to be attached to the legs were dropped and only three Beetles remained.

There were also changes made to the MyoWare Muscle Sensor to Beetle circuitry, initially the idea was to use a sports armband that is capable of holding a smartphone to hold the Beetle and attaching the MyoWare Muscle Sensor to the user. The sports armband intended for use is as shown across Figure 35. However, the armband was long in length and it came in the way when attaching the MyoWare Muscle Sensor onto the user. Therefore, instead of using a sports armband, a pouch was used and this pouch can be seen across Figure 36 below.

Figure 35: Sports armband

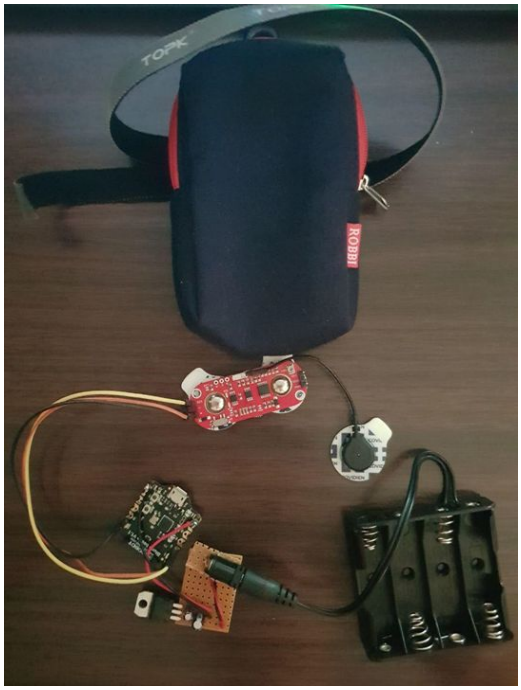


Figure 36: Pouch to hold Beetle for MyoWare Muscle Sensor

The size of this pouch was adequate to store Beetle and its circuitry and it does not come in the way of attaching the MyoWare Muscle Sensor onto the dancer. The same Velcro strap was used

to attach this pouch to the dancer and this strap was made longer to cater for people of different sizes.

Section 3.8.3 Sensors algorithm changes

In terms of the hardware components aspect, there were no changes to the sensors used since the individual subcomponent test. However, we made changes and updates to the algorithms and libraries used on Beetle to gather more sensor data alongside segmenting the sensor data as required of the machine learning models.

Initially, we only intended to use the processed sensor values of yaw, pitch and roll to correctly differentiate between the different relative position and dance moves. However, this has been proven to be ineffective as there were dance moves that did not have much changes to those values as there were no rotations in the three axes during the dance move. Additionally, when moving from one position to another, the rotation in the three axes are minimal dependent on how the dancer moves to their next position. Therefore, the algorithm on Beetle was changed to include gathering of world acceleration data. The world acceleration is the acceleration of the IMU in the three axes that has been adjusted to remove gravity and rotated based on known orientation. This algorithm has been explained in detail in Section 3.7.1 previously.

Additionally, a self-written thresholding algorithm was written to segment the sensor data gathered into two segments, one being the walking phase data and another being the dancing phase data. This thresholding algorithm is essential as it segments our data into two segments and these segments are passed into separate machine learning models to predict the relative position of the dancers and the dance moves performed by the dancers separately. This algorithm has been explained in detail in Section 3.7.2 previously.

Section 3.9 Research: Extension of Hardware 1 Sensors

In this section, discussion on how the hardware sensors components can be extended will be presented. I will first discuss some possible improvements that can be carried out on the current hardware components/devices in terms of design and circuitry to make the hardware sensors more reliable, effective or efficient for this project without changes to the project specifications or requirements. Following that, I will bring the discussion to how these components' functionalities can be brought to another application or work to better realize the component and fully utilise the functionalities.

Section 3.9.1 Further improvements on hardware components

In this section, I will be discussing the improvements that could be carried out to enhance the hardware components to improve the quality of this project. There are two main improvements that I would be discussing with the first one being making changes to the batteries used and the second being usage of more sensors to gather more data to better train our machine learning models and increase the accuracy of our predictions.

The reason behind the first improvement, changes to the batteries used, is because the current batteries used for this project are too bulky and heavy. On top of that, the batteries used were not rechargeable batteries. If we changed to use rechargeable batteries which are usually at 1.2V per cell, there will be a need for at least 5 batteries in series which will lead to the circuit becoming more bulky and heavier. Therefore, we wanted to change the batteries used currently from AA batteries to round batteries and there are two round batteries that we were considering to use. The first round battery would be the Sony CR2032 which is a 3V per cell battery but this battery is not rechargeable which means it is not eco-friendly as more batteries will be required when the batteries go flat. The second round battery that could be used would be the MaxWell ML2032 which is a 3V per cell battery as well but this battery is rechargeable. These round batteries are less bulky and they are definitely lighter as compared to AA batteries. Therefore, this improvement will make the current circuit more compact, lighter and less bulky.

The reason behind the second improvement, having additional sensors, is because we felt that gathering more data from other body parts of the dancer will allow us to better differentiate between the relative positions and dance moves. One of the methods we thought of was to bring back our original design from the initial design report which was to connect 2 IMUs to a Beetle and that will allow us to attach one IMU back to the ankle without having face the issue of sequential data collection as the data will be collected on the same Beetle but in this design, there will be a challenge which is having to consider proper wiring that does not fall off easily and does not obstruct the dancer's dancing. Apart from adding another IMU, there are also other sensors that can be added to get different data such as the flex sensors which can be attached to the elbow to detect elbow bending movements to differentiate between dance moves. However, the main challenge to adding more sensors to one Beetle still applies which is to consider how wiring is to be wired in a way such that it does not hinder the dancer's movement in addition to not falling off and the positioning of the Beetle on the dancer will also be crucial.

Therefore, these two improvements will improve the quality of the project by first making the wearable less bulky and have a lighter weight and second by providing more data of different motion from the dancers to the machine learning models such that it can better differentiate and predict the relative positions alongside the dance moves.

Section 3.9.2 Realizing hardware components (Physiotherapy exercises)

In this section, I will be discussing how my component, hardware sensors, can be extended to another application. I propose a change in the project specifications to assist in physiotherapy exercises instead of predicting relative positions and dance moves in a dance activity. This proposal will be based on two prior works which are the usage of virtual reality game development using accelerometers for post-stroke rehabilitation by Gustavo, et al. (2019) and using surface electromyography in physiotherapy research by Karen and Jennifer (2014). I will be detailing the change of project specifications followed by how these two works support the proposed idea which is potentially better than the current capstone project design.

The current project specifications require us to use sensor data to predict the relative positions between three dancers alongside the dance move that they are performing. In addition to that, we were required to include an EMG sensor to identify whether the dancer is experiencing muscle fatigue. The same set of sensors used in the current project can be brought over to carrying out physiotherapy exercises instead of dancing which will potentially assist physiotherapists around the globe with their duties and lower the strain on our medical workers.

In the paper by Gustavo, et al. (2019), accelerometers were used to identify limbs movements made by the user in their virtual reality game and this information is sent to a smartphone that is executing the developed game. Instead of having a virtual reality game, we can send this information to a server that could perform analysis on the data to identify what sort of movement has been completed by the user and this data can then be analysed by physiotherapists to identify whether any changes has to be made to their rehabilitation programme for the user or identify any potential danger that may arise if the movement completed by the user is improper. In addition to using the accelerometer data, the incorporation of surface EMG data could assist in physiotherapy exercises as well (Karen and Jennifer, 2014). The use of surface EMG data can potentially help to identify the fatigue level of the user's muscles which will allow the physiotherapist to better tweak the rehabilitation programme for the user with knowledge of his/her movement on top of the fatigue level of their muscles when undergoing the programme.

Therefore, by making a tweak to the project specifications to assisting in physiotherapy exercise instead of predicting relative positions and dance moves, we are able to better utilise all the functionalities of the hardware components. In the original project specification, the usage of the surface EMG data only tells whether a dancer is facing muscle fatigue during the dance but this information is not essential in the project scope. However, if we tweak the project specification to assisting in physiotherapy instead, the fatigue level of the muscle will be of greater use to the physiotherapist in allowing him/her to better understand the user's needs and adjust their

rehabilitation programme accordingly. Apart from better utilising the functionalities of the sensor data, this tweak to the project specification will lower the strain on physiotherapists in having to identify movement and fatigue levels manually using their eyes and it will benefit the medical industry greatly.

Section 4 Hardware Details: Hardware 2 - FPGA

Section 4.1 Introduction

In this section, we will be discussing the hardware acceleration of our neural network. We will talk about the 2 approaches(hls4ml and FINN Compiler) that we tried during the course of this project even though we only stick with one of them at the end. The reason for this being is because the inner workings of FINN are in many ways similar to how hls4ml runs and so the knowledge that we gain from one is useful for the other. FINN is in a way easier than hls4ml to use since it is designed in a manner such that users can generate a bitstream to machine learning model with Python alone. In fact, FINN itself provides an end-to-end example which generates a Python Script that loads the bitstream onto the Ultra96, and then performs the neural network inference.

On the other hand, hls4ml requires the user to work with Vivado HLS(High Level Synthesis) projects which generate IPs(Intellectual Property), and connect the generated IP to other IPs within the Block Design Diagram in Vivado itself. Internally, FINN also creates HLS projects to generate IPs. The connection of the Block Design Diagram within FINN is also automated. Therefore, one can think of hls4ml being a lower level tool when compared to FINN.

Section 4.2 Hardware Acceleration with hls4ml tool (Before week 7)

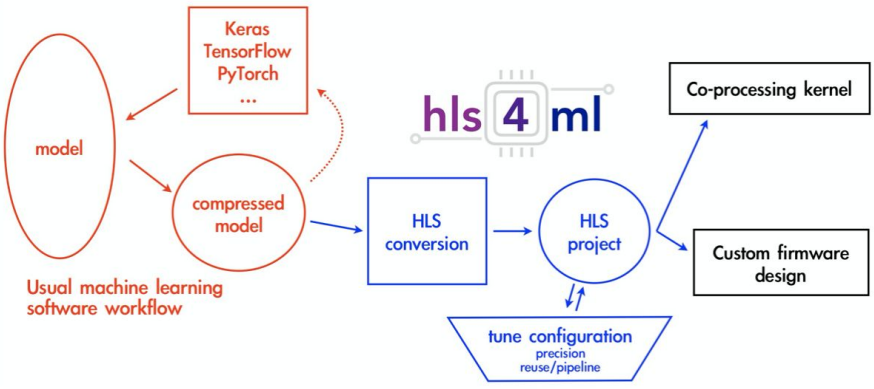


Figure 37: Hls4ml workflow

In this section, we will discuss our attempt to utilize the hls4ml tool to generate a Neural Network. This is the tool of choice up until week 7. Eventually, we settled on using the FINN compiler, but having an in-depth understanding of how to utilize this tool to generate a hardware accelerated neural network gives us a better understanding of how the FINN compiler works as the underlying software/technology that both tools utilizes to generate the bitstream is largely similar.

Hls4ml provides us with an efficient and fast translation of machine learning models to HLS projects. HLS allows us to use higher level languages such as C, C++, without the need to work at the Register Transfer Level(RTL) on Vivado to directly target Xilinx Devices. The resulting HLS project can then be used to produce an IP which can be plugged into more complex designs or be used to create a kernel for CPU co-processing. A more detailed workflow for hls4ml is illustrated in the image above.

Models from open-source packages for training machine learning algorithms are translated into an HLS implementation by hls4ml. When utilizing this tool, PyTorch is our choice of machine learning library. At the current time of writing, hls4ml only supports the generation of Multilayer Perceptron(MLP) on Pytorch Models. However, it has been shown that an accurate MLP Neural network can be trained for human activity recognition(Sousa Lima et al., 2019). Thus, we came to the conclusion that the support of only MLP is already sufficient for our use case.

During our attempt to utilize hls4ml, we trained our neural network with a kaggle dataset. In this dataset, it contains smartphone inertial sensor data, and different types of human activities as the labels. In this dataset, there are 6 different labels, and we utilized 200 accelerometer data points as inputs into the neural network.

Section 4.2.1 Generating HLS and IP with hls4ml tool

With hls4ml, the user can control aspects of their model such as size/compression, precision of calculations, dataflow/resource reuse, control parallel or serial model implementations with varying levels of pipelining. During the attempt to generate the HLS project with hls4ml, We find that tuning the value for resource reuse is the most tricky as the expected utilization estimate can easily exceed the amount of available hardware resource on Ultra96's Programmable Logic(PL). It is also unwise to overestimate the amount of resources that needs to be reused as it could increase the execution speed of inference due to the reduction of parallelization in computation.

After generating the HLS project, the generated HLS source code by hls4ml is insufficient to generate a functioning IP block. This is because the input and output interfaces of the IP block

needs to be manually defined within the HLS source code, and we need to create an appropriate C struct to accommodate the new interface. In our modification, we chose to support the AXI Interface by utilizing the Interface Pragma as follows.

```
#pragma HLS INTERFACE axis port=input_stream,output_stream
#pragma HLS INTERFACE s_axilite port=return
```

Figure 38: Declaring the type of input/output interface with the HLS Interface Pragma

The above code snippet defines both the input_stream and output_stream as AXI-4 Stream(AXIS) Ports so that we can utilize the AXI-4 Stream protocol. We chose to use the AXI-4 Stream protocol over the AXI memory-mapped protocol because our application focuses on data-centric and data-flow paradigm. At the lower level of operation, when we compare the AXI-4 Stream to the memory mapped AXI protocol, the mechanism to move data between IP more efficient using the Stream protocol, and there is no need to involve the concept of address within a system memory space for each block of data that is transferred. Instead, a start and end signal are used to signal the start and end to a stream of data.

From the above code snippet, the s_axilite interface is also defined for the control interface of the IP block. “return” is what the HLS interface calls the control interface of the IP block. The control interface is required to start the operation of the IP block, and also to reset its operation each time we finish a transaction.

As mentioned previously, we also have to declare a suitable structure within the HLS source code in order to make the AXI-4 Stream protocol work. The code snippet below shows the declaration of both the input and output stream utilizing the AXI-4 Stream protocol.

```
typedef struct input_stream {
    struct {
        ap_fixed<32,3> input;
    } data;
    ap_uint<1> user;
    ap_uint<1> last;
} input_stream;

typedef struct output_stream {
    struct {
        ap_fixed<32, 3> output;
        //ap_uint<96> output;
    } data;
    ap_uint<1> user;
    ap_uint<1> last;
} output_stream;
```

Figure 39: The declared structs for input and output AXI streams

In the HLS code, the pointer to the declared structs are passed in as arguments to the overall function. Upon dereferencing the pointer, will we then be able to access and store the relevant data generated by the IP block within each struct.

```
for (int i = 0; i < 200; i++) {
    input[i] = input_stream->data.input;
    input_stream++;
}
```

Figure 40: Code snippet for accessing data from input stream

```
nnet::dense_large<input_t, layer2_t, config2>(input, layer2_out, w2, b2);

nnet::softmax<layer2_t, result_t, softmax_config3>(layer2_out, layer3_out);
```

Figure 41: Code snippet for passing input data to generated HLS neural network function

```
for (int i = 0; i < 6; i++) {
        output_stream->data.output = layer3_out[i];
        output_stream->last = i==5 ? 1: 0;
        output_stream++;
}
```

Figure 42: Code snippet for storing data into output stream

One point to take note is that the "last" variable has to be explicitly set to 1 when we pass the last set of data into the output stream. This sets the TLAST signal to indicate that the last data has been transferred and this is required as part of the AXI-4 Stream Protocol. The TLAST signal informs the AXI Direct Memory Access(DMA) when all the data has been transferred into memory.

Section 4.2.2 Block Design Diagram

After making the above changes to the generated HLS code, we are able to generate an IP block which needs to be connected to the other relevant IP blocks to control its operation.

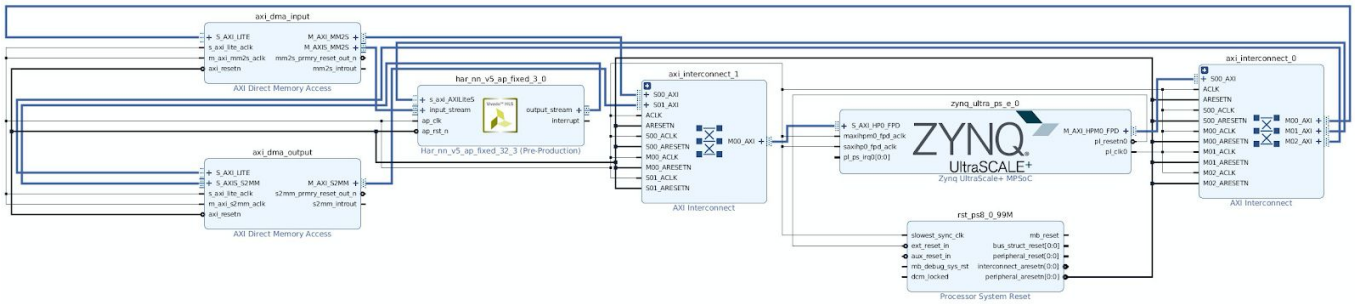


Figure 43: Connected Block Design diagram with generated IP block

The input stream of the generated IP block is connected to the AXI_DMA M_AXIS_MM2S (Master AXIS) port and the output stream is connected to the S_AXIS_S2MM (Slave AXIS) port. Additionally, the s_axi_AXILiteS port is also connected to the M_AXI_HPM0_FPD port of

the Zynq UltraScale+MPSoC IP via the AXI_Interconnect IP. This is required to pass control signals to the IP Block.

Section 4.2.3 Python Script to perform Neural Network inference on FPGA

```
NN_IP_BASE_ADDRESS = 0xA0010000
ADDRESS_RANGE = 0x10000
ADDRESS_OFFSET = 0x0


mmio = MMIO(NN_IP_BASE_ADDRESS, ADDRESS_RANGE)
ap_start = 0x1
ap_auto_reset = 0x80

mmio.write(ADDRESS_OFFSET, ap_start|ap_auto_reset)
```

Figure 44: Code snippet to start and auto reset generated IP block

After generating the bitstream and loading it onto the FPGA of Ultra96. We need to interact with it via the various PYNQ libraries. Firstly, we are required to send certain control signals to the IP Block to start its operation, and we did it via the MMIO class of the PYNQ library. The MMIO Class allows us to access addresses in the System Memory mapped, and is often used to access/control peripherals when the registers/addresses of the peripheral can be accessed.

The memory mapping of each control signal is defined within the hwh file, which is created during the process of generating a bitstream. We can also observe and manually define the address mapping within the address editor of the Block Design. In the above example, the base_address of all control signals for our generated IP block is at 0xA0010000. The address range is the range of address for all control signals. Therefore, given a base_address of 0xA0010000 and an address range of 0x10000, we send control signals to the IP Block by writing to addresses from 0xA0010000 to 0xA001FFFF. To start the operation of the IP Block, we are required to send an ap_start signal, which can be controlled by writing to the 0x1 bit. The ap_auto_reset signal resets and starts the IP block again after each transaction with the DMA and can be controlled by writing to the 0x80 bit.

```
input_buffer = xlnk.cma_array(shape=(200,), dtype=np.float16)
output_buffer = xlnk.cma_array(shape=(6,), dtype=np.float16)

for x in range(len(test_data_array)):
    input_data = test_data_array[x][:-1]
    for i in range(200):
        input_buffer[i] = input_data[i]

    dma_send.sendchannel.transfer(input_buffer)
    dma_recv.recvchannel.transfer(output_buffer)
    dma_send.sendchannel.wait()
    dma_recv.recvchannel.wait()

    if labels[test_data_array[x][-1]] == np.argmax(output_buffer):
        correct_pred += 1;

accuracy = 100. * correct_pred / len(test_data_array)
print('Accuracy: {}'.format(accuracy))
print ("Process time: " + str(time.time() - start) + "s")
```

Figure 45: Code snippet for set up of DMA transfer

The above code snippet is the main section that sets up each DMA transfer to make an inference with the hardware accelerated Neural Network. Two DMA objects, dma_recv and dma_send were defined. An input and output buffer is utilized to send and receive data to and from the DMA. The input buffer is initialized as an array of 200 16 bit floats, which corresponds to 200 input nodes in our neural network, and the output buffer is initialized as an array with 6 16bit floats, which corresponds to the 6 different labels in our dataset.

The input buffer is loaded with the data, and the data is passed into the sending DMA. The output buffer simply waits to receive from the receiving DMA. After receiving the data from the DMA, the argmax of the output buffer is then used to determine the predicted label by the neural network.

Section 4.2.4 Issues faced

On our example dataset, we managed to get an accuracy of 85% with our hardware accelerated neural network. Our neural network architecture has an input layer of 200 nodes, and another output layer of 6 nodes, with a hidden layer of 50 nodes. Our achieved accuracy is slightly lower than the software implemented neural network counterpart which has an accuracy of 96%. However, the main reason that discourages us from continuing with this tool is our inability to solve the following runtime error.

```
RuntimeError: DMA channel not idle
```

This error happens frequently, and whenever it happens, the DMA channel will not get out of the non-idle state, hence preventing us from making any more inference. Our guess on why this happens is that the DMA channel is unable to receive the TLAST signal reliably all the time. The only known way to resolve this issue is by resetting the Ultra96 board, which makes the generated bitstream unusable. Therefore, we decided to try out the FINN Compiler instead. But even then, the inner workings of FINN are in many ways similar to how hls4ml runs and so the knowledge that we gain here is still useful when we try FINN.

Section 4.3 Hardware Acceleration of Quantized Neural Network with FINN

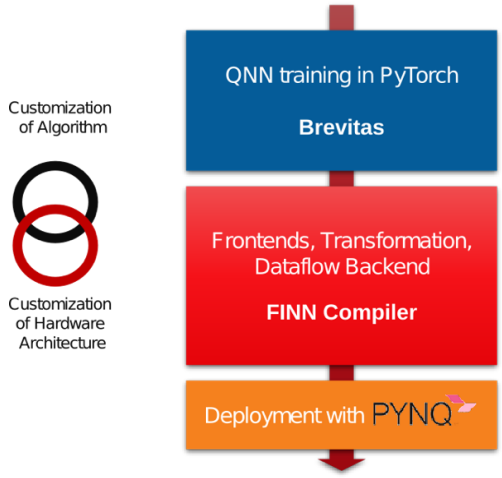


Figure 46: Workflow with Brevitas and FINN Compiler

The Brevitas PyTorch library is used to train a quantized ONNX model, which is an open format to represent machine learning models. This is a plus for us, since we were already using PyTorch when trying out hls4ml, and the transition is easier for us since there is no need to learn a new machine learning library. The FINN Compiler takes this quantized ONNX model, and generates a bitstream that can be hardware accelerated.

One advantage of FINN Compiler over the hls4ml tool is that much of the process of generating a bitstream from an ONNX model is hidden from the user. Users of FINN are only expected to write their scripts in Python with the provided FINN library to do the conversion, and they do not have to deal with technologies that most FPGA designers have to work with(e.g HLS, RTL, Verilog).

But even so, the FINN Compiler is still in its alpha stage of development, and from our experience, there are still several bugs within the software. This requires us to delve deep into the lower level processes, and we have to interact directly with the intermediate files such as

manually altering the Block Design diagram in order to resolve the problem. But still, we expect that this tool is going to become more robust in the future as the FINN Compiler becomes a more reliable and tested software.

Another advantage of utilizing the FINN Compiler is that we are generating a bitstream from a quantized neural network model with all the precisions defined beforehand and the ONNX model is generated according to that precision. In contrast, the hls4ml tool uses a neural network model that is trained on floating points, but still requires the users to manually define the precision of the arbitrary precision datatype when generating the HLS project. This means that users have to pick the precision through trial and error, and the accuracy of the hardware accelerated neural network generated by hls4ml can differ from the original neural network model. However, the main reason why we choose to use the FINN Compiler, is because we are able to generate a bitstream that works reliably without facing the runtime error that we faced when trying out hls4ml.

In our system, we will be using a hardware accelerated neural network for dance action predictions. Movement prediction of dancers with Machine Learning will be done on the CPU instead. The reason for this is that it can take up to 5 seconds to load one bitstream, and only one bitstream can be loaded at any point in time. Managing two bitstreams at once can reduce the performance of the ultra96 significantly as we will be adding another 5 seconds into the inference timing when we switch between bitstreams.

Section 4.3.1 Neural Network Architecture

```
class QNN_HARnn(torch.nn.Module):
  def __init__(self):
    super(QNN_HARnn, self).__init__()
    self.hardtanh0 = qnn.QuantHardTanh(quant_type=QuantType.INT, bit_width=2, narrow_range=True, bit_width_impl_type=BitWidthImplType.CONST,
                                       min_val = -1.0, max_val = 1.0, restrict_scaling_type=RestrictValueType.LOG_FP, scaling_per_channel=False,
                                       scaling_impl_type=ScalingImplType.PARAMETER)
    self.dropout0 = torch.nn.Dropout(p = DROPOUT)
    self.linear1 = qnn.QuantLinear(32, 128, bias=False, weight_quant_type=QuantType.INT, weight_bit_width=4, weight_scaling_stats_op = StatsOp.MAX,
                                   weight_scaling_stats_sigma=0.001, weight_scaling_per_output_channel = True, weight_narrow_range = True,
                                   weight_bit_width_impl_type=BitWidthImplType.CONST)
    self.hardtanh1 = qnn.QuantHardTanh(quant_type=QuantType.INT, bit_width=2, narrow_range=True, bit_width_impl_type=BitWidthImplType.CONST,
                                       min_val = -1.0, max_val = 1.0, restrict_scaling_type=RestrictValueType.LOG_FP, scaling_per_channel=False,
                                       scaling_impl_type=ScalingImplType.PARAMETER)
    self.dropout1 = torch.nn.Dropout(p = DROPOUT)
    self.linear2 = qnn.QuantLinear(128, 64, bias=False, weight_quant_type=QuantType.INT, weight_bit_width=4, weight_scaling_stats_op = StatsOp.MAX,
                                   weight_scaling_stats_sigma=0.001, weight_scaling_per_output_channel = True, weight_narrow_range = True,
                                   weight_bit_width_impl_type=BitWidthImplType.CONST)
    self.hardtanh2 = qnn.QuantHardTanh(quant_type=QuantType.INT, bit_width=2, narrow_range=True, bit_width_impl_type=BitWidthImplType.CONST,
                                       min_val = -1.0, max_val = 1.0, restrict_scaling_type=RestrictValueType.LOG_FP, scaling_per_channel=False,
                                       scaling_impl_type=ScalingImplType.PARAMETER)
    self.dropout2 = torch.nn.Dropout(p = DROPOUT)
    self.linear3 = qnn.QuantLinear(64, 3, bias=False, weight_quant_type=QuantType.INT, weight_bit_width=4, weight_scaling_stats_op = StatsOp.MAX,
                                   weight_scaling_stats_sigma=0.001, weight_scaling_per_output_channel = False, weight_narrow_range = True,
                                   weight_bit_width_impl_type=BitWidthImplType.CONST)


  def forward(self, x):
    x = self.hardtanh0(x)
    x = self.dropout0(x)
    x = self.linear1(x)
    x = self.hardtanh1(x)
    x = self.dropout1(x)
    x = self.linear2(x)
    x = self.hardtanh2(x)
    x = self.dropout2(x)
    x = self.linear3(x)
    return x
```

Figure 47: Brevitas Code snippet describing the Neural Network Architecture on PyTorch

The above code snippet shows a Fully Connected Neural Network with Quantized Hard Tanh function as the activating functions. There are a total of 3 Quantized Linear layers. There is one input layer with 32 nodes, one output layer with 3 nodes (To predict the 3 different dance labels) and one hidden layer with 128 nodes. We decided not to increase the number of layers of the neural network further as not only does it not lead to a noticeable improvement in inference accuracy, it also reduces the time taken to make an inference due to the increase in number of computations performed. Increasing the number of layers also reduces the amount of hardware resources available for loop unrolling. Loop unrolling allows us to parallelize computations, and having lesser hardware resources for loop unrolling decreases the extent of unrolling and increases the time to make an inference.

We will be utilizing a constant precision of 2 bits for Quantized Hard Tanh activation function and a precision of 4 bits for the Linear layers. We choose to define a constant precision as learned precision is still not yet supported within Brevitas. The precisions were chosen by trying out a few combinations, and we found that this combination provides the best result.

One point to note is that the overall neural network architecture is slightly different from the one implemented on Software with the scikit-learn library. This is because we are not able to replicate the same accuracy as the Software variant if we were to utilize the same exact architecture. Another difference with the scikit-learn neural network is that the quantized neural network has 2 additional nodes at the inputs. We will pad the input data with 2 additional 0’s in the hardware accelerated neural network. The reason we made such a change is that we found that it is easier to configure the hardware (e.g FIFO buffers, loop unfolding) when the layers are in powers of 2.

But even with these minor differences, we are still able to maintain a comparable accuracy to the scikit-learn neural network and see improvements in inference timings.

Section 4.3.2 Generating Bitstream with FINN

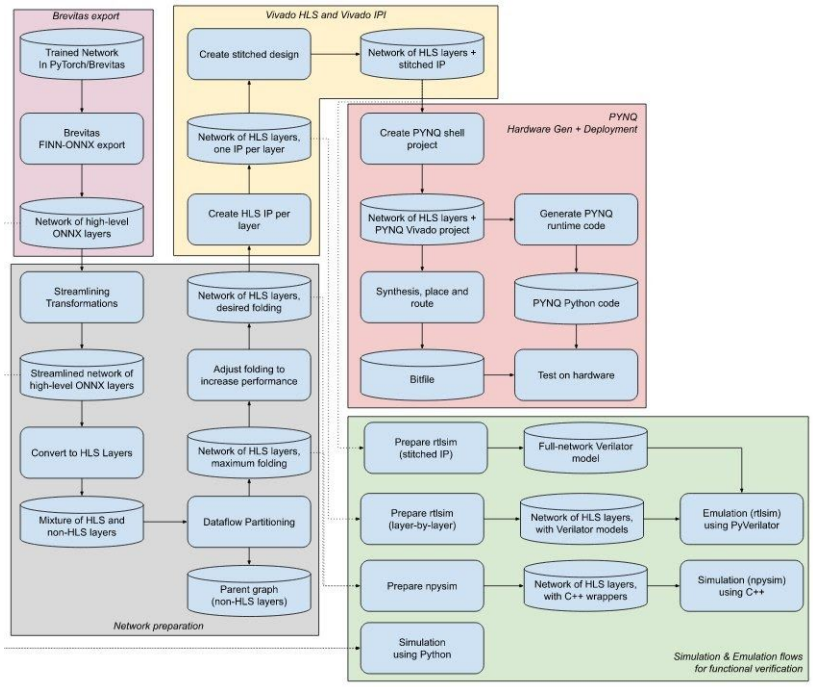


Figure 48: Process to convert a Trained ONNX model in Brevitas to Working Bitstream

An example end-to-end notebook is provided by FINN to demonstrate the whole workflow described above. To generate a bitstream, we customize the end-to-end notebook to work with our ONNX model. The generation of the bitstream comes in several stages. But to summarize, an ONNX model generated by the Brevitas Library undergoes a series of transformations and configurations, and after a series of these changes, several HLS projects describing the ONNX model are generated. One IP block can be generated from each HLS project.

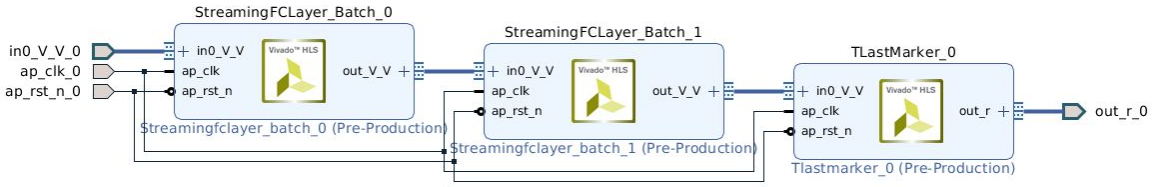


Figure 49: Stitched IP Blocks

These IP blocks are then stitched together to create one overall IP block, and then a Block Design diagram similar to the one we connect ourselves when trying out the hls4ml tool is automatically generated. A bitstream is then generated from this final Block Design. Similar to

how we did it in hls4ml, the IP block also runs with the AXI4-Stream protocol as it generates a dataflow-style architecture for inference.

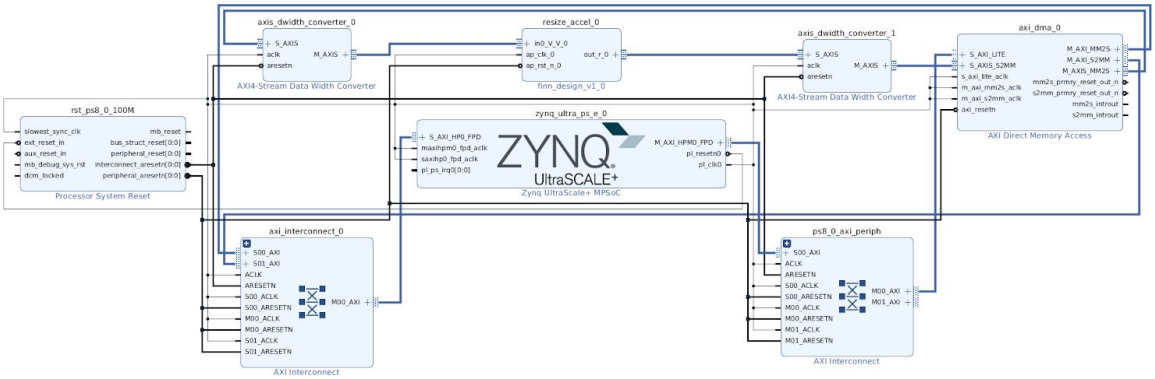


Figure 50: Final Block Design Diagram generated by FINN

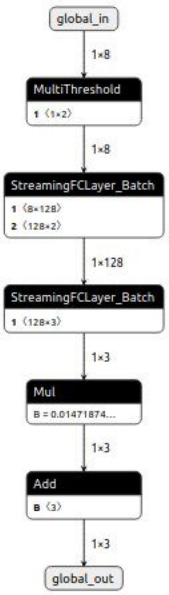


Figure 51: Graph of a processed ONNX model. Only some nodes will be hardware accelerated

However, the FINN compiler only generates a bitstream that hardware accelerates part of the neural network. The parts of the neural network that cannot be loaded onto the FPGA, have to be run on the CPU. This is likely due to the fact that FINN compiler is still in its early stages of development, and the HLS library is still not comprehensive enough to support everything.

As an example, in the image above, only nodes typed as “StreamingFCLayer_Batch” will be hardware accelerated. The first node, typed as “MultiThreshold” is used for the quantization of inputs, which converts a given floating point input to 0, 1 or 2.

$$input\_quntization(x) = \begin{cases} 0, & x < -0.01027827151119709 \\ 1, & -0.01027827151119709 < x < 0.01027827151119709 \\ 2, & x > 0.01027827151119709 \end{cases}$$

Figure 52: Quantization of inputs with 2 threshold values.

The quantization of inputs can be described by the piecewise function above. The threshold values (0.01027827151119708 and -0.01027827151119708) are values determined during the training of the quantized neural network. The quantized input will then be passed into the first hardware accelerated StreamingFCLayer_Batch node, which is the next node in line within the ONNX graph.

The last 2 nodes, typed as "Mul" and "Add" nodes are used for the post processing of output (matrix additions and multiplications). The hardware acceleration of these nodes is also not supported as of now. To perform computation on nodes that couldn't be hardware accelerated, FINN has its own custom library to support some of its own custom operators. For nodes that are defined within the standard ONNX, FINN uses ONNXRuntime to execute these nodes.

Section 4.3.3 Software Setup on the Ultra96

Another issue with utilizing the FINN Compiler, is that their end-to-end flow does not support the running of a neural network fully on the Ultra96. The Ultra96 only runs hardware accelerated nodes within the ONNX model. Nodes that are not hardware accelerated were executed remotely on another PC. Therefore, in order to execute the whole ONNX graph, the remote PC has to set up a ssh connection with the Ultra96 to pass data to and fro. This adds a latency to inference, as the inference of the neural network involves a network connection.

This is not what we wanted because we do not want to pass the collected IMU data from the Ultra96 to a remote PC for preprocessing, only for it to be passed back to the Ultra96 again for hardware acceleration. Therefore, we did a setup of all required libraries on the Ultra96 such that all preprocessing of inputs and postprocessing of outputs can be done on Ultra96.

Without considering the many other dependencies, there are only 2 core libraries that have to be installed on the Ultra96. The first library that we have to install is the FINN library itself. We require it to execute some custom nodes defined by the FINN and not in the normal ONNX standard. The next library that we have to install is the ONNXRuntime library. This is to execute nodes defined within the normal ONNX standard. However, one thing to note when installing the ONNXRuntime library is that it utilizes C extension modules, and since the python wheels of ONNXRuntime for aarch64 architecture (The architecture of the ARM processor that the Ultra96 is running on) is not provided, we have to build the python wheel from source. Building it on the

Ultra96 itself takes around 8 hours, and we recommended having at least 8GB of free space on the memory card when building it from source. Alternatively, we can use QEMU(An emulator that performs hardware virtualization), to emulate the aarch64 architecture to build the python wheel from source.

Section 4.3.4 Python Script to perform Quantized Neural Network Inference on FPGA

The python script for making an inference on the hardware accelerated portion of the Quantized Neural Network generated by FINN is similar to how we did it previously with hls4ml. The inference is done via making a transaction with the AXI DMA, with one input buffer containing the data to be sent and another output buffer waiting for the data to be received.

However, there is also a need to make inference on the non-hardware accelerated portion of the ONNX model. In order to do this, we will have to load and read the ONNX model, to determine which nodes cannot be hardware accelerated, and do the required computation on the CPU.

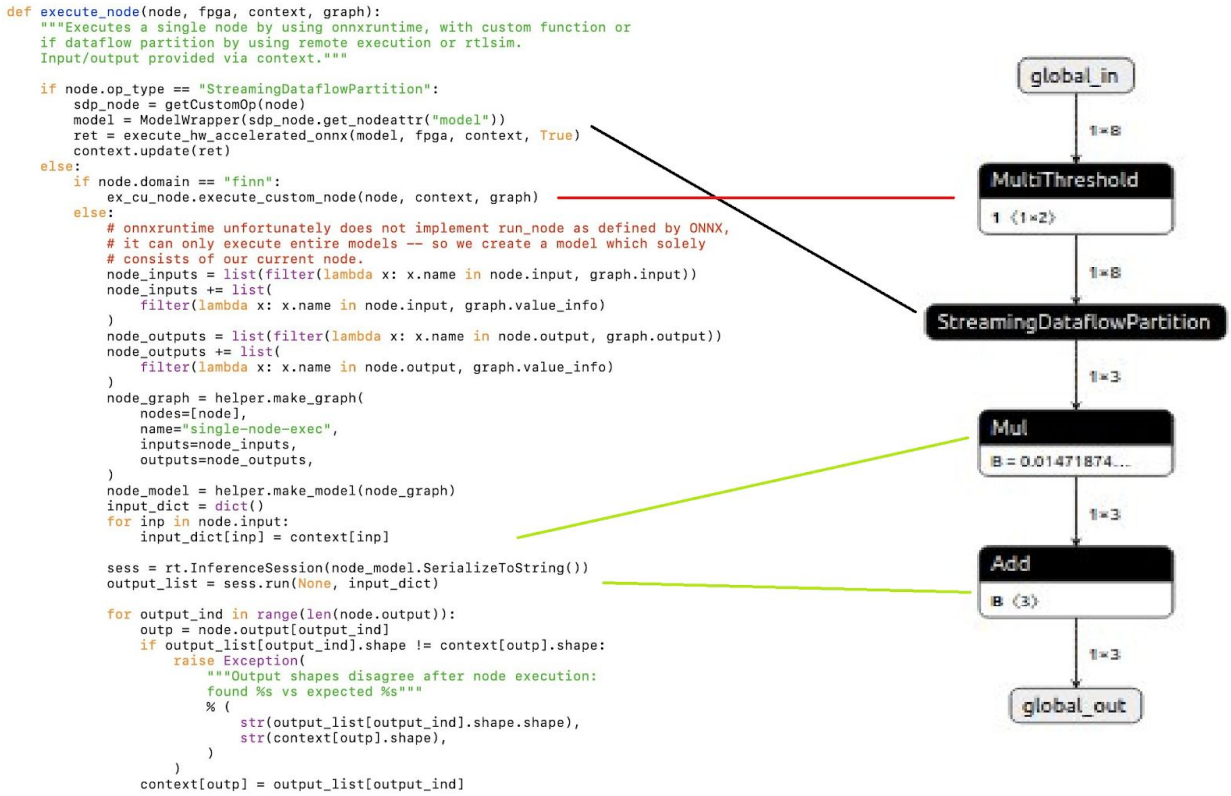


Figure 53: Code snippet of ONNX Graph node execution

Above and on the left, is a code snippet of the execute_node function. On the right is an example of a processed ONNX graph. The lines between the code snippet and the ONNX graph shows how each node within the ONNX graph is executed in code.

The node with op_type “StreamingDataflowPartition” is a partition of all nodes with the type StreamingFCLayer_Batch. Only nodes within the partition will be hardware accelerated. The node with the type “MultiThreshold” is one of FINN’s custom nodes, and custom nodes defined by FINN are executed with the FINN custom_node module. “Mul” and “Add” are nodes that are already defined within the standard ONNX and it is executed with ONNXRuntime.

```
def fpga_single_run(self, input):

    assert input.shape == self.ishape_normal
    ibuf_folded = input.reshape(self.ishape_folded)

    # pack the input buffer, reversing both SIMD dim and endianness
    ibuf_packed = finnpy_to_packed_bytearray(
        ibuf_folded, self.idt, reverse_endian=True, reverse_inner=True
    )
    # copy the packed data into the PYNQ buffer
            np.copyto(self.ibuf_packed_device, ibuf_packed)

    # set up the DMA and wait until all transfers complete
    self.dma.sendchannel.transfer(self.ibuf_packed_device)
    self.dma.recvchannel.transfer(self.obuf_packed)
    self.dma.sendchannel.wait()
    self.dma.recvchannel.wait()

    # unpack the packed output buffer from accelerator
    obuf_folded = packed_bytearray_to_finnpy(
        self.obuf_packed, self.odt, self.oshape_folded, reverse_endian=True, reverse_inner=True
    )

    obuf_normal = obuf_folded.reshape(self.oshape_normal)
    return obuf_normal
```

Figure 54: Code snippet to setup a DMA Transfer for Hardware Acceleration of StreamingDataFlowPartition node

The setup of DMA transfer of data for the hardware acceleration of StreamingDataflowPartition node is similar to how we did it previously with a hardware accelerated neural network generated from the hls4ml tool. One difference however, is in how the data needs to be packed/unpacked before transferring it to the DMA due to the quantization of inputs and outputs.

Section 4.3.5 Software Overhead Optimization

The speed of inference is dependent on how fast we can compute the results of all the nodes within the ONNX Graph, including those that can only be executed on the CPU. From our experiments, we found that it takes the CPU a significant amount of time to process nodes that cannot be hardware accelerated in our Neural Network Architecture. This is due to the extra software overhead from calling the library functions of FINN and ONNXRuntime.

When we are executing a node with ONNXRuntime, we cannot execute that node independently. Instead, we have to construct another full ONNX Graph containing just that specific node. The

process of constructing an ONNX Graph is a computationally intensive process and hence slows down the inference.

Therefore, in order to improve the speed of inference, for each neural network we generated, we tried to hardcode the computation of each node such that we can skip the extra overheads from library calls and the construction of the new ONNX Graphs for ONNXRuntime.

```
self.mt_node_thresholds = np.asarray([[-0.01027827151119709, 0.01027827151119709]])
self.multiply_node_const = 0.014718746766448021
self.add_node_mat = np.asarray([-0.08831246197223663, 0.17662496864795685, -0.058874987065792084])

def fpga_single_run(self, input):

    input = input.reshape(self.ishape_normal)
    input = MT.multithreshold(input, self.mt_node_thresholds)
    assert input.shape == self.ishape_normal
    ibuf_folded = input.reshape(self.ishape_folded)

    # pack the input buffer, reversing both SIMD dim and endianness
    ibuf_packed = finnpy_to_packed_bytearray(
        ibuf_folded, self.idt, reverse_endian=True, reverse_inner=True
    )
    # copy the packed data into the PYNQ buffer
    np.copyto(self.ibuf_packed_device, ibuf_packed)

    # set up the DMA and wait until all transfers complete
    self.dma.sendchannel.transfer(self.ibuf_packed_device)
    self.dma.recvchannel.transfer(self.obuf_packed)
    self.dma.sendchannel.wait()
    self.dma.recvchannel.wait()

    # unpack the packed output buffer from accelerator
    obuf_folded = packed_bytearray_to_finnpy(
        self.obuf_packed, self.odt, self.oshape_folded, reverse_endian=True, reverse_inner=True
    )

    obuf_normal = obuf_folded.reshape(self.oshape_normal)
    obuf_normal = obuf_normal * self.multiply_node_const
    obuf_normal = obuf_normal + self.add_node_mat
    return obuf_normal
```

Figure 55: Optimized code to perform input pre-processing and output post-processing to reduce dependency on the libraries.

The above is an example of how the hardcoding is done. The code highlighted in red are the extra changes we made to the DMA transfer code (Use Fig 54. for comparison), so that input and output processing can be done immediately before and after the DMA transfer. The first line that is highlighted in red involves the usage of another helper function that we wrote ourselves to quantize the inputs to avoid the extra overhead from FINN library. We quantize the input according to the piecewise function in Fig 52. The last 2 lines highlighted in red are for matrix multiplication and matrix addition(post processing of outputs). This allows us to avoid the extra overhead from using the ONNXRuntime library.

With this software overhead optimization, we see an improvement of roughly 2-3x in terms of speed of inference. The amount of improvement that we get is dependent on the ONNX graph, which is dependent on the neural network architecture. The only downside to this is that it is a menial process to read and understand the ONNX graph to hardcode the threshold values and matrices onto the script. The only reason why we are able to use this small little trick somewhat efficiently is because our ONNX graph is small and not a lot of hardcoding needs to be done. For more complicated neural networks, it can be a hassle to utilize hardcoding, and it is up to the developer's own discretion on whether the hassle is worth the extra improvement in speed of inference.

Section 4.4 Hardware Acceleration Evaluation Metrics

	Inference speed
scikit-learn neural network (Accuracy: 77.6%)	0.09536099s
FINN Generated neural network (Accuracy: 77.9%)	0.01762198s
FINN Generated neural network with Software Optimization (Accuracy: 77.9%)	0.00880599s

Figure 56: Inference speed/Accuracy Comparisons

In this section, we will evaluate the results of our hardware accelerated neural network model in comparison to the software variant which utilizes the scikit-learn library. By utilizing hardware acceleration, we manage to achieve an improvement in both accuracy and inference speed. On average, each inference by the scikit-learn neural network takes **0.09536099s**. Each inference by the hardware accelerated neural network, without the software overhead optimization takes **0.01762198s**. With the software overhead optimization, the inference time reduces to **0.00880599s**. This is a 10.8x improvement in inference speed. When evaluating in terms of accuracy, we managed to attain a slight improvement as our hardware accelerated neural network has an accuracy of **77.91%** vs **77.6%** over the scikit-learn neural network.

Section 4.5 Limitations of FINN Compiler and future changes

There are currently several limitations with the FINN Compiler. As discussed previously, FINN Compiler currently does not have support for the quantization of inputs. But there are plans by the developers of FINN to support it in the future as described in the FINN-R paper(Blott et al., 2018).

\The FINN Compiler also provides users with very little flexibility on how they can modify their neural network. Not only does the FINN Compiler support hardware acceleration of Quantized Linear Layers and Quantized HardTanh activation function up to a precision of only 2 bits, they also have to be composed of a series of very specific configurations. Due to this restriction, the potential designs of our Neural Network Architecture for Hardware Acceleration is very limited, and it is why we did not try out other types of Neural Networks during our project. E.g Convolutional Neural Network. But still there are plans for the FINN Compiler to support more types of Neural Networks in the future, with the support of Convolutional Network in the next planned release.

Section 5 Firmware & Communication Details: Communication 1 - Internal

Section 5.1 Introduction

This section talks about the internal communications architecture of the system, and the role it plays in ensuring that the overall system works smoothly and as intended. This section will start off by giving a brief overview of the internal communications component, followed by in-depth explanations of original and the new design after week 7 coupled together with relevant code snippets. Next, there will be an explanation on the protocol to coordinate connections between the Beetles and Ultra96. Finally, the section will end off by talking about the extension of the internal communications component into another relevant application that utilizes the current internal communications functionalities.

Section 5.1.1 Architecture diagram

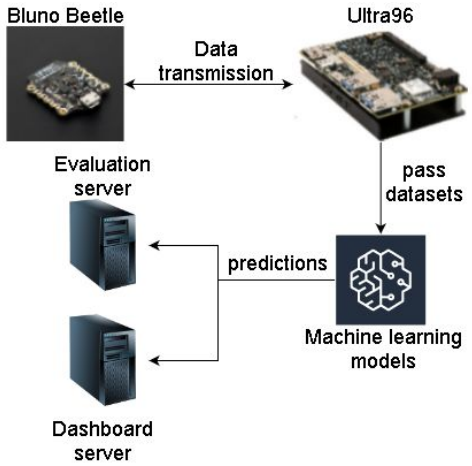


Figure 57: Internal communications architecture

Section 5.2 Brief Overview

Section 5.2.1 Job scope

The main job of the internal communications component was to deliver a smooth and reliable medium for data transmission to occur between the three Beetles and the Ultra96. It has four main branches of roles that can be clearly defined as follows:

- Provide a robust way of transmitting data between the Beetle and Ultra96
- Feed collected datasets into machine learning models for predictions on the side of Ultra96
- Transmit received sensor data to the dashboard server at regular intervals, coupled with machine learning prediction results at the end
- Transmit machine learning prediction results, together with the synchronization delay between dancers, to the evaluation server

Section 5.2.2 Difficulties encountered and alternative solutions

There were difficulties throughout the whole project, related to the physical and software limitations only discovered via countless tests and experiments. Firstly, we originally intended to use 6 Beetles, with each Beetle coming with its own IMU sensor, to transmit sensor data to Ultra96. However, we discovered that there were problems with getting data from 6 Beetles concurrently, without the risk of either one or multiple Beetles getting disconnected midway through the dance run. Even if there were no disconnections, oftentimes, the data would be collected sequentially, which means that there will always be 1 or 2 Beetles whose data will be collected only after the rest of the Beetles. Coupled with the fact that we were facing limited time constraints and there were numerous hardware issues which needed time to be fixed before we could proceed with 6 Beetles, we had to explore alternative solutions. Just like any other engineering projects, we weighed the pros and cons of either proceeding with 6 Beetles as per normal, or drop 3 Beetles and go with just the remaining 3 Beetles for the dance. Finally, we concluded that due to time constraints, we could not risk any hardware or software issues occuring again whilst using 6 Beetles. In order to achieve maximum data collection throughput, we reconfigured the Ultra96's BLE interface settings, which will be further explained in section 5.3.9.

Section 5.3 Design of internal communications

Section 5.3.1 Original design (before Week 7)

The internal communications architecture, up until week 7, only supports the transmission of dummy data from three Beetles to the Ultra96.

```
def establish_connection(address):
    while True:
        try:
            for idx in range(len(beetle_addresses)):
                # for initial connections or when any beetle is disconnected
                if beetle_addresses[idx] == address:
                    if global_beetle[idx] != 0:  # do not reconnect if already connected
                        return
                    else:
                        print("connecting with %s" % (address))
                        beetle = btle.Peripheral(address)
                        global_beetle[idx] = beetle
                        beetle_delegate = Delegate(address)
                        global_delegate_obj[idx] = beetle_delegate
                        beetle.withDelegate(beetle_delegate)
```

Figure 58: Establishing connection with Beetles

The BLE connection with the Beetles is initialized via the establish_connection() function, whereby the device address of the individual Beetles are passed as the Peripheral constructor argument, so that a BLE peripheral object can be instantiated.

```
def initHandshake(beetle_peripheral, address, clocksync_count):
    global timestamp_dict
    global clocksync_flag_dict
    global handshake_flag_dict

    ultra96_sending_timestamp = time.time() * 1000

    for bdAddress, boolFlag in beetles_connection_flag_dict.items():
        if bdAddress == address and boolFlag == False:
            for characteristic in beetle_peripheral.getCharacteristics():
                if characteristic.uuid == UUIDS.SERIAL_COMMS:
                    ultra96_sending_timestamp = time.time() * 1000
                    timestamp_dict[address].append(ultra96_sending_timestamp)
                    characteristic.write(
                        bytes('H', 'utf-8'), withResponse=False)
```

```
if clocksync_flag_dict[address] is True:
    characteristic.write(
        bytes('A', 'utf-8'), withResponse=False)
    print("handshake succeeded with %s" % (
        address))
```

Figure 59: Handshaking with the Beetles

After a BLE connection is successfully established with a Beetle, the handshaking process will now begin via the initHandshake() function. In this function, the Ultra96 will send 'H' to the Beetle, in which the Beetle will then acknowledge by sending 'A' to Ultra96. Upon receiving 'A', Ultra96 will then reciprocate by sending 'A' back to the Beetle to signify that it is ready to receive data from the Beetle.

```
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as data_executor:
    receive_data_futures = {data_executor.submit(
        getBeetleData, beetle): beetle for beetle in global_beetle_periphs}
```

Figure 60: Initializing worker threads to call getBeetleData() for data collection

After doing handshaking with the Beetles, the Ultra96 will then initialize worker threads to call getBeetleData() concurrently. This will allow for the data collection from the Beetles to happen concurrently.

```
def getBeetleData(beetle_peri):
    global packet_count_dict
    global dataset_count_dict
    while True:
        try:
            if beetle_peri.waitForNotifications(20):
                print("getting data...")
                # if number of datasets received from all beetles exceed expectation
                if packet_count_dict[beetle_peri.addr] == 100.0:
                    print("sufficient datasets received from %s. Processing data now" % (
                        beetle_peri.addr))
                    # reset for next dance move
                    packet_count_dict[beetle_peri.addr] = 0.0
                    # reset for next dance move
                    dataset_count_dict[beetle_peri.addr] = 0
                    return
                continue
        except Exception as e:
            print("disconnecting beetle_peri: %s" % (beetle_peri.addr))
            beetles_connection_flag_dict.update(
                {beetle_peri.addr: False})
            reestablish_connection(beetle_peri, beetle_peri.addr)
```

Figure 61: collecting datasets from Beetle

The function that is responsible for collecting datasets from the Beetle is the getBeetleData() function. In this function, the waitForNotifications() function will be called, and will only return

true if there is any incoming data from the Beetle. There is a set timeout value input as a parameter into the function, whereby the function will automatically return false after the timeout has passed.

```
def reestablish_connection(beetle_peri, address):
    while True:
        try:
            print("reconnecting to %s" % (address))
            beetle_peri.connect(address)
            print("re-connected to %s" % (address))
            beetles_connection_flag_dict.update(
                {beetle_peri.addr: True})
            getBeetleData(beetle_peri)
        except:
            print("error reconnecting to %s" % (address))
            time.sleep(1)
            continue
```

Figure 62: Protocol to re-establish connection to disconnected Beetle

If there are any instances whereby the Beetle is disconnected from the Ultra96, a function that will do the reconnection is the reestablish_connection(). Upon successful reconnection, the getBeetleData() function will be called to resume the collection of datasets from the Beetle.

```
strcat(transmit_buffer, ",");
Serial.print(',');
dtostrf(12.23, 5, 2, yaw);
strcat(transmit_buffer, yaw);
strcat(transmit_buffer, ",");
Serial.print(yaw);
Serial.print(',');
dtostrf(15.34, 5, 2, pitch);
strcat(transmit_buffer, pitch);
strcat(transmit_buffer, ",");
Serial.print(pitch);
Serial.print(',');
dtostrf(17.26, 5, 2, roll);
strcat(transmit_buffer, roll);
Serial.print(roll);
```

Figure 63: Transmitting dummy data from Beetle

There were no robustness enhancements implemented in the component due to the fact that dummy data was getting transmitted continuously from the Beetles without stopping. There were also no functionalities to incorporate our machine learning, external communications component and dashboard components together with this component.

The kind of packet types and codes used in this original design was limited, as there was not a need for new packet types related to robustness measures. The only packet codes used in this old design, were the ‘H’ packet code, which is used in sending a hello packet from the Ultra96 to the Beetle for the purpose of handshaking, the ‘A’ packet code, which signifies the acknowledgment in the handshaking process from both Ultra96 and the Beetle, and the ‘D’ packet code, which refers to packets that contains datasets received from the Beetle.

```
def deserialize(buffer_dict, result_dict, address):
    for char in buffer_dict[address]:
        # start of new dataset
        if char == 'D' or end_flag[address] is True:
            # 2nd part of dataset lost or '>' lost in transmission
            if start_flag[address] is True:
                try:
                    # if only '>' lost in transmission, can keep dataset. Else delete
                    if checksum_dict[address] != int(datastring_dict[address]):
                        del result_dict[address][dataset_count_dict[address]]
                except Exception:  # 2nd part of dataset lost
                    try:
                        del result_dict[address][dataset_count_dict[address]]
                    except Exception:
                        pass
                # reset datastring to prepare for next dataset
                datastring_dict[address] = ""
                # reset checksum to prepare for next dataset
                checksum_dict[address] = 0
                comma_count_dict[address] = 0
```

Figure 64: Function to deserialize data packets received from Beetle

The deserialization of received data packets into a correct format packed inside appropriate data structures, is done via a function called deserialize(), which takes in buffer_dict, a dictionary that contains a buffer string that holds the datasets received from the Beetles, and returns result_dict, a dictionary that contains properly formatted datasets that can be properly fed into our machine learning algorithms.

```
pool = multiprocessing.Pool()
workers = [pool.apply_async(processData, args=(address, ))
           for address in beetle_addresses]
result = [worker.get() for worker in workers]
pool.close()
```

Figure 65: Using multiprocessing framework to speed up deserializing operation

Due to the fact that Ultra96 has four processing cores that we can utilize for operations, we have decided to use Python’s multiprocessing library to speed up the rate of cpu operations. As there is a very large number of datasets that needs to be deserialized and stored in new dictionaries in a

properly formatted manner, this operation is considered as a cpu intensive task, which makes multiprocessing an ideal way of reducing latency of the overall system.

The goal of this original design is to ensure that the Ultra96 is able to receive data from three Beetles concurrently for at least a minute, without any bluetooth disconnections. As such, the features implemented at this stage were only designed to meet the requirements of the subcomponent test, instead of being optimized for the eventual integration of other components in the overall system and also to enable the smooth and reliable operation of the system.

Section 5.3.2 New design (after Week 7)

There were numerous modifications being made to the internal communications architecture after week 7's subcomponent test, to ensure smooth integration of the various subcomponents of the entire system with internal communications. Described below are the key points regarding the changes, and we will be explaining each key point in detail.

- Ensure stability of the Beetles by removing FreeRTOS
- Beetles now transmit real data from IMU sensors instead of dummy data
- Beetles only transmit and receive data at appropriate times
- New packet codes for existing and new packet types on both Beetle and Ultra96's side
- Greater robustness in internal communications system after integration
- Integrating with machine learning, external communications and dashboard components
- Collecting EMG data from the Beetle embedded with EMG sensor

Section 5.3.3 Ensuring stability of Beetles

The ATmega328P chip on the Beetle has only a maximum size of 2 Kbytes of SRAM and 32 Kbytes of flash memory (Arduino.cc). We have decided not to use FreeRTOS for any scheduling capabilities.

```
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include <Arduino_FreeRTOS.h>
```

```
Sketch uses 10348 bytes (32%) of program storage space. Maximum is 32256 bytes.
Global variables use 1760 bytes (85%) of dynamic memory, leaving 288 bytes for local variables. Maximum is 2048 bytes.
Low memory available, stability problems may occur.
```

Figure 66: Using FreeRTOS, even without most of the text section of code

As seen in figure 66 above, when we used FreeRTOS library, even with no blocks of code inside Arduino's setup() and loop() functions, 85% of the Uno's dynamic memory (SRAM) was already used up, which can lead to stability problems in the Beetle. If SRAM space runs out, the

sketch program may fail in unexpected ways; it will appear to upload successfully, but not run, or run strangely.

```
text section exceeds available space in board
Sketch uses 118680 bytes (367%) of program storage space. Maximum is 32256 bytes.
Global variables use 2243 bytes (109%) of dynamic memory, leaving -195 bytes for local variables. Maximum is 2048 bytes.
Sketch too big; see http://www.arduino.cc/en/Guide/Troubleshooting#size for tips on reducing it.
Error compiling for board Arduino Uno.
```

Figure 67: Using FreeRTOS, including text section of codes needed for transmission of data

As we can see from figure 67 above, when we try to include chunks of code which is needed for the Beetle to transmit sensor data to Ultra96 properly, we get the error that the sketch is too big (exceeding by more than twice the maximum size of flash memory which is 32 Kbytes!). We also exceeded the SRAM memory usage by 195 bytes, which means that using FreeRTOS is physically not possible due to our sketch design in the grand scheme of things. Also, as we were planning to send the sensor data to Ultra96 as soon as a single dataset is collected from the IMU sensor, there were little reasons to rely on FreeRTOS as we do not need any form of scheduling in our Beetles. Even if we forced the usage of FreeRTOS by reducing our text section of code and also reworked the design of the Arduino sketch to meet the memory requirements, we decided that the risk of going down that route far outweighs the benefits of scheduling functionality that FreeRTOS provides. The extremely low SRAM space available would mean that the Beetle would either stop working or transmit gibberish data to Ultra96, impacting our entire system.

Section 5.3.4 Transmission of real sensor data

A major change which took place on Beetle's side was that it now procures real sensor data from the IMU sensor attached to it, and then transmits these sensor data to the Ultra96, instead of dummy data meant to be used for just testing the ability of the Beetle to transmit any kind of data to Ultra96 and for Ultra96 to receive any kind of data.

```
dtostrf(12.23, 5, 2, yaw);
strcat(transmit_buffer, yaw);
strcat(transmit_buffer, ",");
Serial.print(yaw);
Serial.print(',');
dtostrf(15.34, 5, 2, pitch);
strcat(transmit_buffer, pitch);
strcat(transmit_buffer, ",");
Serial.print(pitch);
```

```
itoa(int(round(ypr[0] * 100 * 180 / M_PI)), yaw, 10);
strcat(transmit_buffer, yaw);
strcat(transmit_buffer, ",");
Serial.print(yaw);
Serial.print(',');
itoa(int(round(ypr[1] * 100 * 180 / M_PI)), pitch, 10);
strcat(transmit_buffer, pitch);
strcat(transmit_buffer, ",");
Serial.print(pitch);
```

Figure 68: dummy data

Figure 69: real sensor data

Figure 68 for the left picture shows the kind of dummy data being transmitted to Ultra96, while figure 69 for the right picture shows the ypr float array, which contains actual sensor data, being referenced, manipulated, and stored into yaw char array, to be transmitted to Ultra96. This is the core step to ensure that actual important data can be received by Ultra96, to be used for the purposes of training our machine learning models, and also to transmit the data to our dashboard server for real-time analytics.

Section 5.3.5 Transmit data only when needed

We have also implemented transmission control mechanisms to ensure that the Beetles only transmit sensor data to Ultra96 only when necessary, which consists of the time window after Ultra96 has already transmitted the machine learning predictions to the evaluation server and is just waiting for the next set of datasets for the next dance to come in from the Beetles.
There are two reasons for making the Beetles transmit data only when necessary, the first of which, is lower consumption of power by the Beetles. With all battery powered devices, power consumption and battery usage optimization is a main concern. At a high level, power consumption is mainly decreased in microcontroller applications, and embedded systems in general, by reducing radio communication and increasing sleep/idle cycles as much as possible. To reduce radio communication as much as possible, we have opted for the Beetles to only transmit when necessary.

```
    if (is_repeat_dance) {
      continue;
    } else {
      break;
    }
  }
  receiveHandshakeAndClockSync();
}
```

```
while (1) {
  if (Serial.available() && Serial.read() == 'T') { // need to do time calibration
    while (1) {
      if (Serial.available() && Serial.read() == 'H') {

    } else if (Serial.available() && Serial.read() == 'A') { // no need to do time calibration
      break;
    }
}
```

Figure 70: Function blocking transmission

Figure 71: blocking while waiting for U96 response

As we can see from figure 70 of the left image, there is a function called receiveHandShakeAndClockSync() which blocks code execution from going back to the top of the loop() function, where our sensor data collection process happens. Inside the blocking function, the Beetle waits for response packets containing either ‘T’ or ‘A’ from Ultra96 indefinitely.

```
def getDanceData(beetle):
    if beetle.addr != "50:F1:4A:CC:01:C4":
        timeout_count = 0
        retries = 0
        incoming_data_flag[beetle.addr] = True
        for characteristic in beetle.getCharacteristics():
            if characteristic.uuid == UUIDS.SERIAL_COMMS:
                while True:
                    if retries >= 10:
                        retries = 0
                        break
                    print(
                        "sending 'A' to beetle %s to collect dancing data", (beetle.addr))
                    characteristic.write(
                        bytes('A', 'UTF-8'), withResponse=False)
                    retries += 1
```

Figure 72: Sending response packet to Beetle to signal data collection

From figure 72 above, we can see that response packets will only get sent to the Beetle whenever getDanceData() is called. In turn, getDanceData() will only be called after dance predictions are sent to the evaluation server. The time window between Ultra96 having collected sufficient datasets, and sending predictions to the server, will be the period in which the Beetle goes into an idle state. This effectively prevents the Beetle from transmitting collected sensor data all the time, wasting power.

The second reason, which is more important, is related to the received input buffer of Bluepy's handleNotification() function. The function gets called automatically whenever there is incoming data from any of the Beetles.

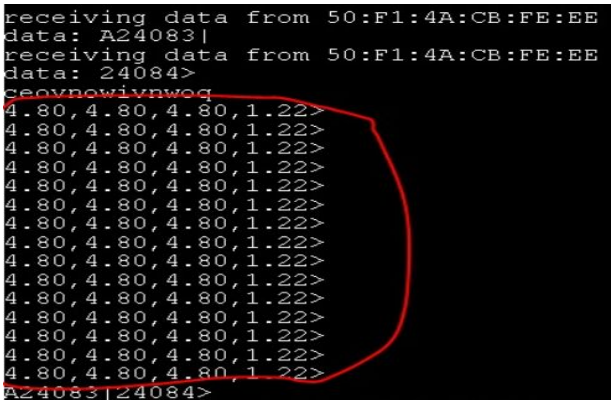


Figure 73: accumulating data in input buffer

As we can see in figure 73 above, if Ultra96 is not expecting any further data from the Beetles due to having received sufficient datasets, it will not call the Beetle's waitForNotifications() function, which basically unblocks Ultra96's input/output task and handles the data stored in the input buffer. The result is accumulating data from the previous dance set, in the buffer due to Beetle's continuous transmission. However, this is undesirable as it causes data inaccuracies or corruption after processing.

Section 5.3.6 New packet types and codes

The table below shows the list of packet types used in the final internal communications architecture and their respective codes

Data packets

Packet Type	Packet Code
Timestamp	T
Sensor data	D

Packets with ‘T’ code are packets that contain the Beetle’s Arduino timestamp data. Packets with ‘D’ code are packets that contain the Beetle’s sensor data.

Response packets

Packet Type	Packet Code
Do time calibration	T
Acknowledge timestamp data received / Skip time calibration and start sensor data transmission	A
Re-transmit timestamp data	R
Re-transmit sensor data	B
Acknowledge sufficient sensor data received	Z

Packets with ‘T’ code are packets that contain the Beetle’s Arduino timestamp data. Packets with ‘A’ code are packets that serve two purposes, with the first purpose being to acknowledge the fact that Ultra96 has received the timestamp from the Beetle, and the second purpose being to tell the Beetle to skip the process of timestamp data collection and start the process of sensor data transmission. Packets with ‘R’ code are packets that tell the Beetle to retransmit timestamp data as the previous timestamp data was lost in transmission, or the Beetle did not transmit timestamp data in the first place. Packets with ‘B’ code are packets that tell the Beetle to retransmit sensor data as the Ultra96 has not received sufficient sensor datasets. Packets with ‘Z’

code are packets that tell the Beetle to stop transmitting sensor data to the Ultra96 and go into blocking mode, until the next set of dance moves arrive.

Section 5.3.7 Greater robustness in internal communications architecture

To ensure greater robustness in the internal communications subcomponent, we have implemented timeout-based re-transmission mechanism, exception handling at every possible important processes, and also added an extra configuration on the BLE interface, on top of the default configurations needed to setup BLE communications.

Re-transmission mechanism

```
while True:
    if retries >= 5:
        retries = 0
        break
    print(
        "Failed to receive timestamp, sending 'Z', 'T', 'H', and 'R' packet to %s" % (beetle.addr))
    characteristic.write(
        bytes('R', 'UTF-8'), withResponse=False)
    characteristic.write(
        bytes('T', 'UTF-8'), withResponse=False)
    characteristic.write(
        bytes('H', 'UTF-8'), withResponse=False)
    characteristic.write(
        bytes('Z', 'UTF-8'), withResponse=False)
    retries += 1
```

Figure 74: Sending response packet to Beetle for timestamp retransmission

```
while True:
    if retries >= 10:
        retries = 0
        break
    print(
        "Failed to receive data, resending 'A' and 'B' packet to %s" % (beetle.addr))
    characteristic.write(
        bytes('A', 'UTF-8'), withResponse=False)
    characteristic.write(
        bytes('B', 'UTF-8'), withResponse=False)
    retries += 1
```

Figure 75: Sending response packet to Beetle for sensor data retransmission

In order to ensure sufficient datasets are received from the Beetles, Ultra96 will attempt to send response packets to the Beetles, signalling the Beetles to retransmit sensor data until Ultra96 has received them. As mentioned earlier, the Beetle will only transmit data for a fixed amount of time, and will stop transmitting after a certain duration has passed to conserve power. If there is no retransmission check on Ultra96's side, the Ultra96 would have kept waiting forever without ever receiving any data.

Handling exceptions at important junctures

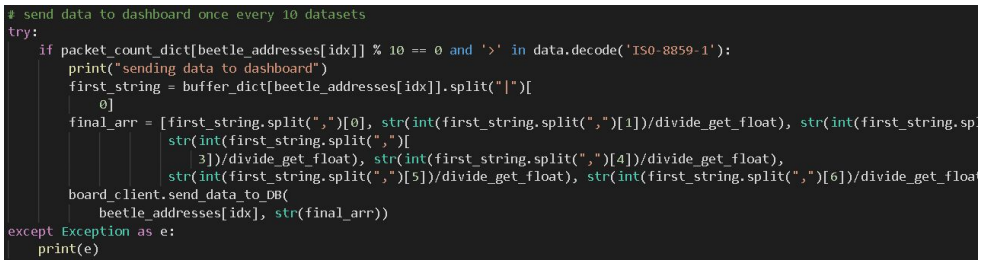


Figure 76: Handling exceptions when sending data to dashboard server

Through endless experimentations and runs, we have realized that handling exceptions at every step of the way, especially at important execution points of the entire code, is essential in preserving the smooth execution of the whole system. The very first important juncture at which exception handling must be implemented, is in the handleNotification() function, whereby sensor data is regularly sent to the dashboard server at fixed intervals via send_data_to_DB(). Sometimes, the received data packet may become corrupted and contain junk values, causing data type exceptions when performing typecasting operations. The exception in turn, causes the Beetle which was responsible for transmitting the corrupted data, to disconnect. This throws the entire system into disarray as much time has to be delayed trying to perform reconnections to the disconnected Beetle, and the reconnection attempts may not even be successful thereafter.

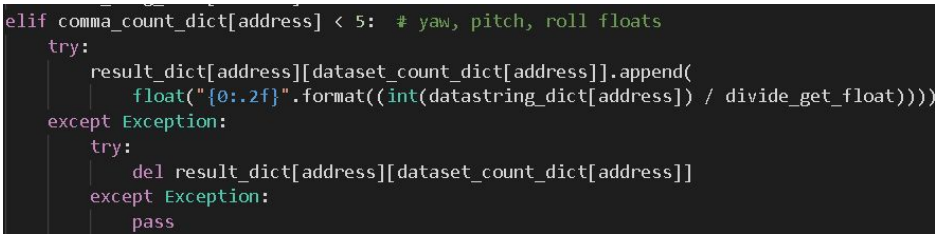


Figure 77: Handling exceptions when deserializing data packets

Another important juncture at which the program can crash happens during the deserialization of data packets inside processData() function. As such, we also added exception handling clauses to account for the fact that sometimes corrupted data can be subjected to typecasting operations, causing exceptions to occur in the program.

Default configuration for BLE interface

Setting up and configuring the BLE interfaces involves four commands. The first command, “echo BT_POWER_UP > /dev/wilc_bt”, powers up the ATWILC300 bluetooth chip, and indicates that the BT requires the chip to be “ON”. The second command, “echo BT_DOWNLOAD_FW > /dev/wilc_bt”, downloads the BT firmware to the ATWILC device’s

Intelligent Random Access Memory (IRAM) using Secure Digital Input Output (SDIO). The third command, “echo BT_FW_CHIP_WAKEUP > /dev/wilc_bt”, acts to prevent the chip from sleeping. The last command, “hciattach /dev/ttyPS1 -t 10 any 115200 noflow nosleep”, is to attach serial UART to bluetooth stack as HCI transport interface. The first argument, “/dev/ttyPS1”, specifies the serial device to attach which is PS1. The second, “-t 10”, specifies an initialization timeout of 10 seconds. The fourth, “115200”, is the baud rate which specifies the UART speed to use. “noflow” means no hardware flow control is forced on the serial link (Maxim Krasnyansky).

Add-on configuration for BLE interface

First, we access the hci0 bluetooth interface of Ultra96 via the command: “cd /sys/kernel/debug/bluetooth/hci0”. Next, we set the minimum and maximum connection interval of the Ultra96’s bluetooth interface to both 10 via the commands, “echo 10 > conn_min_interval” and “echo 10 > conn_max_interval”.
The rationale for choosing the value of 10 for both minimum and maximum connection interval lies in the fact that the lower the maximum and minimum connection interval, the higher the data throughput from transmission of data from the Beetle to Ultra96. Using a higher data throughput ensures that the Beetle Arduino spends less time transmitting sufficient datasets to the Ultra96, therefore prolonging the capacity and lifetime of the battery (Mohammad Afaneh, 2017). Setting the intervals to 10 allowed Ultra96 to receive data from all three beetles concurrently, instead of sequentially that tends to happen if we just used the default value of 20 and 40 for minimum and maximum connection intervals respectively. It would be highly undesirable for sequential data collection to occur, especially when we want to collect data for when the dancers are walking to the new positions. Unlike dancing, walking has a limited time window in which it happens, which means a limited number of datasets being collected. When sequential collection happens, there will usually be the last Beetle whose data has not been collected. However, by the time data is being collected from the final Beetle, the dancers may have already stopped moving, resulting in big inaccuracies in relative position prediction.

Section 5.3.8 Machine learning, evaluation and dashboard server integration

One of the core changes made to the internal communications architecture is the integration of the external communications and dashboard components to enable the system to become complete.

```
try:
    eval_client = eval_client.Client("192.168.43.6", 8080, 6, "cg40024002group6")
except Exception as e:
    print(e)
```

```
try:
    board_client = dashBoardClient.Client("192.168.43.248", 8080, 6, "cg40024002group6")
except Exception as e:
    print(e)

# send data to eval and dashboard server
eval_client.send_data(new_pos, dance, str(sync_delay))
ground_truth = eval_client.receive_dancer_position().split(' ')
ground_truth = [int(ground_truth[0]), int(
    ground_truth[1]), int(ground_truth[2])]
final_string = dance + " " + new_pos
board_client.send_data_to_DB("MLDancer1", final_string)
```

Figure 78: Instantiating evaluation and dashboard client instances

Before the Ultra96 establishes a connection between the Beetles, it instantiates the object instances for evaluation and dashboard servers. The eval_client instance uses the method, send_data() to transmit machine learning predictions, together with the dancers' synchronization delay, to the evaluation server. The board_client instance uses send_data_to_DB() to transmit machine learning predictions to the dashboard server for real-time analytics. As for the machine learning component, two different kinds of dictionaries, one that stores walking datasets, and the other that stores dancing datasets, will be fed into the predict_beetle() function. The function then returns the predicted dance move and relative positions. Section 6.5.4 of the external communications component will explain about this in greater detail.

4. **Sensor data to dashboard server**

Packet format (Sensor data for dashboard)
Sensor data (str) #timestamp|yaw|pitch|roll|acc_x|acc_y|acc_z|
Total size:multiple of 28 bytes

Section 5.3.9 Collecting EMG data

```
Serial.print(maxAmplitude);
Serial.print(',');
Serial.print(meanAmplitude);
Serial.print(',');
Serial.print(rmsAmplitude);
Serial.print(',');
Serial.print(meanFrequency);
Serial.print('>');
delay(1000);
```

Figure 79: Transmitting EMG data

In order to fulfill the additional requirement of displaying muscle fatigue analysis on the dashboard, we transmit four kinds of data on the Beetle's side, maxAmplitude, meanAmplitude, rmsAmplitude and meanFrequency to Ultra96. The transmission interval has been set at 1 second, in line with the requirements of the dashboard graphical interface to display muscle fatigue changes once every second.

```
if beetle_addresses[idx] == "50:F1:4A:CC:01:C4":  # emg beetle data
    emg_buffer[beetle_addresses[idx]
               ] += data.decode('ISO-8859-1')
    if '>' in data.decode('ISO-8859-1'):
        print("sending emg dataset to dashboard")
        packet_count_dict[beetle_addresses[idx]] += 1

        try:
            arr = emg_buffer[beetle_addresses[idx]].split(">")[
                0]
            final_arr = arr.split(",")
            board_client.send_data_to_DB(
                beetle_addresses[idx], str(final_arr))
            emg_buffer[beetle_addresses[idx]] = ""
        except Exception as e:
            print(e)
            board_client.send_data_to_DB(
                beetle_addresses[idx], str(["1", "1", "1", "1"]))
            emg_buffer[beetle_addresses[idx]] = ""
```

Figure 80: Sending collected EMG data to dashboard server

In the handleNotification() function of the Bluepy module, once a full EMG dataset has been received by the Ultra96, denoted by the '>' end byte, the full dataset will be sent to the dashboard server via the send_data_to_DB() method of the board_client dashboard client instance.

Section 5.4 Protocol to coordinate Beetles and Ultra96

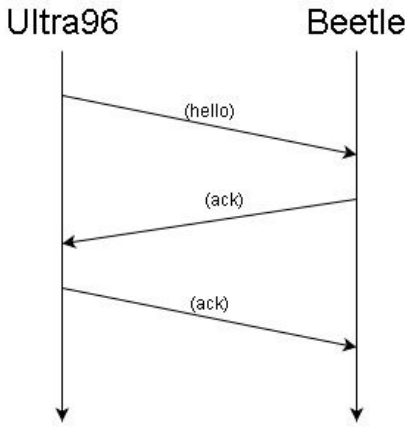


Figure 81: 3-way handshake

The Ultra96 will initialize the 3-way handshake protocol by first sending a “HELLO” packet to the individual Beetle. Ultra96 will then wait to receive “ACK” packets from the Beetles as a form of acknowledgement. Whenever Ultra96 receives “ACK” from any Beetles, it will respond by sending back a “ACK” packet back to the same Beetle. At this point in time, it is not sufficient for Ultra96 to be able to establish a connection with just one Beetle and not all Beetles utilized in our system. The subsequent processes in Ultra96 downstream will be blocked indefinitely until the condition of establishing stable connections with all Beetles involved is met.

Packet format (Sensor data)
Type, timestamp, yaw, pitch, roll, acc_x, acc_y, acc_z, checksum
Total size:27 bytes

```
itoa(int(round(ypr[0] * 100 * 180 / M_PI)), yaw, 10);
strcat(transmit_buffer, yaw);
strcat(transmit_buffer, ",");
Serial.print(yaw);
Serial.print(',');
itoa(int(round(ypr[1] * 100 * 180 / M_PI)), pitch, 10);
strcat(transmit_buffer, pitch);
strcat(transmit_buffer, ",");
Serial.print(pitch);
Serial.print(',');
itoa(int(round(ypr[2] * 100 * 180 / M_PI)), roll, 10);
strcat(transmit_buffer, roll);
strcat(transmit_buffer, ",");
Serial.print(roll);
Serial.print(',');
```

Figure 82: Reduce packet size by converting float to int

Originally, yaw, pitch and roll data from the IMU sensor are supposed to be floats. However, since we needed only a fixed precision of 2 decimal points for the floats, we have decided to multiply every yaw, pitch and roll floats by 100, typecasted them to be integer data types, and then transmit these data as integers. The received yaw, pitch and roll data on the Ultra96's side will then be divided by 100 to obtain the original magnitude of data value collected from the sensors. This allowed us to reduce the overall size of the BLE data having to be transmitted throughout the entire duration of the dance, thus leading to lower rates of data corruption and lost data packets out of the entire datasets received from the Beetle.
The baud rate specifies how fast data is sent over a serial line. Both the Beetle and Ultra96 needs to have the same baud rate configured so that both send and receive data at the same speed (Jim Blom). A baud rate of 115200 bps has been chosen as the standard for our serial communication between the Beetle and Ultra96. It has a fast transmission speed ensuring low latency of communication and yet, not too high that it causes invalid data packets to be received on the receiving end.

Section 5.5 Improvements to existing design

An improvement can be made to facilitate smoother BLE connection setup between the Ultra96 and the Beetles. The issue which we faced when we are trying to conduct a test run, is that more often than not, the Ultra96 will fail to establish a connection with one of the three Beetles. This phenomenon usually happens with the second or last Beetle waiting to establish a connection with Ultra96, and if we configure the minimum and maximum connection interval of the Ultra96's bluetooth interface to both 10, as mentioned in section 5.3.8 under add-on configuration for Ultra96 BLE interface. This problem does not occur if we increase the maximum connection interval to 20 and leave the minimum parameter to 10. Even after conducting various experiments in which we set a timeout delay in between Beetle connections, from the small delay of 1 second to big delay of 5 seconds, we still encountered the same issue repeatedly. It is only after a stroke of luck and numerous repeated tries, that we were able to get Ultra96 to successfully perform a stable connection setup with all three Beetles. This suggests that the internal bluetooth driver of the Ultra96 machine may be faulty, or unable to handle too many BLE connections.

Figure 83: External bluetooth adaptor (adapted from https://images-na.ssl-images-amazon.com/images/I/61Vcrfad%2BQL._AC_SX569_.jpg)

As such, we think that an add-on hardware enhancement in the form of an external bluetooth adaptor may alleviate this problem of BLE connection errors. One advantage of using an external BLE dongle, like the one shown above, is that we do not need to perform configuration and setup for Ultra96's BLE interface as mentioned in section 5.3.8 under default configuration for BLE interface as the firmware is already pre-installed in the external bluetooth adaptor. Thus, we simply need to plug the adaptor into one of the Ultra96's USB type C ports, and we are good to go. With this hardware solution, we feel that we would be able to enable higher data transmission throughput between Ultra96 and the Beetles by setting the lowest possible values of bluetooth minimum and maximum connection interval, without having to worry about the issues we faced as mentioned earlier.

Section 5.6 Extension of internal communications

Section 5.6.1 Rain-Alert-O-Meter

For an extension of my internal communications functionality, I propose a change in the project specifications, in terms of the problem statement. This extension takes on a new problem statement, which is to conduct real time monitoring of the wind speed, temperature and humidity of the external environment. Certain threshold values will be set for each of the collected parameters, and if these thresholds are crossed, an alert will be generated and the alarm buzzer will go off, to notify the user of the impending arrival of rain.

Section 5.6.2 Change in project specifications

The problem statement will be partially morphed, with the real-time monitoring of sensor data still intact. The change lies in the application of the collected data. There are numerous changes in the physical components used in this modified design. There will still be 1 set of Ultra96-V2 machine, with the addition of 1 Arduino UNO board, 1 Ultra96 USB-to-JTAG/UART Pod for serial communications via USB between the Ultra96, 1 DFRobot DHT11 Temperature and

Humidity Sensor, 1 DFRobot wind speed sensor, and 1 DFRobot buzzer module. There will not be any need for the Beetles, unlike the original design for tackling the original problem.

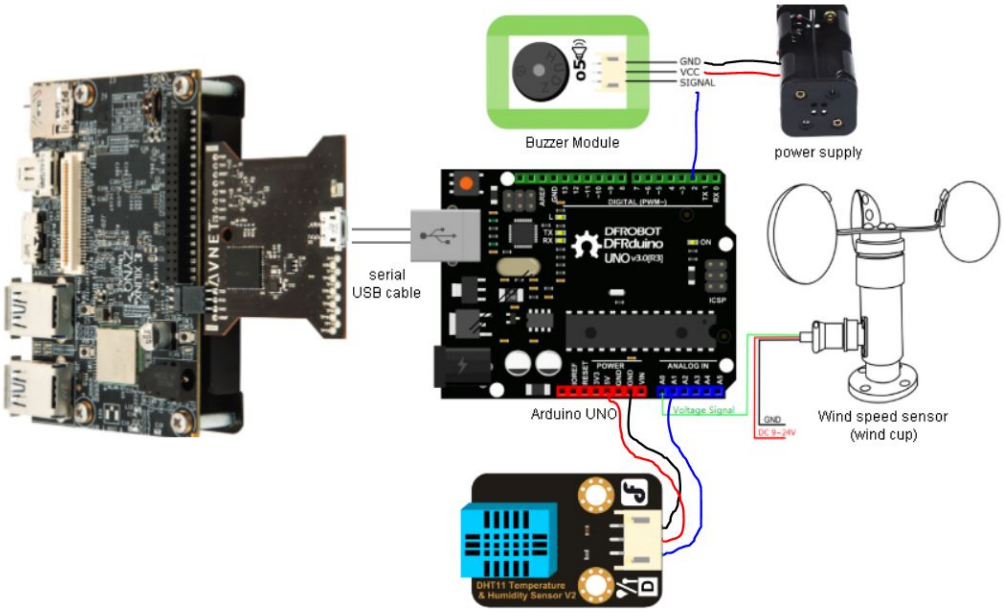


Figure 84: Physical setup of system (Adapted from https://wiki.dfrobot.com/DHT11_Temperature_and_Humidity_Sensor__SKU__DFR0067_, https://wiki.dfrobot.com/Wind_Speed_Sensor_Voltage_Type_0-5V__SKU_SEN0170, https://wiki.dfrobot.com/Buzzer%20Module%20%20(o5)%20SKU:%20BOS0020)

The picture above depicts how the new system would probably look like in design. Another change from the original project specifications, is that instead of using BLE connections as a mode of data transmission between the Arduino UNO and Ultra96, we will be using serial USB connection as the new mode of transmission. However, the essence of the original design, which focuses on the real-time aspects of data transmission and processing, remains unchanged. I feel that an important application such as monitoring for weather changes requires real-time data processing capabilities, and that is where my component's functionalities from my current design comes into play.

Section 5.6.3 Utilizing machine learning to predict weather patterns

To stretch the internal communications architecture further, we can even make use of the power of machine learning to predict the likely state of weather of the surrounding regional area, either in real-time or in any other time intervals we desire. We can do experiments to collect data on temperature, wind and humidity levels over a specified period of time leading to the eventual occurrence of a rain event. In this way, we obtain sufficient datasets that can be fed into a machine learning time-series model that can detect patterns of changes in datasets that lead to the arrival of rain. I believe that the real-time collection and processing of data, which my

component achieves, is vital in allowing the machine learning models to make more accurate, and timely predictions, which in turn, allows for a more preemptive call for action on part of the users to carry out appropriate actions.

Section 5.6.4 Rationale for this extension proposal

I believe that there are many potential useful applications that can be further achieved with this extension. For example, if the system detects an unusual trend in data changes, and if the user is not around at home, it can notify the user via emails sent to their mobile phones, using relevant software packages designed to complete this task. There may be other existing solutions out there that are able to predict weather changes, but they are either too expensive, or there are limited further applications that accompany those original designs. There is obviously a trade-off between design costs involved and the full system functionalities, so my proposal aims to provide a long-term, viable and cheap solution for weather enthusiasts to have an alternative solution besides relying on other large scale weather monitoring systems.

Section 6 Firmware & Communication Details: Communication 2 - External

Section 6.1 Overview and Contributions

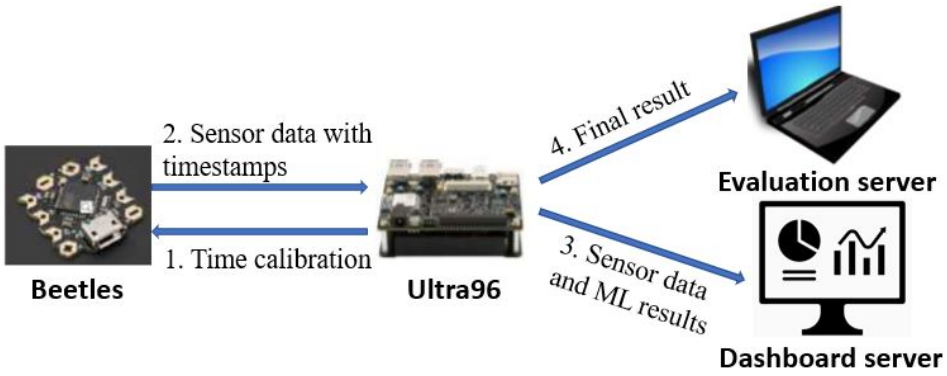


Figure 85: Processes involved in communication external

In overview, communication external has 2 main requirements. The first requirement is the communication between Ultra96 and the servers (3 and 4 in the above figure). And the second requirement is obtaining synchronization delay (1 and 2 in the above figure). The following table summarizes my contributions to the system as well as the modifications after week 7. The detailed implementations and rationale behind the modifications will be explained in each subsection.

Requirement	Feature	Modification after week 7	Tool
Communication between Ultra 96 and the servers	Communication with the evaluation server	The client receives the correct current position of the dancers from the evaluation server after each dance move.	Socket
	Communication with the dashboard	Message format is changed for the dashboard server.	Socket
	Secure communication		AES
Synchronization delay	Detection of start of a dance move and splitting data	Uses different thresholds to determine the start and end of a dance move. And separates the data packets of the walking phase and dancing phase for each action. In the walking phase data, timestamps are set to 0. In the dancing phase data, timestamps are all set to the time of the starting moment of the dance. Walking phase data and dancing phase data are separately passed to dance and position machine learning models.	Beetle
	Clock synchronization protocol	Integrates with the 3-way handshaking process in the communication internal part. Adds retransmission of time calibration packets in case of data loss.	Ultra96 and Beetle
	Time calibration	Frequency of time calibration is set to once every dance action.	Ultra96 and Beetle
	Synchronization delay calculations	Picks the most frequently occurred timestamp in dancing phase data as the Beetle's time at the start of a dance. And checks whether the data from a Beetle is None.	Ultra96

Before week 7, the communication between Ultra96 and the servers was developed. After week 7, the main focus is on obtaining accurate synchronization delay. In the following subsections, I will describe the incorporated features, explain the rationales.

Section 6.2 Communication between Ultra96 and Evaluation Server

Section 6.2.1 Process Overview

- The evaluation server should run first and create a socket and then waits for the client.
- The evaluation client is created on Ultra96 before dancing and establishes connection with the evaluation server.
- Synchronization delay is analyzed and calculated from timestamps obtained from 3 Beetles.
- The position and action are inferred from machine learning algorithms on Ultra96.
- Data is packed into the format '#position|action|syncdelay|' and encrypted using the AES encryption scheme with some secret key for secure communication.
- The evaluation client sends the encrypted data to the evaluation server.
- The evaluation server receives data and decrypts the message following the AES scheme using the same secret key.
- On the client (Ultra96) side, if the action is 'logout', the client closes its TCP connection to the server.
- The evaluation server sends back the correct current position of the dancers.
- The evaluation client receives the current position of the server and passes the information to machine learning for the next round of prediction.

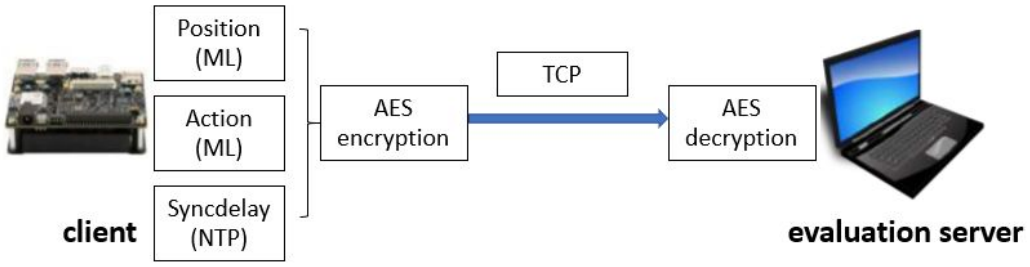


Figure 86: Communication between Ultra96 and evaluation server

Section 6.2.2 Rationale and Benefits for the System

This feature is very straightforward and crucial for the whole system. Sending and receiving data correctly and stably to or from the evaluation server directly affect the performance of the system.

We use socket programming for TCP connection between the server and the client. The reason for using TCP instead of other protocols such as UDP is that TCP provides reliable, ordered, and error-checked data transmission. A socket is one endpoint of a two-way communication link, which is a combination of a port number so that the TCP layer can identify the other application. (Docs.oracle.com, 2020).

The modification after week 7 is that Ultra 96 receives the correct current positions of the dancers for each action after the prediction. Originally, our machine learning algorithm was designed to directly predict the position based on the sensor data. But we found that the accuracy was very low. After analyses, we think the sensor data (from IMU) actually reflect the dynamics and movement of the dancers. So, we decided to add the initial position of the dancers at each round as the baseline and predict the change in position to get the final position. Based on the new machine learning algorithm for position prediction, the evaluation client on Ultra96 needs to receive the correct current position of the dancers and pass it to machine learning for the next round of prediction.

Section 6.2.3 Message Format

#position|action|syncdelay|
Positions can be ['1 2 3', '3 2 1', '2 3 1', '3 1 2', '1 3 2', '2 1 3']
Actions can be ['muscle', 'weightlifting', 'shoutout', 'dumbbells', 'tornado', 'facewipe', 'pacman', 'shootingstar', 'logout']

Section 6.2.4 Socket Implementation

The evaluation server code is provided. This subsection shows the implementations of the evaluation client.

Before dancing, an evaluation client is created on Ultra96. This object is instantiated for only once throughout the evaluation process.

```
eval_client = eval_client.Client("192.168.43.6", 8080, 6, "cg40024002group6")
```

Figure 87: instantiation of evaluation client

During the initialization of the evaluation client, IP, port number and the secret key are provided. With the information, a socket is created and connected to the evaluation client.

```
self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = (ip_addr, port_num)
self.secret_key = secret_key
self.socket.connect(server_address)
```

Figure 88: socket creation in evaluation client

When inferred dance action, position and calculated synchronization results are ready, the following send_data function is called which encrypts the data first and then sends the results via socket to the evaluation server.

```
def send_data(self, position, action, syncdelay):
    encrypted_text = self.encrypt_message(position, action, syncdelay)
    print("[Evaluation Client] encrypted_text: ", encrypted_text)
    self.socket.sendall(encrypted_text)
```

Figure 89: send data from the evaluation client

After the evaluation server receives the results, it logs the results into a file and sends back the correct current dancer position. On the client side, receive_dancer_position function is called to receive the current position and we pass it to the machine learning algorithm for the next round of position prediction.

```
def receive_dancer_position(self):
    dancer_position = self.socket.recv(1024)
    msg = dancer_position.decode("utf8")
    return msg
```

Figure 90: receive position data from the evaluation server

Section 6.3 Communication between Ultra96 and dashboard server

Section 6.3.1 Process Overview

The process is very similar to the process of communication between Ultra96 and the evaluation server. The difference is the data sent and processing of the data on the dashboard server.

- The dashboard server should run first and create a socket and then waits for the client.
- The dashboard client is created on Ultra96 before dancing and establishes connection with the dashboard server.
- Sensor data are sent from Beetles to Ultra96 and predicted position and action are from machine learning on Ultra96.
- Sensor data and machine learning results are encrypted using the AES encryption scheme with some secret key for secure communication.
- The dashboard client sends the encrypted data to the dashboard server.
- The dashboard server receives data and decrypts the message following the AES scheme using the same secret key.

- The dashboard server stores the sensor data and machine learning predictions into the database.

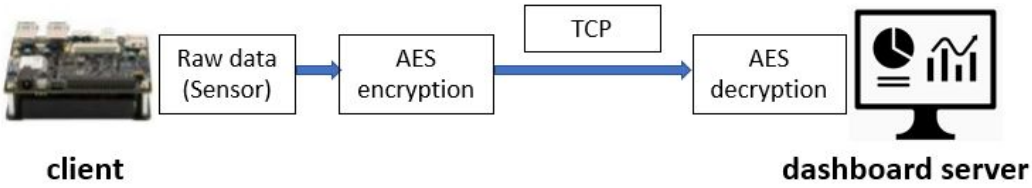


Figure 91: Communication between Ultra96 and dashboard server

Section 6.3.2 Rationale and Benefit for the System

Visualizing sensor data and predictions from machine learning help the user to analyze their movements, which is a core feature of our system. In the process of the implementation and development, sending data for visualization allows us to check the correctness of the machine learning predictions and debug the system. Especially we can easily view the value changes in the dashboard to check whether the Beetles are sending data stably.

After week 7, the message format sent to the dashboard server is modified based on the evaluation client code. The message sent to dashboard server is '#table|data' where table is the database table name to be inserted value into and data are the values to store in the table.

Section 6.3.3 Socket Implementation

The dashboard client code is modified from evaluation client code and dashboard server code is adjusted from evaluation server code.

Before the start of dancing, we create a separate dashboard server which is dedicated to sending data to the dashboard server. The dashboard client and server establish TCP connection the same way as the evaluation client and server.

```
board_client = dashBoardClient.Client("192.168.43.248", 8080, 6, "cg40024002group6")
```

Figure 92: instantiation of dashboard client

After Ultra96 collects sensor data and results from the machine learning algorithm, the dashboard client sends the data together with the database table name to be inserted to the dashboard server.

```
def send_data_to_DB(self, table, data):
    encrypted_text = self.encrypt_message_DB(table, data)
    print("[Dashboard Client] encrypted_text: ", encrypted_text)
    self.socket.sendall(encrypted_text)
```

Figure 93: send data from the dashboard client

Section 6.3.4 Sending Data to Dashboard

The dashboard designer decides to reflect ML results, EMG values and IMU values of the dancers. The following table lists the data we send to the dashboard.

Database Table Name	**Data**	**Sending Frequency**
MLDancer1	dance and position (from ML predictions)	Once every action
EMG	EMG values	Once every second
Beetle1/Beetle2/Beetle3	IMU values of each Beetle	Once every second

Section 6.4 Secure Communication

Section 6.4.1 Main Idea, Rationale and Benefits for the System

Secure communication protects the users' privacy when transmitting data to avoid leakage of the personal data. For encryption of the messages, we use the Advanced Encryption Standard (AES) scheme. AES is both secure and fast which is suitable for our system (Quora.com, 2020).

AES has 3 types AES-128, AES-192 and AES-256 depends on how many bits the key is. The longer the key is, the more secure the encryption is and the slower the encryption process can be. For our product, we adopt AES-128 because it provides sufficient security and is faster (sluiter, 2019). So, when running the evaluation and dashboard clients, a secret key of length 16 is provided (16 chars = 16 bytes = 128 bits).

Section 6.4.2 AES Encryption and Decryption Scheme

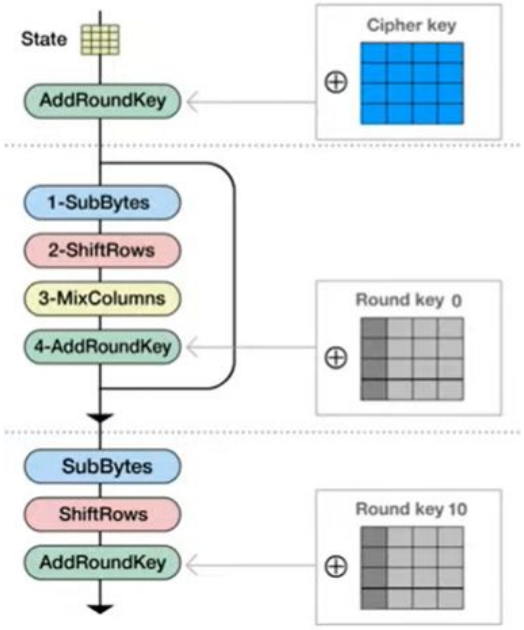


Figure 94: AES Encryption and Decryption Scheme

To encrypt the text, in the first step, the original text needs to perform XOR operation with a computed round key from the secret key. Then in the SubBytes process, Rijndael S-box lookup table is used to replace the blocks in the state. Next, the rows in the state are shifted and the columns are mixed by multiplying them with some matrices. Finally the round key is added to the state and the result is the ciphertext.

Decryption of ciphertext is done through the inverse process of the above encryption process.

Section 6.4.3 Encryption Implementation

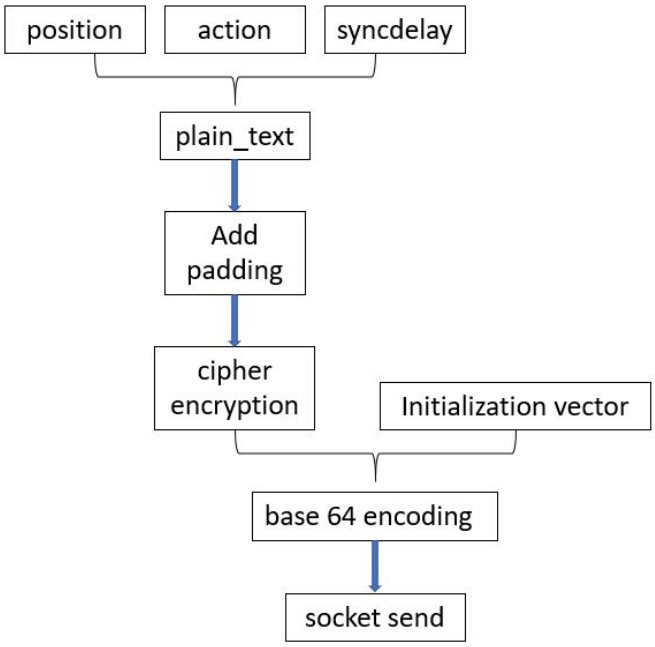


Figure 95: AES Encryption Algorithm

Following the above AES encryption algorithm, the following code snippet shows how the message is encrypted. Firstly, we create a string according to the message format and add paddings to it to meet the encryption requirement. Then an initialization vector and a cipher encryption are generated and they are passed to a base 64 encoding function to finally produce the AES encrypted text.

```
def encrypt_message(self, position, action, syncdelay):
    plain_text = '#' + position + '|' + action + '|' + syncdelay + '|'
    print("[Evaluation Client] plain_text: ", plain_text)
    padded_plain_text = self.add_padding(plain_text)
    iv = Random.new().read(AES.block_size)
    aes_key = bytes(str(self.secret_key), encoding="utf8")
    cipher = AES.new(aes_key, AES.MODE_CBC, iv)
    encrypted_text = base64.b64encode(iv + cipher.encrypt(bytes(padded_plain_text, "utf8")))
    return encrypted_text
```

Figure 96: message encryption function

Section 6.5 Detection of Start of a Dance Move and Splitting Data

Section 6.5.1 Main Idea, Rationale and Benefits for the System

Synchronization delay measures how synchronous the dancers are. It is calculated based on the time difference among the dancers' start of dance. And the project requires that the error of synchronization delay should be less than 1 ms. It is crucial to find an accurate and fast method to decide the moment of the start of dance because:

- Accuracy of the start time directly impacts the precision of the synchronization delay.
- Correctly separating the walking phase data and dancing phase data affect the accuracy of machine learning predictions. The walking phase data are used for the position prediction model and dancing phase data are fed to the action type prediction model. The purity of the datasets is important for the correctness of the final prediction.
- The speed of the method decides the speed of our system. For every dance action, this method needs to be applied to determine whether the dancers start dancing. So, the method should be fast to reduce the response time of our system

Before week 7, there were 2 ideas for determining the start of dance. The first idea is we set proper threshold values for the sensor data on Beetles to distinguish the walking (moving to the correct position) and dancing (actual dancing) data. The second idea is we pass the sensor data to another machine learning model to tell whether the dance starts. After week 7, we decided to adopt the first idea which uses thresholding values to separate the walking phase data and dancing phase data on Beetles. The rationale is that machine learning accuracy is not guaranteed and is much slower than directly splitting on the Beetles.

Section 6.5.2 Thresholding Implementation

To find proper thresholds, we conducted some experiments on Beetles to observe the value changes of the sensor data. There are 2 conditions to find out: transition from walking phase to dancing phase and transition from dancing phase to walking phase.

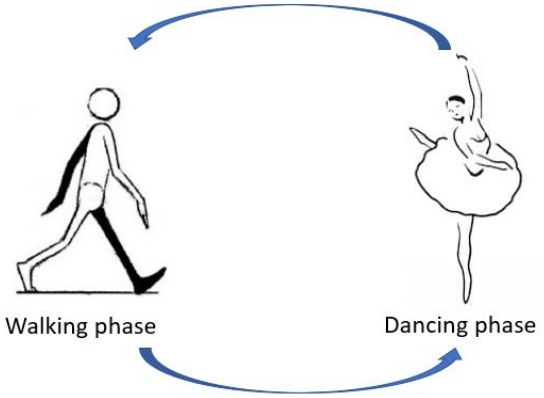


Figure 97: Transition between the walking phase and dancing phase

When the sensor data start to drastically change, we take that moment as the start of the dancing phase. We find the following condition composite of acceleration, yaw, etc. based on experiment results to set the flag of dancing phase.

```
if (!setOnce && stoppedMoving && ((abs(yawDiffSum) >= 15 || abs(pitchDiffSum) >= 15 || abs(rollDiffSum) >= 15)
&& (abs(accelXDiffSum) >= 5000 || abs(accelYDiffSum) >= 5000 || abs(accelZDiffSum) >= 5000))) {
  stoppedMoving = false;
  setOnce = true;
  timestamp = millis();
}
```

Figure 98: Transition from walking phase to dancing phase

When the sensor data values drop and become steady, we set the flag of stoppedMoving and assume now sensor data come from the walking phase.

```
if (tmp > 40 && !setOnce && (abs(yawDiffSum) <= 10 || abs(pitchDiffSum) <= 10 || abs(rollDiffSum) <= 10)
&& (abs(accelXDiffSum) <= 2000 || abs(accelYDiffSum) <= 2000 || abs(accelZDiffSum) <= 2000)) {
  stoppedMoving = true;
}
```

Figure 99: Transition from dancing phase to walking phase

Section 6.5.3 Splitting Data with Thresholding

With the above 2 conditions of thresholds, we can successfully separate walking phase data and dancing phase data on Beetles and obtain the Beetles' time of the start of a dance. For each Beetle, we use a moving dictionary to store walking phase data and a dancing dictionary to store dancing phase data.

```
beetle1_moving_dict = {"50:F1:4A:CB:FE:EE": {}}
beetle2_moving_dict = {"78:DB:2F:BF:2C:E2": {}}
beetle3_moving_dict = {"1C:BA:8C:1D:30:22": {}}
beetle1_dancing_dict = {"50:F1:4A:CB:FE:EE": {}}
beetle2_dancing_dict = {"78:DB:2F:BF:2C:E2": {}}
beetle3_dancing_dict = {"1C:BA:8C:1D:30:22": {}}
```

Figure 100: dictionaries for split data

In the dancing phase, we use a variable timestamp to keep the timestamp of the start of the dancing phase. And this timestamp is kept the same in packets of the same dancing phase.

```
{"1C:BA:8C:1D:30:22": {"56": [87051, -55.97, 48.21, 7.4, 15, -214, -13], "57": [87051, -55.64, 47.91, 7.46, 35, -305, -258],
"58": [87051, -55.5, 47.65, 7.42, 31, 321, 94], "59": [87051, -55.56, 47.5, 7.29, 30, -311, -285],
```

Figure 101: sample dancing phase data (highlighted part is the timestamp)

In the walking phase, timestamps are all 0 in the packets. When Ultra96 receives the data with 0 timestamp, it is able to tell the data are from the walking phase and should be passed to the position prediction machine learning model.

```
{"50:F1:4A:CB:FE:EE": {"2": [0, -72.39, -70.65, -19.2, 78, -84, 229],
"3": [0, -72.37, -70.63, -19.22, 46, -74, 205], "4": [0, -72.36, -70.61, -19.24, 20, -44, 193],
```

Figure 102: sample walking phase data (highlighted part is the timestamp)

Section 6.5.4 Passing Split Datasets to Machine Learning

With split datasets of walking phase and dancing phase, we can pass them separately to position model and dance model. The dancing data and walking data are first normalized for the prediction. Machine learning models predict the action and movement for each Beetle. And the final result takes the most frequent predictions for the 3 Beetles.

```
beetle1_dance = predict_beetle(beetle1_dance_data_norm, mlp_dance)
beetle2_dance = predict_beetle(beetle2_dance_data_norm, mlp_dance)
beetle3_dance = predict_beetle(beetle3_dance_data_norm, mlp_dance)
dance_predictions = [beetle1_dance, beetle2_dance, beetle3_dance]
dance = most_frequent_prediction(dance_predictions)
```

Figure 103: passing dancing phase data to dance action model

```
try:
    beetle1_move = predict_beetle(beetle1_moving_norm, mlp_move)
except Exception as e:
    beetle1_move = 'S'
try:
    beetle2_move = predict_beetle(beetle2_moving_norm, mlp_move)
except Exception as e:
    beetle2_move = 'S'
try:
    beetle3_move = predict_beetle(beetle3_moving_norm, mlp_move)
except Exception as e:
    beetle3_move = 'S'
# Find new position
new_pos = find_new_position(ground_truth, beetle1_move, beetle2_move, beetle3_move)
```

Figure 104: passing walking phase data to position model

Section 6.6 Clock Synchronization Protocol

Section 6.6.1 Main Idea, Rationale and Benefits for the System

Synchronization delay helps the users to evaluate how synchronous their dance is. The challenge is the 3 Beetles have their own clock time which are not synchronized. In order to obtain synchronization delay, the Beetles' time should be converted to standard Ultra96 time. In this project, we design our clock synchronization protocol based on the simplified version of Network Time Protocol (NTP). Network Time Protocol (NTP) is a networking protocol for clock synchronization between computer systems (En.wikipedia.org, 2020). The rationale for leveraging NTP is that this is a broadly used protocol for time calibration and has proved its success in many systems.

After week 7, we increase the robustness of clock synchronization protocol. We add retransmission of timestamp data if they are lost to deal with unexpected corner cases. The rationale is that we found the Beetles are sometimes unstable and time calibration packets may be lost. To guarantee the success of clock synchronization, retransmission and acknowledgement functionalities are added.

Section 6.6.2 Protocol Implementation

In our system, there is only 1 Ultra96 so we take its time as the standard reference time and convert the Beetles' time to Ultra96 time.

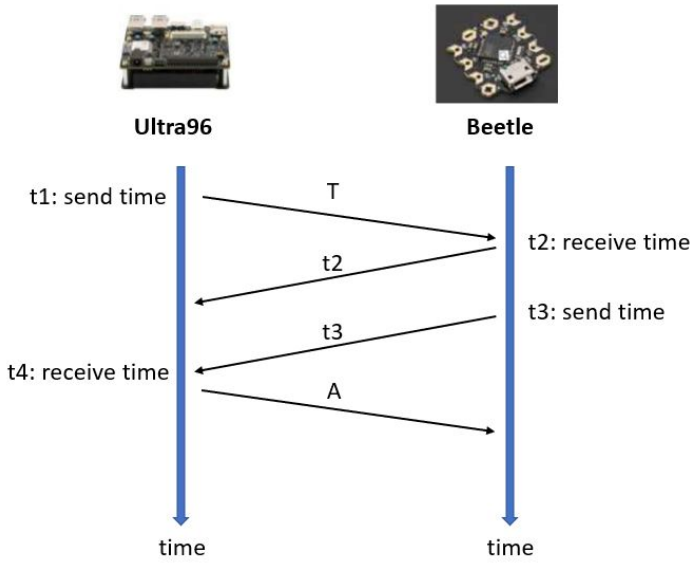


Figure 105: Clock synchronization protocol (success)

For each Beetle, Ultra96 initiates the time calibration process by sending 'T' to the Beetle and records its Ultra96 sending time t1. When the Beetle receives the 'T' packet, it keeps down its

local Beetle clock time t2 when receiving ‘T’ packet and clock time t3 when sending the packet back to Ultra96. Ultra96 takes down its Ultra96 time t4 after receiving the 2 packets from Beetles. In the end, Ultra96 sends ‘A’ to the Beetle as the acknowledgement of successfully receiving the packets.

However, sometimes the Beetles can be unstable and time calibration packets are lost. If Ultra96 fails to receive the 2 time packets from the Beetle, it will send ‘R’ to restart the time calibration process. Until the complete data packets are received, Ultra96 sends ‘A’ as the end of the time calibration process.

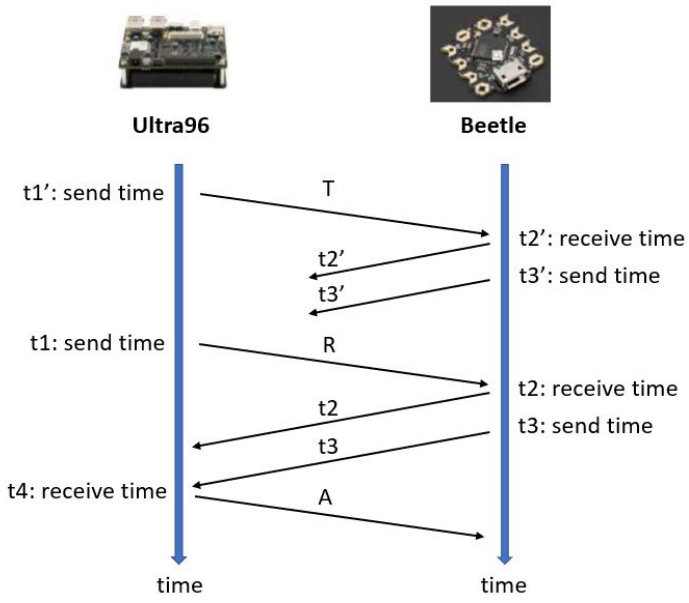


Figure 106: Clock synchronization protocol (retransmission)

The Beetle local clock time is acquired by millis() function, the unit is millisecond.

```
tmp_recv_timestamp = millis();

tmp_send_timestamp = millis();
```

Figure 107: Timestamp of Beetles

Ultra96 obtains its time using time(), and converts the time unit to millisecond.

```
ultra96_sending_timestamp = time.time() * 1000

ultra96_receiving_timestamp = time.time() * 1000
```

Figure 108: Timestamp of Ultra96

Section 6.6.3 Packet Format

When the Beetle sends its receiving and sending timestamps, it separates them into 2 packets instead of concatenating them into 1 string. The rationale of the design is to reduce the packet size. Each timestamp is long type which is 8 bytes. However, if we concatenate the timestamps to a string, the packet size will be the number of characters in the string, which is much larger than sending over long type messages separately.

Section 6.7 Time Calibration

Ultra96 employs our designed clock synchronization protocol to obtain the Beetles' local sending and receiving timestamps and records down its own sending and receiving timestamps. Based on the data, we need to calculate the clock offset to synchronize the Beetles' clock to Ultra96 reference time. And during the long dancing process, the Beetles' clock time keeps changing. So we need to determine the most suitable frequency of time calibration to trade off between accuracy and speed.

Section 6.7.1 Clock Offset Calculation

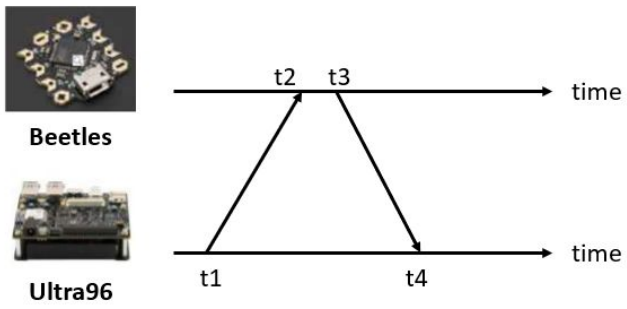


Figure 109: Time calibration process

Ultra96 collects t1, t2, t3 and t4. And it keeps clock offset variables for each Beetle. The formula for calculating clock offset is the following:

$$RTT = (t4 - t1) - (t3 - t2)$$

$$\text{One-way communication delay} = RTT/2$$

$$\text{Clock offset} = t2 - t1 - RTT/2$$

The code snippet is as follows. We deal with the corner case that the Beetle timestamps are lost.

```
def calculate_clock_offset(beetle_timestamp_list):
    if(len(beetle_timestamp_list) == 4) :
        RTT = (beetle_timestamp_list[3] - beetle_timestamp_list[0]) \
              - (beetle_timestamp_list[2] - beetle_timestamp_list[1])
        clock_offset = (beetle_timestamp_list[1] - beetle_timestamp_list[0]) - RTT/2
        return clock_offset
    else:
        print("error in beetle timestamp")
        return None
```

Figure 110: clock offset calculation

Section 6.7.2 Frequency of Time Calibration

Before week 7, we had 2 plans. Plan A is that we only do this calibration at the beginning and then stick to the same clock offset and communication delay in the following evaluations. Plan B is that we calibrate every polling interval. Short polling intervals update the parameters frequently but are sensitive to jitter and random errors. Long intervals may require larger corrections with significant errors between the updates (Ntp.org, 2020). Normal operating system uses 1024s as the polling interval.

After week 7, we decide to calibrate time for each dance action. In the experiments, we found that only the first calculated synchronization delay is accurate after time calibration, and later on, the values are very unreasonable (larger than 20 seconds) because the Beetles' clocks are easily disturbed by drastic movements. For the sake of accuracy, we sacrifice speed and calibrate time for each dance move.

Section 6.8 Synchronization Delay Calculation

Section 6.8.1 Timestamp of Start of a Dance for Beetles

Ideally, the Beetles separate packets into walking phase data where timestamps are 0 and dancing phase data where timestamps are all the time of the start of the dance. However, in the real case, the sensor values can suddenly drop and increase when the dancers move. The thresholding conditions in Beetles misinterpret this value change as a new dance action and assigns a new starting timestamp to the later packets. So the timestamps in dancing phase data for an action may not be the same.


```
{"1C:BA:8C:1D:30:22": {"1": [87051, -64.77, 48.03, 6.36, -75, -239, 121], "2": [87051, -64.48, 47.68, 7.45, -175, -40, -297],
"84": [103100, 103.37, -77.34, -7.3, 38, -273, 113], "85": [103100, 103.47, -77.42, -6.98, -148, 310, 172],
```


Figure 111: timestamp changes in dancing phase data

To deal with this challenge, we pick the most frequently occurred timestamp in the first 5 packets of a dancing phase dataset as the actual starting time of a Beetle. We pick the timestamps in the first 5 packets because the earlier drastic changes are more likely to be the start of a dance. But instead of picking the first timestamp, we pick the most frequently occurred timestamps because we want to rule out noises in the data caused by large movements in the walking phase.

Section 6.8.2 Asynchrony among the Dancers

Base on the clock offsets calculated in time calibration process and dance starting time of the Beetles, the asynchrony among dancers is calculated on Ultra96:

Time of start of a dance for Beetle 1, 2, 3 are t1, t2, t3

Clock offset of Beetle 1, 2, 3 are o1, o2, o3

Start time of actions on Beetle 1, 2, 3 (Ultra96 time): t1+o1, t2+o2, t3+o3

Asynchrony between the dancers: max(t1+o1, t2+o2, t3+o3) - min(t1+o1, t2+o2, t3+o3)

The following code snippet shows conversion of Beetles' time to Ultra96 time.

```
time_beetle = most_frequent(value_list)
time_ultra96 = time_beetle - clock_offset
return time_ultra96
```

Figure 112: convert Beetles time to Ultra96 time

When calculating synchronization delay, we consider the corner cases that some Beetles lose connection and fail to send its data packets to Ultra96. If any Beetle's data is None, we calculate synchronization delay using the other 2 Beetles to get a rough estimation. Checking and dealing with the corner cases increase the robustness of our system.

```
def calculate_sync_delay(beetle1_time_ultra96, beetle2_time_ultra96, beetle3_time_ultra96):
    sync_delay = 850
    if beetle1_time_ultra96 is not None and beetle2_time_ultra96 is not None and beetle3_time_ultra96 is not None:
        sync_delay = max(beetle1_time_ultra96, beetle2_time_ultra96, beetle3_time_ultra96) \
                     - min(beetle1_time_ultra96, beetle2_time_ultra96, beetle3_time_ultra96)
    elif beetle1_time_ultra96 is None:
        sync_delay = max(beetle2_time_ultra96, beetle3_time_ultra96) - min(beetle2_time_ultra96, beetle3_time_ultra96)
    elif beetle2_time_ultra96 is None:
        sync_delay = max(beetle1_time_ultra96, beetle3_time_ultra96) - min(beetle1_time_ultra96, beetle3_time_ultra96)
    elif beetle3_time_ultra96 is None:
        sync_delay = max(beetle1_time_ultra96, beetle2_time_ultra96) - min(beetle1_time_ultra96, beetle2_time_ultra96)
    return sync_delay
```

Figure 113: calculate synchronization delay

Section 6.9 Further Development of Communication External Components

Section 6.9.1 Better Design of Clock Synchronization Protocol

In our current design, time calibration and handshaking process share the same function receiveHandshakeAndClockSync(). The function contains some if-else to tell whether it is the time calibration message (with ‘T’) or the handshaking (with ‘H’) message. However, before the evaluation, we found that it’s not convenient to experiment with polling intervals of time calibration because it is not implemented in an independent function.

According to the software engineering principle separation of concerns ("Separation of concerns", 2020), computer programs should be separated to different sections with each section addressing a separate concern. So our future plan is to separate the handshaking and clock synchronization processes apart into different functions.

Section 6.9.2 Applying Secure Communication to Email Services

In this project, to protect the users’ privacy, we use AES encryption to ensure secure communication. However, in real life, internet security remains a problem and especially our emails’ privacy and security are not well respected. According to Wikipedia (Email encryption, 2020), some emails are still sent in a clear form without encryption.

The standard protocol for sending emails is Simple Mail Transfer Protocol (SMTP) which doesn’t support encryption and authentication. And some email service providers set no encryption as default (Email is completely insecure by default. | Viget, 2020). The implication is that we can apply AES to message transmission to secure the data privacy in the email services.

Section 7 Software Details

Section 7.1 Segmenting the sensed data

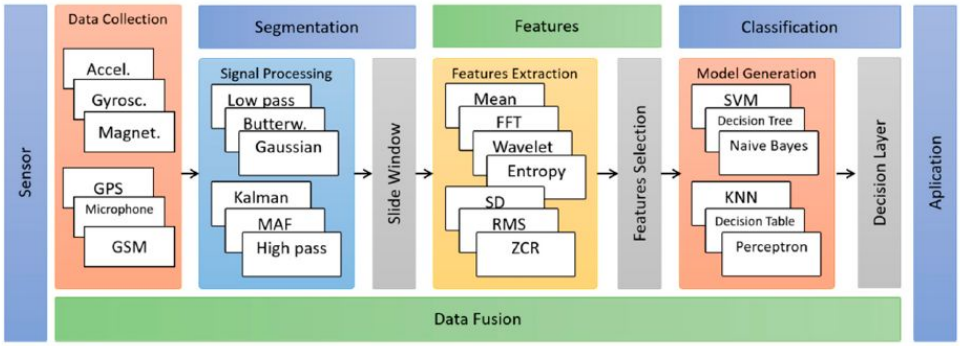


Figure 114: Overall machine learning process

Segmentation is the process of separating data into meaningful subgroups that share the same traits. Subgroups are represented by signal segments in a given time interval. The objective of segmentation is for each segment to contain enough characteristics to allow recognition of a dance move at a specific moment in time.

Data is collected at 20Hz from the Beetles. Studies have shown that frequencies of 20Hz contain enough information about physical activities, and thus it is believed that this would be sufficient to capture enough information about dance moves.

The data is then pre-processed by using the *I2Vdevlib* by *jrowberg* as mentioned in Section 3.7.1. This is done to remove noise and gravity and to obtain orientation values which are more representative of the dancer's actions.

Initially, it was decided that every dataset collected at 20Hz would be fed to the machine learning model and predictions would be made based on this data. However, prediction accuracy was poor at about 45%. We then decided that sensor data will be divided into consecutive segments so that each one of them can be analysed separately and sequentially. These segments are known as time windows. We used tumbling window-based segmentation, which divides sensor data into windows over time, as shown in Figure 115 below. It is believed that each window would be able to capture enough characteristics of a movement that would aid the machine learning model in determining and separating those characteristics. This showed, as the machine learning model's prediction accuracy improved to about 77.6%.

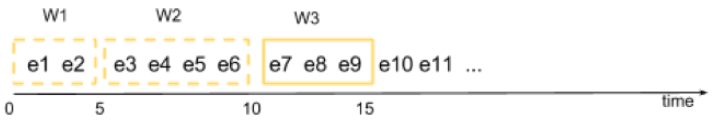


Figure 115: Tumbling window-based segmentation (Cloudera, 2020)

Section 7.2 Features to be explored

In the case of Support Vector Machine (SVM) and K-Nearest Neighbors (KNN) models, time domain and frequency domain features would have to be extracted for best results.

In the time domain, we can utilise statistical and non-statistical functions. Statistical functions include mean, variance, median while non-statistical functions include areas and bins distribution. The magnitude of x, y and z values can be used to assess level of movement intensity, and it is independent of the orientation of sensors on the user's body.

In the frequency domain, we can explore features such as the spectral energy, entropy, peak frequency and power of the signal. This is obtained via Fourier transform, which can be useful for capturing repetitive patterns of signals in terms of frequency. These features however, are dependent on the orientation of the sensors on the user's body. The Fourier transform will be implemented via software, such as this Python library: https://docs.scipy.org/doc/numpy/reference/routines.fft.html

Time domain features have a lower computational cost when compared to frequency domain features, but features in the frequency domain can better represent contextual information in terms of signal patterns. Thus, an appropriate number of features need to be picked from each domain so that it is not too computationally heavy while also maintaining a reliable level of accuracy. These features would aid with the accuracy of prediction results on SVM and KNN models.

However, studies have shown that Neural Networks (NNs) are able to extract higher level features on their own with provided raw data. Hence, we did not extract any time and frequency domain features for use with our NN model as NNs work well with raw data. As such, the only features we used were yaw, pitch, roll, and linear body accelerations in the X, Y and Z direction (accX, accY, accZ), making for a total of 6 features: ***yaw, pitch, roll, accX, accY and accZ.***

Section 7.3 Machine learning models under consideration, and optimizing chosen machine learning algorithm

SVMs are widely used in classification problems. It finds a hyperplane in an N-dimensional space that has the maximum distance between the data points of 2 or more classes, N being the number of features. The classes will represent the different types of dance moves. The scikit-learn documentation for SVM can be found at: https://scikit-learn.org/stable/modules/svm.html

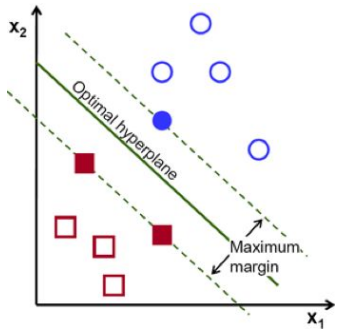


Figure 116: Visual representation of the SVM algorithm

Figure 116 above shows a case where the data can be classified linearly. However, for a dataset as shown in Figure 117 below, a linear classifier would not be suitable:

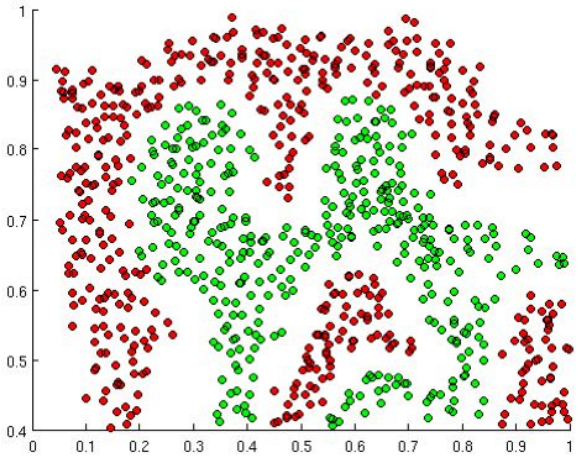


Figure 117: Dataset that cannot be classified linearly

In this case, a kernel for the SVM would have to be used. An appropriate kernel to use for this problem would be the Radial Basis Function (RBF) kernel:

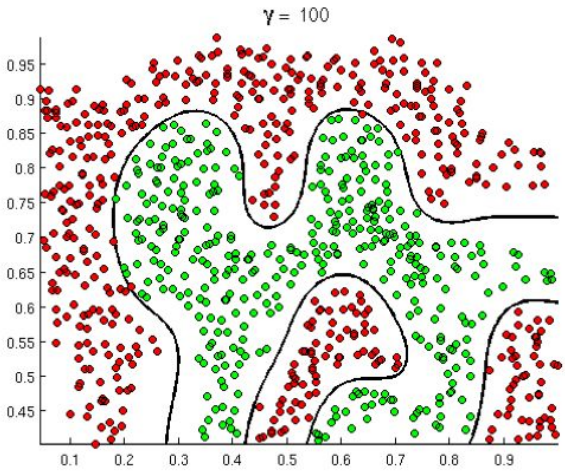


Figure 118: Dataset classified using the RBF kernel

The kernel can be specified in the scikit-learn library as documented in the link above.

The KNN algorithm assumes that in a dataset, the closer the data points are, they refer to the same thing. It commonly uses straight-line (Euclidean) distance to calculate the distance between the data points. It acquires the K nearest neighbors by finding the distances between a query and all the examples in the data, adds them to an ordered collection, sorts them in ascending order and picks the first K values from the sorted collection. The algorithm must be tried with different values of K, and the value of K which gives the lowest number of errors and hence stable predictions should be selected for the algorithm to be reliable.

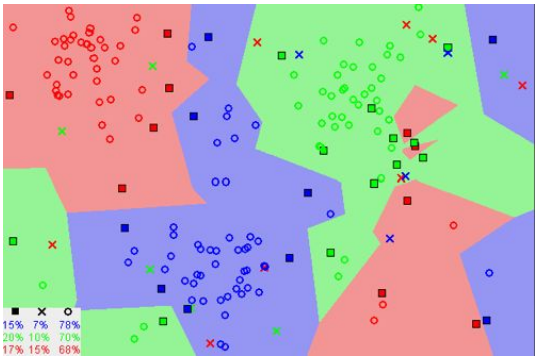


Figure 119: Visual representation of the KNN algorithm

The KNN algorithm is versatile as it delivers reliable results across different classification problems. However, it gets slower as the number of data points increases. Details for implementing KNN in scikit-learn can be found at:
https://scikit-learn.org/stable/modules/neighbors.html

A neural network is a machine learning algorithm that is able to capture complex patterns in datasets by using many hidden layers and non-linear functions associated with those layers. As shown in Fig 25 below, it takes in x1, x2, … , xN as inputs and outputs f(X). For our project, the inputs will be our datasets segmented according to features and the outputs will be the labels of dance moves according to the number (1, 2, … , 8). The scikit-learn implementation for feed-forward neural networks can be found at:
https://scikit-learn.org/stable/modules/neural_networks_supervised.html

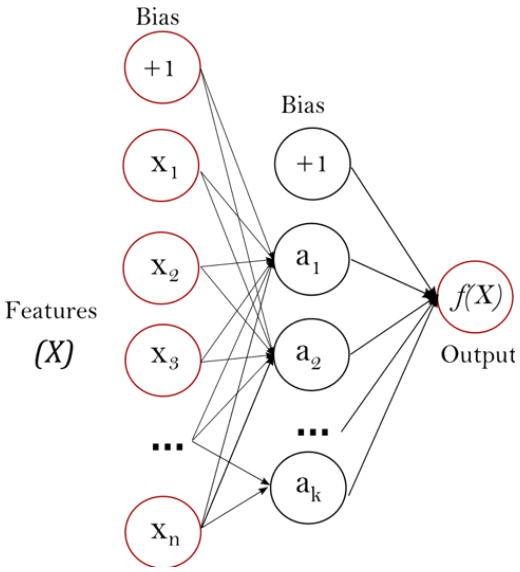


Figure 120: Feed-forward neural network

The number of middle hidden layers can be tweaked to obtain a model that is able to predict accurately and efficiently.

For our project, we used the MLPClassifier library provided by scikit-learn. It utilises a Multi-Layer Perceptron (MLP) neural network, which is a supervised learning algorithm that learns a unique function by training on a given set of data. The MLPClassifier trains on two arrays: an array X with the training samples as the rows and features as the columns, and an array Y which holds the labels of the respective training samples. In the case of our project, X would contain the data with respect to each feature (yaw, pitch, roll, accX, accY, accZ) and Y would contain the respective label for each sample. For the sample data in Figure 121, X would be [144.72, 64.74, 24.1, -304, -89, -213] while Y would be [‘shoutout’].

```
yaw_hand        144.72
pitch_hand       64.74
roll_hand         24.1
acc_X_hand        -304
acc_Y_hand         -89
acc_Z_hand        -213
Dance         shoutout
```

Figure 121: Sample dance data

After training the MLP model, it is able to predict labels for new samples of data.

Section 7.4 Training and testing the model

For the dance coaching wearable, each problem the machine had to solve could be looked at in two phases: the movement phase and the dancing phase. The movement phase occurs when the dancers move to the new positions and the dancing phase occurs when the dancers attempt to perform the dance move shown on the screen. The machine attempts to predict the new positions of the dancers based on data collected during the moving phase and the dance move based on data collected during the dancing phase. The data collected from the Beetles is separated into the 2 phases using thresholding mentioned in Section 6.5.2.

We trained two different MLP models: one for dance prediction (*mlp_dance*) and one to predict the movement of each dancer (*mlp_move*). Predicting dancer movement aids in relative position prediction which will be explained in a later section. Data collected during the movement phase and dance phase is fed to *mlp_move* and *mlp_dance* respectively. We believe that it is better to use two separate machine learning models so that it eliminates the possibility of the model mistakenly classifying a dance move as a general movement in the left or right direction.

MLP is sensitive to feature scaling (Scikitlearn, 2020), so we used the StandardScaler library from scikit-learn to scale our training data before feeding it to the MLP models. The StandardScaler library was fit on both training and test data, and in hindsight it should have only been fit on training data as information about the test set should not be used to transform the test set (Brownlee J, 2020).

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

# Now apply the transformations to the data:
scaler.fit(X_train)
X_train = scaler.transform(X_train)

scaler.fit(X_test)
X_test = scaler.transform(X_test)
```

Figure 122: Scaling the training and testing data

We opted for a 80-20 train-test split to train and test the performance of our model. A validation set was not used as we deemed that our datasets were not large enough to split into train, validation and test sets. The *train_test_split* library from scikit-learn was used to split the sensor data into training and test datasets, stratified by the data labels, as shown in Figure 123.

```
from sklearn.model_selection import train_test_split
y = data.Dance
X = data.drop('Dance', axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=123,
                                                    stratify=y)
```

Figure 123: Splitting dance data into training and testing datasets

Both models utilised an MLP containing 3 hidden layers with 256 neurons each. They were then fit onto the dance and movement datasets, with a loss of about 0.001 and 0.002 respectively.

```
import sklearn.neural_network as nn

mlp_dance = nn.MLPClassifier(hidden_layer_sizes=(256,256,256,),
                             max_iter=500, verbose = True)

MLP_dance = mlp_dance.fit(X_train, y_train)
Iteration 57, loss = 0.00114788
Training loss did not improve more than tol=0.000100 for 10 consecutive
epochs. Stopping.
```

Figure 124: Training *mlp_dance*

```
import sklearn.neural_network as nn

mlp_move = nn.MLPClassifier(hidden_layer_sizes=(256,256,256,),
                            max_iter=500, verbose = True)

MLP_move = mlp_move.fit(X_train, y_train)
Iteration 90, loss = 0.00154962
Training loss did not improve more than tol=0.000100 for 10 consecutive
epochs. Stopping.
```

Figure 125: Training *mlp_move*

After training the model, it is vital that we test it to see that it is making accurate predictions. To evaluate the performance of a model, we can use metrics that inform, in mathematical terms, how reliable the model is in detecting an activity. An example of an evaluation metric is accuracy.

Accuracy is the fraction of correct results among the total number of cases.

Accuracy = number of correctly classified activities / total number of instances

Accuracy for our models was measured using the *accuracy_score* library provided by scikit-learn. m*lp_dance* performed with an accuracy of 78.9% while *mlp_move* performed with an accuracy of 64.1%.

```
from sklearn.metrics import accuracy_score

# Accuracy Metric
y_pred_mlp_dance = mlp_dance.predict(X_test)
print('Accuracy score:', accuracy_score(y_test, y_pred_mlp_dance))
```

```
Accuracy score: 0.7893835616438356
```

Figure 126: Computing accuracy for *mlp_dance*

```
from sklearn.metrics import accuracy_score

# Accuracy Metric
y_pred_mlp_move = mlp_move.predict(X_test)
print('Accuracy score:', accuracy_score(y_test, y_pred_mlp_move))
```

```
Accuracy score: 0.6405228758169934
```

Figure 127: Computing accuracy for *mlp_move*

Confusion matrices were also generated to assess the models' performance. We used the function *plot_confusion_matrix* by *vikashrajluhaniwal*, which utilises the pyplot library from matplotlib to plot the confusion matrix generated using scikit-learn's *confusion_matrix* library. The confusion matrices for mlp_dance and mlp_move are shown as follows:

```
# confusion matrix
# Comparing the predictions against the actual observations in y_val
cm_mlp_dance = confusion_matrix(y_pred_mlp_dance, y_test)
print(cm_mlp_dance)
```

```
[[124  23  13]
 [ 26 210  22]
 [ 11  28 127]]
```

Figure 128: Confusion Matrix for *mlp_dance* using *confusion_matrix*

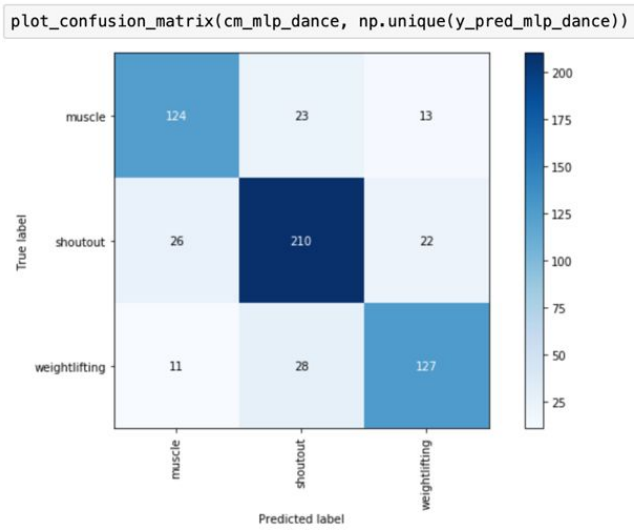


Figure 129: Confusion Matrix for *mlp_dance* using *plot_confusion_matrix*

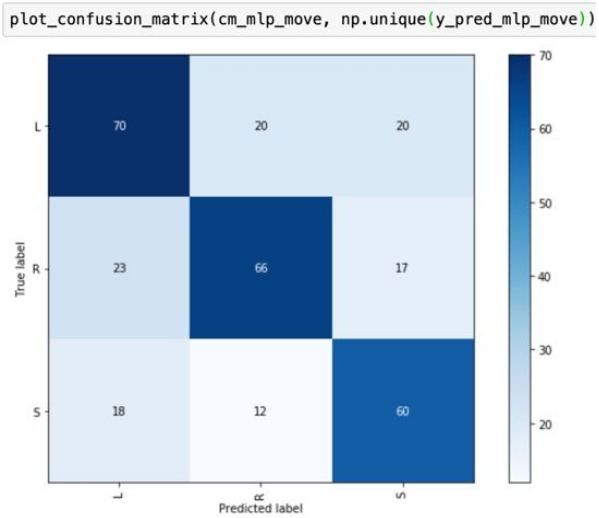


Figure 130: Confusion Matrix for *mlp_move*

Section 7.5 Storing the incoming sensor data

For model training, we wanted to simulate the exact test environment so that the model would be able to gather data in test conditions and it is believed that this would provide the model with realistic data to make optimal predictions during the actual testing of our system. To achieve this, we ran the evaluation server to display the dance move and new positions, and collected sensor data while simulating the actual test.

We modified the evaluation server script to write the dance moves and the positions which it displayed to *labels.txt* as shown in Figure 131:

```
with open('labels.txt', 'a') as label_file:
    line_file = my_server.action + " " + my_server.dancer_positions
        + "\n"
    label_file.write(line_file)
```

Figure 131: Writing evaluation server labels to labels.txt

```
shoutout 2 3 1
shoutout 1 3 2
shoutout 1 2 3
shoutout 1 3 2
muscle 2 3 1
weightlifting 3 2 1
muscle 1 2 3
shoutout 2 3 1
weightlifting 3 1 2
muscle 3 2 1
shoutout 1 2 3
shoutout 3 2 1
```

Figure 132: Sample *labels.txt*

The file *labels.txt* would then be parsed via the *parse_labels* function, as shown in Figure 133:

```
def parse_labels(filename):
    # beetle_1, 2, 3 are addresses of the beetles in string

    dance_labels = [] # dance performed
    movement_labels = {'1':[], '2':[], '3':[]}
    # track beetle movement in dict, key = beetle_num

    pos_old = ['1', '2', '3'] # initial pos

    # parse labels.txt
    with open(filename,'r') as inf:
        for line in inf:
            labels = line.split()

            # store dance move
            dance_labels.append(labels[0])

            # store movement based on prev numbers
            # e.g. 1 2 3 -> 3 1 2
            pos_new = [labels[1], labels[2], labels[3]]

            for i in range(3): # for a person in the old position
                for x in range(3): # find a person in the new position
                    if pos_old[i] == pos_new[x]: # person found
                        if i < x:
                            # he moved right
                            movement_labels[pos_old[i]].append('R')
                        elif i > x:
                            # he moved left
                            movement_labels[pos_old[i]].append('L')
                        else:
                            # he stood still
                            movement_labels[pos_old[i]].append('S')
                        break
            pos_old = pos_new

    return (dance_labels, movement_labels)
```

Figure 133: *parse_labels* function

The function *parse_labels* extracts the dance moves and stores them sequentially in the list *dance_labels*. The positions of each Beetle is also extracted and the logic above determines the movement of each Beetle and stores them sequentially in a dictionary, with each key representing the Beetle number and and the value representing a list which contains the movement of each beetle sequentially throughout the dance evaluation process. The function finally returns the sequential dance and movement labels associated with each beetle, to be used for labelling the data later.

Data from the three Beetles during the movement phase is collected and stored in three separate dictionaries: *beetle1_move_dict*, *beetle2_move_dict* and *beetle3_move_dict* while the data collected during the dancing phase is stored in *beetle1_dance_dict*, *beetle2_dance_dict* and *beetle3_dance_dict* accordingly.

The sensor data for the movement and dancing phases was collected in the form of these dictionaries and written to *position.txt* and *dance.txt*. Every line of data represents sensor readings collected from 1 beetle, and every 3 lines of data represents 1 dance during the evaluation (as 1 dance phase would produce dance data readings from 3 beetles). An example of data from 1 beetle in *dance.txt* is shown in Figure 134:


```
{"50:F1:4A:CB:FE:EE": {"58": [113545, 102.63, -76.19, -12.47, -298, -71, -213], "59": [113545, 102.48, -76.23, -12.39, 278, -145, -124],
"60": [113545, 102.36, -76.25, -12.31, 98, -279, 44], "61": [113545, 102.33, -76.21, -12.31, -130, 252, 244], "62": [113545, 102.36,
-76.13, -12.37, -296, 158, 313], "63": [113545, 97.73, -80.79, -8.56, 198, -61, 170], "64": [113545, 98.11, -80.55, -8.94, -6, -114,
-121], "67": [113545, 99.3, -79.09, -10.47, -98, -280, -311], "68": [113545, 99.28, -78.83, -10.69, 312, 0, 219], "69": [113545, 99.44,
-78.48, -11.05, -82, -93, 47], "70": [113545, 99.79, -78.19, -11.44, 301, -69, 193], "71": [113545, 100.93, -66.64, 4.83, 133, -124, -29],
"72": [113545, 100.9, -65.74, 5.37, -117, -281, 88], "73": [113545, 101.23, -64.87, 5.49, -314, 26, -48], "74": [113545, 101.82, -64.15,
5.27, -184, -267, 245], "75": [113545, 102.41, -63.64, 5.05, 210, 204, -249], "76": [113545, 102.89, -63.39, 4.95, -61, 100, 6], "77":
[113545, 103.35, -63.37, 4.91, 188, 89, -146], "78": [113545, 110.35, -74.56, -0.23, -79, 210, 66], "80": [113545, 109.96, -75.48, -1.45,
-261, 207, 290], "81": [113545, 109.85, -75.74, -1.61, -115, -308, 112], "82": [113545, 109.6, -76.02, -1.56, 122, -194, -126], "83":
[113545, 109.27, -76.27, -1.33, 236, 35, -183], "84": [113545, 108.99, -76.35, -1.06, 236, 184, -143], "85": [113545, 108.77, -76.29,
-0.98, 239, 8, -54], "86": [113545, 108.78, -71.11, 2.77, 116, -234, -119], "87": [113545, 108.74, -71.33, 2.75, 203, -188, -126], "88":
[113545, 108.62, -71.62, 2.7, 240, -2, 38], "89": [113545, 108.49, -71.94, 2.62, 178, 263, -311], "90": [113545, 108.42, -72.41, 2.32,
-19, -5, 177], "91": [113545, 108.63, -72.68, 1.83, -50, 189, 177], "92": [113545, 108.69, -72.78, 1.69, 42, -311, 143], "93": [113545,
76.86, -24.01, 28.43, -59, 220, 82], "94": [113545, 70.61, -16.49, 30.97, 234, -90, 49], "95": [113545, 67.04, -12.53, 32.08, 4, 193,
-133], "96": [113545, 63.56, -8.78, 32.59, 320, 45, 50], "97": [113545, 60.49, -5.0, 32.47, 293, -107, -288], "98": [113545, 57.6, -1.2,
31.83, -249, 307, 284], "99": [113545, 39.36, 32.09, 14.38, 162, -54, 24], "100": [113545, 39.67, 32.88, 13.92, 237, -194, -253], "101":
[113545, 39.73, 33.75, 13.46, -325, -278, 61], "102": [113545, 39.55, 34.67, 12.94, -234, 319, -91], "103": [113545, 39.19, 35.53, 12.31,
-279, -297, 132], "107": [113545, 36.46, 38.15, 9.61, 31, 56, -183], "108": [113545, 44.1, 38.41, 17.67, 148, 225, 47], "109": [113545,
44.66, 37.24, 18.2, -116, -315, -115], "110": [113545, 45.06, 36.03, 18.8, -230, -223, -319], "111": [113545, 45.34, 33.46, 20.46, -318,
-172, 23], "112": [113545, 45.44, 32.16, 21.18, -136, -34, -241], "113": [113545, 45.68, 30.98, 21.55, -167, -299, -82], "114": [113545,
37.03, 42.3, 11.72, -1, 249, -116], "115": [113545, 36.5, 42.57, 11.29, -79, 323, 188], "116": [113545, 36.01, 42.76, 10.89, -237, -288,
-152], "117": [113545, 35.71, 42.89, 10.47, 218, -178, 136], "118": [113545, 35.53, 42.96, 10.11, 38, -105, -158], "119": [113545, 35.26,
43.11, 9.68, 320, -209, 295], "120": [113545, 35.32, 43.23, 9.52, 244, -294, 223], "121": [113545, 35.5, 43.41, 9.33, 260, -251, 102],
"122": [113545, 42.8, 34.65, 19.3, 323, -218, -283], "123": [113545, 43.12, 33.27, 19.81, 157, 46, -149], "124": [113545, 43.99, 30.97,
20.2, 136, 146, -288], "125": [113545, 44.3, 29.89, 20.4, 146, 22, 98], "126": [113545, 44.55, 28.9, 20.58, 63, -140, -81], "128":
[113545, 44.77, 27.49, 20.9, -241, -170, 312], "129": [113545, 44.68, 27.05, 21.06, 266, -153, -258], "130": [113545, 34.0, 42.81, 13.74,
-3, 95, -93], "131": [113545, 33.36, 43.41, 12.88, 238, 113, -201], "133": [113545, 32.87, 43.61, 12.07, -313, 252, 18], "134": [113545,
32.85, 43.75, 11.86, 300, 292, -209], "135": [113545, 33.06, 43.87, 11.68, -195, 242, 211], "136": [113545, 33.36, 43.89, 11.49, -36, 178,
119], "139": [113545, 36.56, 40.89, 18.83, -137, 169, 98], "140": [113545, 36.73, 39.96, 19.4, -101, 258, -174], "141": [113545, 38.02,
37.68, 20.39, 117, 108, -214], "142": [113545, 38.79, 36.56, 20.88, -26, 13, -194], "143": [113545, 39.16, 37.86, 19.67, 315, 288, -291],
"144": [113545, 37.82, 39.65, 18.02, 166, 284, -318], "145": [113545, 37.36, 40.35, 17.25, 58, -216, 35], "146": [113545, 37.02, 40.95,
16.53, -23, -194, -291], "147": [113545, 36.68, 41.52, 15.81, -90, -129, -13], "148": [113545, 36.24, 42.05, 15.13, -108, -24, 225],
"149": [113545, 35.69, 42.5, 14.53, -139, 137, -174], "150": [113545, 35.04, 42.09, 16.29, 325, 154, -207], "151": [113545, 35.24, 41.7,
16.69, 155, -323, 99], "153": [113545, 35.6, 40.76, 17.67, -64, -200, -166], "154": [113545, 35.79, 40.24, 18.29, 35, -313, -23], "156":
[113545, 36.56, 38.98, 19.46, 192, -163, 143], "157": [113545, 36.87, 38.18, 19.94, 71, 22, -11], "158": [113545, 37.04, 37.3, 20.47, -95,
4, -293], "159": [113545, 34.83, 34.42, 23.35, 63, 172, 30], "160": [113545, 34.29, 35.17, 22.68, -50, -243, -59], "161": [113545, 33.58,
36.55, 21.4, -301, 159, -50], "162": [113545, 33.44, 37.23, 20.84, -206, 155, -78], "163": [113545, 33.42, 37.88, 20.28, -3, 41, -139],
"164": [113545, 33.38, 38.49, 19.66, 151, -4, -171], "165": [113545, 33.13, 39.05, 19.05, 117, -2, -66], "166": [113545, 29.53, 44.21,
11.47, 163, -197, -287], "167": [113545, 30.33, 44.13, 11.59, 202, -145, -300]}}
```


Figure 134: Sample dance data for 1 beetle written to *dance.txt*

We developed a logic to parse dance data into time windows of 5 readings, as mentioned earlier in Section 7.1. This logic is shown in Figure 135:

```
window_size = 5
def func(x):
    return int(x[0])

def parse_dance_data(filename, dance_labels):
    # collect beetle data
    data = []
    # every 3 lines evaluated, next dance label
    a = 1
    b = 0
    z = 0 # time series counter


    with open(filename,'r') as inf:
        for line in inf:
            # evaluate 3 lines (beetles) for 1 label

            dic_file = eval(line)
            for k,v in dic_file.items(): # k = beetle address
                if v: # dict is not empty
                    window_count = 0
                    ypr = []
                    for data_num, data_value in sorted(v.items(), key = func):
                        window_count += 1
                        for i in range(1,7):
                            ypr.append(data_value[i])

                        if (window_count%window_size == 0):
                            dance_label = dance_labels[b] # take current dance label
                            ypr.append(dance_label)
                            data.append(ypr)
                            ypr = []

            if a%3 == 0:
                b += 1 #increment pointer

            a += 1
    return(data)
```

Figure 135: Logic for parsing sensor data

The function *parse_dance_data* sequentially takes in and splits the data into lists containing 5 sets of 6 readings (yaw, pitch, roll, accX, accY, accZ) each, totalling to 30 readings per label, and appends these lists with the respective dance label obtained from the *dance_labels* list returned by the *parse_labels* function as explained earlier. For every 3 lines in dance.txt (representing 3 beetles during collection of 1 dance), the pointer is incremented to point to the next dance move shown during the evaluation phase.

Similarly, we developed a logic to parse movement data into time windows of 5 readings, as shown in Figure 136:

```
window_size = 5
def func(x):
    return int(x[0])

def parse_move_data(filename, movement):
    # collect data
    data = []
    a = 1
    b = 0 # point to first movement
    z = 0 # time series counter

    with open(filename,'r') as inf:
        for line in inf:

            dic_file = eval(line)
            for k,v in dic_file.items(): # addr = beetle address
                if v: # not empty dict
                    window_count = 0
                    ypr = []

                    for data_num, data_value in sorted(v.items(), key = func):
                        window_count += 1
                        for i in range(1,7):
                            ypr.append(data_value[i])

                        if (window_count%window_size == 0):
                            movement_label = movement[k][b] # get label
                            ypr.append(movement_label)
                            data.append(ypr)
                            ypr = []

            if a%3 == 0:
                b +=1
            a +=1

    return(data)
```

Figure 136: Logic for parsing movement data

```
dance_labels, movement_labels = parse_labels(label_filename)

movement = {}
movement[beetle_1] = movement_labels['1']
movement[beetle_2] = movement_labels['2']
movement[beetle_3] = movement_labels['3']
```

Figure 137: Mapping movement of dancer no to beetle address

```
beetle_1 = "50:F1:4A:CB:FE:EE"
beetle_2 = "1C:BA:8C:1D:30:22"
beetle_3 = "78:DB:2F:BF:2C:E2"
```

Figure 138: Beetle addresses

The sequentially stored movement labels in the form of lists from the parsing of labels.txt will be mapped to the Beetle address in the dictionary *movement*, where *beetle_1*, *beetle_2* and *beetle_3* are the addresses of the Beetles used. The dictionary *movement* is then passed to *parse_move_data*, where the sensor readings are collected and stored in sets of 5 totaling to 30 readings per label in a similar manner to *parse_dance_data*. The key difference is that each beetle key as read in the dictionaries in *position.txt* would have to be pointed to the appropriate list in the dictionary *movement*, where the list of sequential movements are stored.

To complete the collection of the dataset, a header is needed to describe the data being passed to the MLP model. Thus, we created a list describing the features of the data being passed to the MLP. The list would be created based on the length of the time window, as shown in Figure 139:

```
time_series = 5

headings = ['y_', 'p_', 'r_', 'accX_', 'accY_', 'accZ_']
header = []

for i in range(1,time_series+1):
    for j in range(6):
        header.append(headings[j] + str(i))

header.append('Movement')

move_headings = header
print(move_headings)
```

```
['y_1', 'p_1', 'r_1', 'accX_1', 'accY_1', 'accZ_1', 'y_2', 'p_2', 'r_2', 'accX_2', 'accY_2'
, 'accZ_2', 'y_3', 'p_3', 'r_3', 'accX_3', 'accY_3', 'accZ_3', 'y_4', 'p_4', 'r_4', 'accX_4
', 'accY_4', 'accZ_4', 'y_5', 'p_5', 'r_5', 'accX_5', 'accY_5', 'accZ_5', 'Movement']
```

Figure 139: Header containing the feature titles for a time window of length 5

For the case of movement data, this header *move_headings* is then appended to the data before the .txt files are parsed and stored as lists. The end result is a 2D array (list of lists), in which the first element is the list describing the headings and the second element onwards would be the data collected in time windows of 5 datasets and labelled accordingly to the movement (L – left, R – right, S – still), as shown in Figure 140:

```
final_move_data = []

final_move_data.append(move_headings)

move_data = parse_dataset_move('position.txt', 'labels.txt')

final_move_data.extend(move_data)

move_data = parse_dataset_move('position2.txt', 'labels2.txt')

final_move_data.extend(move_data)

move_data = parse_dataset_move('position3.txt', 'labels3.txt')

final_move_data.extend(move_data)

move_data = parse_dataset_move('position4.txt', 'labels4.txt')

final_move_data.extend(move_data)

print(final_move_data[:3])
```

```
[['y_1', 'p_1', 'r_1', 'accX_1', 'accY_1', 'accZ_1', 'y_2', 'p_2', 'r_2', 'accX_2', 'accY_2
', 'accZ_2', 'y_3', 'p_3', 'r_3', 'accX_3', 'accY_3', 'accZ_3', 'y_4', 'p_4', 'r_4', 'accX_
4', 'accY_4', 'accZ_4', 'y_5', 'p_5', 'r_5', 'accX_5', 'accY_5', 'accZ_5', 'Movement'], [-7
2.39, -70.65, -19.2, 78, -84, 229, -72.37, -70.63, -19.22, 46, -74, 205, -72.36, -70.61, -1
9.24, 20, -44, 193, -72.37, -70.6, -19.25, 24, -2, 173, -72.36, -70.59, -19.26, 57, 20, 171
, 'R'], [-72.33, -70.56, -19.29, 91, 41, 157, -72.29, -70.52, -19.33, 101, 62, 144, -72.24,
-70.47, -19.38, 64, 66, 136, -72.21, -70.43, -19.42, 5, 70, 122, -71.99, -70.51, -19.39, -6
9, 144, 271, 'R']]
```

Figure 140: Overall process of parsing and appending data

Once all the data is parsed and stored in the list of lists *final_move_data,* it is then written to a csv file to be used for MLP training and testing. Figure 141 describes the flow of writing to the csv file:

```
import csv

csv_filename = "move_data.csv"

with open(csv_filename, "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(final_move_data)
```

Figure 141: Writing data to csv file

The pandas csv reader is then used to read the csv file into a DataFrame for the MLP to process, as shown in Figure 142:

```
# Load movement data
import pandas as pd
data = pd.read_csv("move_data.csv")
data
```

	y_1	p_1	r_1	accX_1	accY_1	accZ_1	y_2	p_2	r_2	accX_2	...	accX_4	accY_4	accZ_4	y_5
0	-72.39	-70.65	-19.20	78	-84	229	-72.37	-70.63	-19.22	46	...	24	-2	173	-72.36
1	-72.33	-70.56	-19.29	91	41	157	-72.29	-70.52	-19.33	101	...	5	70	122	-71.99
2	-71.98	-70.54	-19.36	-70	181	244	-71.96	-70.59	-19.33	-68	...	-62	190	183	-71.93
3	-71.81	-70.93	-19.03	-85	224	181	-67.37	-68.51	-21.45	-225	...	163	-57	-33	-66.76
4	-66.71	-67.80	-22.18	199	-55	16	-68.16	-68.68	-21.32	-285	...	-11	-283	-305	-68.13
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1525	100.99	-79.52	-6.72	-3	-294	-33	98.58	-77.38	-3.45	317	...	173	-252	-127	98.25
1526	97.98	-77.74	-2.45	12	19	-19	97.73	-77.87	-2.11	-95	...	-148	72	-205	97.58
1527	95.78	-76.28	1.41	-179	-134	60	95.78	-76.16	1.42	-246	...	274	130	-86	95.83
1528	96.08	-75.84	1.15	288	158	65	96.37	-75.67	0.93	-268	...	-33	-153	-311	97.27
1529	97.18	-73.40	0.20	-255	-94	170	97.13	-73.37	0.22	-236	...	-158	-169	253	97.13

1530 rows × 31 columns

Figure 142: Read csv file into DataFrame for processing

After training and testing the model as described in Section 7.4, the MLP model is then exported to a *.joblib* file using the *joblib* library, ready for use in predicting real time data. The process is described in Figure 143:

```
from joblib import dump, load
dump(mlp_move, 'mlp_movement.joblib')
```

```
['mlp_movement.joblib']
```

Figure 143: Exporting MLP to .joblib file for real time prediction

Section 7.6 Real Time Streaming

Real time sensor data is stored in the dictionaries mentioned in Section 7.5: *beetle1_move_dict*, *beetle2_move_dict*,*beetle3_move_dict*, *beetle1_dance_dict*, *beetle2_dance_dict* and *beetle3_dance_dict*.

The same logic detailed in Section 7.5 is used to parse real time data. The only difference is that instead of reading the dictionaries from .txt files, the dictionaries are directly passed to the data parsing function *parse_data*, as shown in Figure 144:

```
def func(x):
    return int(x[0])

def parse_data(dic_data, beetle):
    # collect hand data
    window_size = 5
    data = []
    for k,v in dic_data.items(): # k = beetle address
        if v: # dict is not empty
            window_count = 0
            ypr = []
            for data_num, data_value in sorted(v.items(), key = func):
                window_count += 1
                for i in range(1,7):
                    ypr.append(data_value[i])

                if (window_count%window_size == 0):
                    data.append(ypr)
                    ypr = []
    return (data)
```

Figure 144: Parsing real-time data

The data returned from *parse_data* is then normalised using scikit-learn's *StandardScaler* function described in Section 7.4 before it is sent to the MLP model for prediction. This is done via the normalise_data function, as shown in Figure 145:

```
def normalise_data(data):
    scaler = StandardScaler()
    scaler.fit(data)
    data = scaler.transform(data)

    return data
```

Figure 145: Normalising the parsed sensor data

Similar to the issue described in Section 7.4, it would have been a better practice to export the scaler used on the train data for training the MLP and import it into the main code to normalise the parsed sensor data before passing it to the MLP for prediction. This is to ensure consistency over the scaling used for the training and prediction datasets.

Before processing the normalised data for prediction, the exported MLP is loaded by the main script using *joblib*, as shown in Figure 146:

```
from joblib import dump, load

# Load MLP NN model
mlp_dance = load('mlp_dance.joblib')

# Load Movement ML
mlp_move = load('mlp_movement.joblib')
```

Figure 146: Importing trained MLP models for prediction

Section 7.7 Transition and dance prediction

To make a prediction, the MLP predicts a label for every time window, and the most predicted label is returned as the prediction as shown in Figure 147:

```
def predict_beetle(beetle_data, model):
    pred_arr = model.predict(beetle_data)
    unique, counts = numpy.unique(pred_arr, return_counts=True)
    pred_count = dict(zip(unique, counts))
    prediction = max(pred_count.items(), key=operator.itemgetter(1))[0]
    return prediction
```

Figure 147: Using the imported model to make a prediction

To predict the final positions of the dancers, the function *find_new_position* takes in the ground truth, and evaluates the movement of each Beetle based on the *mlp_move* prediction. Based on the movements of each Beetle (*b1_move, b2_move* and *b3_move*) and the initial positions from *ground_truth*, using the logic as described in Figure 148, the function is able to return the dancers' new positions:

```
def find_new_position(ground_truth, b1_move, b2_move, b3_move):

    dic = {1: b1_move, 2: b2_move, 3: b3_move}

    # Determine movement of the beetle in that position
    p1_movement = dic[ground_truth[0]]
    p2_movement = dic[ground_truth[1]]
    p3_movement = dic[ground_truth[2]]

    if p1_movement == "R" and p2_movement == "S" and p3_movement == "L":
        # output = [3, 2, 1]
        output = [ground_truth[2], ground_truth[1], ground_truth[0]]
    elif p1_movement == "R" and p2_movement == "L" and p3_movement == "S":
        # output = [2, 1, 3]
        output = [ground_truth[1], ground_truth[0], ground_truth[2]]
    elif p1_movement == "R" and p2_movement == "L" and p3_movement == "L":
        # output = [2, 3, 1]
        output = [ground_truth[1], ground_truth[2], ground_truth[0]]
    elif p1_movement == "S" and p2_movement == "R" and p3_movement == "L":
        # output = [1, 3, 2]
        output = [ground_truth[0], ground_truth[2], ground_truth[1]]
    elif p1_movement == "S" and p2_movement == "L" and p3_movement == "S":
        # output = [2, 1, 3]
        output = [ground_truth[1], ground_truth[0], ground_truth[2]]
    else:
        # output = [1, 2, 3]
        output = ground_truth
        position = str(output[0]) + " " + str(output[1]) + " " + str(output[2])

    return position
```

Figure 148: Logic to determine new position

Data from all the Beetles are needed to predict the new position. If data is only received from 2 or less Beetles, then a random final position is predicted.

With regards to dance move prediction, we have 3 scenarios:

1. Data is received from all 3 Beetles (best case scenario)
2. Data is received from only 2 Beetles or 1 Beetle
3. No data is received from any Beetle (worst case scenario)

In scenario 1, we pick the most commonly predicted dance of the 3 Beetles, whereas in scenario 2, we pick a prediction from any Beetle and in scenario 3, we randomly output a prediction. We designed an algorithm for each scenario so that the case of no prediction is avoided and potential crashes are minimized, improving the stability of the wearable as a result. The methods are shown as follows:

```
# find most frequent element in a list
def most_frequent_prediction(pred_list):
    return max(set(pred_list), key = pred_list.count)
```

Figure 149: Pick most commonly predicted dance in Scenario 1

```
def eval_1beetle(beetle_dict_1, beetle_1):
    # Get beetle data from dictionaries
    beetle1_data = parse_dance_data(beetle_dict_1, beetle_1)
    # Predict dance move of each beetle
    beetle1_dance = predict_beetle(beetle1_data, mlp_dance)
```

Figure 150: Evaluate one Beetle in scenario 2

Finally, the code snippet below describes the overall flow for data collection and prediction:

Code Snippet:

```
ground_truth = [1,2,3]

# BEETLE ADDRESSES
beetle1 = "50:F1:4A:CB:FE:EE"
beetle2 = "1C:BA:8C:1D:30:22"
beetle3 = "78:DB:2F:BF:2C:E2"

ACTIONS = ['muscle', 'weightlifting', 'shoutout']
POSITIONS = ['1 2 3', '3 2 1', '2 3 1', '3 1 2', '1 3 2', '2 1 3']

if beetle1_move_dict[beetle1] and beetle2_move_dict[beetle2] and beetle3_move_dict[beetle3]:
    # Get MOVE data from dictionaries in arguments
    beetle1_move_data = parse_move_data(beetle1_move_dict, beetle1)
    beetle2_move_data = parse_move_data(beetle2_move_dict, beetle2)
    beetle3_move_data = parse_move_data(beetle3_move_dict, beetle3)

    # Normalise MOVE data
    beetle1_move_data_norm = normalise_data(beetle1_move_data)
    beetle2_move_data_norm = normalise_data(beetle2_move_data)
    beetle3_move_data_norm = normalise_data(beetle3_move_data)

    # Predict MOVE DIRECTION of each beetle
    beetle1_move = predict_beetle(beetle1_move_data_norm, mlp_move)
    beetle2_move = predict_beetle(beetle2_move_data_norm, mlp_move)
    beetle3_move = predict_beetle(beetle3_move_data_norm, mlp_move)

    # Find new position
    new_pos = find_new_position(ground_truth, beetle1_move, beetle2_move, beetle3_move)
```

```
else:
    # worst case scenario
    new_pos = random.choice(POSITIONS)

# PREDICT DANCE
if beetle1_dance_dict[beetle1] and beetle2_dance_dict[beetle2] and beetle3_dance_dict[beetle3]:
    # Get DANCE data from dictionaries in arguments
    beetle1_dance_data = parse_dance_data(beetle1_dance_dict, beetle1)
    beetle2_dance_data = parse_dance_data(beetle2_dance_dict, beetle2)
    beetle3_dance_data = parse_dance_data(beetle3_dance_dict, beetle3)
    #print(beetle1_data)

    # Normalise DANCE data
    beetle1_dance_data_norm = normalise_data(beetle1_dance_data)
    beetle2_dance_data_norm = normalise_data(beetle2_dance_data)
    beetle3_dance_data_norm = normalise_data(beetle3_dance_data)
    #print(beetle1_data_norm)

    # Predict DANCE of each beetle
    beetle1_dance = predict_beetle(beetle1_dance_data_norm, mlp_dance)
    beetle2_dance = predict_beetle(beetle2_dance_data_norm, mlp_dance)
    beetle3_dance = predict_beetle(beetle3_dance_data_norm, mlp_dance)
    #print(beetle1_dance)

    dance_predictions = [beetle1_dance, beetle2_dance, beetle3_dance]

    dance = most_frequent_prediction(dance_predictions)

elif beetle2_dict[beetle2] and beetle3_dict[beetle3]:
    dance = eval_1beetle(beetle2_move_dict, beetle2)

elif beetle1_dict[beetle1] and beetle3_dict[beetle3]:
    dance = eval_1beetle(beetle1_move_dict, beetle1)

elif beetle1_dict[beetle1] and beetle2_dict[beetle2]:
    dance = eval_1beetle(beetle1_move_dict, beetle1)

elif beetle1_dict[beetle1]:
    dance = eval_1beetle(beetle1_move_dict, beetle1)
```

```
elif beetle2_dict[beetle2]:
    dance = eval_1beetle(beetle2_move_dict, beetle2)

elif beetle3_dict[beetle3]:
    dance = eval_1beetle(beetle3_move_dict, beetle3)

else:
    # worst case scenario
    dance = random.choice(ACTIONS)
```

Section 7.8 Extensions to machine learning model

Section 7.8.1 Possible Model Improvements

Sliding window segmentation – Moving Average

Tumbling window segmentation was used to segment sensor data, as described in Section 7.1. However, there are better alternatives to segment the data, such as the sliding window approach shown in Figure 151:

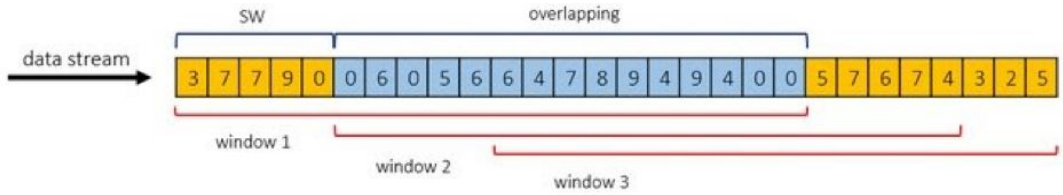


Figure 151: Sliding window-based segmentation (Christian, 2017)

The mean of each window can be extracted to get a moving-average signal, which further smoothens out noise in signal readings and shows long-term trends in the data (Chegg.com, 2020). The amount of overlapping of the windows can be initially set at 50% and determined empirically after training and testing the model.

Using metrics other than accuracy

Accuracy is a good choice of evaluation metric for classification problems which are well balanced. However, it treats every activity as equally important, and is inefficient in the case where there are imbalanced datasets.

Hence, we can use metrics other than accuracy to assess the performance of our model. These metrics include precision, recall and f1 score (Agarwal, 2019).

Precision is the fraction of predicted positives among the true positives.

Precision = True Positives / (True Positives + False Positives)

Precision is looked at when we want to be very sure that the model's prediction is correct. In the case for our project, if we want to be very sure that the model predicts every dance move correctly, we should look at the precision metric.

Recall is the proportion of actual positives that is correctly classified.

Recall = True Positives / (True Positives + False Negatives)

Recall is useful when we want to capture as many positives as possible. For example, we should look at recall if we want our model to recognise a dance move even if it is not too sure.

F1 score is a tradeoff between precision and recall, and this makes it a very useful metric.

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

Figure 152: Formula for F1 score

In many instances, we want to have a model with both good precision and recall, so F1 is a useful metric in that regard. However, it gives equal weight to both precision and recall and there would be instances in which we would want to factor in more recall or more precision when looking at our metric. To do this, we can create a weighted F1 metric by adjusting β as shown in Figure 153:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}.$$

Figure 153: Formula for weighted F1 metric

Use of validation with more data

For our project, we used the holdout method for training and testing the model. However, there are other potentially better methods that can be used (Khandelwal, 2019), which will be detailed as follows.

The holdout method involves splitting the data set into 2 sets: a training set and a test set. The proportion can be chosen as desired. For example, the dataset can be split into 70% for training and 30% for testing, as shown in Figure 154:

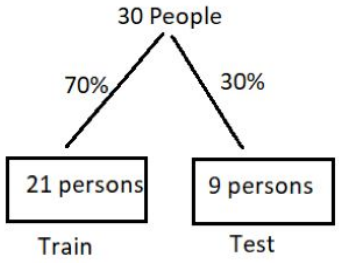


Figure 154: Holdout method – train-test splitting

We opted to use 80% of the dataset for training and 20% for testing. However, it could result in the case where the model makes better predictions on the training data but poorer predictions on the test data. This is caused by sample variability between training and test sets.

The cross validation method eliminates this problem. The data is divided into subsets, the model is trained on a few subsets and the other subsets are used to evaluate the model's performance. To reduce variability, many rounds of cross validation are performed with different subsets from the same data, and the results are combined to come up with an estimate of the model's prediction ability.

One type of cross validation is the Leave One Out Cross Validation (LOOCV) method. One observation is used for testing and the rest of the dataset is used for training the model. In essence, if the dataset contains n observations, then the training set has n-1 observations and the

testing set has 1 observation. This is repeated until every observation has been used as the test set.

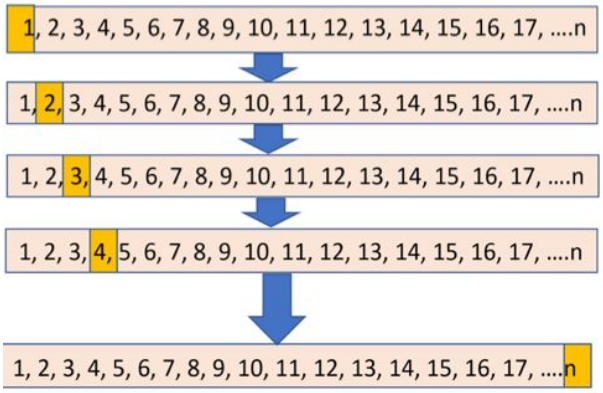


Figure 155: LOOCV method

The strongest benefit of the LOOCV method is that it has very little bias. However, it requires a high computational cost as the model has to be trained n times, n being the number of observations in the dataset. Another point to consider is that if the test observation is an outlier, then the variability of test results is high, which will affect evaluation metrics.

Another type of cross validation is the K-fold cross validation method. The dataset is randomly separated into k groups of about equal size. The first group is used for testing, the model is trained on k-1 groups. This is repeated k times until every group has been used as the test set. This method is similar to LOOCV, except that k = n in the case of LOOCV.

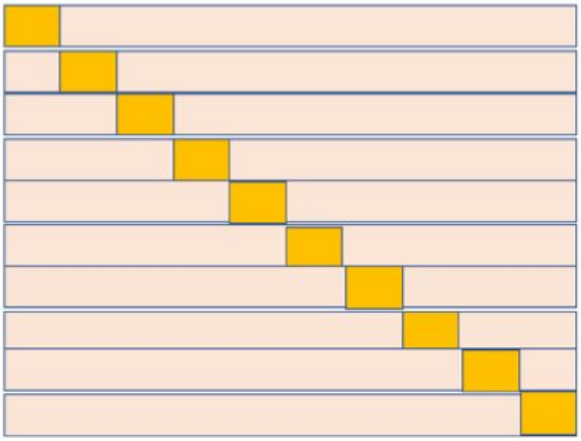


Figure 156: K-fold cross validation method

K-fold cross validation has less computational cost compared to LOOCV as the model has to be only trained k times, where k < n. There is also reduced bias as compared to the holdout method,

and the variance of the test set is reduced compared to LOOCV due to multiple observations in the test set. However, if the value of k is large, then there would be a high computational cost incurred, similar to the case of LOOCV. Hence, it is desirable to pick a value of k which has a good trade off between accuracy and computational cost.

Due to the unfortunate circumstances surrounding the project, we were unable to collect numerous amounts of data for our MLP model. However, with more data, we can use a validation dataset and employ the K-fold cross validation method to validate our model before proceeding to test it.

K-fold cross validation can also be used to find the best hyperparameters for the MLP model. This is done via the use of GridSearchCV provided by scikit-learn which employs stratified K-fold cross validation to find a suitable parameter *alpha,* which is a L2 regularisation term that helps in preventing overfitting of the model by penalising model weights with large magnitudes [2]. A suitable choice of alpha can further increase the model performance.

Use of other methods in position transition prediction

The algorithm proposed in Section 7.7 to determine the new position can be used to detect the final positions of dancers based on the initial positions of the dancers, but there are some obvious limitations.

Firstly, the initial position has to be known in order for the final position to be determined. If the initial position is wrong or is not available, then the final position would not be able to be determined.

Secondly, data has to be collected from every Beetle in order for the algorithm to work. Once there is missing data from one Beetle, then the algorithm would not have enough information to determine the final location.

Lastly, in the case of more than 3 dancers, it would be difficult to develop a similar algorithm to determine the final positions of the dancers from the initial positions. Many scenarios would have to be coded out and this means that the code can be long and susceptible to errors, and it is difficult to adapt the code for different uses.

Thus, other methods can be used to determine the positions of the 3 dancers. One such method is via localisation using LIDAR on each dancer. This uses a point set registration algorithm, where the real-world position of every point is known. To determine the location of each dancer, we

can run the point set registration algorithm to find the best match for the LIDAR scan acquired for each user in the LIDAR mapping process. The algorithm would then return the coordinates of the dancers which can be used in finding dancer position (Liu, 2019).

If the dance coaching is intended to be used indoors, then another method to determine dancer positions is the use of an indoor positioning system. This system can consist of Bluetooth Low Energy (BLE) devices, or Wi-Fi Access Points, which will act as beacons. The beacons can be placed at different points of the room and each beacon would correspond to a dance position (i.e. the dancer at position 1 would have the strongest signal to beacon 1) (Mapwize, 2020). We can determine the position of a dancer by determining the beacon which has the strongest signal strength relative to his or her position.

Section 7.8.2 Applying MLP in spam email detection

MLP has many applications in our everyday lives. One such application of MLP is in spam email detection.

Spam email refers to non-voluntary email messages sent in bulk that aim to deceive the user into making poor choices such as falling for scams and accidentally clicking a link and downloading malware. It is a problem because people eventually fall for them (Hoffman C, 2016).

MLP is one of the machine learning techniques widely adopted to detect and filter out spam emails (Software D, 2017). The objective of MLP in this case would be to be able to accurately predict if an email is spam or not. Supervised learning is employed, meaning that the MLP model would be trained on email data labelled as spam or not spam. Since spam email is sent in bulk and there are many spam emails that exist, the MLP can be fed with huge amounts of data from these bulks of spam emails. These huge amounts of data enable the MLP to learn effectively and thus it is able to accurately predict spam emails, being used widely in spam email filters today. Figure 157 shows the flow of employing MLP in a spam email classification:

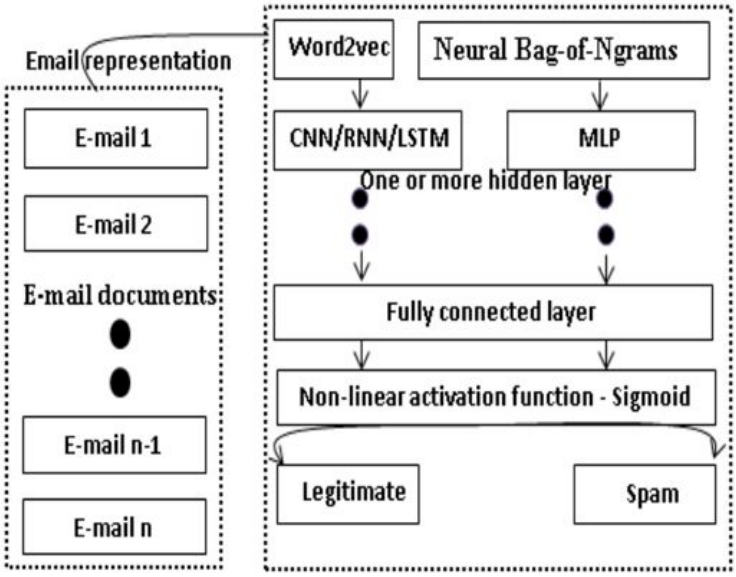


Figure 157: Using MLP in spam email detection (Ganesh, 2019)

Section 8 Software Details: Software 2 - Dashboard and Client-Server

Section 8.1 Architecture Design

This section contains a high-level overview of Software 2's contribution to the project. This includes an architecture diagram and the code that Software 2 had implemented in this project.

Section 8.1.1 Architecture Diagram

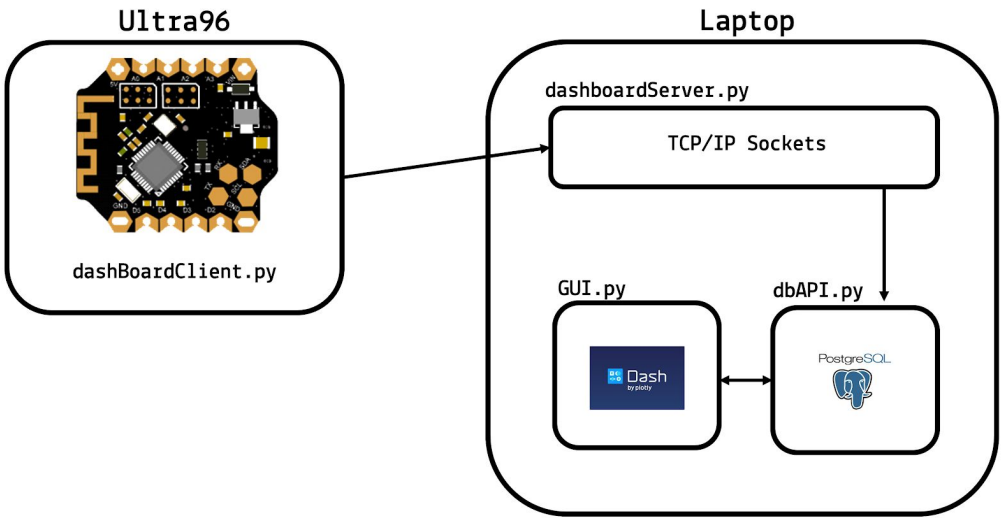


Figure 158: Architecture Diagram of Software

Section 8.1.2 Code Contributions to Project

No	Component	Rationale
1	dashBoardClient.py	Dashboard client to send information to Dashboard server
2	dashboardServer.py	Dashboard server will call functions in dbAPI to store data received by Dashboard Client into Postgresql database
3	GUI.py	GUI constantly queries Postgresql database using dbAPI to display real-time data on the graph
4	dbAPI.py	dbAPI contains an application programming interface to interact with the database

Section 8.2 Front-End: Dashboard Design

This section showcases iterative changes made in the design process throughout the project. This section includes Screenshots of Design Process, Changes from Initial Dashboard Design, Final Dashboard Design, Approach to Real-Time Streaming and Explanation of Front-End Code and Explanation of Performance Enhancing Front-End Code.

Section 8.2.1 Screenshots of Design Process

Figure 159: Screenshot Of Final Dashboard Design

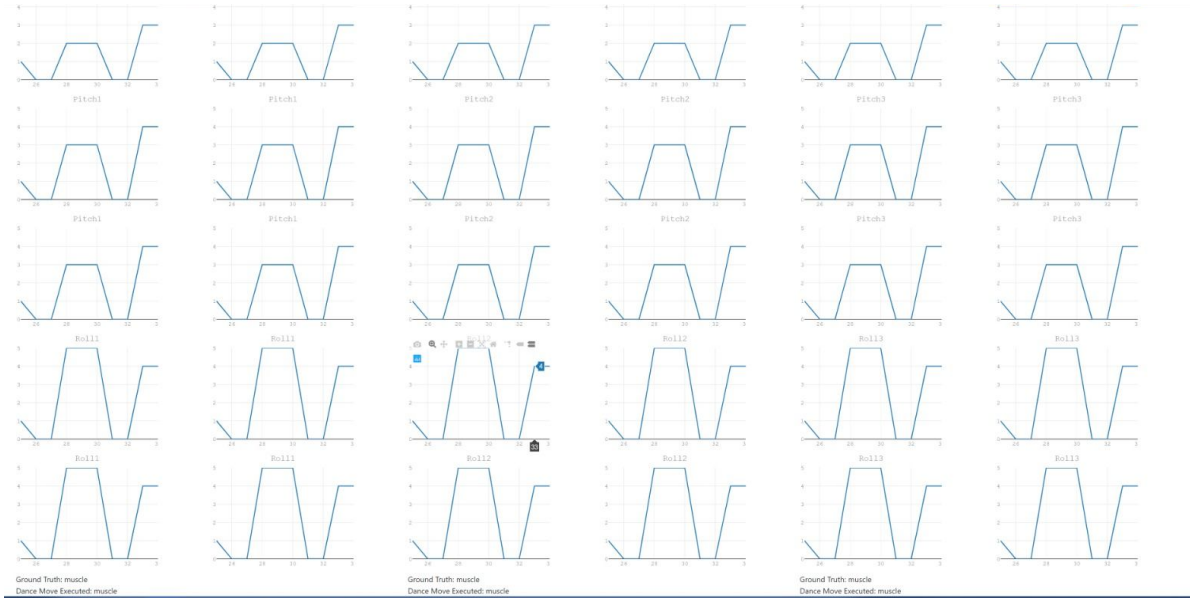


Figure 160: Sample of Initial Dashboard Design

Section 8.2.2 Changes from Initial Dashboard Design

No	Initial Design(Week 7)	Final Design	Rationale for Change
1	Each Beetle's parameters would have a single graph. Hence a graph for Yaw, Pitch, Raw, X-Axis, Y-Axis and Z-Axis. There would be a total of 6 beetles used for the project. Each dancer will use 6 beetles of information.	*Change from 6 Beetles for 3 dancers to 1 Beetle per dancer.* One Beetle responsible for each dancer with IMU and Accelerometer plots: 1) One plot for IMU to display Yaw, Pitch, and Roll 2) One plot for the accelerometer to display the X-axis, Y-axis, and Z-axis. Similarly, for the last Beetle that is connected to the EMG sensor: 1) One plot for Mean Amplitude, RMS Amplitude and Max Amplitude. 2) One plot for Mean Frequency.	For Dash, more graph plots meant more of the computer's resources were used for rendering. There is a limit to the amount of plots that can be rendered on the dashboard . Therefore, to minimize the number of plots and workload on the computer. The final design implementation was considered. Further, a lesser number of plots allowed the dashboard to be viewed on one page with a clean interface. The EMG features were only introduced after week 7 of the project as we decided to start working on it on week 6.
2	Simply just showing the graphs, dance output, and ground truth as dynamic plain text.	User Interface Changes: 1) Adding headers to specify each dancer's labels. 2) Adding colors such as the blinking green box on to the "dance move executed" and graph color lines. 3) Changing font sizes and creating borders.	Front-end design principles for dashboards were considered to make the dashboard more appealing and better user interaction(Minhas, 2019).
3	No EMG output	EMG Output	Following the article provided by the project's hardware designer, EMG analytics was developed to enable users to understand if the dancer were able to dance or were too tired to dance.

Section 8.2.3 Final Dashboard Design

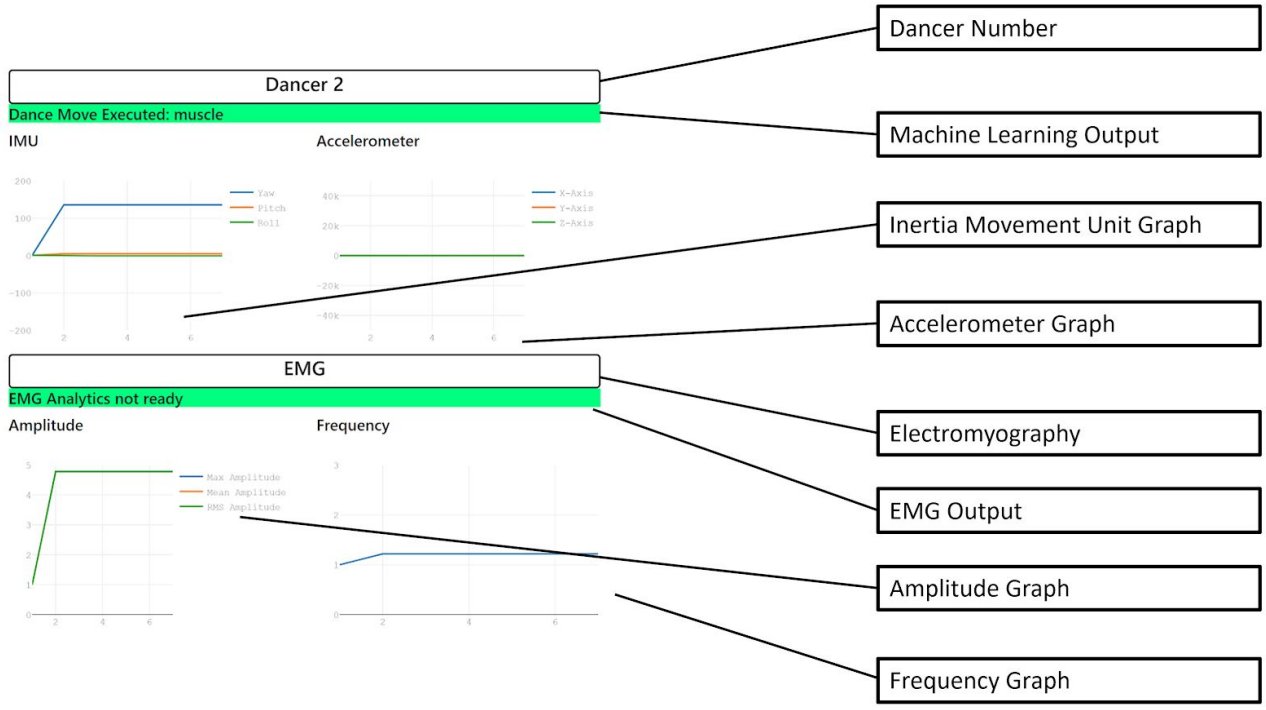


Figure 161: Labelled Diagram of Dashboard

Features:

No	Feature	Explanation
1	Dancer	There are a total of **3 dancers** with the following features: 1) Dancer Number. 2) Machine Learning Output: dance move executed and relative position. 3) Inertial Measurement Unit Graph displays Yaw, Pitch, and Roll. 4) Accelerometer Graph displays X-Axis, Y-Axis, and Z-Axis.
2	EMG	1) Electromyography will be measured from 1 dancer. 2) EMG output displays data analytics on the state of the wearer. 3) Amplitude Graph shows Max Amplitude, Mean Amplitude and RMS Amplitude. 4) Frequency Graph that shows Mean Frequency.

Section 8.2.4 Approach to Real-Time Streaming Data

Dash for Beginners is an article that explains how Dash and Plotly enables real-time streaming of graphs. Dash is built on top of a Flask server, using React.JS, React and Plotly.JS to render the front-end. Throughout this project, Dash proves to be a fast prototyping tool with reliable results. An alternative would be to use a MERN stack. However, from conversations with groups that used a MERN stack, it was time-consuming to learn the technology. Dash provides an out of the box front-end and flask server that has a small learning curve. Also, Plotly was a tool that helped to define graphs easily for visualization and React provided aesthetic visualizations. Further, we had the ability to incorporate bootstrap grid systems to have the layout seen.

Real-time streaming was mainly done through callbacks and is explained by *Szabgab*. More information will be elaborated through **section 8.2.5**. The front-end code will query the database every 1s to get data to animate the graph.

Section 8.2.5 Explanation of Front-End Code

The high-level sequential steps that the application follows according to the code in **GUI.py**:

1) App Renders Display
2) Single Live Update of Component

App renders Display

To simplify understanding, we used bootstrap to implement the 4 sections of 3 Dancers and 1 EMG as such:

Dancer 1	Dancer 2
Dancer 3	EMG

Further, Dash components were used as this would allow us to use aesthetic UI features without the need to code them from scratch. The next page will contain the code for implementing the UI elements for additional reference.

```
app.layout = html.Div([
html.H6("CG4002 Group 6"),

dbc.Row([
    #Dancer 1 graphs
    dbc.Col([
        html.Div(style ={'text-align':'center','border': '2px solid black','border-radius':'5px'}, children = html.H5('Dancer 1')),
        html.Div(style={'font-size':'80%'},children = html.H6(id='Dancer1Output')),
        dbc.Row([
            dbc.Col([html.Div(style ={}, children = html.H6('IMU')),
                     html.Div(children=html.Div(id='graphsD1', className = "row")),
                dcc.Interval(
                    id='graph-update',
                    interval= 1000,
                    n_intervals=0
                    )
                    ]),
            dbc.Col([html.Div(style ={}, children = html.H6('Accelerometer')),
                     html.Div(children=html.Div(id='graphsD2', className = "row"))
             ])
    ]),

        ]),
    #Dancer 2 graph
    dbc.Col([
        html.Div(style ={'text-align':'center','border': '2px solid black','border-radius':'5px'}, children = html.H5('Dancer 2')),
        html.Div(style={'font-size':'80%'},children = html.H6(id='Dancer2Output')),
        dbc.Row([
            dbc.Col([html.H6('IMU'),
                     html.Div(children=html.Div(id='graphsD3', className = "row")),
                     ]),
            dbc.Col([html.H6('Accelerometer'),
                     html.Div(children=html.Div(id='graphsD4', className = "row")),
                     ])
                ]),
        ]),
    ]),

##add the EMG to a new row and colum grid with dancer 3
dbc.Row([

        dbc.Col([
                html.Div(style ={'text-align':'center','border': '2px solid black','border-radius':'5px'}, children = html.H5('Dancer 3
                html.Div(style={'font-size':'80%'},children = html.H6(id='Dancer3Output')),
                dbc.Row([
                    dbc.Col([html.H6('IMU'),
                             html.Div(children=html.Div(id='graphsD5', className = "row")),
                             ]),
                    dbc.Col([html.H6('Accelerometer'),
                             html.Div(children=html.Div(id='graphsD6', className = "row")),
                             ]),
                    ]),
            ]),
        ##EMG Graphs
        dbc.Col([
            html.Div(style ={'text-align':'center','border': '2px solid black','border-radius':'5px'}, children = html.H5('EMG')),
            html.Div(style={'font-size':'80%'},children = html.H6(id='EMGOutput')),
            dbc.Row([
                dbc.Col([html.H6('Amplitude'),
                         html.Div(children=html.Div(id='graphsD7', className = "row")),
                         ]),
                dbc.Col([html.H6('Frequency'),
                         html.Div(children=html.Div(id='graphsD8', className = "row")),
                         ]),
                ]),
            ]),
    ]),


], className="container",style={'width':'98%','margin-left':10,'margin-right':10,'max-width':5000000}
```

Single Live Update of Components

This feature is done by the update_graph(n) function will constantly change these variables every 1 second:

a) Yaw, Pitch, Roll, X-axis, Y-axis, and Z-axis of each dancer
b) EMG Output
c) Machine learning dance output

To simplify understanding, the code is commented with explanations and is displayed in the annex.

Summary of update_graph(n):

1) There is a callback method @app.callback(x,y,z) before the update_graph(n). This callback method will determine what values to return based on their HTML id. Hence, this is the method to update values on static HTML elements.

   One of the parameters is an Interval, which determines the time taken for the function to update. For interval = 1000, means 1000 millisec per update as reflected in the code segment below:

```
dcc.Interval(
    id='graph-update',
    interval= 1000,
    n_intervals=0
    )
```

2) Upon every update, the database will be queried for new data. Below is the code that calls dbAPI.py to query information from the database to take the most recent value from the database to update.

```
lastRow1 = getLastRow("Beetle1")
lastRow2 = getLastRow("Beetle2")
lastRow3 = getLastRow("Beetle3")
lastRow4 = getLastRow("EMG")
MLDancer = getLastRow("MLDancer")
```

3) The queried data is used to update each graph. Every y-axis value(such as Yaw/Pitch/Roll/X-Axis/Y-Axis/Z-Axis) is stored in a deque with length of 10, this is to fit inside the Dash component to render graphs.

```
Yaw1.append(lastRow1[0])
Pitch1.append(lastRow1[1])
Roll1.append(lastRow1[2])
X1.append(lastRow1[3])
Y1.append(lastRow1[4])
Z1.append(lastRow1[5])
Yaw2.append(lastRow2[0])
Pitch2.append(lastRow2[1])
Roll2.append(lastRow2[2])
X2.append(lastRow2[3])
Y2.append(lastRow2[4])
Z2.append(lastRow2[5])
Yaw3.append(lastRow3[0])
Pitch3.append(lastRow3[1])
Roll3.append(lastRow3[2])
X3.append(lastRow3[3])
Y3.append(lastRow3[4])
Z3.append(lastRow3[5])
MaxAmp.append(lastRow4[0])
MeanAmp.append(lastRow4[1])
RMSAmp.append(lastRow4[2])
MeanFreq.append(lastRow4[3])
```

4) After updating the value of each deque to store the y-axis of the graph, we update a deque called X to store the time, x-axis, of all graphs. Next, we prepared the data structures needed for a graph to be real-time. The sample code below explains how the EMG1 plot of Max Amplitude, Mean Amplitude and RMS Amplitude is rendered. Go is a plotly object that allows the graphs to be fit into a scatter plot class and EMG1 is an array to append the graphs MaxAmplitude, MeanAmplitude and RMSAmplitude. This method of formatting all graphs are similarly done for EMG2(Mean Frequency), Dancer 1, Dancer 2 and Dancer 3.

```
X.append(X[-1]+1)

MaxAmplitude = (go.Scatter(
    x=list(X),
    y=list(data_dict['MaxAmp']),
    name='Max Amplitude',
    mode= 'lines'
    ))
MeanAmplitude = (go.Scatter(
    x=list(X),
    y=list(data_dict['MeanAmp']),
    name='Mean Amplitude',
    mode= 'lines'
    ))
RMSAmplitude = (go.Scatter(
    x=list(X),
    y=list(data_dict['RMSAmp']),
    name='RMS Amplitude',
    mode= 'lines'
    ))

#Appends graph to EMG time graph
EMG1.append(dbc.Col(html.Div(dcc.Graph(
            id="EMG1",
            animate=True,
            ##Change the value of 'data' to fig add in MeanAmplitudem RMSAmplitude
            figure={'data': [MaxAmplitude, MeanAmplitude, RMSAmplitude],'layout' : go.Layout(xaxis=dict(range=[min(X),max(X)])
                                                    yaxis=dict(range=[0,5]),#change to 250,250
                                                    margin={'l':30,'r':1,'t':30,'b':30},
                                                    title='{}'.format(''),
                                                    font=dict(family='Courier New, monospace', size=12, color='#7f7f7f'),
                                                    height = 250,
                                                    width = 360
                                                    )}
        ))))
```

5) Finally, the function will return the outputs of each graph(each dancer and EMG), output of Machine learning result, EMG output result and colours of blinking boxes(green blinking machine learning output).

```
return [YPR1,
        XYZ1,
        YPR2,
        XYZ2,
        YPR3,
        XYZ3,
        EMG1,
        EMG2,
        "Dance Move Executed: " + MLDancer1[0],
        "Dance Move Executed: " + MLDancer1[0],
        "Dance Move Executed: " + MLDancer1[0],
        output,
        {'background':'springgreen'} if (n%2) else {'background': 'white'},
        {'background':'springgreen'} if (n%2) else {'background': 'white'},
        {'background':'springgreen'} if (n%2) else {'background': 'white'},
        {'background': EMGColour(n,output)},
        ]
```

The full code of the update function will be copy and pasted as reference if needed.

```
#Function that updates and animates all the graphs and the different changes in colour and text
@app.callback(
    [dash.dependencies.Output('graphsD1','children'),
     dash.dependencies.Output('graphsD2','children'),
     dash.dependencies.Output('graphsD3','children'),
     dash.dependencies.Output('graphsD4','children'),
     dash.dependencies.Output('graphsD5','children'),
     dash.dependencies.Output('graphsD6','children'),
     dash.dependencies.Output('graphsD7','children'),
     dash.dependencies.Output('graphsD8','children'),
     dash.dependencies.Output('Dancer1Output','children'),
     dash.dependencies.Output('Dancer2Output','children'),
     dash.dependencies.Output('Dancer3Output','children'),
     dash.dependencies.Output('EMGOutput','children'),
     dash.dependencies.Output('Dancer1Output','style'),
     dash.dependencies.Output('Dancer2Output','style'),
     dash.dependencies.Output('Dancer3Output','style'),
     dash.dependencies.Output('EMGOutput','style'),
    ],
    [dash.dependencies.Input('graph-update', 'n_intervals')]
    )
def update_graph(n):
    #Stops updates when the graph is intitally loading, this increasing performance
    if n is None:
        raise PreventUpdate

    #To be used to display graph for graphs 1,2 are for dancer 1, graphs 3 & 4 are for dancer 2, graphs 5 & 6 are for dancer 3
    YPR1 = []
    XYZ1 = []
    YPR2 = []
    XYZ2 = []
    YPR3 = []
    XYZ3 = []
    EMG1 = []
    EMG2 = []
    #A list of data names that are needed to update
    data_names = ['Beetle1', 'Beetle2', 'Beetle3']

    #Functions to query the database for the last saved input to PostgreSQL
    lastRow1 = getLastRow("Beetle1")
    lastRow2 = getLastRow("Beetle2")
    lastRow3 = getLastRow("Beetle3")
    lastRow4 = getLastRow("EMG")
    MLDancer = getLastRow("MLDancer")


    #Append new data to deque
    Yaw1.append(lastRow1[0])
    Pitch1.append(lastRow1[1])
    Roll1.append(lastRow1[2])
    X1.append(lastRow1[3])
    Y1.append(lastRow1[4])
    Z1.append(lastRow1[5])
    Yaw2.append(lastRow2[0])
    Pitch2.append(lastRow2[1])
    Roll2.append(lastRow2[2])
    X2.append(lastRow2[3])
    Y2.append(lastRow2[4])
    Z2.append(lastRow2[5])
    Yaw3.append(lastRow3[0])
    Pitch3.append(lastRow3[1])
    Roll3.append(lastRow3[2])
    X3.append(lastRow3[3])
    Y3.append(lastRow3[4])
    Z3.append(lastRow3[5])
    MaxAmp.append(lastRow4[0])
    MeanAmp.append(lastRow4[1])
    RMSAmp.append(lastRow4[2])
    MeanFreq.append(lastRow4[3])
```

```
#Determines the Analytics needed
output = EMGAnalytics(MaxAmp,MeanFreq)


#Increase the value of the X-axis by 1 for all graphs
X.append(X[-1]+1)

MaxAmplitude = (go.Scatter(
    x=list(X),
    y=list(data_dict['MaxAmp']),
    name='Max Amplitude',
    mode= 'lines'
    ))
MeanAmplitude = (go.Scatter(
    x=list(X),
    y=list(data_dict['MeanAmp']),
    name='Mean Amplitude',
    mode= 'lines'
    ))
RMSAmplitude = (go.Scatter(
    x=list(X),
    y=list(data_dict['RMSAmp']),
    name='RMS Amplitude',
    mode= 'lines'
    ))

#Appends graph to EMG time graph
EMG1.append(dbc.Col(html.Div(dcc.Graph(
            id="EMG1",
            animate=True,
            ##Change the value of 'data' to fig add in MeanAmplitudem RMSAmplitude
            figure={'data': [MaxAmplitude, MeanAmplitude, RMSAmplitude],'layout' : go.Layout(xaxis=dict(range=[min(X),max(X)]
                                                    yaxis=dict(range=[0,5]),#change to 250,250
                                                    margin={'l':30,'r':1,'t':30,'b':30},
                                                    title='{}'.format(''),
                                                    font=dict(family='Courier New, monospace', size=12, color='#7f7f7f'),
                                                    height = 250,
                                                    width = 360
                                                    )}
        ))))

MeanFrequency = (go.Scatter(
x=list(X),
y=list(data_dict['MeanFreq']),
```

```
name='Mean Frequency',
mode= 'lines'
))

#Appends graph to EMG frquency graph
EMG2.append(dbc.Col(html.Div(dcc.Graph(
        id="EMG2",
        animate=True,
        ##Change the value of 'data' to fig
        figure={'data': [MeanFrequency],'layout' : go.Layout(xaxis=dict(range=[min(X),max(X)]),
                                                yaxis=dict(range=[0,2]),#change to 1.0 - 2.0
                                                margin={'l':30,'r':1,'t':30,'b':30},
                                                title='{}'.format(''),
                                                font=dict(family='Courier New, monospace', size=12, color='#7f7f7f'),
                                                height = 250,
                                                width = 360
                                                )}
    ))))
```

```
#Iterate through all the data to append to the relevant deque that stores Yaw, pitch, roll, x, y , z of beetles
for data_name in data_names:
    dancerNo =  data_name[-1]
    YAW = (go.Scatter(
    x=list(X),
    y=list(data_dict['Yaw' + dancerNo]),
    name='Yaw',
    mode= 'lines'
    ))
    PITCH = (go.Scatter(
    x=list(X),
    y=list(data_dict['Pitch' + dancerNo]),
    name='Pitch',
    mode= 'lines'
    ))
    ROLL = (go.Scatter(
    x=list(X),
    y=list(data_dict['Roll'+dancerNo]),
    name='Roll',
    mode= 'lines'
    ))
    XAxis = (go.Scatter(
    x=list(X),
    y=list(data_dict['X'+dancerNo]),
    name='X-Axis',
    mode= 'lines'
    ))
    YAxis = (go.Scatter(
    x=list(X),
    y=list(data_dict['Y'+dancerNo]),
    name='Y-Axis',
    mode= 'lines'
    ))
    ZAxis = (go.Scatter(
    x=list(X),
    y=list(data_dict['Z'+dancerNo]),
    name='Z-Axis',
    mode= 'lines'
    ))

    YPR = eval("YPR" + dancerNo)
    XYZ = eval("XYZ" + dancerNo)

    YPR.append(dbc.Col(html.Div(dcc.Graph(
            id=data_name,
            animate=True,
            ##Change the value of 'data' to fig
            figure={'data': [YAW,PITCH,ROLL],'layout' : go.Layout(xaxis=dict(range=[min(X),max(X)]),
                                                  yaxis=dict(range=[-200,200]),#change to 250,250
                                                  margin={'l':30,'r':1,'t':30,'b':30},
                                                  title='{}'.format(''),
                                                  font=dict(family='Courier New, monospace', size=12, color='#7f7f7f'),
                                                  height = 250,
                                                  width = 360
                                                  )}
        ))))
    XYZ.append(dbc.Col(html.Div(dcc.Graph(
            id=data_name,
            animate=True,
            ##Change the value of 'data' to fig
            figure={'data': [XAxis,YAxis,ZAxis],'layout' : go.Layout(xaxis=dict(range=[min(X),max(X)]),
                                                  yaxis=dict(range=[-30000,30000]),# <X<
                                                  margin={'l':30,'r':1,'t':30,'b':30},
                                                  title='{}'.format(''),
                                                  font=dict(family='Courier New, monospace', size=12, color='#7f7f7f'),
                                                  height = 250,
                                                  width = 360
                                                  )}
        ))))



return [YPR1,
        XYZ1,
        YPR2,
        XYZ2,
        YPR3,
        XYZ3,
        EMG1,
        EMG2,
        "Dance Move Executed: " + MLDancer1[0],
        "Dance Move Executed: " + MLDancer1[0],
        "Dance Move Executed: " + MLDancer1[0],
        output,
        {'background':'springgreen'} if (n%2) else {'background': 'white'},
        {'background':'springgreen'} if (n%2) else {'background': 'white'},
        {'background':'springgreen'} if (n%2) else {'background': 'white'},
        {'background': EMGColour(n,output)},
        ]
```

Section 8.2.6 Explanation of Performance Enhancing Front-End Code

In this section, there will be explanations on important front-end code that affected the performance or design in a major way. These code snippets are in **GUI.py** that stores the front-end of the project:

1) Improving the performance of CPU
2) Bootstrap Components to improve UI design

Improving the performance of CPU

1) Rewriting the app launch function

The solution was found from a plotly forum that discussed the current faults with the dash app loading function and how there can be a workaround for this to reduce CPU(Me, 2018) .

```
def main():
    init_layout(1_000)
    app.run_server(host='127.0.0.1')
```

Above is a code snippet, which has an initial layout before the app runs. The typical method would be to run the server then run the application. The improved performance allows the GUI layer to be loaded first and then render the layer on the server. This "hack" increases CPU drastically. Previously, we were only able to render 8 graphs that updated every 1 second. However, from this method, we were able to render about 27 graphs that live every 1 second. A downside was that the application did not reload it's UI components every time there was a code change in the UI. But, this project had little need for UI and this did not hinder the developing process much.

2) **Preventing live updates from occurring before the app is fully loaded**

```
def update_graph(n):
    #Stops updates when the graph is intitally loading, this increasing performance
    if n is None:
        raise PreventUpdate
```

The code above was constructed to prevent any exception errors that cause poor performance. Currently, Dash apps are made to just load their live updates when the app is launched. This is a problem for our project as there were a lot of UI components that had to load before live updates. Hence, this workaround prevented many errors and the app loaded quicker.

Section 8.3 Back-End Design

This section will cover the implementation of the back-end design of this project in the sections of Choice of Database, API Implementation and Client-Server Implementation.

Section 8.3.1 Choice of Database

MySQL vs PostgreSQL(Hristozov, 2019)
MySQL and PostgreSQL are very similar in performance. Hence, either would be good choices as they are compatible with Dash.

Benefits of PostgreSQL over MySQL:

1) PostgreSQL adheres to SQL standards more compared to MySQL.
2) PostgreSQL has more data types/files available. Since choosing a database was in the initial phases of the project, choosing a database that had more flexibility in the event we need other data types/files.
3) PostgreSQL has Multiversion Concurrency Control which allows multi-query plans from multiple sources. This would allow the server script to save data points to the database and the GUI script from querying data points to be displayed.

Benefits of MySQL over PostgreSQL:

1) MySQL is a more popular language. It would be easier to look online for a solution from forums as well as look for 3rd party extensions.
2) MySQL read speed is faster than PostgreSQL for a lot of read queries. But, for such a project performance may be negligible as there are a maximum of 5 queries to the database per second.

After considering the benefits of each language and discussion with team members, **PostgreSQL** was decided to be part of our project.

Section 8.3.2 API Implementation

Database API (dpAPI.py)
The script contains functions to connect to the database, CRUD of Tables and CRUD of rows of a Table. The database API was created to interact with the Postgres database to allow code reuse when the dashboard server saves data points into the database and when the GUI queries the database for update data points. The articles *8 advantages of API developers(BBVAOpen4U, 2018)*, explains how having APIs can be beneficial. The most beneficial part of having an API would be to allow seamless integration of different components of Software 2 together by using the API instead of copying the same code into every script. The next page will be the code.

85 lines (70 sloc) | 2.62 KB

Raw Blame

```python
#Developed by Gerald Chua Deng Xiang
import psycopg2 #import for DB

#Create the connection
connection = psycopg2.connect(user = "postgres",
                                  password = "Cg4002",
                                  host = "localhost",
                                  port = "5432",
                                  database = "postgres")
cursor = connection.cursor()

#Functions for CRUD

#Function to add a row
def addValue(tableName, *data):
    dataList = list(data) #convert multiple arguments into a list
    query = "INSERT INTO " + tableName + " VALUES ("+ str(dataList).strip('[]') +")" #inserts value into the table
    #query += "RETURNING " + str(dataList).strip('[]') #stores the last value that is being added
    cursor.execute(query)
    connection.commit()
    count = cursor.rowcount
    #lastRow = cursor.fetchone()#contains last row being stored
    print(count, "Value saved into " + tableName)
    print(query)
    #return lastRow

def addML(table, data):
    print(data)
    query = "INSERT INTO " + table + " VALUES ("+ data +")" #inserts value into the table
    #query += "RETURNING " + str(dataList).strip('[]') #stores the last value that is being added
    cursor.execute(query)
    connection.commit()
    count = cursor.rowcount
    #lastRow = cursor.fetchone()#contains last row being stored
    print(count, "Value saved into " + table)
    print(query)


#Function to show all rows in Table
#Parameter: tableName - to specify the table name to be shown
def showTable(tableName):
    query = "SELECT * from " + tableName
    cursor.execute(query)
    print("Queried table from: " + tableName)
    table = cursor.fetchall()
    print(table)

#Function to get last row in table
def getLastRow(tableName):
    query = "SELECT * from " + tableName
    cursor.execute(query)
    #print("Queried table from: " + tableName)
    table = cursor.fetchall()
    return table[-1]


#function to clear table
def clearTable(tableName):
    query = "TRUNCATE " + tableName + "; DELETE FROM " + tableName +";"
    cursor.execute(query)
    connection.commit()

#function to create a new table
def createTable(tableName, *columnNames):
    columns = list(columnNames)
    query = "CREATE TABLE " + tableName + " ("
    first = True
    for column in columns:
        if first:
            query += "\n" + column + " NUMERIC NOT NULL"
            first = False
        else:
            query += ", \n" + column + " NUMERIC NOT NULL"
    query += ");"
    cursor.execute(query)
    connection.commit()
    print("Table created successfully in PostgreSQL ")
```

Section 8.3.3 Client-Server Implementation

Storing of incoming information is done through the dashboard client-server implementation. The Client will be on the Ultra96 to send data to the dashboard Server which will be on Software 2's laptop. The server will immediately store data into the tables of the local database of Software 2's laptop. Detailed explanations can be found in the communication external section of the report and detailed code can be found in the dashboard.zip file.

Dashboard Client (dashBoardClient.py)

The Client implementation is developed similar to External Communication in Section 6.3 of the report.

The Client runs code in the sequential order as such on the **Ultra96's script "final.py"**:

1) During the setup phase of the combined.py, a dashboard client object will be created. This dashboard client is imported by dashBoardClient.py.
2) The client will create its own TCP/IP and bind to the IP address and port of the dashboard server. The client has the secret key "cg40024002group6" and will connect to the dashboard server.
3) After a connection is established, whenever the Ultra96 sends data, it will add padding to ensure the message length is made up of blocks of 16 bytes and encrypt the message according to AES and send out to the server.
4) Code Explanation of sending Data:

```
def send_data_to_DB(self, Beetle, data):
    #dataList = list(data.values())
    encrypted_text = self.encrypt_message_DB(Beetle, data)
    print("encrypted_text: ", encrypted_text)
    sent_message = encrypted_text
    print("[Dashboard Client] sent_message length: ", len(sent_message))
    self.socket.sendall(sent_message)
```

The Ultra96 will call this function from it's instantiated object. The client will send a string containing important data and the Beetle's address. The next page will display the dashboard client class code.

```
BLOCK_SIZE = 16
PADDING = ' '
testDict = {}


class Client():
    def __init__(self, ip_addr, port_num, group_id, secret_key):
        super(Client, self).__init__()

        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_address = (ip_addr, port_num)
        self.secret_key = secret_key
        self.socket.connect(server_address)
        # self.timeout = 60
        print("[Dashboard Client] client is connected!")

    def add_padding(self, plain_text):
        pad = lambda s: s + (BLOCK_SIZE - (len(s) % BLOCK_SIZE)) * PADDING
        padded_plain_text = pad(plain_text)
        print("[Dashboard Client] padded_plain_text length: ", len(padded_plain_text))
        return padded_plain_text

    def send_data_to_DB(self, table, data):

        #dataList = list(data.values())
        encrypted_text = self.encrypt_message_DB(table, data)
        print("[Dashboard Client] encrypted_text: ", encrypted_text)
        sent_message = encrypted_text
        print("[Dashboard Client] sent_message length: ", len(sent_message))
        self.socket.sendall(sent_message)

    def encrypt_message_DB(self, table, dataDict):
        plain_text = "#" + table + " | " + str(dataDict)
        print("[Dashboard Client] plain_text: ", plain_text)
        padded_plain_text = self.add_padding(plain_text)
        iv = Random.new().read(AES.block_size)
        aes_key = bytes(str(self.secret_key), encoding="utf8")
        cipher = AES.new(aes_key, AES.MODE_CBC, iv)
        encrypted_text = base64.b64encode(iv + cipher.encrypt(bytes(padded_plain_text, "utf8")))
        print(type(encrypted_text))
        return encrypted_text

    def stop(self):
        self.connection.close()
        self.shutdown.set()
        self.timer.cancel()
```

Dashboard Server (dashboardServer.py)

The script is modified from the **eval_server.py** to connect with the dashboard client to set up a connection through TCP/IP sockets. The process of receiving data follows in sequential order when the script is executed:

1) In Command Prompt, "python dashboardServer.py <IP Address>" is executed
2) The script will call a Server class to create a TCP/IP socket and bind to the default port number "8080".
3) The socket will listen and wait for a connection.
4) When a connection is received, a secret key has to be input to ensure that the client connection is secure.
5) Code Explanation on How Data is Received

```
def run(self):
    while not self.shutdown.is_set():
        data = self.connection.recv(1024)
        if data:
```

Inside the run(self) function is the code for data being received, decrypted and stored into the database through the database API. Advanced Encryption Standard (AES) is used for data decryption when receiving data from the client. More information can be found in the communications section of the report.

The code below is an example of how data points are stored. The server will receive a string. This string will contain the address of the Beetle and data points. For the example below, the Beetle is Beetle 1, and the data points sent are in the format of an array but as a string data type. eval(splitStr[1]) will convert the string into an array to be stored into the database using addValue. There are many elif statements that enable different Beetles to send information to be stored in the database.

```
if(splitStr[0] == "50:F1:4A:CB:FE:EE"):
    table = "Beetle1"
    dataPoint = eval(splitStr[1])
    addValue(table, dataPoint[1], dataPoint[2],dataPoint[3],dataPoint[4],dataPoint[5],dataPoint[6])
```

The full code is displayed in the next page for further clarification.

```
from dbAPI import showTable,addValue,addML #import function from dbAPI.py

MESSAGE_SIZE = 3 # position, 1 action, sync
class Server(threading.Thread):
    def __init__(self, ip_addr, port_num, group_id):
        super(Server, self).__init__()

        # Create a TCP/IP socket and bind to port
        self.shutdown = threading.Event()
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_address = (ip_addr, port_num)
        #server_address = (ip_addr, port_num)
        print('starting up on %s port %s' % server_address, file=sys.stderr)
        self.socket.bind(server_address)
         # Listen for incoming connections
        self.socket.listen(1)
        #set up a connection
        self.client_address, self.secret_key = self.setup_connection()

    def setup_connection(self):
        # Wait for a connection
        print('waiting for a connection', file=sys.stderr)
        self.connection, client_address = self.socket.accept()
        print("Enter the secret key: ")
        secret_key = sys.stdin.readline().strip()
        print('connection from', client_address, file=sys.stderr)
        if len(secret_key) == 16 or len(secret_key) == 24 or len(secret_key) == 32:
            pass
        else:
            print("AES key must be either 16, 24, or 32 bytes long")
            self.stop()
        return client_address, secret_key # forgot to return the secret key

    #Function that runs the receiving of messages
        def run(self):
            while not self.shutdown.is_set():
                data = self.connection.recv(1024)
                if data:
                    try:
                        message = self.decrypt_message(data)
                        #print(message)
                        if(message != ""):
                            splitStr = message.split(' | ')
                            print(splitStr)
                            try:
                                table = ''
                                #Position 1
                                if(splitStr[0] == "50:F1:4A:CB:FE:EE"):
                                    table = "Beetle1"
                                    dataPoint = eval(splitStr[1])
                                    addValue(table, dataPoint[1], dataPoint[2],dataPoint[3],dataPoint[4],dataPoint[5],dataPoint

                                #Position 2
                                elif(splitStr[0] == "1C:BA:8C:1D:30:22"):
                                    dataPoint = eval(splitStr[1])
                                    table = "Beetle2"
                                    addValue(table, dataPoint[1], dataPoint[2],dataPoint[3],dataPoint[4],dataPoint[5],dataPoint
                                #Position 3
                                elif(splitStr[0] == "78:DB:2F:BF:2C:E2"):
                                    dataPoint = eval(splitStr[1])
                                    table = "Beetle3"
                                    addValue(table, dataPoint[1], dataPoint[2],dataPoint[3],dataPoint[4],dataPoint[5],dataPoint
                                #Emg Data
                                elif(splitStr[0] == "50:F1:4A:CC:01:C4"):
                                    print("received: " + splitStr[0])
                                    dataPoint = eval(splitStr[1])
                                    print("eval: " + str(dataPoint))
                                    table = "EMG"
                                    addValue(table, dataPoint[0], dataPoint[1],dataPoint[2],dataPoint[3])
                                #Machine Learning Output
                                elif(splitStr[0] == "MLDancer1"):
                                    print(splitStr)
                                    addValue("MLDancer1", splitStr[1])
                                    print("completes function")
                            except Exception as e:
                                print("Error:" + splitStr + "Error Message: " +  str(e))
                    except Exception as e:
                        print("Data failure from: " + str(e) )
                else:
                    print('no more data from', self.client_address, file=sys.stderr)
                    self.stop()

    def stop(self):
        self.connection.close()
        self.shutdown.set()
        self.timer.cancel()

    def decrypt_message(self, cipher_text):
        decoded_message = base64.b64decode(cipher_text)
        iv = decoded_message[:16]
        secret_key = bytes(str(self.secret_key), encoding="iso-8859-1")
        cipher = AES.new(secret_key, AES.MODE_CBC, iv)
        decrypted_message = cipher.decrypt(decoded_message[16:]).strip()
        decrypted_message = decrypted_message.decode('iso-8859-1')
        decrypted_message = decrypted_message[decrypted_message.find('#'):]
        decrypted_message = bytes(decrypted_message[1:], 'iso-8859-1').decode('iso-8859-1')
        return decrypted_message
def main():
    ip_addr = sys.argv[1]
    my_server = Server(ip_addr, 8080, 6)
    my_server.start()

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

Database Tables Implementation

For the project, we are using 4 Beetles. 3 Beetles will be connected to an IMU and an accelerometer. The last Beetle is connected to the EMG Sensor.

Below is the list of tables in the Database:

Table Name	Data Columns
Beetle1	Yaw, Pitch, Roll, X-Axis, Y-Axis, Z-Axis
Beetle2	Yaw, Pitch, Roll, X-Axis, Y-Axis, Z-Axis
Beetle3	Yaw, Pitch, Roll, X-Axis, Y-Axis, Z-Axis
EMG	MeanAmplitude, RMSAmplitude, MaxAmplitude, MeanFrequency
MLResult	DanceMove

Section 8.4 Data Analytics

Data Analytics of EMG

The section explains the collaboration between the hardware personale and Software 2 on creating data analytics for EMG output.

From the hardware personale, we needed an algorithm to determine whether the dancer wearing the EMG is fatigued and should stop dancing. From this article, it explains when the values of Mean Amplitude, RMS Amplitude and Max Amplitude start increasing gradually and Mean Frequency starts decreasing gradually, muscle fatigue is imminent(Toro, et al., 2019). The solution was to use a function that detects if a series of points follows a decreasing or increasing polynomial function. The resource of how this is implemented can be found in the StackOverflow discussion on *How can I detect if a trend is increasing or decreasing in a time series*(Singhal, 2019). From using the numpy polyfit function, we could estimate the most recent 5 points of the RMS Amplitude, Mean Amplitude, Max Amplitude and Mean Amplitude graphs to determine if it is an increasing or decreasing function. When the amplitude graphs are increasing and the mean frequency graph is decreasing, then the algorithm will detect fatigue and output fatigue in the live graph.

Code in GUI.py for Data Analytics of EMG

1) determineTrend() - determines if a trend is increasing or decreasing .
2) EMGAnalytics() - uses determineTrend() to determine if the dancer is fatigued.

```
def determineTrend(data):
    index = [1,2,3,4,5]
    coeffs = np.polyfit(index,list(data), 1)
    slope = coeffs[-2]
    trendVal = float(slope)
    trend = ''
    if(trendVal>0):
        trend = "increasing"
    elif(trendVal<0):
        trend = "decreasing"
    print(float(slope))
    return trend

#Determines when frequency is decreasing and time is increasing that the dancer wearing the EMG is fatigued
def EMGAnalytics(MaxAmp,MeanAmp,RMSAmp,MeanFreq):
    #Taking the last 5 datapoints
    try:
        MaxAmpData = (list(MaxAmp))[-5:]
        MaxAmpTrend = determineTrend(MaxAmpData)
        MeanAmpData = (list(MeanAmp))[-5:]
        MeanAmpTrend = determineTrend(MeanAmpData)
        RMSAmpData = (list(RMSAmp))[-5:]
        RMSAmpTrend = determineTrend(RMSAmpData)
        FreqData = (list(MeanFreq))[-5:]
        FreqTrend = determineTrend(FreqData)
        if(FreqTrend == "decreasing" and MaxAmpTrend == "increasing" and MeanAmpTrend == "increasing" and RMSAmpTrend == 'increasing' ):
            return "Fatigued"#fatigue
        else:
            return "No issues present"#??Ready to dance?
    except:
        return("EMG Analytics processing")
```

Section 8.5 Software 2 Project Extensions

In this section, we attempted at researching extensions that could have been possible if we were given more time in developing the dashboard:

1) Integrating the full stack application with the dashboard server
2) Dashboard Server Reconnection
3) Using an existing dashboard design pattern to develop a dashboard
4) Following UI/UX Design Life Cycle to design a system

Integrating the full stack application with the dashboard server

Currently, we need to run 2 scripts side by side, the dashboard server and GUI separately. In the final implementation, we could have considered combining these two scripts together to have one script to launch both dashboard server and GUI. This would give a better user experience and simply the need to have two terminals running at the same time.

Dashboard Server Reconnection

The current process of setting up our prototype was tedious. We had to run the dashboard server script and Ultra96.py script at the same time. When an exception occurred, we had to restart the whole system as the dashboard and client servers would not prompt for a reconnection. For the client and server set up, we could have added code to allow reconnections. This would reduce the need for manual reconnections and allow us to spend time debugging other aspects of the project.

Using an existing dashboard design pattern to develop a dashboard

If given more time, we would have followed the principles of design for dashboards(Durcevic, 2019) and consider design templates for dashboards(Rokr, 2020). This would enhance the dashboard aesthetics and produce a more professional dashboard.

Following UI/UX Design Life Cycle to design a system

There are many stages that are done in the UI/UX Design process when creating a product. The article, *7 UX Deliverables: What will I be making as a UX designer?* (Komninos, 2020), explains how UI/UX professionals ideate and create a product that is most suited for users. These ideas were explained in CS3240, Interaction Design, in NUS, and were the best methods to develop a product that was the best fit for the user. Using such a process, even though it is time consuming will allow the best product to be developed and can be applied to creating the dashboard. However, being the first batch of CG4002, the process of ideation and contextual inquiry can be done on past capstone projects to understand what is the best dashboardashboard that can be created.

Section 9 Societal and Ethical Impact

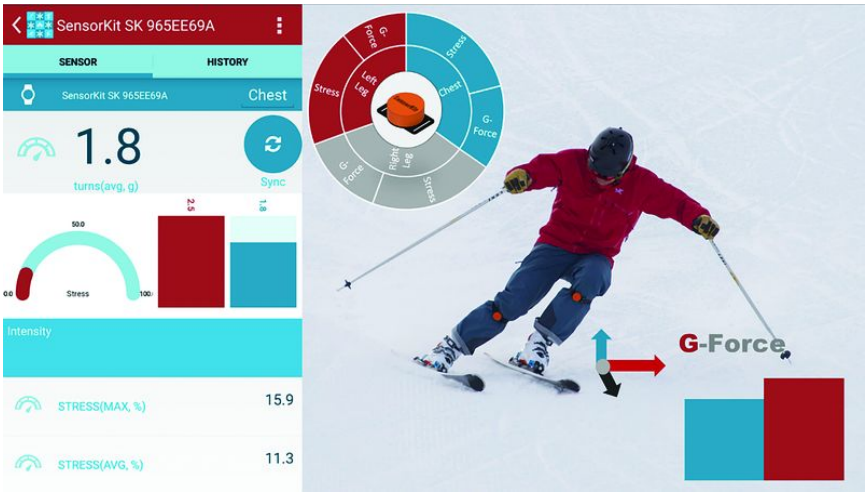
This section will weigh the pros and cons that surround the societal and Ethical impact of human activity detection. The section will be supported by relevant research papers that are discussing this topic.

Applications of Human Activity Detection:

1) Performance of Athletes
2) Early Intervention of Elderly in Emergencies
3) Hand Gesture Recognition to Operate Machinery

Performance of Athletes

The ideas developed from this subsection are from the article *Sensors in Sports: Analyzing Human Movement with AI* (Kexugit, 2018) . The article explains how data can be collected from a sensor kit. Sensors will be attached to athletes to gather data with data analytics that will determine the movements of athletes and G-force or stress that an athlete can take. They used this kit on a skier to classify movement and to calculate the force that can be stressed on a person's body.



Picture from the article

The goal of this article is to classify movement by skiers. This data can be used to develop a system that can teach beginners the right movements or help advance skiers fine-tune their craft. This system can be inducted to take in different sports as well. Therefore, with such a system,

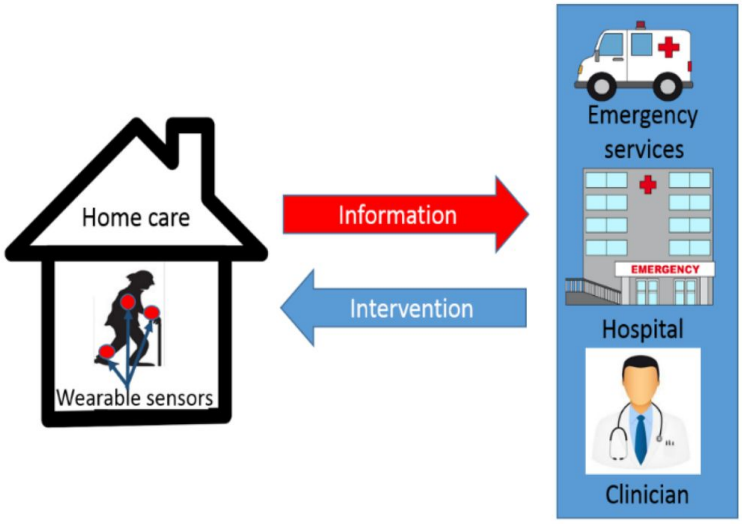
people will be to create commercial products to teach humans sports in a way that mimics high-performance athletes.

Societal and Ethical Impacts

Pros	Cons
Users need not constantly rely on coaches to teach them a sport. They can use a system to tell them the right or wrong movements. This may bring more jobs for engineers to develop such systems.	This system will take away jobs from coaches.
This system can be extended to movements as well. For example, if a person is lifting objects "wrongly", they will be more prone to injury. Such a system can be a method to teach people how to use their move properly to prevent injuries. It may also help people that are in rehabilitation to learn movements again.	High-performance athletes may not be the best people to mimic. Their moves may be specific to their body type and cannot be generalized to all humans.

Early Intervention of Elderly in Emergencies

The article, *Physical Human Activity Recognition Using Wearable Sensors (Attal, et al., 2015)*, is on an experiment to provide early intervention when an elderly is facing an emergency. Sensors will be attached to the elderly for their daily activities to be tracked in real-time so that help professionals may be able to predict unexpected medical incidents and provide medical attention. They use IMUs, camera imaging, and other hardware to determine the activity that the elderly is doing. An example of the classification of activities is sitting down, lying down, walking or standing. Therefore, when a classification such as falling is detected, healthcare professionals are able to detect these issues.



Picture from the Article

Societal and Ethical Impacts

Pros	Cons
Early Intervention will enable the elderly to get medical attention before it reaches a critical condition. This will definitely save a lot of lives as the current solution would be to have a caregiver to monitor the elderly.	This may be an invasion of privacy as people can be tracked constantly or do not want their data to be used for experiments.
This solution will also benefit patients with disabilities or high risk of ailments. From the article, *Robust Human Activity and Sensor Location Corecognition via Sparse Signal Representation (Xu, Zhang, Sawchuk, & Sarrafzadeh, 2012)*, it mentions that human activity detection can determine the high risks of obesity, cerebral palsy or autism.	Since different individuals will have variances in their movement trajectories and mannerisms, sufficient datasets need to be collected from a large sample of patients in order for detection to be accurate in patients across various age groups and body physiques.

Hand Gesture Recognition to Operate Machinery

The article, *A real-time gesture recognition system using near-infrared imagery(Mantecón, Del-Blanco, Jaureguizar, & García, 2019)*, mentions how hand gestures can be used to operate Unmanned Aerial Vehicles(UAV) or other types of real-time operating systems. Currently, there is a company called *Ultraleap,* that has developed a system called a leap motion device that is used in Kinect game consoles to detect movement. Other than predicting gestures as part of a

game mechanism, Ultraleap aims to be able to use machine learning to train models on hand gestures, in hope of developing a system that can adequately recognize sign language.

Hence, future technologies will be good enough to recognize hand gestures that can be used to instantly translate sign language, or create systems such as a virtual keyboard.

Societal and Ethical Impacts

Pros	Cons
A technology breakthrough would result in better imaging technology to recognize tasks that would be faster to perform by gestures instead of physically touching the object.	There may be some outliers that have different bone structures that may be able to participate in such technology,
Games would be better developed to create Virtual Reality that is more realistic than a current set up. Similar to the fictional world of *Ready Player One,* as everyone would have a virtual avatar and able to control their virtual self similar to in reality,	The games need to take into account the immediate physical surroundings of the player, so that they do not end up walking into obstacles.

Concluding Thoughts

From the research articles and pondering of the future possibilities of human activity detection, there is a world of opportunity for technology to benefit humanity. However, there will always be unethical individuals that will want to exploit current technology for malicious gain. The pros heavily outweigh the cons for our research of this technology and the world is moving to a place where Big Data and IoT will be used to make our lives easier. Having real-time operating systems along with Artificial Intelligence may be the step into the future to test the boundaries of Virtual Reality for us all. However, the exploitation of personal data will be an alarming problem that professions developing this technology will always have to face.

However, to prevent the misuse of personal data, it is possible to anonymize the data collected. All data collected must remove all traces of attribution to individuals. Also, data collected would need to be with the consent of individuals. There could be similar legislation to the Personal Data Protection Act that would ensure that data collected would not need to have important "fingerprints" that are associated with individuals.

Lastly, backdoors are common methods that have been implemented in code so that manufacturers are easily able to access systems without the need for authentication of passwords. Ethical impacts regarding such practices can result in data being stolen from users. Hence, there

needs to be legislation or ethical principles inculcated to developers to prevent privacy infringement.

Section 10 Project Management

Section 10.1 Team Structure

For this team project, we adopted egoless team structure. For each week, each of us takes turns to be the team leader. Team members will post our questions and discussion topics and the team leader is in charge of collecting the questions, managing progress for the week and hosting the lab meeting efficiently.

Figure 162: Structure of egoless team

Section 10.2 Progress tracking

We mainly use Github milestones and issues to track the progress of each team member. Other than that, we also have a team weekly update on our own internal Whatsapp group. We decided to not use Trello as we felt that the features provided on Github are sufficient for our use case.

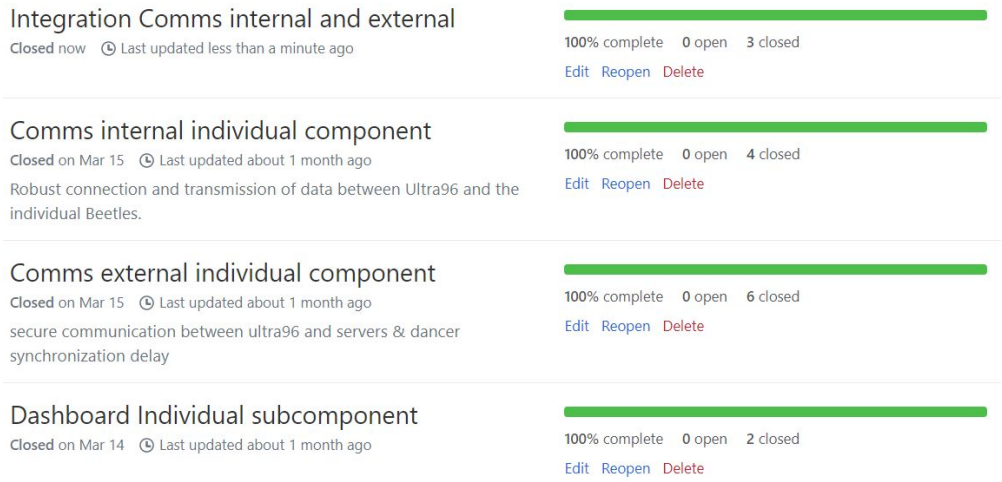


Figure 163: Our Github milestones

Week	Team	HW1	HW2	Comms1	Comms2	SW1	SW2
Week 3	Research	Research on individual component Prepare and work on individual design report					
Week 4	Design Report	* Completed initial design report and research * Purchased the required hardware components	* Complete Report and research * Learn the basics of PyTorch and Brevitas, with online tutorials	*Complete Report and research *Setting up device scanning, and connecting a single Beetle to Ultra96	*Complete Report * Get socket code running on my laptop * Research on NTP for timing protocol and synchronization	*Complete Report and research *Try running sample ML code in python on any dataset	*Complete Report and research on technologies to use for dashboard *Explore on creating dashboard client and server for dashboard
Week 5	Prototyping of individual components	* Explored the libraries and algorithms available for IMU * Gathered sensor readings from sensors using Beetle * Wired one set of hardware prototype and explore the changes in sensor reading	* Learn the basics of Neural Networks. This includes its concepts and technologies. *Start reading up on FINN Compiler and find out how to use it	*Get a Beetle to be able to maintain stable connection and send data to Ultra96 for at least 1 minute	*Run socket code on Ultra96 *Design clock synchronization protocol	*Obtain a suitable dataset modeling human activities *Apply machine learning libraries *Determine appropriate segmentation methods and features	*Test code for database storage on PostgresSQL *Find out what data points will be needed for the dashboard
Week 6	Progress Checkpoint First individual	* Experimented the battery circuitry for Beetle and	* Realized that FINN Compiler in its current state does not	*Get 3 Beetles to be able to maintain stable concurrent	* Get clock synchronization protocol running between BLE	*Apply k-fold model validation *Output	*Integrate Dash and PostgreSQL to run data that is sent

	prototype test	Ultra96 and tested whether they are at operational voltages for the hardware components * Attached the battery circuit to the hardware prototype * Gathered EMG sensor readings from MyoWare Muscle Sensor	support the implementation of MLP, seek for other alternatives. *Choose to use hls4ml instead.	connections and send data to Ultra96 for at least 1 minute	and Ultra96	metrics of models - classification accuracy and confusion matrix *Determine relative location	from Ultra96 to laptop *Implementing dbAPI for interaction between front-end and back-end
Recess week		* Wired up the other sets of hardware components based on the initial prototype including the battery circuitry * Explored and researched on features to extract from EMG data	*Learn multiple new concepts/tools to get started with hls4ml (e.g AXI-protocols, Fixed Precision point, Vivado HLS, PYNQ MMIO and DMA Libraries, How to connect IPs in Block Design Diagram etc.)	Prepare design and implementation for connecting and receiving data from 3 Beetles to the Ultra96			*Complete Full Stack GUI *Complete dashboard client and server
Week 7	Individual subcomponent test	* Ensured all hardware components are working as expected	* Continue trying to implement a working neural	*Ensure robust concurrent connections between 3	*Ensure socket connection is reliable between	*Ensure ML works efficiently on dataset and metrics	*Integrate dashboard and DB with the Ultra96 to send

		and fixed any hardware issues faced * Extracted certain features from EMG sensor reading * Gathered sensor data by performing certain dance moves and checked the changes in sensor reading	network with hls4ml. But unable to do so. * Went back to try out the FINN Compiler because there was an update last week that supports the generation of bitstream for MLP.	Beetles and Ultra96	Ultra96 and evaluation server *Ensure clock offset is correctly obtained	are readily available *Review algorithm for relative location	information through tests and discussion with internal comms *Understand EMG related topic
Week 8	Integration of subcompo nents into main system	* Integration with Internal Communicati on to ensure sensor data can be sent to Ultra96 * Debugging of issues that arise from the integration * Ensured that Beetles can function properly on batteries after integration of code * Extracted more features from EMG sensor	* Trained a working neural network with Brevitas and generated an ONNX Model on the example dataset * Generated a working bitstream with the ONNX Model.	*Integrating sensor packet data received from the Beetles to the machine learning model *Improve speed and reliability of data transmission whenever possible	*Implement thresholding values to detect start of the dance	*Work with HW2 to amend ML algorithms for FPGA acceleration	*Makes to received feedback *Decide and implement trends for EMG data

		reading, in particular extracting frequency domain features					
Week 9	Testing of Integrated prototype	* Gathered sensor data for machine learning models * Coded new thresholding algorithm to segment data appropriately for machine learning models * Debugged and fixed any hardware issues faced during data collection * Integrated EMG code with Internal Communication and ensure features extracted are sent to dashboard	* FINN Compiler does not directly support the inference with the hardware accelerated neural network directly on the Ultra96. Have to do a manual software setup of the Ultra96.	*Perform limit testing on the entire internal communications protocol in real time *Fix any bugs arising and ensure communication is robust	*Experiment to determine the frequency of time calibration	*Test ML algorithm on the data collected from actual sensors *Collect data from users	*Improve the GUI to look more readable and aesthetic.
Week 10	First evaluation test (3 dancer, 3 moves)	Mock evaluation tests to check connections as we started can work remotely. Continue with data collection for Machine Learning					

Week 11	First Evaluation Test(3 dancers, 3 moves)	* Fine-tune the integrated system	* Train a neural network with newly collected data *Generate a working bitstream for hardware acceleration	*Fine-tuning the integrated system	*Fine-tune the integrated system	*Fine tune based on the results from the first evaluation test, collect more data if needed	*Work on documentation and report
Week 12	Final Design report	Prepare and work on Final design report. Peer review					
Week 13	Final Design report submission	Finalize Design report Final Design report submission					

Table: Timeline of our progress

Section 11 References

[1] Dfrobot. (n.d.). Beetle BLE – The smallest Arduino Bluetooth BLE (4.0). Retrieved February 2, 2020, from https://www.dfrobot.com/product-1259.html

[2] Learning about Electronics. (n.d.). What is a LM7805 Voltage Regulator?. Retrieved April 14, 2020, from http://www.learningaboutelectronics.com/Articles/What-is-a-LM7805-voltage-regulator

[3] Theorycircuit. (2016, July 16). Myoware Muscle Sensor Interfacing with Arduino. Retrieved February 2, 2020, from http://www.theorycircuit.com/myoware-muscle-sensor-interfacing-arduino/

[4] Hareendran T.K. (n.d.). XL4015 Power Supply Module - Secret Talks. Retrieved April 14, 2020, from https://www.electroschematics.com/xl4015-power-supply-module-secret-talks/

[5] Jirapong Manit. (2019, June 6). How do I select window size while recording EMG using Biopac Single Channel?. Retrieved April 16, 2020, from

https://www.researchgate.net/post/How_do_I_select_window_size_while_recording_EMG_using_Biopac_Single_Channel

[6] Mads Aavik. (2017, August 10). What Is FFT and How Can You Implement It on an Arduino?. Retrieved April 16, 2020, from https://www.norwegiancreations.com/2017/08/what-is-fft-and-how-can-you-implement-it-on-an-arduino/

[7] Wikipedia. (n.d.). Spectral density. Retrieved April 16, 2020, from https://en.wikipedia.org/wiki/Spectral_density

[9] Gustavo R.P., Bruno A.M., Andre F.P., Malki-cedheq B.C., Marco A.B. (2019). Virtual Reality Game Development Using Accelerometers for Post-stroke Rehabilitation. XXVI Brazilian Congress on Biomedical Engineering. IFMBE Proceedings, vol 70/1. doi: 10.1007/978-981-13-2119-1_89

[10] Karen L.G., Jennifer E.M. (2014). Using Surface Electromyography in Physiotherapy Research. Australian Journal of Physiotherapy Vol 29/1. Doi: 10.1016/S0004-9514(14)60659-0

[11] Arduino.cc. *Arduino Memory*. Retrieved 14 April, 2020, from https://www.arduino.cc/en/tutorial/memory.

[12] Mohammad Afaneh. (2017). *BLE power consumption optimization: The comprehensive guide.* Retrieved 14 April, 2020, from https://www.novelbits.io/ble-power-consumption-optimization.

[13] Maxim Krasnyansky. *hciattach(8) - Linux man page*. Retrieved 14 April, 2020, from https://linux.die.net/man/8/hciattach

[14] Jim Blom. *Serial Communication*. Retrieved 14 April, 2020, from https://learn.sparkfun.com/tutorials/serial-communication/all.

[15] Docs.oracle.com. (2020). *What Is a Socket? (The Java™ Tutorials > Custom Networking > All About Sockets)*. [online] Available at: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html [Accessed 2 Feb.

2020].

[16] Quora.com. (2020). *Cryptography: What are the advantages and disadvantages of AES over Triple-DES? - Quora*. [online] Available at:

https://www.quora.com/Cryptography-What-are-the-advantages-and-disadvantages-of-AES -over-Triple-DES [Accessed 3 Feb. 2020].

[17] sluiter, s. (2019). *How does AES encryption work? Advanced Encryption Standard*. [online] YouTube. Available at: https://www.youtube.com/watch?v=lnKPoWZnNNM [Accessed 2 Feb. 2020].

[18] En.wikipedia.org. (2020). *Network Time Protocol*. [online] Available at: https://en.wikipedia.org/wiki/Network_Time_Protocol [Accessed 2 Feb. 2020].

[19] Ntp.org. (2020). *How does it work?*. [online] Available at: http://www.ntp.org/ntpfaq/NTP-s-algo.htm#Q-ALGO-POLL-BEST [Accessed 2 Feb. 2020].

[20] Separation of concerns. (2020). Retrieved 14 April 2020, from https://en.wikipedia.org/wiki/Separation_of_concerns

[21] En.wikipedia.org (2020) *Email Encryption*. [online] Available at: https://en.wikipedia.org/wiki/Email_encryption [Accessed 14 April 2020]

[22] https://www.viget.com (2020) *Email Is Completely Insecure By Default. | Viget*. [online] Available at: <https://www.viget.com/articles/email-is-completely-insecure-by-default/> [Accessed 14 April 2020]

[23] Sousa Lima, W., Souto, E., El-Khatib, K., Jalali, R. and Gama, J.(2019) Human Activity Recognition Using Inertial Sensors in a Smartphone: An Overview. *Sensors*, [online] 19(14), p.3213. Available at: https://www.mdpi.com/1424-8220/19/14/3213/htm [Accessed 5 February 2020]

[24] Blott, M., Preußer, T., Fraser, N., Gambardella, G., O'brien, K., Umuroglu, Y., Leeser, M. and Vissers, K.(2018) FINN- R. *ACM Transactions on Reconfigurable Technology and Systems*, [online] 11(3), pp.1-23. Available at: https://arxiv.org/pdf/1809.04570 [Accessed 18

March 2020].

[25] Minhas, S. (2019, November 27). 10 Rules of Dashboard Design. Retrieved April 14, 2020, from https://medium.muz.li/10-rules-of-dashboard-design-f1a4123028a2

[26] Dash for Beginners. (n.d.). Retrieved April 14, 2020, from https://www.datacamp.com/community/tutorials/learn-build-dash-python

[27] Szabgab. (n.d.). Plain function or Callback - An example in Python. Retrieved April 14, 2020, from https://code-maven.com/function-or-callback-in-python

[28] Me. (2018, December 18). High cpu in the browser and python. Retrieved April 14, 2020, from https://community.plotly.com/t/high-cpu-in-the-browser-and-python/17073

[29] Hristozov, K. (2019, July 19). MySQL vs PostgreSQL -- Choose the Right Database for Your Project. Retrieved from https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres

[30] BBVAOpen4U. (2018, June 25). 8 advantages of APIs for developers. Retrieved April 14, 2020, from https://bbvaopen4u.com/en/actualidad/8-advantages-apis-developers

[31] Toro, S. F. D., Santos-Cuadros, S., Olmeda, E., Álvarez-Caldas, C., Díaz, V., & Román, J. L.
S. (2019). Is the Use of a Low-Cost sEMG Sensor Valid to Measure Muscle Fatigue? *Sensors*, *19*(14), 3204. doi: 10.3390/s19143204

[32] Durcevic, S. (2019, July 11). 14 Dashboard Design Principles & Best Practices To Convey Your Data. Retrieved from https://www.datapine.com/blog/dashboard-design-principles-and-best-practices/

[33] Rokr. (2020, February 17). 37 Best Free Dashboard Templates For Admins 2020. Retrieved from https://colorlib.com/wp/free-dashboard-templates/

[34] Komninos, A. (2020, April 7). 7 UX Deliverables: What will I be making as a UX designer? Retrieved from https://www.interaction-design.org/literature/article/7-ux-deliverables-what-will-i-be-making-as-a-ux-designer

[35] Singhal K. (2019, April 12). How can I detect if a trend is increasing or decreasing in time

series? [Online Forum]. Retrieved from https://stackoverflow.com/questions/55649356/how-can-i-detect-if-trend-is-increasing-or-decreasing-in-time-series

[36] Kexugit. (2018, April). Machine Learning - Sensors in Sports: Analyzing Human Movement with AI. Retrieved April 14, 2020, from https://docs.microsoft.com/en-us/archive/msdn-magazine/2018/april/machine-learning-sensors-in-sports-analyzing-human-movement-with-ai#sensor-kit-overview

[37] Attal, F., Mohammed, S., Dedabrishvili, M., Chamroukhi, F., Oukhellou, L., & Amirat, Y. (2015). Physical Human Activity Recognition Using Wearable Sensors. *Sensors*, *15*(12), 31314–31338. doi: 10.3390/s151229858

[38] Xu, W., Zhang, M., Sawchuk, A. A., & Sarrafzadeh, M. (2012). Robust Human Activity and Sensor Location Corecognition via Sparse Signal Representation. *IEEE Transactions on Biomedical Engineering*, *59*(11), 3169–3176. doi: 10.1109/tbme.2012.2211355

[39] Mantecón, T., Del-Blanco, C. R., Jaureguizar, F., & García, N. (2019). A real-time gesture recognition system using near-infrared imagery. *Plos One*, *14*(10). doi: 10.1371/journal.pone.0223320

[40] Ultraleap. (n.d.). Digital worlds that feel human. Retrieved April 14, 2020, from https://www.ultraleap.com/

[41] Ready Player One (film). (2020, April 13). Retrieved April 14, 2020, from https://en.wikipedia.org/wiki/Ready_Player_One_(film)

[42] Understanding Sliding and Tumbling Windows (n.d.). Retrieved from https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.6.1/bk_storm-component-guide/content/storm-windowing-concepts.html

[43] Neural network models (supervised). (n.d.). Retrieved from https://scikit-learn.org/stable/modules/neural_networks_supervised.html

[44] sklearn.preprocessing.StandardScaler. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[45] sklearn.model_selection.train_test_split. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[46] sklearn.metrics.accuracy_score. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

[47] Vikash Raj Luhaniwal. (2019, December 16). EDA all classification algorithms with 96% acc. Retrieved from https://www.kaggle.com/vikashrajluhaniwal/eda-all-classification-algorithms-with-96-acc

[48] sklearn.metrics.confusion_matrix. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

[49] pandas.read_csv. (n.d.). Retrieved from https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

[50] running Python functions as pipeline jobs. (n.d.). Retrieved from https://joblib.readthedocs.io/en/latest/

[51] Brownlee, J. (2020, January 28). How to Save and Reuse Data Preparation Objects in Scikit-Learn. Retrieved from https://machinelearningmastery.com/how-to-save-and-load-models-and-data-preparation

-in-scikit-learn-for-later-use/

[52] sklearn.neural_network.MLPClassifier. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier

[53] Hoffman, C. (2016, September 22). Why Is Email Spam Still A Problem? Retrieved from https://www.howtogeek.com/180604/htg-explains-why-is-spam-still-a-problem/

[54] Software, D. (2017, November 30). 9 Applications of Machine Learning from Day-to-Day Life. Retrieved from https://medium.com/app-affairs/9-applications-of-machine-learning-from-day-to-day-life-112a47a429d0

[55] R, V., Ganesh, B., Kumar, A., & KP, S. (n.d.). DeepAnti-PhishNet: Applying Deep Neural Networks for Phishing Email Detection. Retrieved from http://ceur-ws.org/Vol-2124/paper_9.pdf

[56] Liu, H., Ye, Q., Wang, H., Chen, L., & Yang, J. (2019). A Precise and Robust Segmentation-Based Lidar Localization System for Automated Urban Driving. *Remote Sensing*, *11*(11), 1348. doi: 10.3390/rs11111348

[57] Mapwize. (n.d.). Retrieved from https://www.mapwize.io/posts/how-indoor-positioning-works.html

[58] Christian, A. (2017, December). Analyzing User Emotions via Physiology Signals. Retrieved February 9, 2020, from https://www.researchgate.net/publication/323935725_Analyzing_User_Emotions_via_Physio logy_Signals

[59] sklearn.model_selection.GridSearchCV. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[61] Chegg.com. (n.d.). Retrieved from https://www.chegg.com/homework-help/describe-give-advantages-disadvantages-moving-averages-b-exp-chapter-8-problem-10q-solution-9780134156323-exc

[62] Agarwal, R. (2019, November 26). The 5 Classification Evaluation metrics every Data Scientist must know. Retrieved from https://towardsdatascience.com/the-5-classification-evaluation-metrics-you-must-know-aa97784ff226

[63] Khandelwal, R. (2019, January 25). K fold and other cross-validation techniques. Retrieved from https://medium.com/datadriveninvestor/k-fold-and-other-cross-validation-techniques-6c03a2563f1e