CG4002 Embedded System Design Project
AY2020/21 Semester 2

Final Report

B18	Name	Student #	Sub-Team	Specific Contribution
#1	Lim Ting Wei	A0184280E	HW Sensors	Section 2.2 Section 3.1 Section 6 Section 7
#2	Mohideen Imran Khan	A0181321U	HW FPGA	Section 2.1 Section 3.2 Section 6 Section 7
#3	Mohamed Riyas	A0194608W	Comms Internal	Section 2.2 Section 4.1 Section 6 Section 7
#4	Zeng Hao	A0177355W	Comms External	Section 1 Section 4.2 Section 6 Section 7
#5	Ng Wei Jie, Brandon	A0184893L	SW ML	Section 2.3 Section 5.1 Section 6 Section 7
#6	Jeevan Neralakere Somashekhar	A0167779A	SW Dashboard	Section 1 Section 5.2 Section 6 Section 7

Table of Contents

Table of Contents 2

Section 1 System Functionalities 5

Section 1.1 Introduction 5

Section 1.2 User Story 5

Section 1.2.1 Coach 5

Section 1.2.2 Trainee 6

Section 1.3 Use Case Diagram with Description 6

Section 1.4 Feature Lists 7

Section 2 Overall System Architecture 9

Section 2.1 System Architecture Diagram 9

Section 2.2 System Components and Interaction 10

Section 2.3 Algorithm For Activity Detection Problem 12

Section 2.3.1 Algorithm To Classify Dance Moves 13

Section 2.3.2 Algorithm To Detect Dancer Positions 13

Section 2.3.3 Algorithm To Measure Muscle Fatigue 13

Section 3 Hardware Details 15

Section 3.1 Hardware sensors 15

Section 3.1.1 Summary of components and devices 15

Section 3.1.2 Pin table 16

Section 3.1.3 Schematics 16

Section 3.1.4 Power requirements 17

Section 3.1.5 Algorithms and libraries 18

Section 3.1.6 Changes made to hardware design 23

Section 3.1.7 Changes made to arduino code 25

Section 3.2 Hardware FPGA 27

Section 3.2.1 Ultra96 Synthesis Setup 27

Section 3.2.2 Ultra96 Simulation Setup 28

Section 3.2.3 Neural Network Design 29

Section 3.2.4 PYNQ Overlay 31

Section 3.2.5 Evaluation of Hardware Accelerator 32

Section 3.2.6 Ultra96 Power Management 34

Section 4 Firmware & Communications Details 37

Section 4.1 Internal Communications 37

Section 4.1.1 Introduction 37

Section 4.1.2 Multithreading on Beetle 37

Section 4.1.5 Tasks running in Beetle 45

Section 4.1.6 Components that make up the data packet that gets transmitted to the laptop from the beetle 50

Section 4.1.7 Periodic Push by Arduino to Transmit Data between Beetle and Laptop 51

Section 4.1.8 No Retransmission of Data Packets 51
Section 4.1.9 Task management using loops instead of FreeRTOS and Protothread 52
Section 4.1.10 Setup and configuration for BLE interfaces 53
Section 4.1.12 2-way Handshake 60
Section 4.1.13 No processing of IMU sensor data as no need to fit data packet within 20 bytes 61
Section 4.1.14 Checksum 62
Section 4.1.15 Packet Format 62
Section 4.1.16 Packet Types 63
Section 4.1.17 Baud Rate & Connection Rate 63
Section 4.1.17 No Packet Reassembly 64
Section 4.1.16 Ensuring reliability & robustness 64

Section 4.2 External Communications 66
Section 4.2.1 Introduction 66
Section 4.2.2 Communication between laptops and Ultra96 66
Section 4.2.3 Communication between Ultra96 and Dashboard/Evaluation Servers 67
Section 4.2.4 Detailed Message Format 67
Section 4.2.5 Libraries and APIs 67
Section 4.2.6 Processes running on the Ultra96 68
Section 4.2.6 Clock Synchronization and Dance Synchronization Delay 69

Section 5 Software Details 72
Section 5.1 Software Machine Learning 72
Section 5.1.1 Initial Method (Dance Moves) 72
Section 5.1.2 Initial Method (Dance Positions) 72
Section 5.1.3 Data Segmentation 73
Section 5.1.4 Feature Extraction 73
Section 5.1.5 Detection Of Dance Moves 77
Section 5.1.5.1 Training Model 78
Section 5.1.5.2 Validation And Evaluation 80
Section 5.1.6 Detection Of Relative Position 81
Section 5.2 Software Dashboard 84
Section 5.2.1 Introduction 84
Section 5.2.2 Dashboard Design 84
Section 5.2.2.1 User Flow 84
Section 5.2.2.2 Features to be Supported 86
Section 5.2.2.3 Wireframes & Actual Implementation 87
Section 5.2.3 System Architecture 98
Section 5.2.4 Technical Stacks 99
Section 5.2.4.1 Database - MongoDB 99
Section 5.2.4.2 Backend Framework - Express 100
Section 5.2.4.3 Frontend Library - ReactJS 101

Section 5.2.4.4 Runtime Environment - NodeJS 102
Section 5.2.4.5 Queue Service - RabbitMQ 102
Section 5.2.5 Storing Incoming Sensor Data 102
Section 5.2.6 Real-time Streaming 103
Section 5.2.6.1 Data flow between Ultra96 and MongoDB 103
Section 5.2.6.2 Data flow between MongoDB and server-side 103
Section 5.2.6.3 Data flow between server-side and client-side 103
Section 5.2.7 Authentication 104
Section 5.2.8 User Survey 104
Section 5.2.8.1 Pre-User Survey 104
Section 5.2.8.1 Post-User Survey 106

Section 6 Societal and Ethical Impact **107**
Section 6.1 Future Use Cases 107
Section 6.2 Privacy Concerns 108
Section 6.3 Ethical 108

Section 7 Project Management Plan **109**

Bibliography **112**

Section 1 System Functionalities

Section 1.1 Introduction

Welcome to DanceEdge! A wearable system that detects and coaches dance moves for groups of up to 3 people! This system is especially designed to help both coaches and trainees alike train virtually in the COVID-19 Pandemic. The pandemic has undoubtedly affected physical interaction. As a result, dance groups are unable to meet up as often as they would have liked. Fear not! DanceEdge is here to solve that problem!

So, how does DanceEdge work? Trainees are to each wear a pair of wearable devices on their wrists and ankles. The system will act as dance coach by evaluating the accuracy of their individual dance choreography. These wearable devices will assist by sending relevant data to our machine learning model which eventually feedbacks the predicted data to a dashboard that can be accessed by both coach and trainees. The system computes the position of dancers, the dance moves itself and the synchronization delay between the fastest and slowest dancer. This computed output will be compared against the correct output to allow trainees to identify their mistakes. To increase accuracy, the correct output will be used to recalibrate the system for any misclassification. All in all, trainees and coaches will both be able to view the analysis of the results on the dashboard.

This report focuses on the initial design of DanceEdge.

Section 1.2 User Story

Section 1.2.1 Coach

User stories from the point of view of a dance coach:

- [Priority: HIGH] I want to know if the members of my group are performing the right actions so that I can guide them to better perform the routine
- [Priority: HIGH] I want to know if the members of my group are synchronised so that I can advise which dancer to slow down or speed up
- [Priority: MEDIUM] I want to know how fatigued my group members are so that I can analyse the strenuity of the dance actions
- [Priority: MEDIUM] I want to see if the overall performance and accuracy of my group members after a dance routine so that I know who to focus my attention to and coach individually

Section 1.2.2 Trainee

User stories from the point of view of a trainee:

- [Priority: HIGH] I want to know if I am doing the right dance move so that I can better follow the routine
- [Priority: HIGH] I want to know if I am too slow or too fast from my other dance members so that I adjust my movements to better synchronize the group's movement
- [Priority: HIGH] I want the wearable device to be power efficient so that I do not have to frequently and inconveniently keep changing the batteries
- [Priority: HIGH] I want the wearable device to be lightweight and easy to put on so that I not only feel comfortable wearing it for long periods of time but also not feel restrained by it. Essentially, I would want to wear it because it makes me feel good.
- [Priority: MEDIUM] I want to know my overall statistics after a dance routine so that I can pinpoint where to improve myself
- [Priority: MEDIUM] I want to be able to dance anywhere such that I do not have to worry about COVID-19 physical interaction restrictions and still be able to learn to dance.

Section 1.3 Use Case Diagram with Description

- Use Case 1 - Correct Position and Dance Move
  - Main Success Scenario:
    1. Coach logins in into DanceEdge dashboard and registers up to three trainees who will be involved in dance coaching exercise.
    2. Evaluation server outputs respective positions of each trainee along with an action to perform.
    3. Trainees will transit to their new position and perform the aforementioned action.
    4. Dashboard visually shows their predicted position and action as well as the real-time raw sensor data that was used to derive those predictions. In addition, the synchronization delay between the trainees as well as muscle fatigue level of one trainee will be displayed.

       Use case ends

- Use Case 2 - Final Logout Dance Action

- Main Success Scenario:
  1. Evaluation server outputs “Logout” and the positions the trainees should transit to.
  2. Trainees transit to their new position and perform final logout dance action.
  3. Dashboard displays that the dance routine has come to an end. Trainees and coaches can view their overall statistics for the dance exercises.
     Use case ends

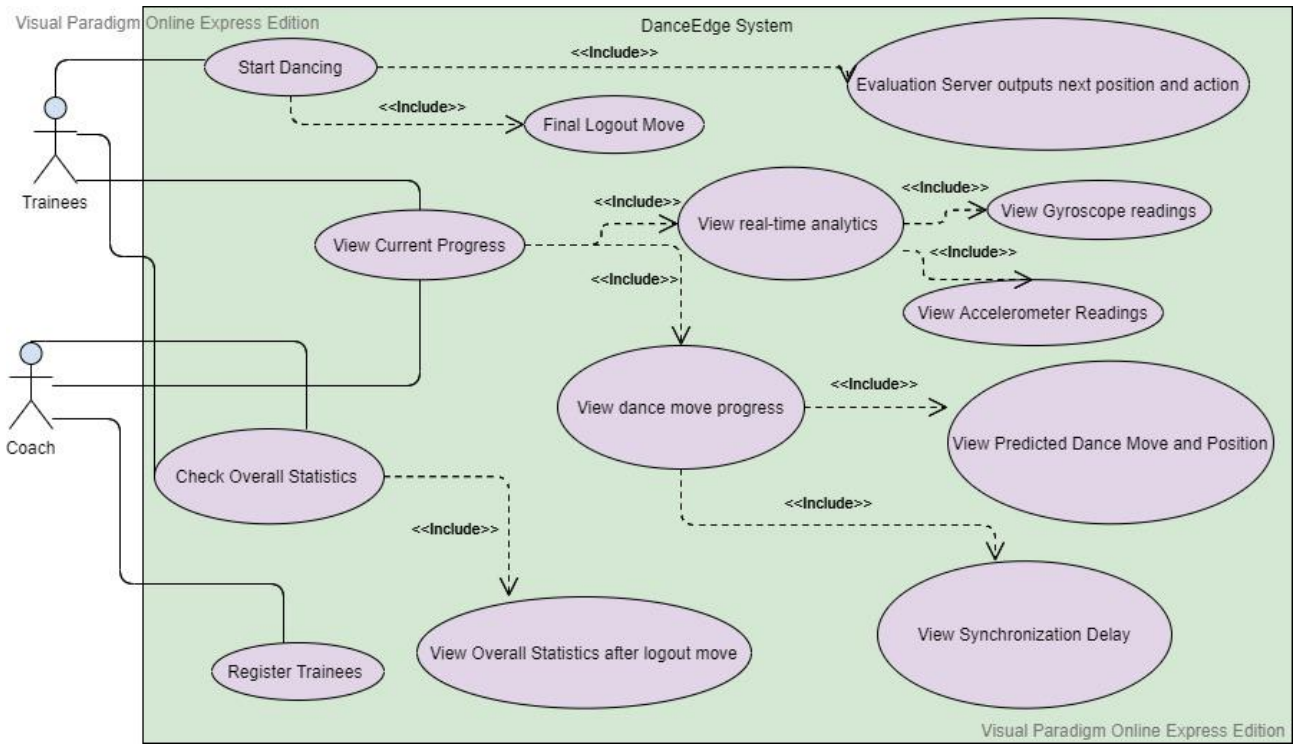


Figure 1.3-1: Use Case Diagram

Section 1.4 Feature Lists

Below is a list of features and its concept ordered by priority levels:

1. Priority Level: HIGH
   a. Accurate Sensor Readings
   b. Accurate Machine Learning Classification for Actions and Positions
   c. Accurate Calculation of Synchronisation Delay
   d. Secure and Reliable Transmission between Sensors
   e. Secure and Reliable Transmission between Laptop and Server
   f. Energy Efficient Power Consumption

2. Priority Level: MEDIUM

a. Easy-to-use and Aesthetically-pleasing User Interface
b. Real-time updates on Dashboard
c. Lightweight Wearable Device

3. Priority Level: LOW
    a. One size that fits anyone for wearable device
    b. Comfortable wearable device

Section 2 Overall System Architecture

Section 2.1 System Architecture Diagram

The planned architecture of the system is as shown in the diagram below:

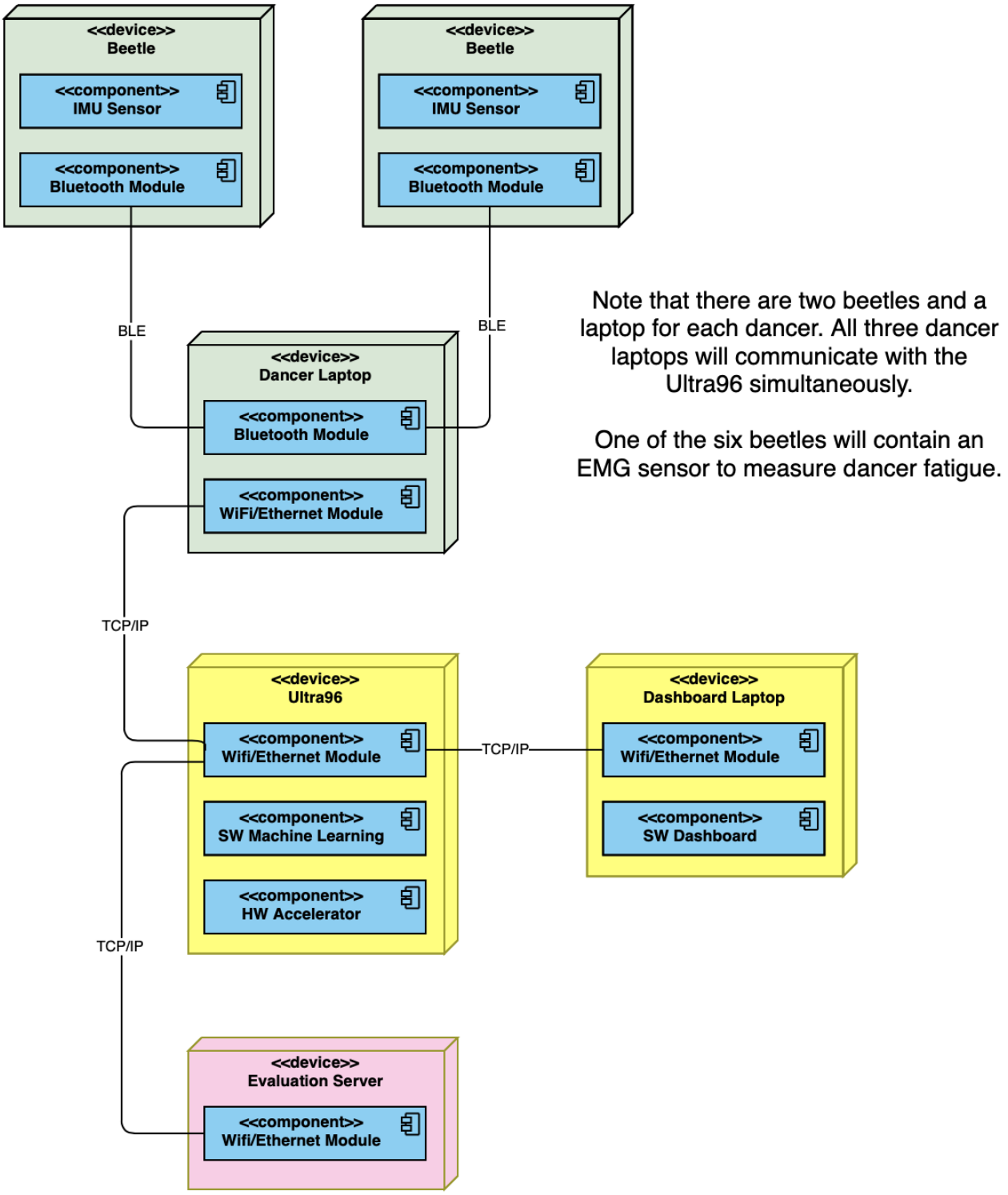


Figure 2.1 Overall system diagram

Section 2.2 System Components and Interaction

The following table details the components of the system, their purpose and how they interact with other components.

Component	Purpose	Interaction
Beetle	The purpose of the beetle is to read and process the data given by the IMU and transmit it to the dancer laptop through properly formatted data packets. Each data packet will contain the yaw, pitch, roll, accelerometer and gyroscope values along with other attributes. On top of the above responsibilities, one special beetle will also be responsible for collecting, processing and transmitting the required EMG signal attribute to the laptop.	The beetle receives data from the sensors through Serial, which then transmits the data to the laptop via Bluetooth using the CC2540 chip built on the beetle itself. Additionally, every trainee will be equipped with two beetles connected to a dancer laptop.
Bluno USB Link	The purpose of the bluno USB link is to receive data from the beetle through the pyserial library. Each link will be able to establish connection and receive data from 1 beetle. Hence, we will be using 6 bluno USB links.	The bluno USB link will auto connect with the beetle once it is attached to one of the serial ports in the laptop and the beetle is switched ON. Then, it will receive the data from the beetle.
Dancer Laptop	The purpose of dancer laptops are to receive the data transmitted from the beetles of the respective dancer. It will further process the data received and send the processed data to Ultra96.	After receiving the data transmitted from the dancers through bluetooth, the laptop will modify the data and send the data to the Ultra96 through Wifi.
Ultra96	The Ultra96 receives processed sensor data from the dancer laptops. 1. The data is fed into the SW machine learning model that uses the on-chip FPGA to accelerate dance move inference. 2. The data is used to infer relative dancer positioning through machine learning models. 3. The data is used to infer dancer	The dancer Laptop communicates with the Ultra96 through TCP/IP. Again, TCP/IP is used for communication between the Ultra96 and the evaluation server and between the Ultra96 and the Dashboard Laptop.

	synchronisation delay. The inferred information is sent to the dashboard for display and to the evaluation server for grading.	
Dashboard Laptop	To provide an interface for trainees and coaches to view and analyse their present and past performances in their dance routines.	The dashboard interacts with the database hosted on the cloud.
Evaluation Server	The evaluation server is to send dancers movement and locations for the instruction purpose and receive the data sent from Ultra96 for the evaluation purpose.	

Drawing of the final form of the system

Figure 2.2 Final form of the wearable

Figure 2.2 shows the user wearing two sets of devices. Each set consists of an arm pouch, one beetle, one IMU, and one battery pack. One of the dancers is required to wear a

MyoWare Muscle sensor too. The reason why we picked arm pouch as our wearable container is because of the comfort and the convenience of the user. The strap of the arm pouch allows the user to tighten as well as loosen the pouch with ease. With the help of zippers, the pouch is able to prevent its contents from slipping out, and cushions them from any potential damage.

Section 2.3 Algorithm For Activity Detection Problem

This section outlines the algorithms to classify the dance moves and detect dance positions. It also covers the muscle fatigue measurement. Each dancer wore two wearable devices - one on the right hand and the other on the waist. The readings from the right hand were purely for detecting dancer moves. The readings from the waist were purely for detecting left and right turns to determine the dancer positions.

A window of 250 readings were collected on the Ultra96 from each of the wearable devices. The data was sent to Ultra96 in a buffer of 5 packets. When the window of 250 readings were collected, Ultra96 used a deep learning model to classify the dance moves and thresholds to detect left or right turns. Refer to Figure 2.3-1 for the sequence diagram for activity detection. Note that the model ran on Ultra96.

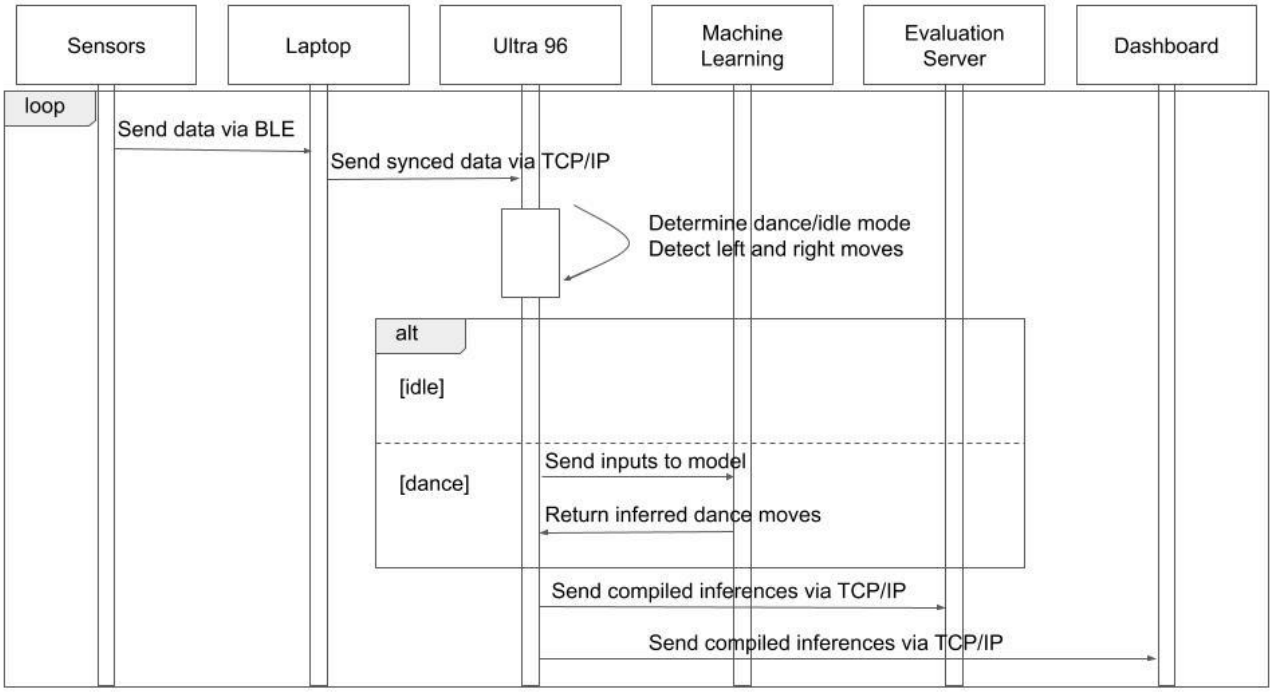


Figure 2.3-1 Sequence Diagram

Section 2.3.1 Algorithm To Classify Dance Moves

1. Collect data from wearable device on the hand
    a. Sample the 3-axial acceleration and 3-axial gyroscope at 25Hz
    b. Calculate the orientation angles yaw, pitch, and roll angle
    c. Send data that contains acceleration, gyroscope, and angles to laptop via bluetooth
2. Transmit data on laptop to Ultra96
    a. Send data that contains acceleration, gyroscope, and angles to Ultra96 via TCP in batch size of 5
3. Classify dance moves on Ultra96
    a. Collect a window of 250 readings (about 10 seconds of data)
    b. Perform feature extraction on the last 60 readings
    c. Normalize the features
    d. Infer the dance moves using deep learning model
    e. Send inferences to evaluation server via TCP and dashboard via RabbitMQ

Section 2.3.2 Algorithm To Detect Dancer Positions

4. Collect data from wearable device on the waist
    a. Sample the 3-axial acceleration and 3-axial gyroscope at 25Hz
    b. Calculate the orientation angles yaw, pitch, and roll angle
    c. Send data that contains acceleration, gyroscope, and angles to laptop via bluetooth
5. Transmit data on laptop to Ultra96
    a. Send data that contains acceleration, gyroscope, and angles to Ultra96 via TCP in batch size of 5
6. Detect dancer positions on Ultra96
    a. Find the index of the roll angle in the window that exceeds the threshold
    b. Determine whether the dancer make a left or right turn
    c. Compute the dancer positions from the turns
    d. Send inferences to evaluation server via TCP and dashboard via RabbitMQ

Section 2.3.3 Algorithm To Measure Muscle Fatigue

7. Collect data from myoware sensor
    a. Sample the EMG signals at 25Hz
    b. Send data that contains the EMG signals to laptop via bluetooth

8. Train data to dashboard server
    a. Send data to dashboard server via RabbitMQ

Section 3 Hardware Details

Section 3.1 Hardware sensors

Section 3.1.1 Summary of components and devices

Bluno Beetle, DFR0339, is a microcontroller that is suitable for wearables due to its small size. Its functions are similar to Arduino Uno. It has a Bluetooth chip, CC2540, which allows serial communication between devices via Bluetooth.

Datasheet link: https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339

The Inertial measurement unit (IMU) board, GY-521, has a chip called MPU-6050. MPU-6050 has a three-axis accelerometer and a three-axis gyroscope. Furthermore, it has a Digital Motion Processor (DMP) which removes the need of a magnetometer (Walsh, 2013). It is used to determine the dance movements from the user.

Datasheet link:
https://components101.com/sites/default/files/component_datasheet/MPU6050-DataSheet.pdf

MyoWare Muscle Sensor, AT-04-001, detects EMG signals from muscle response, and these signals are used to determine muscle fatigue of the user.

Datasheet link:
https://github.com/AdvancerTechnologies/MyoWare_MuscleSensor/blob/master/Documents/AT-04-001.pdf

Four AAA batteries with a battery holder are used to power up the Beetle for a period of time.

Pin headers are used to be soldered on IMU, Beetle and muscle sensor pins. Jumper wires are used to connect IMU, Beetle and muscle sensor together.

Section 3.1.2 Pin table

Bluno Beetle Pin	Connects to	Via
5V	IMU Vcc	Wire
GND	IMU GND	Wire
SCL	IMU SCL	Wire
SDA	IMU SDA	Wire
VIN	Battery holder	Battery holder's wire
GND	Battery holder	Battery holder's Wire

Table 3.1.2-1: Pin table for connections from beetle to IMU and battery holder.

Bluno Beetle Pin	Connects to	Via
5V	MyoWare Sensor '+'	Wire
GND	MyoWare Sensor '-'	Wire
A0	MyoWare Sensor SIG	Wire

Table 3.1.2-2: Pin table for connections from beetle to muscle sensor. Note that this is only applicable to 1 dancer.

Section 3.1.3 Schematics

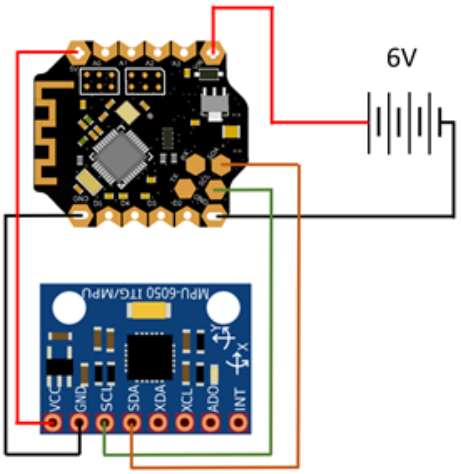


Figure 3.1.3-1: Schematics without MyoWare Muscle Sensor

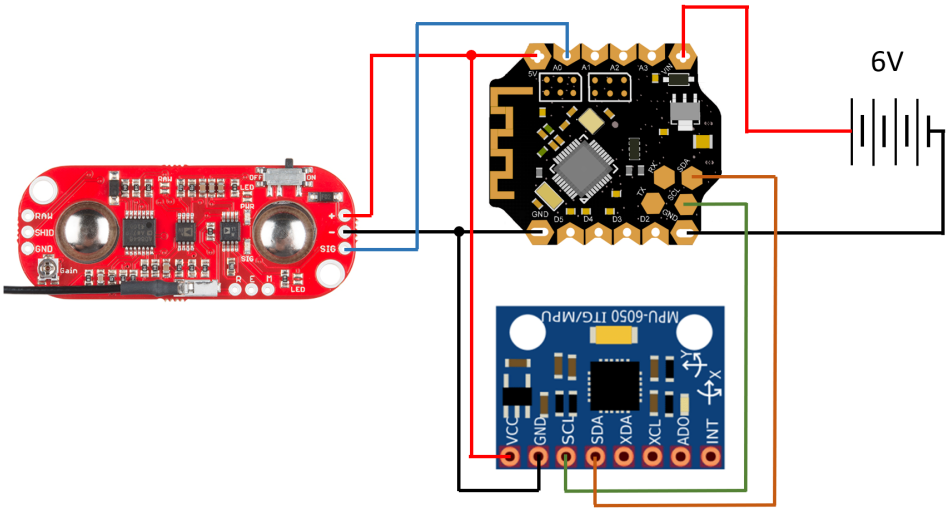


Figure 3.1.3-2: Schematics with MyoWare Muscle Sensor

Section 3.1.4 Power requirements

Bluno Beetle:

- Operating voltage: 5V
- Operating current: 10mA (cliffgi, 2016) + 0.2mA (Atmel, n.d.)

To power up the beetle without using the micro-USB port, an external voltage ranging from 5V to less than 8V is suitable (DFRobot, n.d.). Hence, we decided to use 4 AAA batteries which produce 6V.

IMU board:

- Operating voltage: 2.375V - 3.46V
- Operating current: 3.9mA (InvenSense, n.d.)

To power up the IMU board, the beetle's 5V pin is connected to IMU's Vcc pin.

MyoWare Muscle Sensor:

- Operating voltage: 3.3V or 5V
- Operating current: 9mA (MyoWare, n.d.)

To power up the muscle sensor, the beetle's 5V pin is connected to the sensor's '+' pin.

The overall power is 5*10.2m + 3.46*3.9m + 5*9m = 109.5mW, which results in roughly 21.9mA in current. As the practical application is usually higher than theory, we bumped it to 25mA. Referring to the specifications sheet of Energizer Max battery, one set is able to run for about 12-15 hours (Energizer, n.d.). After 12-15 hours, the batteries reach 5V or below and are required for replacement.

Section 3.1.5 Algorithms and libraries

First, we need to find a variable that is suitable for thresholding to detect the start of dance moves. After trial and error, total acceleration is considered to be suitable for thresholding. To compute total acceleration, acceleration x, y, and z values are used. Here is the formula as shown:

```
total_acceleration = sqrt(pow(aaReal.x/ 8192.0,2)+ pow(aaReal.y/ 8192.0,2)+ pow(aaReal.z/ 8192.0,2));
```

Figure 3.1.5-1: Formula of total acceleration

When computing total acceleration, the sensitivity is reduced by 1g (acceleration due to gravity), which represents 8192. Next, to make use of the total acceleration, a finite state machine is used as reference for the algorithm.

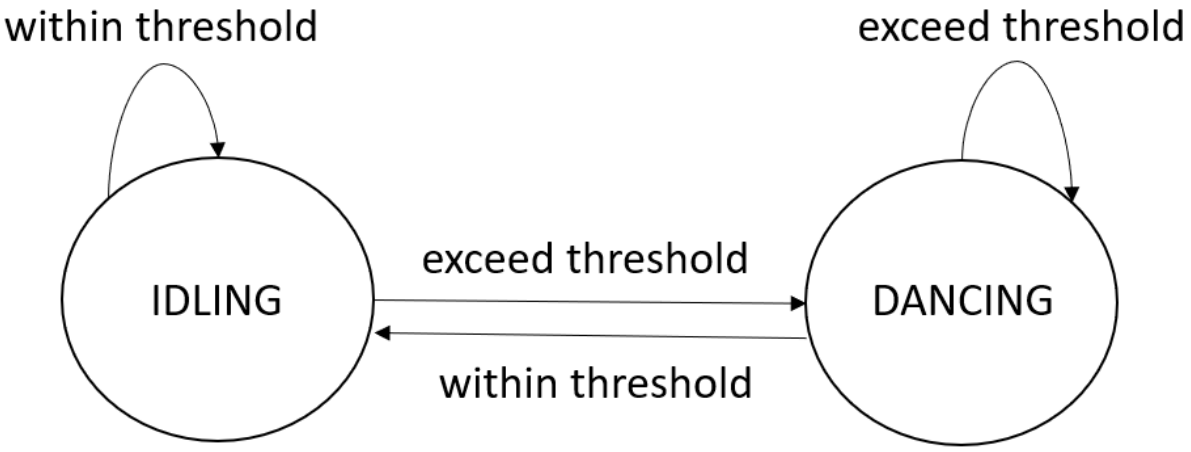


Figure 3.1.5-2: Finite State Machine to detect the start of dance

A finite state machine model will be used to detect the start of dance of the user. The two states are IDLING and DANCING. The initial state is IDLING, and it remains in that state when the user is not moving. When the user starts to dance, the value of total acceleration and other values change and exceed the threshold's values, which results in the state shifting from IDLING to DANCING state. When the user stops moving, the value of total acceleration and other values return to within the threshold, and the state shifts back from DANCING to IDLING state.

The following shows how the algorithm works in changing of states whenever the user dances or stops dancing:

Test case 1: The user starts to dance

1. Check current beetle state
    a. If the current state is IDLING, move to step 2.
    b. If the current state is DANCING, check for threshold, results in exceeding threshold, no change of state.
2. Check for threshold
    a. Total acceleration > 0.53
    b. Either absolute pitch or absolute roll > 10

    If both 2a and 2b conditions are met, move to step 3.

3. There is a counter that keeps track of how many times the threshold is exceeded. If the threshold is exceeded at least 3 times in a row, change beetle state to DANCING.

Test case 2: The user stops dancing

1. Check current beetle state
    a. If the current state is IDLING, check for threshold, results in within threshold, no change of state.
    b. If the current state is DANCING, move to step 2.
2. Check for threshold
    a. Total acceleration <= 0.53
    b. Both absolute pitch and absolute roll <= 10

    If either 2a or 2b conditions are met, move to step 3.

3. There is a counter that keeps track of how many times the threshold is not exceeded. If the threshold is not exceeded at least 4 times in a row, change beetle state to IDLING.

Here is the code snippet where the above algorithm is implemented:

```
int readValues () {
  if (total_acceleration > 0.53 && ((abs_pitch > 10) || (abs_roll > 10)))
     return 1; //DANCING at this moment
  }
  return 0; //IDLING at this moment
}

bool isDancing_2() {                    //current state is idling, check i
  if (readValues() == 1) {            //read DANCING
    count_dancing++;
    if (count_dancing >= 3) {
      return true;                    //Finally, idle->dancing
    }
  } else {                            //read IDLING, dancing is false alarm
    count_dancing = 0;
  }

  return false;                      //return false
}

bool isIdling() {                     //current state is moving/dancing, ch
  if (readValues() == 0) {            //read IDLING
    count_idling++;
    if (count_idling >= 4) {
      return true;                    //Finally, move->idle or dance->idle
    }
  } else {                           //read MOVING, idling is false alarm
    count_idling = 0;
  }
  return false;                      //return false
}

void checkBeetleState() {
  switch (bState) {               //Depending on the state
    case IDLING: {
      if (isDancing_2()) {            //Check if it is dancing
        bState = DANCING;
        count_idling = 0;
        //count_moving = 0;
        count_dancing = 0;
      }
      break;
    }
    case DANCING: {
      if (isIdling()) {            //Check if it is idling
         bState = IDLING;
         count_idling = 0;
         //count_moving = 0;
         count_dancing = 0;
      }
       break;
    }
  }
}
```

Figure 3.1.5-3: Code snippet to detect start of dance

In order to read the accelerometer and gyroscope of the IMU board, a MPU6050 library maintained by Electronic Cats will be used to detect yaw, pitch, roll, gyroscope's x, y, and z axis as well as acceleration's x, y and z axis. These values can be used to detect dance movements, as well as the positions of the dancers.

Electronic Cats MPU6050 library link: https://github.com/ElectronicCats/mpu6050

In order to determine the fatigue level of the user, three indicators are needed. They are mean absolute value (MAV), root mean square (RMS), and mean frequency (MNF) (Toro et al., 2019). To retrieve amplitude from the muscle sensor, a sample code provided by AdvancerTechnologies is used. Noise filtering is not required as the EMG signal has been rectified and integrated by the hardware (MyoWare, n.d.). As fourier transform is needed to retrieve values for MNF, a library called arduinoFFT will be used.

arduinoFFT library link: https://github.com/kosme/arduinoFFT

First, in order to convert analog signal to voltage value, the following formula is used to find voltage:

```
sensorValue = analogRead(analogInPin);
float voltage = sensorValue * (5.0 / 1023.0);
```

Figure 3.1.5-4: Formula of finding voltage from analog signal (sparkfun, n.d.)

Next, to compute MAV, RMS, and MNF, the following formulas are as shown:

$$MAV = \frac{1}{N}\sum_{i=1}^{N}|x_i| \qquad RMS = \sqrt{\frac{1}{N}\sum_{i=1}^{N}x^2} \qquad MNF = \frac{\sum_{j=1}^{M} f_j P_j}{\sum_{j=1}^{M} P_j}$$

Figure 3.1.5-5: Formulas of MAV, RMS and MNF (Toro et al., 2019)

In MAV, it is calculated as the average of N absolute voltages, where N is the total number of samples. In RMS, it is calculated as the square root of the average of N number of square of voltages (Toro et al., 2019). However for MNF, we used a slightly different formula to compensate for the lack of memory in a bluno beetle.

$$\mathrm{MNF} = \frac{\sum_{k=1}^{N} \left(\sum_{j=1}^{\frac{M}{2}} f_j P_j \right)_k}{\sum_{k=1}^{N} \left(\sum_{j=1}^{\frac{M}{2}} P_j \right)_k}$$

Figure 3.1.5-6: Formula of updated MNF

In MNF from figure 3.1.5-6, it is calculated as the sum of N batches of the summation of the product of frequency interval and its respective power spectrum, divided by the sum of N batches of the summation of power spectrum, where N represents the number of batches that the beetle has recorded, and M represents the number of recorded samples in a batch (Toro et al., 2019).

The beetle can only store 64 samples of voltages at a time due to lack of memory, thus M is 64. It is sampled at 1 kHz, updating MAV, RMS, and MNF every 64 ms.

Here is the code snippet of the implementation of MAV, RMS, and MNF:

```
for (int i = 0; i < samples; i ++) {
  accu_mav += abs(vReal[i]);
  accu_rms += pow(vReal[i],2);
  vImag[i] = 0; //Important, prevent ovf
}

overall_s_count += samples; //include new samples
mav = accu_mav*1.0/overall_s_count;
rms = sqrt(accu_rms*1.0/overall_s_count);
```

```
//Only Fourier transform AFTER getting MAV and RMS, due to reusing array
FFT.Compute(vReal, vImag, samples, FFT_FORWARD); /* Compute FFT */

for (int i = 0; i < (samples/2)+1; i++)
{
  double vPower = (pow(vReal[i],2) + pow(vImag[i],2)) / (samples*samplingFrequency);

  if (i >= 1 && i < (samples/2))
  {
    vPower = 2*vPower;
  }

  sum_freq_power += (i*1.0*samplingFrequency/samples)*vPower;
  sum_power += vPower;
}

mean_freq = sum_freq_power*1.0/sum_power;
```

Figure 3.1.5-7: Code snippet to compute MAV, RMS, and MNF

In order to send data from beetle to laptop at 25Hz, while sampling EMG signal values at 1kHz, multithreading is needed. A library called ArduinoThread is used to enable multithreading.

ArduinoThread library link: https://github.com/ivanseidel/ArduinoThread

Section 3.1.6 Changes made to hardware design

After the evaluation in week 9, we found out that there are some issues with the initial design. Here is the initial design as shown:

Figure 3.1.6-1: A beetle set excluding battery holder (left) and a beetle set with battery holder (right)

Based on the initial design, we have suspected a number of potential issues:

1. Some headers are soldered to bluno beetles and IMUs perpendicularly, resulting in potential disconnections due to the sharp bending of wires.
2. The battery holder slides back and forth when the dancer starts dancing, which may collide with the connections between the beetle and the IMU.
3. There is no switch to turn on and off easily.
4. The pouch cover may be interfering with the bluetooth connectivity between the beetle and laptop.

With these issues in mind, we have decided to revise the design to solve the above issues. In week 11, here is the revised design as shown:

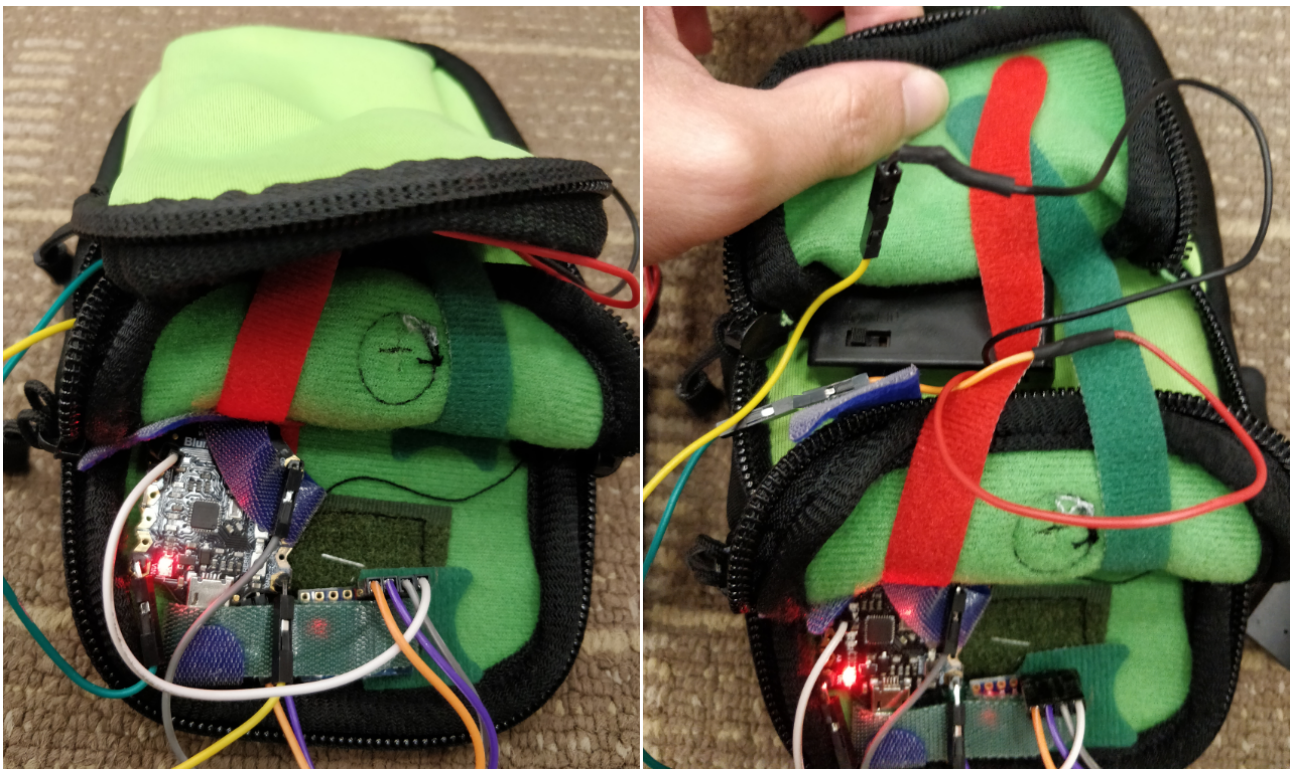


Figure 3.1.6-2: A beetle set with hidden battery holder (left) and a beetle set with exposed battery holder (right)

The revised design has eliminated all of the above issues that have been mentioned earlier, making the beetle sets more robust, while having the flexibility to swap out components when necessary. Unfortunately, there is an issue with this design, which is the potential loose connections between the beetle and the battery holder. This is due to the female headers of the jumper cable being worn out fairly easily and loose over time. The person in charge of hardware sensors has already thought out of a few solutions:

1. Perform heat shrinking on connections between beetle and battery holder
2. Perform soldering on connections between beetle and battery holder
3. Mount both beetle and IMU onto a strip board to strengthen the connection

However, some of us have different views to this revised design and decided to rework the design. Here is the final design as shown:

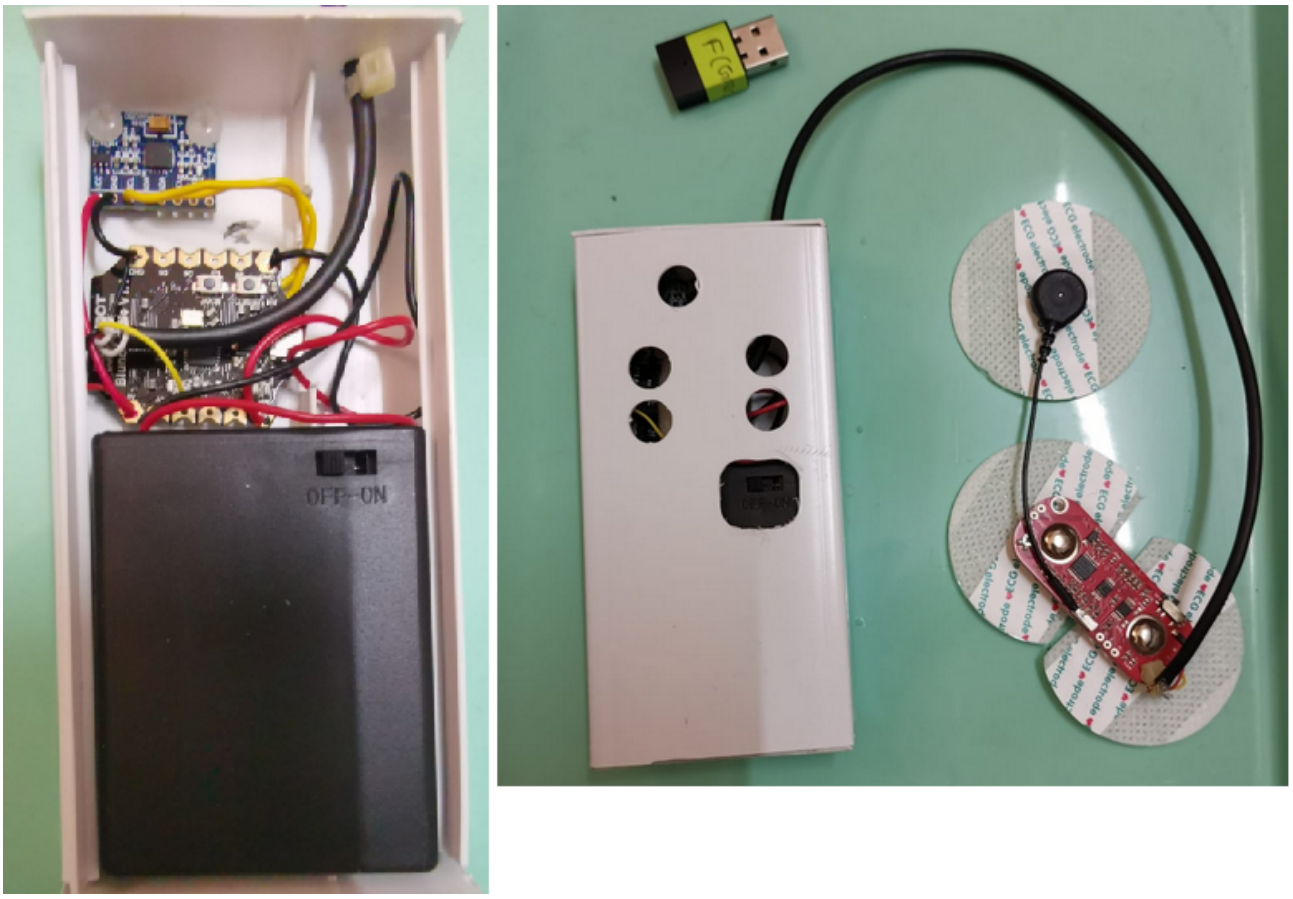


Figure 3.1.6-3: A beetle set in an opened container (left) and a beetle set in a closed container (right)

The final design has eliminated the above issue that is mentioned in the revised design, by soldering and placing the battery holder, beetle, and IMU in a container. As there could be potential interference of bluetooth connectivity due to the hard casing, the casing has been drilled with a few holes to improve connectivity.

Section 3.1.7 Changes made to arduino code

In week 7, the thresholds to detect the start of dance move were different from the final thresholds. There were three states: IDLING, MOVING, and DANCING. In order to reach DANCING state from IDLING state, it has to switch from IDLING to MOVING state first, before attempting to switch from MOVING to DANCING state. To change state from MOVING to DANCING state, a few steps are carried out:

1. Retrieve 5 most recent values of gyro y, and 5 recent values from gyro z
2. Check for threshold
    a. Total acceleration > 0.4
    b. Third value of gyro y > 100 and is greater than first and fifth value
    c. Third value of gyro z > 150 and is greater than first and fifth value

3. If 2a and either 2b or 2c conditions are met, change state from MOVING to DANCING.

We decided to simplify the thresholding and reduce the states to just IDLING and DANCING as we realised that the MOVING state is redundant and it delays the response time by at least a second. Here is the code snippet of thresholding in week 7:

```
bool is_startdance() {
  int five_gyro_y[] = {0,0,0,0,0};
  int five_gyro_z[] = {0,0,0,0,0};
  int j = 0;
  for (int i = startpt; i < endpt; i++) //sizeof(arr)/sizeof(arr[0])
  {
      five_gyro_y[j] = arr[i%6];
      j++;
  }
  j = 0;
  for (int i = startpt; i < endpt; i++) //sizeof(arr)/sizeof(arr[0])
  {
      five_gyro_z[j] = arr_2[i%6];
      j++;
  }
  if ( ((five_gyro_y[2] > 100) && (five_gyro_y[2] > five_gyro_y[0]) && (five_gyro_y[2] > five_gyro_y[4]))
     ){
      return true;
  } else if ( ((five_gyro_z[2] > 150) && (five_gyro_z[2] > five_gyro_z[0]) && (five_gyro_z[2] > five_gyro_z[4]))
            ){
             return true;
  }
  return false;
}

int readDanceValues () {

  if (total_acceleration > 0.4 && is_startdance()) {
     return 1; //DANCING at this moment
  }
  return 0; //NOT DANCING at this moment
}
```

Figure 3.1.7: Code snippet to detect start of dance in week 7

Section 3.2 Hardware FPGA

Specialised coprocessors are becoming increasingly common in consumer devices. Recent SoCs such as Apple M1 contain dedicated units for accelerating neural networks and handling security-related tasks (Frumusanu, 2020). These coprocessors perform a specific set of tasks in a fast and power-efficient manner as compared to general-purpose processors which frees the main processor to perform other tasks (Howard, 2014).

Therefore, an FPGA-based hardware accelerator is implemented to make the dance detection machine learning model more responsive. We hope that this will enhance our user experience while reducing the overall system power consumption.

Section 3.2.1 Ultra96 Synthesis Setup

Development of the hardware accelerato rwas done using Vivado High-Level Synthesis (HLS) tools. The neural network model as described in section 3.2.3 was implemented in C++ within Vivado HLS. A simple C++ program created using Vivado HLS is shown in the image below while the full code is uploaded to LumiNUS as instructed:

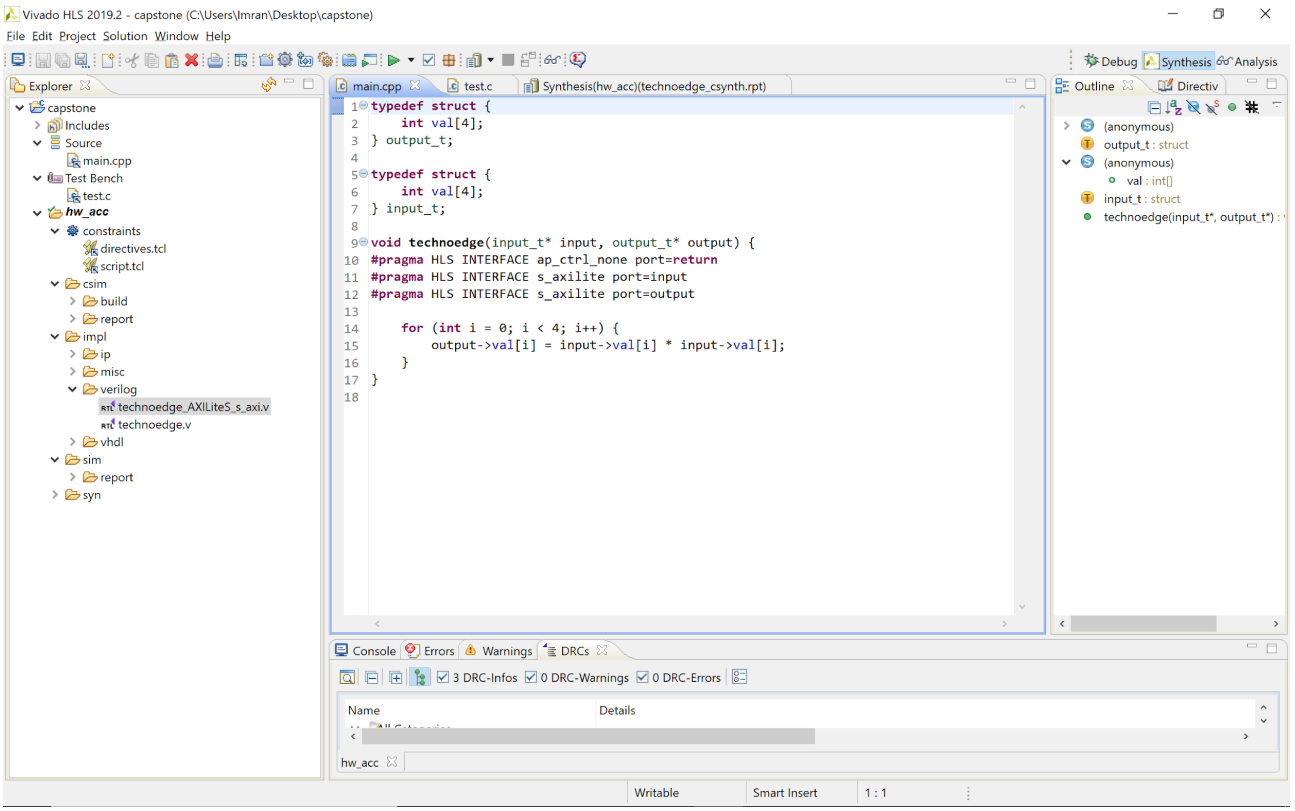


Figure 3.2.1 HLS Screenshot

The synthesised C++ model was then packaged as an Intellectual Property (IP) core and imported into Vivado. This IP interfaces with the Zynq processing system using AXI4-LITE,

a memory-mapped input/output (MMIO) interface. The block diagram of the system is as shown:

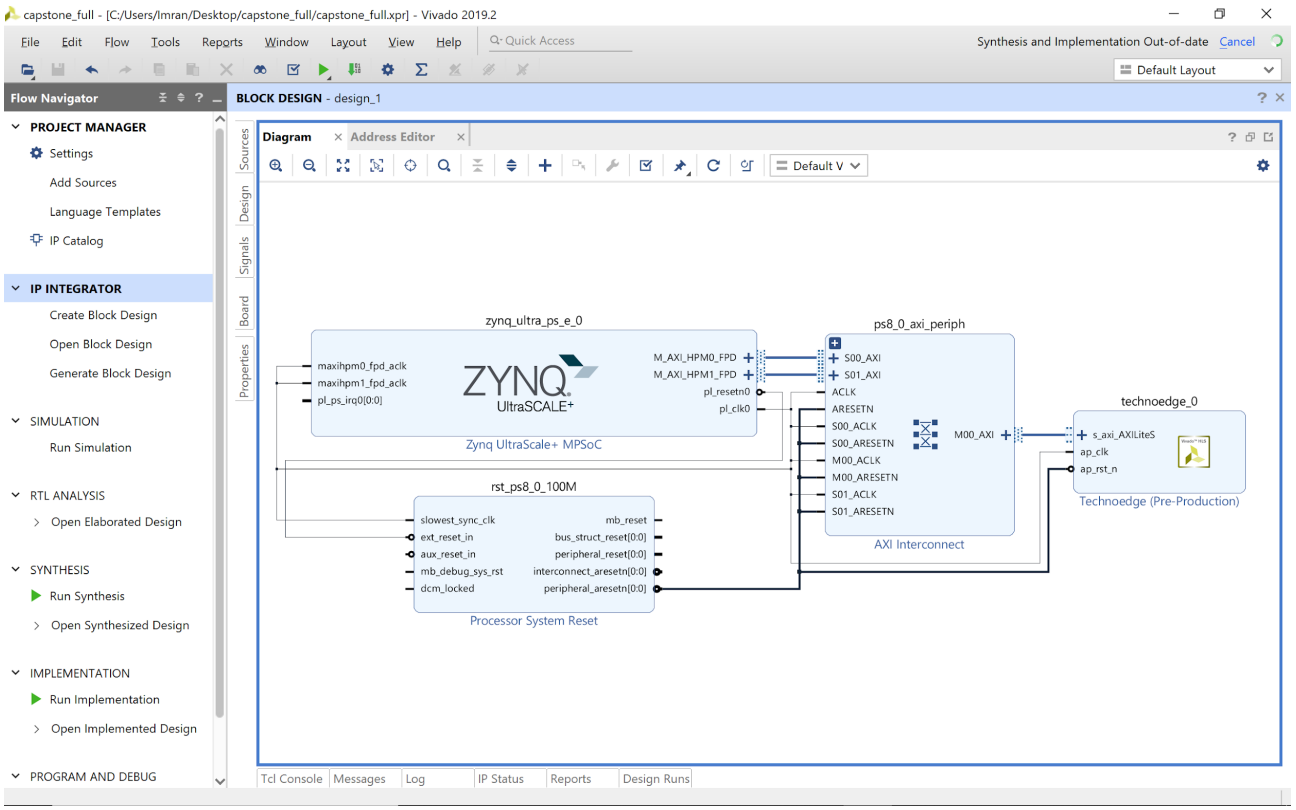


Figure 3.2.2 Block Design Screenshot

A MMIO interface was chosen over a streaming interface such as AXIS as the benefits of streaming are diminished for our neural network implementation where the entire input has to be received before the processing could begin.

Section 3.2.2 Ultra96 Simulation Setup

The IP core as exported from Vivado HLS was verified with cocotb, a Python-based cosimulation library. An example testbench for the IP shown in the previous section is as follows:

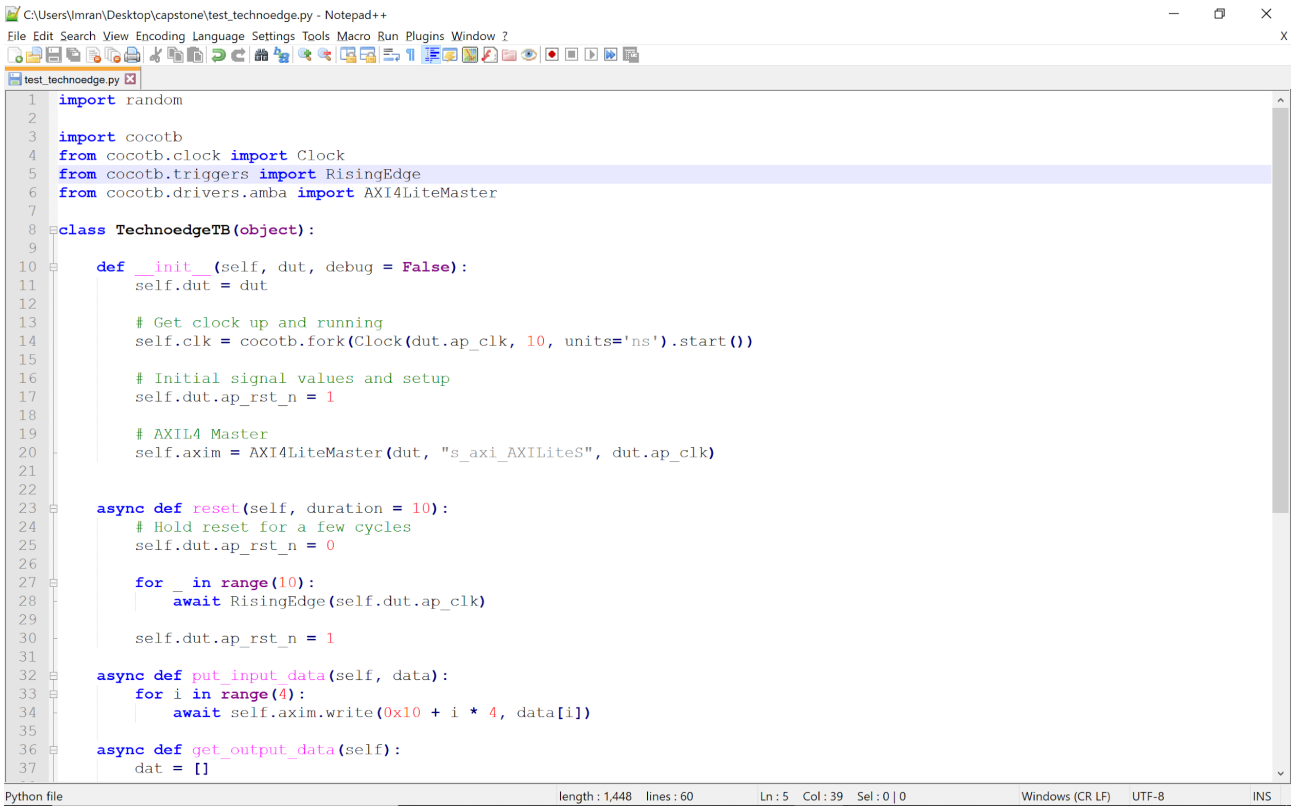


Figure 3.2.3 Testbench screenshot

Our machine-learning models were primarily designed and validated in software using Python. Thus, using Python for verification integrates the software and hardware development processes by allowing for code reuse across the two platforms. This would not be possible with Vivado HLS-based C++ testbenches where the same logic has to be duplicated in C++.

Section 3.2.3 Neural Network Design

The machine learning model on the FPGA uses a fixed point system with 8 bits for the sign and integer components and 24 bits for the fractional component. Thus, the results are very close to floating-point computation and the accuracy loss is minimal at just 0.001.

We designed the following components for our neural network:

Component	**Mathematics**
1D Convolution Layer	The convolution layer “convolves” the input with one or more *kernels*. In lay terms, the kernel slides along the 1D input vector with one or more channels, performing an element-wise multiplication and accumulation. This then results in a new 1D vector with one channel per kernel used. Bias terms are then added to these vectors to

	produce the final output. Now, let N be the batch size. In our case, it will just be 1 (one inference at a time). Let C be the number of channels and L be the length of the 1D input vector (number of samples). Then the output is defined by the following equation (*Conv1D*, n.d.): $$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$ where (*) is the "convolution" operation. Note that this is not the same as the mathematical convolution operation (CG2023), although it is somewhat similar in principle.
ReLU	This is the activation function that takes as input a value and returns the value as it is if it's non-negative or returns zero otherwise.
1D Max Pooling	The max pooling layer is similar to the convolution layer but instead of performing a multiplication and accumulation operation it simply returns the maximum value within the sliding kernel.
Flatten	This layer simplifies a multi-channel vector of samples to a single-channel vector.
Linear	This is the classic neural network layer with a set number of nodes. Each node connects to every other node in the previous layer with a particular weight. The output of the node is the sum of the products of the weights and the corresponding node's output, with an additional bias term. Mathematically, it can be expressed as follows (*Linear*, n.d.): $$y = xA^T + b$$ where x is the output vector of the previous layer, A is the matrix of weights, b is the bias vector and y is the computed output vector of the current layer.

We used the following architecture:

Layer	**Description**	**No. of Nodes**
Input	126 extracted features	126
Linear	64-node hidden layer	64
Linear	16-node hidden layer	16
Linear	9-node output layer	9

We have considered two different methods of implementing the above architecture:

1. Independent IP cores that corresponding to layer types and stitching them together in Vivado Block Design
2. Single IP core in C++ with reusable software components (using C++ templates)

While approach (1) makes the neural network implementation more modular and easier to test with RTL simulation (we can test each component independently), we have decided on approach (2) for improved latency as it doesn't have the overhead of streaming data from one IP to another.

Section 3.2.4 PYNQ Overlay

Overlays are PYNQ abstractions that allow user applications to span across both the Processing System (PS), the general-purpose ARM processors, and the Programmable Logic (PL), the FPGA. Such an abstraction allows the application developer to take advantage of hardware acceleration without in-depth knowledge of hardware engineering.

Therefore, we developed a PYNQ overlay that handles the following tasks:

1. Program the FPGA using the generated bitstream
2. Handle low-level read and writes to the FPGA

This overlay can then be used by other developers who do not need to know the details of the underlying hardware.

A screenshot of the overlay is as follows:

```
class TechnoEdge:
    def __init__(self, filename):
        self.overlay = Overlay(filename)
        self.techno = self.overlay.technoedge_0

        self.write_arr = self.techno.mmio.array[DATA_ADDR // 4:DATA_ADDR // 4 + FC1_IN]
        self.read_arr = tc.techno.mmio.array[RESULT_ADDR // 4: RESULT_ADDR // 4 + FC3_OUT]

    def write(self, data):
        self.write_arr[:] = to_int_arr(data)

    def run(self):
        self.techno.write(STATUS_REGISTER, 1)
        while self.techno.read(STATUS_REGISTER) & 0x2 == 0:
            pass

    def run_benchmark(self, reps = 1000):
        data = to_int_arr([0] * FC1_IN)

        start = time()
        for _ in range(reps):
            self.write_arr[:] = data
            self.run()
            _ = self.read_arr
        end = time()

        return (end - start)/reps*1000000

    def get_result(self):
        return [ctypes.c_int(x).value / FIXED_FACTOR for x in self.read_arr]

    def put_weights(self, wts):
        #
        # FC1 weight and bias
        #
        arr = self.techno.mmio.array[FC1_WT_ADDR // 4:FC1_WT_ADDR // 4 + len(wts['fc1_wt'])]
        arr[:] = to_int_arr(wts['fc1_wt'])
        arr = self.techno.mmio.array[FC1_BIAS_ADDR // 4:FC1_BIAS_ADDR // 4 + len(wts['fc1_bias'])]
        arr[:] = to_int_arr(wts['fc1_bias'])

        #
        # FC2 weight and bias
        #
        arr = self.techno.mmio.array[FC2_WT_ADDR // 4:FC2_WT_ADDR // 4 + len(wts['fc2_wt'])]
        arr[:] = to_int_arr(wts['fc2_wt'])
        arr = self.techno.mmio.array[FC2_BIAS_ADDR // 4:FC2_BIAS_ADDR // 4 + len(wts['fc2_bias'])]
        arr[:] = to_int_arr(wts['fc2_bias'])

        #
        # FC3 weight and bias
        #
        arr = self.techno.mmio.array[FC3_WT_ADDR // 4:FC3_WT_ADDR // 4 + len(wts['fc3_wt'])]
        arr[:] = to_int_arr(wts['fc3_wt'])
        arr = self.techno.mmio.array[FC3_BIAS_ADDR // 4:FC3_BIAS_ADDR // 4 + len(wts['fc3_bias'])]
        arr[:] = to_int_arr(wts['fc3_bias'])
```

Section 3.2.5 Evaluation of Hardware Accelerator

The hardware accelerator was evaluated based on the following factors:

1. **Accuracy**

   The accuracy of the predictions produced by the hardware accelerator would be compared against the accuracy of the pure software implementation.

   Actual accuracy: The software and FPGA predictions are identical to an accuracy of 0.001. This was possible due to the usage of large fixed-point multiplication in the FPGA.

2. **Time taken to make one inference from software**

   However, communication overhead between the PS and PL has to be taken into account as well. Therefore, we plan to make a large number of inferences, say 1000, through the PYNQ overlay and get the average time to make a single inference. This can then be compared against a pure software implementation.

   Actual prediction: the FPGA takes about 11 microseconds per prediction including communication overhead. The software takes about 1-2ms per prediction.

   The code used for FPGA benchmarking is as shown:

```
In [275]: tc = TechnoEdge('/home/xilinx/jupyter_notebooks/frontier/capstone_full.bit')

In [276]: # load weights from file
          f = open("wts", "rb")
          wts = np.load(f)
          tc.put_weights(wts)
          f.close()

In [277]: print("It takes %dus per inference on average." % tc.run_benchmark(1000))
          It takes 111us per inference on average.
```

3. **Hardware resource usage**

   Additionally, we measured the hardware resources (LUTs, RAMs, DSPs, etc) used by the neural network implementation. Using fewer resources allows us to use a smaller FPGA which in turn reduces both the cost and power consumption. Although changing FPGA is not possible within the scope of this project, we measure this metric to provide a more complete picture. The screenshot below shows the estimated resource utilisation. Owing the small size of the neural network, we are well below full utilisation.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	1	-	-	-
Expression	-	12	0	781	-
FIFO	-	-	-	-	-
Instance	30	-	676	600	-
Memory	1	-	64	8	0
Multiplexer	-	-	-	281	-
Register	-	-	270	-	-
Total	31	13	1010	1670	0
Available	432	360	141120	70560	0
Utilization (%)	7	3	~0	2	0

4. **Power consumption**

   This is elaborated in the next section.

Section 3.2.6 Ultra96 Power Management

The Ultra96 board has some support for measuring power usage through PMBus. This facility was used to measure the power consumption of the board when the system is in use.

With respect to the FPGA, we are prioritising performance over power even though it is possible to direct hardware design tools to reduce resource usage and prioritise lower power usage over performance. We believe that the gain in user experience from a more performant system outweighs the cost of increased power usage, especially given the fact that the Ultra96 is currently not used as a mobile device.

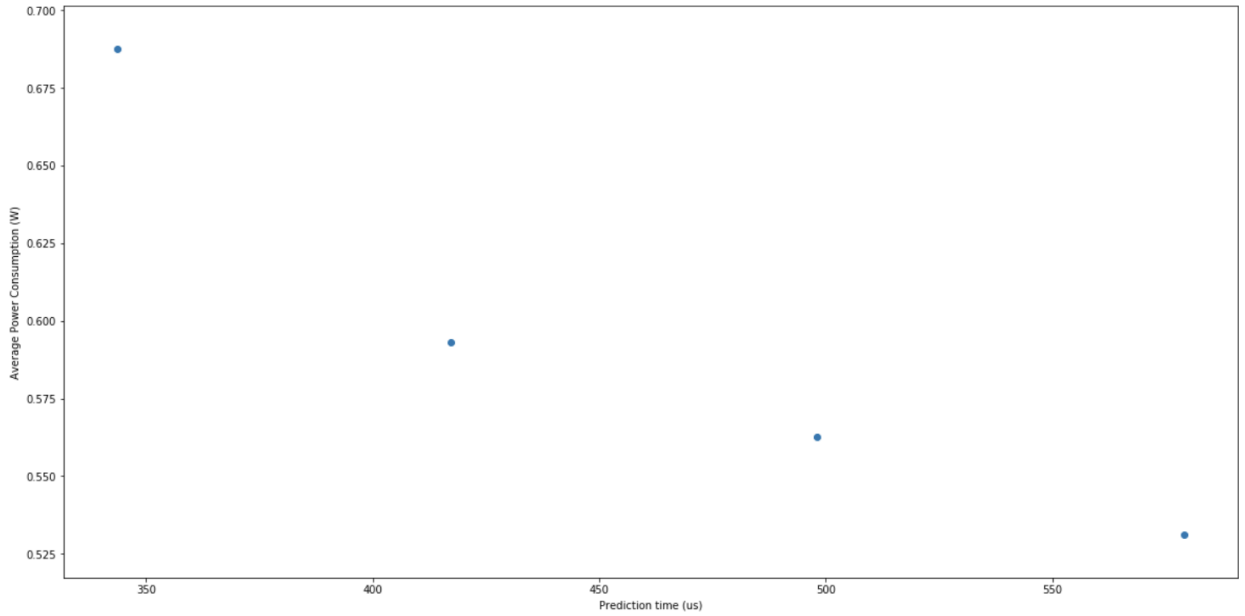
That said, the following measures would still be undertaken to keep power usage of the FPGA as low as possible:

1. Control signals to the IP block to signal the start and end of the computation instead of running the computations perpetually to reduce dynamic power dissipation

2. If the performance of the FPGA is more than adequate, we will go for a more balanced approach towards performance and resource usage

Additionally, we benchmarked the power consumption of the system against different CPU and FPGA clock speeds.

Here is the benchmark of the CPU power consumption with respect to clock speed and prediction times:



And the code to change CPU clock frequency:

```
def set_cpu_freq_mhz(freq):
    if freq == 300:
        freq = 200000
    if freq == 400:
        freq = 350000
    if freq == 600:
        freq = 400000
    if freq == 1200:
        freq = 600000
    !echo $freq > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
    !cat  /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_cur_freq
    !echo $freq > /sys/devices/system/cpu/cpu1/cpufreq/scaling_setspeed
    !cat  /sys/devices/system/cpu/cpu1/cpufreq/cpuinfo_cur_freq
```

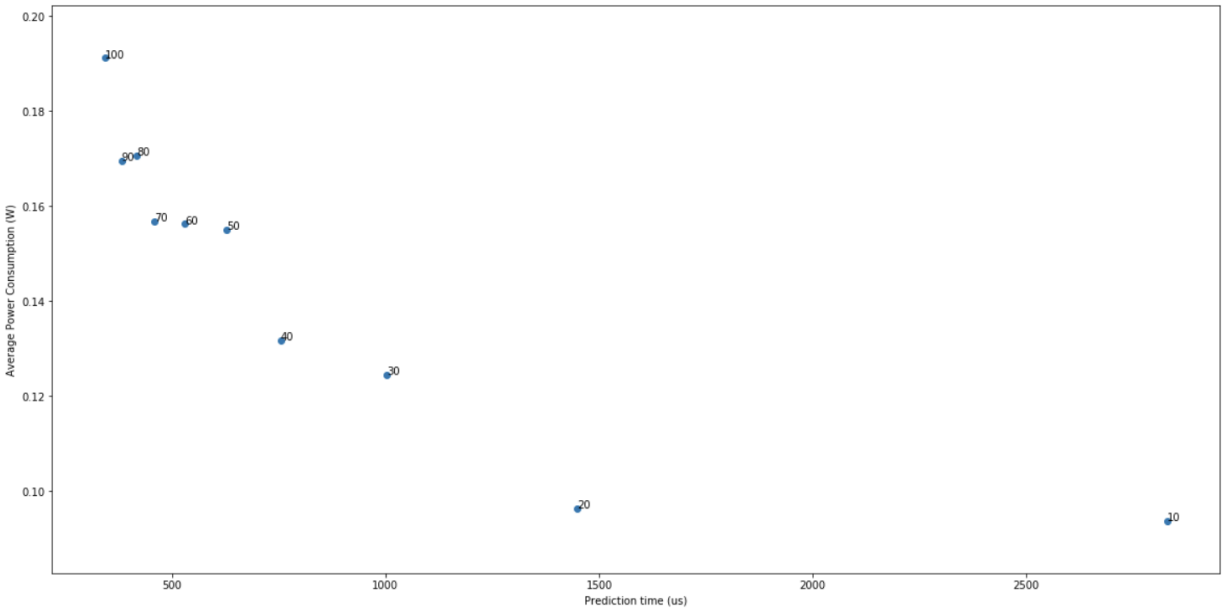
```
tc = TechnoEdge('/home/xilinx/jupyter_notebooks/frontier/capstone_full.bit')
rails = pynq.get_rails()
```

```
powers = []
times = []

for freq in [300, 400, 600, 1200]:
    set_cpu_freq_mhz(freq)
    recorder = pynq.DataRecorder(rails['PSINT_FP'].power)
    with recorder.record(0.2):
        duration = tc.run_benchmark(10000)
    power = recorder.frame['PSINT_FP_power'].mean()
    times.append(duration)
    powers.append(power)

plt.clf()
plt.figure(figsize=(20,10))
plt.scatter(times, powers)
plt.xlabel("Prediction time (us)")
plt.ylabel("Average Power Consumption (W)")
```

Here is the FPGA power consumption with respect to clock frequency and prediction times:



And here is the code used to measure it:

```
def set_fpga_freq(freq):
    pynq.ps.Clocks.fclk0_mhz = freq
    pynq.ps.Clocks.fclk1_mhz = freq
    pynq.ps.Clocks.fclk2_mhz = freq
    pynq.ps.Clocks.fclk3_mhz = freq
```

```
iterations = 10000
powers = []
times = []
freqs = []

for freq in range(10, 110, 10): # rails.keys():
    set_fpga_freq(freq)
    freqs.append(freq)
    rail = 'INT'

    recorder = pynq.DataRecorder(rails[rail].power)
    with recorder.record(0.2):
        pred_time = tc.run_benchmark(iterations)

    power = recorder.frame[recorder.frame.columns[1]].mean()
    times.append(pred_time)
    powers.append(power)


plt.clf()
plt.figure(figsize=(20,10))
plt.scatter(times, powers)
plt.xlabel("Prediction time (us)")
plt.ylabel("Average Power Consumption (W)")

for i in range(0, len(times)):
    plt.text(times[i], powers[i], freqs[i])
```

As seen in the diagrams above, lowering clock frequency decreases power consumption but increases prediction time. Since the Ultra96 is already a power-optimized board, lowering clock frequencies severely affected performance.

Section 4 Firmware & Communications Details

Section 4.1 Internal Communications

Section 4.1.1 Introduction

The main goal of the internal communications component is to provide a robust, reliable and concurrent mode of data transmission between the six beetles (two per trainee) and their respective assigned laptops through the use of Bluetooth. The various tasks that it needs to accomplish are as shown below:

- Obtain sensor data from both the IMUs and MyoWare muscle sensor and process them
- Transmit these data as packets from the respective beetles to their specified laptop while also ensuring reliability by handling any issues of disconnections, data corruption or packet fragmentation

In the coming sections, we will discuss how the internal communication system was designed, with a few considerations in mind, to achieve these objectives. In addition, we will also be talking about how some of the subcomponents were modified and redesigned after the week 9 evaluation for better performance due to some limitations that we faced. We will first start off by discussing various aspects of the overall design of the internal communications system before moving on to explain more about the protocol design that coordinates data transmission between the beetles and the laptops.

Section 4.1.2 Multithreading on Beetle

Multithreading, to put it simply, refers to the process of executing multiple threads simultaneously. In our case, having multithreading on the beetles allows the overall communication system to be more responsive and achieve improved performance and concurrency. Furthermore, as we don't foresee having any chokepoints that might need a linear execution, there won't be any additional overhead computation time as well.

Though hardware level threading cannot be done (since beetles only have one core, capable of executing one instruction anytime (Alden, 2016)), we can instead do software side multithreading with the help of certain libraries. In the below section 4.1.5, we will discuss more about the tasks that will run in the beetles .

Section 4.1.3 Original design before Week 9 Evaluation

The original design before the week 7 evaluation was to use six beetles, two per trainee. In addition, we planned to use the BluePy library for bluetooth communication between the Arduino and the laptop.

Summary of the main flow

The program running in the laptop will first attempt to connect to the respective beetle through the Bluepy library by using the beetle's MAC address. Once the connection has been established, the respective beetle will start transmitting the data to the laptop through the 'Serial.print' function. Firstly, the beetle will be reading the sensor values and will be packing the data into a struct and then will be sending it to the laptop. In the laptop side, the data will be received by the 'handleNotification' function provided by the BluePy library where the data will be unpacked and the data attributes will be extracted. Below, we will further explain some of the key functions in the program.

Connection and reconnection function

As seen from figure 4.1.3-1 below, connection and reconnection of beetles were done using the same function called "establish_connection". In order to connect with a beetle, we first create a peripheral class. We then set the notification delegate to receive notifications from the beetle and append the peripheral object to a self created global peripheral list called "global_beetle". This is important for reconnection in cases of disconnection. Then, we set the notification delegate to receive data from the beetle. Once the connection is set up, we will call the "initHandshake" function to do a 2-way handshake where the laptop will first send a character to the beetle followed by the beetle sending back a time packet as an acknowledgement which consists of its timestamp data.

Any disconnections will be either detected by the BTLEDisconnectError exception or the lack of any data being received by the laptop over a predefined amount of time. In order to reconnect after any disconnections, firstly, the bluetooth connection set up with the beetle will be completely disconnected via the global_beetle[idx]._stopHelper() and the global_beetle[idx].disconnect() function. The peripheral object appended to the global peripheral list will be removed. Then, a new connection will be set up. Though this might take some time, we found this to be more reliable than other reconnection techniques and decided to stick with this.

```
def establish_connection(address):
    allow_imu[address] = False
    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:  # disconnect before reconnect
                        global_beetle[idx]._stopHelper()
                        global_beetle[idx].disconnect()
                        global_beetle[idx] = 0
                    if global_beetle[idx] == 0:  # just stick with if instead of else
                        print("connecting with %s" % (address))
                        # creates a Peripheral object and makes a connection to the device
                        beetle = btle.Peripheral(address)
                        global_beetle[idx] = beetle
                        # creates and initialises the object instance.
                        beetle_delegate = Delegate(address)
                        global_delegate_obj[idx] = beetle_delegate
                        # stores a reference to a “delegate” object,
                        # which is called when asynchronous events
                        # such as Bluetooth notifications occur.
                        beetle.withDelegate(beetle_delegate)
                        # total_connected_devices += 1
                        initHandshake(beetle)
                        print("Connected to %s" % (address))
                        allow_imu[address] = True
                        return
        except KeyboardInterrupt:
            print(traceback.format_exc())
            if global_beetle[0] != 0:  # disconnect
                global_beetle[0]._stopHelper()
                global_beetle[0].disconnect()
                global_beetle[0] = 0
            sys.exit()
        except Exception:
            print(traceback.format_exc())
            establish_connection(address)
            return
```

Figure 4.1.3-1 Connect and reconnect function

Sending data

```
void sendData()
{
  unsigned long tmp_recv_timestamp = 0;
  unsigned long tmp_send_timestamp = 0;
  if (Serial.available())
  { // Handshake
    msg = Serial.read();
    msg = (char)msg;
    if (msg == 'H')
    {
      TimePacket packet;
      packet.type = 2;
      packet.padding1 = 0;
      tmp_recv_timestamp = millis();
      packet.rec = tmp_recv_timestamp;
      tmp_send_timestamp = millis();
      packet.sent = tmp_send_timestamp;
      packet.padding2 = 0;
      packet.padding3 = 0;
      packet.checksum = getTimeChecksum(packet);

      Serial.write((uint8_t *)&packet, sizeof(packet));
    }
  }
  // else if (msg == 'A') {
  if (isReadyToSendData)
  {
    DataPacket packet;
    packet.type = 1;
    packet.yaw = int(round(ypr[0] * 100 * 180 / M_PI));
    packet.pitch = int(round(ypr[1] * 100 * 180 / M_PI));
    packet.roll = int(round(ypr[2] * 100 * 180 / M_PI));
    packet.accx = int(aaReal.x);
    packet.accy = int(aaReal.y);
    packet.accz = int(aaReal.z);
    packet.padding1 = 0;
    packet.padding2 = 0;
    packet.checksum = getChecksum(packet);

    Serial.write((uint8_t *)&packet, sizeof(packet));
  }
}
```

Figure 4.1.3-2 Function responsible for sending packets from Arduino

As seen from the above figure, in the Arduino, the packets were sent using a struct. The reason for doing so was because the BluePy library documentation recommended the use of structs for better and reliable transmission. There were two different types of packets that were sent. One was the time packet which would be sent at the handshaking process or when a reconnection happens. It carried the timestamp data to calculate the synchronisation delay between the trainees/dancers. But, after week 9, as we changed the way to calculate synchronisation delay, we did not had the need for Arduino's

timestamp data and removed it. The second packet was the IMU data packet where the yaw, pitch, roll, acceleration in the x axis, acceleration in the y axis and acceleration in the z axis were sent. Back then, the gyroscope values were not seen required and thus not sent. For both the packets, the packet ID was sent so that the packets can be distinguished in the laptop. In addition, both the packets also contained a computed checksum value by XORing the other values in the packet. This was to detect any data corruption. In the laptop, a dedicated function helped to check the checksum value by computing its own checksum value based on the data received and then comparing it with the received checksum value. Also, padding was used in both the packets to ensure that the size of the packets are always fixed to be the maximum packet length of 20 bytes. This is so that serialising and deserialising would be easier and less complicated to implement. Beyond that, if the packets are not 20 bytes, we also realised that there is a chance that two packets that are waiting in the buffer can be read as one chunk of 20 bytes packet.

Receiving data

Once the connection and handshake has been completed, the function, “getDanceData” will be called. As can be seen from figure 4.1.3-3, this function will call the “waitForNotification” function provided by the Bluepy library to receive the data from the beetle. If no data is received, it might mean a disconnection. In such a case, we have a manual way of counting in the “getDanceData” function before deciding to disconnect and reconnect using the connection function written above.

```
def getDanceData(beetle):
    waitCount = 0
    while True:
        try:
            if beetle.waitForNotifications(2):
                return
            else:
                waitCount += 1
                if waitCount >= 10:
                    waitCount = 0
                    establish_connection(beetle.addr)
                    return

        except KeyboardInterrupt:
            print(traceback.format_exc())
            if global_beetle[0] != 0:  # disconnect
                global_beetle[0]._stopHelper()
                global_beetle[0].disconnect()
                global_beetle[0] = 0
            sys.exit()
        except Exception:
            print(traceback.format_exc())
            establish_connection(beetle.addr)
            return
```

Figure 4.1.3-3 Function for receiving dance data

The “waitForNotification” function will trigger the “handleNotification” function which will receive the data from the beetle as a struct (since we are sending the data as a struct in the Arduino). As seen from figure 4.1.3-4 below, we first check for the size of the packet and ensure its length is 20 bytes. That is because in the Arduino, we ensured every packet being sent out is 20 bytes. If the packet is lesser than 20 bytes, a reassembly algorithm has been implemented where if the previous packet is also lesser than 20 bytes, both these packets will be merged and then processed if the length has now become 20 bytes. If not, the current fragmented packet will be kept in case the next packet coming in is less than 20 bytes. We did not consider the other scenarios as based on our experimentations, other scenarios happened very rarely and also required a lot of logic. Hence, we decided that it is not worth the effort to implement a complicated logic.

In the processing stage, the first byte will be extracted and based on its value, we will be able to see whether it is a handshake time packet (sent during the handshake phase) or it is a data packet. Based on this information, the packets are unpacked and then the values are extracted and stored in a buffer.

```
if size < 20:
    if buffer[address] != b"":
        packet = buffer[address] + data
        if len(packet) == 20:
            processData = True
            buffer[address] = b""
        else:
            buffer[address] = packet
    else:
        buffer[address] = packet

if (size == 20) or (processData == True):

    buffer[address] = b""

    packetUnpacked = False

    laptop_receiving_timestamp = time.time()

    try:
        packetType = struct.unpack("<h", packet[:2])[0]
        # time packet
        if packetType == 0:
            packet = struct.unpack("<hhLLhhL", packet)
            packetUnpacked = True
            if verbose:
                print(packet)
        # imu data packet
        elif packetType > 0:
            packet = struct.unpack("<hhhhhhhhhh", packet)
            packetUnpacked = True
            if verbose:
                print(packet)
    except Exception:
        print(traceback.format_exc())
```

Figure 4.1.3-4 Function that receives and unpacks the data

Issues and problems with code

However, soon we realised several issues and limitations with our approach. The biggest of them was reliability. Getting data from the six beetles concurrently without the risk of either one or multiple beetles getting disconnected was difficult even with our reconnection and reassembly functions. This was especially observed in crowded environments like the lecture hall where we felt the interference of other bluetooth devices such as phones and other beetles were causing our beetles to get disconnected frequently from the laptop.

In addition, in order to use the Bluepy library, one had to use a linux environment. Only one of us in the group had a built-in linux environment, For the others, we had to use virtualbox. Compared with a built-in linux environment, we felt that the use of virtualbox was not reliable and was also causing excessive delays and disconnections at times. The virtualbox even crashed without reasons at times during our testing phase.

In addition, we had also decided to use six beetles instead of three so as to accurately predict the position change of the dancers when they move. WIth three beetles, the reliability issues were still not so bad. But with six beetles, the number of disconnections and lost and fragmented packets were really very high. As such, we decided to look for a more reliable option and hence decided to use the bluno USB links for our bluetooth connections.

Section 4.1.4 New design after week 9 evaluation

Due to the reasons specified above, we decided to use bluno USB links, as seen in figure 4.1.4-1, for our bluetooth connection. Using the bluno USB links gave us very good reliability as long as the beetle is within a certain range from the bluno USB link. The number of lost or fragmented packets were very low such that we didn't even felt and had the need to handle any lost or fragmented packets. Thus, switching to bluno USB links instead of using the BluePy library significantly improved the performance of the bluetooth communication. And since internal communication affects other sections like external communications and machine learning, the use of bluno USB links increased the performance of the other sections as well.

Figure 4.1.4-1 Bluno USB link

The way this works is that, we first have to insert a bluno USB link on the laptop. Then, we have to switch ON the beetle. The beetle and link will auto connect without any code or human intervention. When the connection has been set-up, a blue light will blink in the USB link and a green light will blink in the beetle. Once that is done, we can run the code where the beetle will send data using the "Serial.print" function and the laptop will receive the data by using the "get_line" function in the pyserial library. Basically, the Arduino will transmit data using the "Serial.println" function. In the laptop, in order to send and/or capture any messages, we have to use the pyserial library.

Limitations with new design

There were definitely some issues and limitations with the new design. The first was that we had to do some BLE settings on both the beetles and the bluno USB links so that the right beetle can connect with the right bluno USB link. If not, if a number of beetles (that are ON) and bluno USB links (that are connected to laptops) are nearby, they will randomly connect with each other. This is not just troublesome for us (as we will not know which beetle is connected with which bluno USB link) but also other groups as their beetles can also get connected with our bluno USB links. The settings that we had to configure will be explained in section 4.1.10. Secondly, each bluno USB link can only connect with one beetle. So, to use six beetles, we needed six bluno USB links. We were only given one. We decided to loan three from other groups that were not using their assigned bluno USB links and brought two more since we were also allocated budget to buy extra materials. Third, in a crowded space like the lecture hall, the bluno USB link can take very long to connect with the beetle. We think this is because of the many bluetooth devices nearby and the interference of such devices. Nonetheless, we resolved that by doing our connection in an uncrowded and secluded place where there is not much bluetooth interference. In the School of Computing, the staircase leading to Dean's Office is usually not crowded and connecting our devices there takes seconds (as compared to up to two minutes in the lecture hall). So, whenever we needed to connect our devices, we went there.

As we were able to resolve all these limitations and constraints and the benefits of having good reliability was something we didn't want to lose, we decided to use bluno USB links instead of the BluePy library.

Section 4.1.5 Tasks running in Beetle

Communication tasks running in Beetle			
No.	**Name of Tasks**	**Bef Week 9**	**Aft Week 9**
		Priority Level	**Priority Level**
1	Transmitting data packets to its specified laptop	Most Highest	Equal
2	Receiving any data packets from its specified laptop (got removed after the week 9 evaluation)	2nd Highest	-
3	Reading sensor data and processing the data	High	Equal

	before they are ready to be transmitted to the laptop (For one special beetle that will be connected to a MyoWare muscle sensor, this task is also responsible for collecting EMG data)		

Figure 4.1.5-1 Table showing the different communication tasks running in the beetle before and after week 9 evaluation.

As seen from figure 4.1.5-1 above, before the week 9 evaluation, there were three tasks running in the beetles continuously. To ensure that the corresponding task runs at its assigned time, we used our main loop to check for the respective task's schedule time. However, after the week 9 evaluation, the number of tasks were reduced to just two and the priority levels were also changed. In addition, the tasks were also modified. We will first explain what the respective tasks do before explaining the changes that we made and the reason behind the change of design.

Task 1: Transmitting data packets to its specified laptop

The first task is responsible for mainly sending data packets to its specified laptop. Before week 9, there were three kinds of data that the respective beetles were transmitting to their specified laptop.

The first is processed IMU sensor data which will be later transmitted to the Ultra96 where the respective dance move and position will be predicted. Here, we did not used a poll approach where the laptop first sends a request for data to the respective beetles, followed by the beetles transmitting the data. Instead, we implemented a push approach where the beetles will send out data packets to their laptops as soon as the data packets are ready for transmission.

Moving on, the second kind of data packet that was transmitted from the beetles to their respective laptops was the 'ACK' packet when the respective laptops sent out a 'HELLO' packet to their corresponding beetles to initiate a connection. When we were using the Bluepy library before week 9, this was done at the start (before the dancers start performing their first move) or in the event of a reconnection, as part of a 2-way handshaking protocol.

Lastly, the third kind of data sent will be the EMG data. Before the week 9 evaluation, this was only applicable for the one special beetle that was connected to the MyoWare muscle sensor. In other words, except for this one special beetle, the rest of the beetles were only transmitting the other two kinds of data as explained above depending on the respective situations. And for the transmission of the EMG data, it will be done in a similar way as the transmission of IMU sensor data where the special beetle sends out the EMG data packet once it is available. As for the reason for choosing this approach for data transmission instead of the poll approach, it will be explained in the next section.

In addition, before the week 9 evaluation, this task was given the highest priority compared to the other tasks. This is so as it was necessary to establish a strong and stable connection with the laptop and send out the data packets to the laptop as soon as they are available especially once the trainees start performing their dance moves, to prevent any bottlenecks.

Now, we will talk about the changes made to this task after the week 9 evaluation. Firstly, instead of having two separate packets, one for the IMU data and another for the EMG data, we decided to combine both into one packet. The main reason for doing so was to further reduce any time delay in the bluetooth communication. Based on our experimentation using the bluno USB link after the week 9 demo, we realised that sending the EMG data as a separate packet reduced the frequency of the IMU data packets that were getting transmitted. In theory, it should not have any effect but in reality, the presence of EMG packets reduced the frequency of the IMU packet being transmitted by as much as 3 Hz which increased the time delay. Furthermore, the transmission of EMG packets also increased the number of fragmented packets that we were receiving at the laptop. Earlier, we mentioned how using the bluno USB links severely reduced any lost or fragmented packets. But with EMG packets, there was a notable increase of such lost and fragmented packets. Thus, we came to a conclusion that unlike the Bluepy library where the increase in size of a packet can cause the lost and/or fragmented packets to increase, for the bluno USB link, the increase in the number of packets getting sent almost simultaneously will increase the number of lost and/or fragmented packets. For the bluno USB links, even when the packet size was increased, it did not cause the number of lost and/or fragmented packets to go up. As such, we decided to use one packet for the IMU data values and the EMG values. Hence, we changed the Arduino program such that all the beetles, regardless whether they are connected to the EMG sensor or not, will be reading the value

from pin A0 (through analog) where the EMG sensor will be attached for one of the beetles. So, for that one beetle that is connected to the EMG sensor, the beetle will be receiving the EMG analog voltage value and sending this as part of the other IMU data values. For the other beetles, since they are not connected to any EMG sensor, they will not be receiving any EMG values and in the place of the EMG value, will be sending the value of 0 together with the other IMU data values. In the external communication code, this will get ignored. The second change we made was with respect to the transmission of the 'ACK' packet as part of handshaking or any reconnection. Instead of the laptop first sending a 'HELLO' packet, and then the Arduino sending out a 'ACK' packet, we changed the design such that when the Arduino prints out a certain sentence which in our case was, "Send any character to begin DMP programming and demo: ", it will be received by the laptop. The laptop will then send a random character (which "A" in our case) that will be received by the Ardunio. Then, the Arduno can start transmitting the data packets. The Arduino function where this takes place can be seen below in figure 4.1.5-2.

```
void setup() {
  // join I2C bus (I2Cdev library doesn't do this automatically)
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
  Wire.begin();
  Wire.setClock(400000); // 400kHz I2C clock. Comment this line if having compilation difficulties
#elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
  Fastwire::setup(400, true);
#endif

  Serial.begin(115200);
  while (!Serial);

  mpu.initialize();
  if (!mpu.testConnection()) {
    Serial.println(F("!MPU6050 connection failed"));
  }

  Serial.println(F("\nSend any character to begin DMP programming and demo: "));
  while (Serial.available() && Serial.read()); // get 'ACK' from laptop
  devStatus = mpu.dmpInitialize();
```

Figure 4.1.5-2 Arduino function which does the handshaking.

Due to this change, we were able to reduce any delay in handshaking and didn't had to have another packet for handshake. Furthermore, since the rate of disconnections was extremely small with the use of the bluno USB links, we also decided to not invoke any handshaking for reconnection. If a disconnection happens, both the beetle and the bluno USB link will auto connect and as long as the beetle was not switched OFF, the laptop will start receiving the IMU values from the beetle once the connection re-establishes itself automatically.

Task 2: Receiving any data packets from its specified laptop

The second task that was running in the beetles before the week 9 evaluation was receiving any data packets sent from the laptops to the beetles. To be more specific, there were two instances when the laptop needed to send a data packet to a beetle, hence creating the need for this task. The first was as part of the handshaking protocol, when the laptop first sent out a 'HELLO' packet to initiate the connection to the respective beetles. The second and final instance was our sole usage when we required the laptop to send a dummy value to the beetles as part of the testing phase to either send out a message or verify certain things. Due to the importance of handshake at the start or in the event of a reconnection, this task was given a higher priority than the last task.

However, after switching to the bluno USB link, as mentioned earlier, the handshaking process was changed. The only place where the laptop will send something to the beetle was the handshake. And since the handshake was only done at the start at the "void setup()" stage in the Arduino, there was no need for this process unless we wanted to send something to the Arduino for solely testing purposes. Furthemore, for our design, in the case of lost data, data corruption (identified if the computed checksum value by the laptop does not match the checksum specified in the data packet) or data fragmentation, the laptop is not going to send a request to the respective beetle to retransmit the packet again. The rationale behind this will be explained in section 4.1.8.

As such due to the above reasons, we felt this task was unnecessary and removed this.

Task 3: Reading sensor data and processing the data

Lastly, moving on to the final task, this task was responsible for receiving sensor data from the various sensors and processing them (for IMU sensor data) before they get transmitted to the laptop. To start off with, each of the six beetles will be receiving data from the one IMU sensor that each of them are connected to. In addition, one special beetle, as discussed before, will also be receiving data from the MyoWare muscle sensor that it is connected to. For the IMU data, once the respective beetles have received the data from the sensor, the next stage is processing these IMU sensor data. Under this stage, more specifically, the yaw, pitch and roll values obtained by the beetle were scaled so that they will be within the -180 and 180 range for the machine learning model. Before the week 9 evaluation, the data received by the respective beetles (which are of floating point type),

were converted to integer type by being multiplied by a fixed scalar value and then being subjected to zero offset to account for negative value. This was so that the number of bytes allocated for the respective attributes will be reduced. But after changing over to the bluno USB link, since packet size is not a concern anymore, we didn't had the need to scale the values unless needed by the machine learning model. For the EMG values, since the analog voltage value itself was sufficient, no processing was done.

Assigning priority to the different tasks

As said earlier, after week 9, we only had two tasks in the Arduino. Since connection between the Arduino and bluno USB link is guaranteed and is taken care of naturally by the beetle and the bluno USB link, we didn't had the need to give a higher priority to the first task anymore. Furthermore, in our design, after handshaking, transmission was only one-way from the beetle to the laptop. Thus, the only important thing that we needed to ensure was the transmission of the data packet once it is ready for transmission. For that, giving equal priority for both the collection of sensor values and the transmission of the collected sensor values was sufficient. This made the code much simpler to not just implement and also understand. In addition, it was a lot easier to debug the code as well.

Section 4.1.6 Components that make up the data packet that gets transmitted to the laptop from the beetle

The transmission of the data packets in the Arduino were done using the "Serial.print" function. For the transmission of the IMU values, we sent out 9 attributes. These were yaw, pitch, roll, acceleration in the x axis, acceleration in the y axis, acceleration in the z axis, gyroscope value in the x axis, gyroscope value in the y axis and lastly gyroscope value in the z axis. For the EMG values, we only sent out the absolute voltage value. For the beetles that were not connected with any EMG sensor, this value was just 0.

Furthermore, in the Arduino, we did not had the need to send the beetle ID to identify the beetles as it was taken care of by the external communications section. Also, there was no need for any transmission of the Arduino's timestamp data as well. To calculate the synchronization delay between trainees, the timestamp data of the laptops were used. In addition, the part whether a trainee is dancing or not was taken care of and done by the machine learning algorithms. Hence, we didn't had the need to send any computed flag variable, which stores whether it is the start of a new dance move or not.

As such, the only values that were sent out were the 9 IMU attributes, the EMG value, start byte, which was the character, “#” (to identify a new data packet in the laptop) and lastly a computed checksum value. The IMU calculated checksum value will be used by the laptop as a fingerprint to make comparisons against the checksum value that it computes, to detect any data corruption.

Section 4.1.7 Periodic Push by Arduino to Transmit Data between Beetle and Laptop

As discussed in the above section 4.1.5, data transmission of IMU values and EMG data were done using a periodic push by Arduino approach instead of a periodic poll by the laptop. In the latter approach, the laptop will have more control as to when and from which beetle, it needs the data packets. It can then send out a poll packet to signal the respective beetle to transmit the required data. However, besides the issue of Ardunio losing data due to its small memory space if polling is not done frequently, we also felt that this approach will incur more time than the former approach as there is going to be some time incurred in transmitting a poll request by the laptop and the beetle receiving the request and responding with the data packets. And there are also issues if this request gets lost. As such, we will implement a periodic push of data packets by the Ardunio to the laptop as soon as the packets are ready to be transmitted.

That being said, this approach has its own issues as well too. For instance, if too many data packets are transmitted to the laptop at the same time, the laptop might either be unable to handle them or not have enough space to store them even with a buffer. To prevent or at least reduce the occurrences of these situations, we designed the algorithm such that upon receiving a data packet, if there are no issues with the checksum value, the packet will be immediately sent to the external communications algorithm where the packets will be stored and sent in batches to the Ultra96. Despite it however, if the laptop is still unable to receive a packet at times (but it is very rare), there will be no retransmission of data packets between the beetle and the laptop, just like in situations of data corruption or fragmentation.

Section 4.1.8 No Retransmission of Data Packets

As discussed in section 4.1.5 and the previous section, in the event of any lost data packet, data corruption or fragmentation, the data packet will not be retransmitted by the respective beetles to the laptop again. There were three reasons why we made this decision.

The first is due to the need to reduce any latency as much as possible so that the rest of the stages such as the prediction of dance move can be done in a quick manner, without any bottlenecks.

The second reason behind our decision is that even if there are any issues with some of the data received, we feel that we will have sufficient valid datasets that we can use to accomplish the various functionalities of the system without a need for retransmission. This is so as we will be using a push approach to transmit one full dataset per transmission. As such, if there are issues with any one particular packet, we can still use other full datasets from either the previous or upcoming packets. And in the worst case scenario where if the data for a particular trainee is totally unavailable (due to the reasons explained), we can instead rely on datasets from other trainees and still produce a prediction and estimation of the synchronisation delay though this might affect our accuracy. However, based on our experimentation, the effect was small and so we proceeded to go with no retransmission.

And the last reason which was indeed the main reason for having no retransmission was the reliability provided by the bluno USB links. When we used to do bluetooth communication through BluePy library, there were severe unreliability issues such as disconnections and lost and fragmented packets. But after we switched to the bluno USB links, reliability greatly improved. In fact, as long as the beetle is not far away from the bluno USB link, the number of disconnections or any fragmented or lost packet were very low. Due to that, we decided that there was not a need for retransmission of packets. Also, we didn't find the need for any reassembly functions as well when a packet gets fragmented. As opposed, when we used the BluePy library, despite having a reassembly function, we were still unable to recover the whole packet at times.

Section 4.1.9 Task management using loops instead of FreeRTOS and Protothread

In section 4.1.5 above, we have discussed the various tasks that will be running in the beetles. In order to properly schedule the two tasks that were explained, a loop was sufficient.

Back before the week 9 evaluation, we had three tasks and a decision had to be made regarding which library to utilize to ensure that the tasks get scheduled to run when they need to.

The two libraries considered were the FreeRTOS library which is included in the Arduino sketches and the “ArduinoThread" library which could be used to implement the Protothread mechanism. Considering that the ATmega328P chip on the beetle has only a maximum size of 2 Kbytes of SRAM and 32 Kbytes of flash memory (*Arduino Memory Information*, 2018), we decided not to use FreeRTOS for any scheduling capabilities as the FreeRTOS library incurs high RAM requirements (*FreeRTOS Support*, 2020). Using the FreeRTOS library also required us to have high RAM usage which might adversely affect the performance of the beetles. There was also a possibility of the beetles transmitting garbage data or even stop working due to insufficient SRAM space.

As such, we went with our next choice which was to implement the ProtoThread mechanism using the “ArduinoThread” library. Protothreads work by having the application to repeatedly call a protothread function which will run till it exits or blocks, thus giving us a “feeling” of multi-threading. Therefore, the “ArduinoThread” library allows us to create and schedule many parallel tasks with a fixed or variable time defined between runs. As a protothread does not require its own stack, protothreads are lightweight and allows us to create multiple “threads” in memory constrained environments like in our case (Alden, 2016). And indeed before the changes were made after week 9, we used the Protothread library.

But after the changes were made, there was no need for scheduling as all the two tasks were given equal priority. Using a protothread for that seemed an overkill especially since a loop can carry out the same thing without any inefficiency. Hence, we decided to just use a loop to run both the tasks. The way this was done would be shown in section 4.1.11.

Section 4.1.10 Setup and configuration for BLE interfaces

In our design, the bluno USB link will be the central device and the Arduino beetles will be the peripheral devices.

To configure the BLE on the Arduino beetle, the following steps were done on the Arduino IDE (*DFRobot*, n.d.):

1) Inside the Arduino IDE, under Tools->Port, choose the correct serial port.
2) Open the Serial monitor (by clicking the upper right corner of the IDE window) and under the pull-down menu, select the "No line ending" and “115200 baud” options.
3) Type "+++" and press the send button.
4) If the AT Command Mode is successfully accessed, the message "Enter AT Mode" will be displayed in the monitor.
5) Under the pull-down menu, select the "Both NL & CR" and “115200 baud” options.
6) To set the Arduino Beetle to the default peripheral settings, enter the command, “AT+SETTING=DEFPERIPHERAL” and press the send button. If the message “Ok” appears in the monitor dialog window, that means that the BLE has been successfully configured.
7) Then, the command “AT+MAC=?” was entered to know the MAC address of the respective beetle.
8) Then, to ensure that the beetle only connects to a specified bluno link, we had to bind the beetle to the bluno link with the link’s MAC address (found using “AT+MAC=?” for that link). Once the link’s MAC address is known, the command “AT+BIND=MAC” was entered with MAC being the MAC address of the link.
9) Then, the command, "AT+CMODE=UNIQUE" was set to make the connection binding.
10) The command, “AT+EXIT” was entered to exit the AT mode

To configure the BLE on the bluno USB link, the following steps were done on the Arduino IDE (*DFRobot*, n.d.):

1) Inside the Arduino IDE, under Tools->Port, choose the correct serial port.
2) Open the Serial monitor (by clicking the upper right corner of the IDE window) and under the pull-down menu, select the "No line ending" and “115200 baud” options.
3) Type "+++" and press the send button.
4) If the AT Command Mode is successfully accessed, the message "Enter AT Mode" will be displayed in the monitor.
5) Under the pull-down menu, select the "Both NL & CR" and “115200 baud” options.
6) To set the Arduino Beetle to the default peripheral settings, enter the command, “AT+SETTING=DEFCENTRAL” and press the send button. If the message “Ok”

appears in the monitor dialog window, that means that the BLE has been successfully configured.

7) Then, the command “AT+MAC=?” was entered to know the MAC address of the respective beetle.
8) Then, to ensure that the bluno link only connects to our specified beetle, we had to bind the bluno link to the beetle with the beetle’s MAC address (found using “AT+MAC=?” for that beetle). Once the beetle’s MAC address is known, the command “AT+BIND=MAC” was entered with MAC being the MAC address of the beetle.
11) Then, the command, "AT+CMODE=UNIQUE" was set to make the connection binding.
12) Lastly, the command, “AT+EXIT” was entered to exit the AT mode

Previously, in order to run the BluePy library for the beetle-laptop communication, we required a Linux operating system to run the individual laptops. Back then, we used Ubuntu hosted within VirtualBox as it was lightweight, easy to use and install. However, it caused delays and also crashed unexpectedly many times. However, with the use of bluno USB links, we don’t really need to use a Linux operating system. However, to use Windows, slight changes have to be made to the laptop python code where serial ports are specified.

Section 4.1.11 Program flow

Arduino

```
void setup() {
  // join I2C bus (I2Cdev library doesn't do this automatically)
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
  Wire.begin();
  Wire.setClock(400000); // 400kHz I2C clock. Comment this line if having compilation difficulties
#elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
  Fastwire::setup(400, true);
#endif

  Serial.begin(115200);
  while (!Serial);

  mpu.initialize();
  if (!mpu.testConnection()) {
    Serial.println(F("!MPU6050 connection failed"));
  }

  Serial.println(F("\nSend any character to begin DMP programming and demo: "));
  while (Serial.available() && Serial.read()); // get 'ACK' from laptop
  devStatus = mpu.dmpInitialize();
```

Figure 4.1.11-1 Function showing handshake in the beetle

As explained earlier, before the laptop program runs, it is vital to ensure that both the beetle and the bluno USB link have connected. This can be verified once a blue light blinks in the USB link and a green light blinks in the beetle. Once that is done, the laptop program can start to run.

As seen from figure 4.1.11-1, the beetle will print out, “Send any character to begin DMP programming and demo: " which will be captured by the bluno USB link and subsequently by the laptop. The laptop will then send out a random character (character “A” in our case) to the beetle. And upon receiving the character, the beetle will do the other configurations and start to send out the data.

```
void loop() {
  // if programming failed, don't try to do anything
  if (!dmpReady) return;

  mpu.resetFIFO();
  fifoCount = mpu.getFIFOCount();
  while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

  while (fifoCount >= packetSize) {
    mpu.getFIFOBytes(fifoBuffer, packetSize);
    fifoCount -= packetSize;
  }

  mpu.dmpGetQuaternion(&q, fifoBuffer);
  mpu.dmpGetGravity(&gravity, &q);
  mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
  mpu.dmpGetAccel(&aa, fifoBuffer);
  mpu.dmpGetLinearAccel(&aaReal, &aa, &gravity);
  mpu.dmpGetGyro(&gyro, fifoBuffer);
```

Figure 4.1.11-2 Collection of IMU data

```
    Serial.print("#");

    Serial.print(ypr[0] * 180 / M_PI);
    Serial.print(",");
    Serial.print(ypr[1] * 180 / M_PI);
    Serial.print(",");
    Serial.print(ypr[2] * 180 / M_PI);
    Serial.print(",");

    Serial.print(gyro.x);
    Serial.print(",");
    Serial.print(gyro.y);
    Serial.print(",");
    Serial.print(gyro.z);
    Serial.print(",");

    Serial.print(aaReal.x);
    Serial.print(",");
    Serial.print(aaReal.y);
    Serial.print(",");
    Serial.print(aaReal.z);

    Serial.print(",");
    Serial.print(analogRead(EMG));
    Serial.print(",");

    int checksum = int(ypr[0] * 180 / M_PI) ^ int(ypr[1] * 180 / M_PI) ^ int(ypr[2] * 180 / M_PI) ^
      gyro.x ^ gyro.y ^ gyro.z ^ aaReal.x ^ aaReal.y ^ aaReal.z ^ int(analogRead(EMG));
    Serial.println(checksum);

    Serial.print("\n");

    // Experimentally determined to send data at a reliable 25Hz
    delay(27);
}
```

Figure 4.1.11-3 Sending the data packet

As can be seen from figure 4.1.11-2, the collection of the sensor values will be done in the loop. After which, as can be seen from figure 4.1.11-3, the values will be sent together with a start byte (“#” in our case) and a computed checksum value. For the beetle connected with the EMG sensor, the “analogRead(EMG)” will be the absolute EMG voltage value. For the other beetles not connected to an EMG sensor, this value will simply be ignored by the external communications program. The individual data attributes are split by “,” so that they can be split into their respective attributes in the laptop. Finally, “delay(27)” is used to collect and send data at 25 Hz.

Laptop

```
import argparse
import time

import serial

SERIAL_PORTS = ["/dev/ttyACM0", "/dev/ttyACM1", "/dev/ttyACM2"]


def check(line):
    try:
        yaw, pitch, roll, gyrx, gyry, gyrz, accx, accy, accz, emg, cksum = line.split(",")
        val = int(yaw) ^ int(pitch) ^ int(roll) ^ int(gyrx) ^ int(gyry) ^ int(gyrz) ^ int(accx) ^ int(accy) ^ int(accz) ^ int(emg)
        line = ""
        if (val == int(cksum)):
            line = yaw + "," + pitch + "," + roll + "," + gyrx + "," + gyry + "," + gyrz + "," +
                accx + "," + accy + "," + accz + "," + emg
            return line
        else:
            return line
    except Exception as e:
        print (e)
        return line

class IntComm:
    def __init__(self, serial_port):
        self.ser = serial.Serial(SERIAL_PORTS[serial_port], 115200, timeout=0.5)
        self.ser.reset_input_buffer()
        self.ser.reset_output_buffer()
        self.ser.flush()
        while True:
            line = self.ser.readline()
            if b"Send any character to begin DMP programming and demo:" in line:
                self.ser.write("A".encode())
        print("Opened serial port %s" % SERIAL_PORTS[serial_port])

    def get_line(self):
        ln = self.ser.readline().decode().strip()
        if (ln[0] == "#")
            if (check(ln[1:]) != ""):
                print ("checksum passed")
                return ln
        else:
            return self.get_line()
```

Figure 4.1.11-4 Intcomm.py running in the laptop

In the laptop, the program that we run can be seen in the above figure 4.1.11-4. In the program, we define the IntComm class. The function “serial.Serial" from the pyserial library opens the serial port where the bluno USB link is connected. Here , “115200” refers to the baud rate that we are sending while “0.5” refers to the timeout. This means that for subsequent reading of data from the serial port, the program will only wait for 0.5 seconds and if the buffer is still empty, then the program will proceed to the next line and will not forever wait for any input to arrive at the serial. After which, we flush both the input and output buffer. This is so that any old data from previous communication that got left behind can get flushed. Then, we run the while loop to receive the line, “Send any character to begin DMP programming and demo: " from the beetle which will signify the handshake. Once we receive the line, we will send the character, “A” as an acknowledgement. Once the beetle receives it, it will start sending the data packets.

In order to capture the data packets, the function, “get_line(self)” needs to be called. This function will first attempt to read a line within 0.5 seconds. Then, it will check for the start byte, “#”, which will indicate a new packet. It will then pass this line to another function called “check()” where if the computed checksum matches the checksum in the packet, the line will be deemed as a good packet. The function will return a new line without the checksum value. Otherwise, if the checksum value does not match or there is some error captured in the try catch block, the function will return an empty string and the whole line will be discarded and the “get_line(self)” will be called again.

```
if __name__ == "__main__":

    # Arguments
    parser = argparse.ArgumentParser(description="Internal Comms")
    parser.add_argument(
        "--serial", default=0, help="select serial port", type=int, required=True
    )
    args = parser.parse_args()
    serial_port_entered = args.serial

    # Initialise intcomm
    int_comm = IntComm(serial_port_entered)

    count = 0
    start = time.time()

    while True:
        line = int_comm.get_line()
        count = count + 1

        print(line)

        if count % 100 == 0:
            end = time.time()
            print("Receiving data at %f Hz" % (100 / (end - start)))
            start = time.time()
```

Figure 4.1.11-5 Main function

As can be seen from figure 4.1.11-5, the main function will capture the input from the user, so that the right serial port, where the USB bluno link is connected, will be opened. Then, the IntComm class will be created and the “int_comm.get_line()” will be called.

No reconnection function

We don’t have a reconnection function but that is because reconnection is naturally handled by the bluno USB link. As long as the Arduino did not get switched OFF, when there is a disconnection, the Arduino will be sending out the data packets but will not be captured by the bluno USB link and the subsequently by the laptop till the bluno USB link manages to connect back with the Arduino again. Once the connection has been established again, the laptop can again receive the packets. No termination or restart of

the program is required unless the Arduino gets switched OFF. In that case, because the handshake needs to be done again, the program needs to run again.

That being said, the rate of disconnections is very low. In our week 11 and 13 evaluation, we did not experience any disconnection either. As such, the bluetooth connection with a bluno USB link is very strong and reliable.

Communication between External and Internal Communication

In the external communication part, a separate file called “Laptop.py” is used and this file communicates with the internal communication by calling the Intcomm.py file, creating the IntComm class and then calling the “get_line()” function under the “collect_data()” function to receive the data as can be seen from figure 4.1.11-6.

```
class Laptop:
    def __init__(self):
        self.intcomm = IntComm(IS_POSITION)
        self.buffer = []
        self.counter = 1
        self.start_time = time.time()

    def collect_data(self):
        # data: #yaw,pitch,roll,gyrx, gyry, gyrz,accx,accy,accz,emg
        try:
            data = self.intcomm.get_line()
```

Figure 4.1.11-6 Laptop.py calling the IntComm.py

In the “Laptop.py” file, the EMG value for the beetles that are not connected to an EMG sensor will be ignored. The IMU values and the EMG value (for the beetle that is connected with an EMG sensor) are sent to the dashboard and the Ulta96 in the “Laptop.py” file.

Section 4.1.12 2-way Handshake

In our design, we will use a 2-way handshake. The Arduino will first print out, “Send any character to begin DMP programming and demo: ". The laptop’s python program uses the pyserial library which will capture this in the serial buffer. Upon receiving this line, the laptop will send a random character, “A” in our case. And upon receiving this, the Arduino will start sending the data packets. Upon establishing connection, the respective beetles can start transmitting data and don’t need to wait for other laptops and beetles to complete

their handshake as we don't feel the need to since the respective beetles will transmit data once it is available and don't wait for anything such as a laptop poll request.

As mentioned earlier, other than the handshaking phase, the laptop does not send any packets to the Arduino at all. We hence felt that the laptop does not need to know whether the beetle has acknowledged its acknowledgement. Anyways, after the handshake, the beetle will start transmitting data immediately as well. Hence, we established a 2 way handshake instead of a 3-way handshake, to reduce connection overhead.

Section 4.1.13 No processing of IMU sensor data as no need to fit data packet within 20 bytes

Previously, before the changes after the week 9 evaluation, the IMU sensor values received by the beetles had to undergo some encoding before they were transmitted to the laptops. To be more specific, these data values had to undergo two stages of encoding before being packaged into a packet. The first involved converting the data to integer type and the second involved adding a zero offset.

The IMU sensor values are by default, floating point values. If these original values are sent without modification, then one additional byte will be required for each of the data values that are not of integer type and have a decimal point. However, because the maximum bytes supported by the BluePy library were 20 bytes after accounting for overheads (*Stack Overflow*, n.d.), we can't afford to waste any precious space and had to think about how to properly squeeze values belonging to one full dataset within a packet. As such, in the first stage of encoding, we decided to convert these values into integer type by scaling them by an appropriate value so that the decimal part is retained as much as possible, while still being able to fit it within a packet. Once the first stage of encoding was done, these data values then went through the next stage of encoding where a zero offset was included so that negative readings can also be transmitted without the need for additional bytes. In addition, padding was also added to ensure that each reading value is of their respective required bytes.

However, after switching to bluno USB links, we didn't have the requirement to ensure that the packet is not more than 20 bytes. In fact, using an Android app or bluno USB link, we tested and found out that one can send packets more than 20 bytes. So, after week 9, we

removed the encoding and directly sent the values. That made the code much easier to understand, modify and debug.

Section 4.1.14 Checksum

To ensure that the laptop receives the correct readings, a checksum value was calculated and used. We considered two algorithms to calculate the checksum.

The first approach was using an 8 bit algorithm by DIGI. Under this approach, we first have to add all the byes of the packet except the start delimiter and the length. Then, we have to keep the lowest 8 bits from the result. We then have to subtract this quantity from 0xFF. On the laptop side, we have to add all the bytes including the checksum and don't include the delimiter and length. If the checksum is correct, then the last two digits on the far right of the sum would be 0xFF. When we used this approach, we realised that there can be issues with this algorithm at times, For instance, the value of the last two digits can still be 0xFF despite adding different values. The main issue with this algorithm is that it is only checking for the last byte as the reference point and errors can still be undetected.. Hence, at the end, we decided to go with the simpler approach of using XOR.

Under this approach, we XOR each byte in the data packet. This value was then appended at the end of that respective data packet before getting transmitted to the laptop. At the laptop's end, a function took care of computing the checksum value based on what it has received and then subsequently compared it with the stored checksum value in the packet to detect any corrupted data. This approach was much simpler to code and understand than the other approach.

Section 4.1.15 Packet Format

As said earlier, since we are using the bluno USB link, we don't have to ensure that the packet size is less than 20 bytes.

Data	Bytes used
Start byte	1
Yaw	2
Pitch	2
Roll	2

Accelerometer	x	2
	y	2
	z	2
Gyroscope	x	2
	y	2
	z	2
EMG		2
Checksum		2
Total bytes used: 23 bytes		

Figure 4.1.15-1 Table showing the format of the data being sent

As seen from figure 4.1.15-1 above, the start byte, “#” takes up 1 byte. Two bytes are used for yaw, pitch, roll, each accelerometer attribute and each gyroscope attribute. In addition, checksum takes up 2 bytes as well. As said earlier, we will not be transmitting any timestamp data (since synchronization delay is computed using the laptop timestamp data), beetle ID (since taken care by external communication) and start of dance flag (since taken care by machine learning code).

Section 4.1.16 Packet Types

The only special packet type used is the ‘ACK’ packet by the laptop as part of handshake which is simply just, “A”. Apart from that, we are not using any ‘HELLO’ packet. We will be sending both the IMU and EMG data in the same packet as seen above.

Section 4.1.17 Baud Rate & Connection Rate

Baud rate refers to the rate at which a particular data can be transmitted over a communication channel. In our design, both the beetle and its respective laptop need to be of the same baud rate to send and receive data at the same rate. We have chosen our baud rate to be 115200 bps to ensure the right balance between having a fast enough transmission speed to attain low latency, but not too high such that the receiving party does not receive the packet at all or is unable to receive the packet properly.

The connection rate is defined as the number of packets transmitted per second. In our design, due to the requirements of the machine learning model, we will be setting the connection rate to be 25 Hz or in other words, 25 packets per second. Based on our experimentation, 25 Hz also causes no significant lost or fragmented packets.

Section 4.1.17 No Packet Reassembly

Before the changes after the week 9 evaluation, we had implemented a packet reassembly function. If a data packet is less than 20 bytes, and if the previous packet is also less than 20 bytes, then both these packets were merged and then processed if the length became 20 bytes. If not, the previous fragmented packet was discarded and the current fragmented packet was kept in case the next incoming packet is less than 20 bytes. We did not consider the other scenarios as based on our experimentations, other scenarios happened very rarely and also required a lot of logic. Hence, we decided that it was worth the effort to implement a complicated logic.

After the changes were made, as explained earlier, the rate of data fragmentation was very low. So, we did not feel the need to have a packet reassemble function in the laptop. Whether a packet has been received properly or not is determined by two things: the start byte, "**#**" and the correct checksum. If a packet does not satisfy them, it is simply discarded.

Section 4.1.16 Ensuring reliability & robustness

Just like any other communication protocol, reliability and robustness is essential and in the above sections, we have indeed discussed some of the ways in which this will be achieved in our design.

As mentioned in the above sections, to ensure data verification, checksum value will be appended at the end of every data packet and will be affirmed on the laptop's side. In the event of obtaining an incorrect checksum value (suggesting data corruption), data fragmentation or lost data, we will be simply discarding the data. We don't have a data reassembly function or a retransmission function in the event of such cases. However, that is because using the bluno USB link, the rate of such cases happening were really low. And as such, the occurrences of these cases did not had any notable impact on our performance and prediction. Given that, having a retransmission or a reassembly function would not have resulted in a lot of benefits but rather might have caused more

disadvantages by increasing the time delay and possibly creating a bottleneck for the rest of the stages like external communication. A reassembly function, if not coded properly, could also cause error in one packet to cascade into the subsequent good packets, thus corrupting them as well. As such, we decided that it was best to not have any retransmision or reassembly functions. In addition, we also used an appropriate baud rate of 115200 bps which reduced the occurrences of lost or invalid data packets.

On a related note, there can be instances when the connection between a beetle and its respective laptop might get disrupted and lost. In such circumstances, we don't have an explicit reconnection function, but that is again because of the use of the bluno USB links. As said earlier, connection and reconnection is auto handled by the bluno USB links. As such, when a disconnection happens, as long as the Arduino has not been switched OFF, after some time, the connection between the Arduino and the bluno USB links will get reestablished again without any code or human intervention. If the Arduino gets switched OFF, then we will need to run the python program in the laptop again so that the handshake can be carried out again. This can be inconvenient. Based on our experiments, we realized that the Arduino can only get switched OFF on its own if the batteries are weak or if the wires are loose. To fix that, the hardware sensor in-charge, Ting Wei along with machine learning in-charge Brandon and I made some changes to the hardware design to further reduce the chances of loose wires and also frequently changed the batteries. Besides, the bluno USB link itself guarantees reliable bluetooth connection most of the time. As such, after the above efforts, there was almost no disconnections or reconnections, compromising the bluetooth communication.

Lastly, on a similar note, in order to prevent any exceptions from crashing and terminating our program, we identified exceptions in our testing phase and handled them so as to prevent any disruption in our program flow, and hence improving the robustness of the overall system.

Section 4.2 External Communications

Section 4.2.1 Introduction

External Communication is the connection bridge for each component. The external communication will help transfer the data from dancers' laptops to the Ultra96 and send the processed data to the dashboard for the monitoring and evaluation server. With the help of time stamps, it shall be able to calculate synchronization delay for the three dancers.

Section 4.2.2 Communication between laptops and Ultra96

Transmission Control Protocol (TCP) will be used to build connections between the laptops and Ultra96. TCP protocol is a stable connection protocol that ensures the message accuracy between the client side and server side. TCP ensures that the connection

between the laptops and Ultra96 is stable and consistent. For each dancer laptop, there will be a TCP client running on the dancer's laptop which sends dancer's movement data regularly. On the Ultra96, there will be three server threads under the Ultra96 process which receive the data from the dancers' laptops.

Section 4.2.3 Communication between Ultra96 and Dashboard/Evaluation Servers

Transmission Control Protocol(TCP) will also be used for the communication between Ultra96 and Evaluation Server. After processing the data using machine learning and FPGA, the prediction result and positions will be sent to the evaluation server.

For the communication between Ultra96 and Dashboard, we use the message queue. When we receive the data from different clients and processed these data, the

Section 4.2.4 Detailed Message Format

1. The detail message format for the message sent from evaluation client to the evaluation server: "**positions[0] positions[1] positions[2] | dance_move | Synchronization Delay**"
2. The detail message format for the message sent from dancers' laptops to the Ultra96: "**Dancer ID | Is Position| yaw, pitch, roll, gyrox, gyroy, gyroz, accx, accy, accz, emg|**"
3. The detail message format for the message sent from the Client to the dashboard :" **yaw, pitch, roll, gyrox, gyroy, gyroz, accx, accy, accz, emg**"
4. The detail message format for the message sent from the Ultra96 to the dashboard : "**positions[0] positions[1] positions[2] | dance_move | new_positions[0] new_positions[1] new_positions[2]|sync_delay|accuracy**"

Section 4.2.5 Libraries and APIs

Library	**APIs**	**Purpose**
socket	socket.socket(), socket.connect(), socket.sendall(), socket.recv()	Build TCP connections between client side and server side.
Crypto	AES Random	Encryption and decryption of messages.
threading	threading.event() threading.Timer()	Build multiple threads under one process.
time	time.time()	Create the time stamp, which will be used to

		calculate the synchronization delay.
twisted	twisted.internet twisted.internet.protocol twisted.protocols.basic	Build stable TCP connection between clients and servers.

Python socket library: will be used to build a server socket and multiple client sockets. The socket library is mainly to build TCP connections between client side and server side.

Python Crypto library: will be mainly used for the encryption and decryption of messages.

Python threading library: will be used for the multithreading purpose. Since there are three dancers' laptops sending messages to the Ultra96 simultaneously , the threading library will help create three different server threads and ensure that the messages sent from three dancers' laptops can be received properly.

Python time library: will also be used to create timestamps for the synchronization purpose.

Python twisted library: twisted library provides a better solution for the TCP connection which enables the auto reconnection ability when some of the clients are actually disconnected.

Section 4.2.6 Processes running on the Ultra96

Processes created on Ultra96		
Process Name	Process Purpose	Priority
Ultra96:	This process is the server process that receives the data sent from the three dancers'	Highest

	laptops. This process will have three threads which connect to the three different dancers' laptops respectively. Besides that, this process will also pass the collected data from three different dancers to the ML process and receive the results of dance move detection from different dancers respectively. This process will also send data to the eval server and dashboard.	
ML:	This process is for the machine learning part. After receiving the data from the dancers laptops, this process will determine the dancers' positions and dance moves. Afterwards, the final results will be sent to the Ultra96 process.	High

Section 4.2.6 Clock Synchronization and Dance Synchronization Delay

Network Time protocol will be used for the clock synchronization. Whenever the dancer starts a new move, the ML process will detect the start of the dance move based on the data sent from laptops. A proper threshold is set to determine whether this is a starting of the dance move.

The clock of Ultra96 will be used as the reference clock. To calculate the synchronization delay between different dancers, it is necessary to synchronize the clocks since different dancers' laptops may have different time delays. While it is hard to synchronize the clocks of Ultra96, dancers' laptops and beetles, I will split the whole system into two parts, which are beetles(Client) and dancer's laptop(Server) and the dancer's laptop(Client) and Ultra96(Server). For each part, the Client will send the timestamp at t0 and the Server will receive the time stamp at t1. Afterwards, the Server will send back the timestamp at t2 and the Client will receive the timestamp at t3. Hence we can calculate the Round Trip Time (RTT) and clock offset:

Round Trip Time:RTT = t1-t0+t3-t2

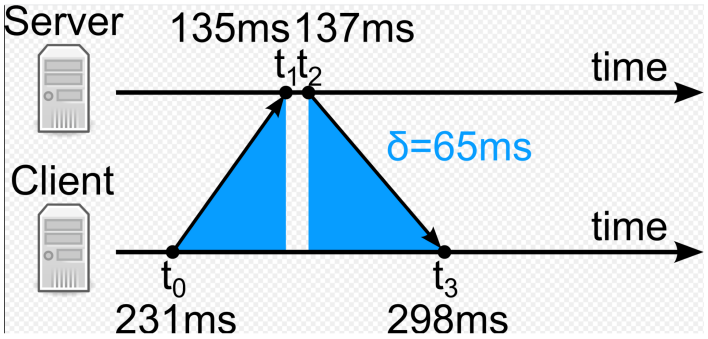
Clock Offset: Offset = t1 - t0 - RTT/2.

Detailed code implementation is as follows:

```
raw_data = get_raw_data()
t1 = time.time()
print("t1: " + str(t1))
my_client.send_message(str(dancer_id) + '|' + str(RTT) + '|' + str(offset) + '|' + str(raw_data) + '|')
timestamp = my_client.receive_timestamp()
t4 = time.time()
print("t4: " + str(t4))
t2 = float(timestamp.split("|")[0])
print("t2: " + str(t2))
t3 = float(timestamp.split("|")[1])
print("t3: " + str(t3))

RTT = (t4 -t3 + t2 - t1)
print("RTT: " + str(RTT))
offset = (t2 - t1) - RTT/2
print("offset: " + str(offset))
```

To calculate the starting time of the dancer's movement, we simply need to record the time when the timestamp package arrived at Ultra96 as **T**, and use the **T** to minus the clock offset between Ultra96 and dancer's laptop and the half of RTT between dancer's laptop and beetles.



Network Time Protocol

Detailed code implementation is as follows:

```
def calculate_ultra96_time(beetle_time, offset):
    return beetle_time - offset
```

After multiple testing based on the dancer's different dancing locations. It turns out that the different locations do not make a huge impact on the delay. Hence, we will not constantly

update the RTT and offset for each dancer. Instead, we will only update the offset when it is necessary.

To get the actual sync delay, the system will use the offset to calculate the starting time based on the Ultra96. With the three starting times, the sync delay will simply be the absolute difference between the earliest dance move and latest dance move.

Detailed code implementation is as follows:

```
def calculate_sync_delay(start_time0, start_time1, start_time2):
    if start_time0 and start_time1 and start_time2:
        return abs(
            max(start_time0, start_time1, start_time2)
            -
            min(start_time0, start_time1, start_time2)
        )
```

Section 5 Software Details

Section 5.1 Software Machine Learning

This section covers the steps for data segmentation, feature exploration, and machine learning algorithms to detect the different dance moves. It also covers the thresholds set to detect the dance positions. It also highlights the differences in the initial method and the final implementation. Finally, this section covers the system integration of the model. The main changes to the initial method are summarized below.

- Added a second wearable device on the waist to detect dance positions
- Measured left and right turns instead of lateral movements to detect dance position
- Implemented DNN instead of CNN to infer dance moves

Section 5.1.1 Initial Method (Dance Moves)

The initial method for detecting dance moves was using deep learning models. The first model was a Dense Neural Network (DNN) trained on features extracted from a non-time series dataset. The second model was a Convolutional Neural Network (CNN) trained on a time series dataset. CNN was previously preferred due to its ability to learn pattern and sequences of signal theoretically.

The first issue with this design was that CNN had a large model architecture. This meant that a lot of more training data was required to train a CNN. DNN was found to achieve the same accuracy as CNN with the same amount of training data. Moreover, DNN was found to perform much better than CNN during integration testing.

The second issue with this design was the CNN required longer inference time due to its large model architecture. This resulted in higher power usage as compared to the power usage for DNN. CNN did not provide a better predictive ability than DNN and consumed more power.

Section 5.1.2 Initial Method (Dance Positions)

The initial method for detecting dance positions was to set a threshold to detect a left step or right step using the acceleration values. One imu sensor on the right hand was used to detect both dance move and dance positions. A time frame was set to capture the left and right steps to predict the dance positions. Note that the initial design report mentioned that there was a second wearable on the ankle for detecting dance position. However, this was not followed through in the early stages of implementation due to bluetooth issues.

The first issue with this design was the acceleration values were not responsive. When the dancer stops dancing, the acceleration readings take a while to stabilize. This resulted in false triggers in left and right steps if the acceleration readings from the previous window ate into the current window.

The second issue with this design was that the left and right steps to detect the dance positions were not very accurate. Each dancer had different ways of taking left and right steps and was very hard to generalize. Left and right turns were easier to generalize and accurate to determine the dancer positions.

The third issue with this design was that the time frame to detect when the dancer took a left and right turn was very difficult. Each dancer again had different reaction times, and the left and right turns might be missed if the dancer reaction was too fast or slow. A second imu sensor on the waist to detect the turns without time frame was found to be more accurate. Note that the final stage of implementation includes a second wearable on the waist.

Section 5.1.3 Data Segmentation

Each dance move required a lot of hand movements that were distinct from one another. Imu readings from the hand were sufficient to classify the dance moves. The model received a window consisting of 9 types of readings to predict the dance move. They were the 3-axial acceleration, 3-axial gyroscope and 3 orientation angles.

The imu sensor sampled stable readings at 25Hz with the bluetooth link. Experimentally, it was found that setting a window size of 60 readings provided the best features to train the model. The model was provided about 2.5 seconds of readings to infer the dance moves. To train the model, a window of 60 readings with 50% overlap was employed to train the model. Figure 5.1.3-1 illustrates the sliding window with 50% overlap.

Section 5.1.4 Feature Extraction

The DNN model to predict dance moves took in an array of features extracted from a window of 60 readings. The features extracted are listed in Table 5.1.4-1 below and are classified as time-domain features and frequency-domain features. These are also common feature extraction methods for human activity recognition problems.

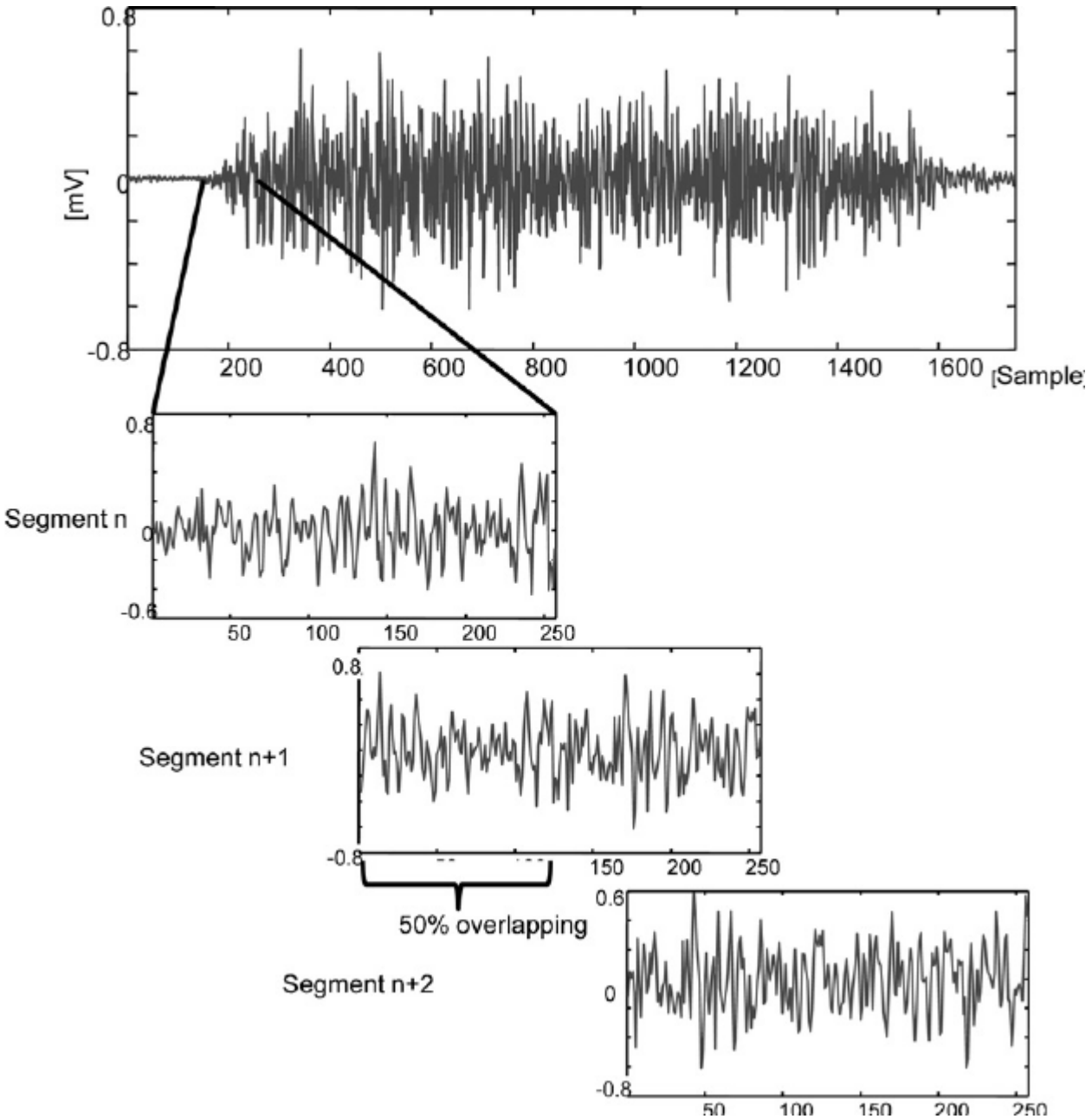


Fig 5.1.3-1 Sliding window with 50% overlap (Orosco et al., 2018)

Category	Feature
Time	Mean
	Variance
	Mean absolute deviation
	Root mean square
	Zero crossing rate
	Interquartile range

	75th percentile
	Kurtosis
	Min-max
	Signal magnitude area
Frequency	Spectral centroid
	Spectral energy
	Spectral entropy
	Principal frequency

Table 5.1.4-1 Features extracted to infer dance moves

Boxplots and T-SNE were the data exploration techniques used to decide the features to extract to train the model. Boxplot helps to visualize the spread of the data and determine whether the feature will be useful in differentiating some dance moves. On the other hand, T-SNE helps to understand if all the features extracted can differentiate the dance moves. If any one of the clusters in T-SNE overlaps, then better features are needed to differentiate the classes where the clusters overlap.

Figure 5.1.4-1 shows the boxplot for kurtosis on z-acceleration, and it suggests that if the value of kurtosis is below 0, the dance moves are probably side pump, listen, wipetable or logout.

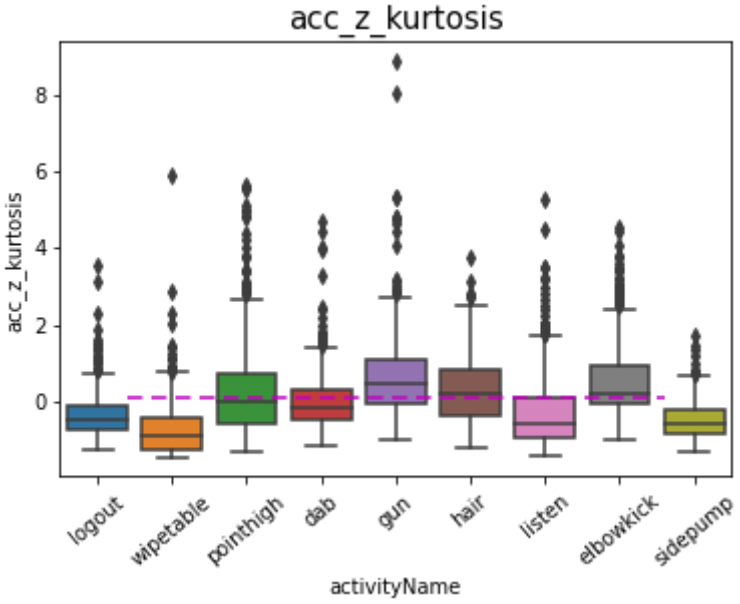


Figure 5.1.4-1 Boxplot for kurtosis on z-acceleration

Figure 5.1.4-2 shows the boxplot for the principle frequency for pitch, and it suggests that if the values spread across -1 and 1, the dance moves are probably logout, pointhigh or dab.

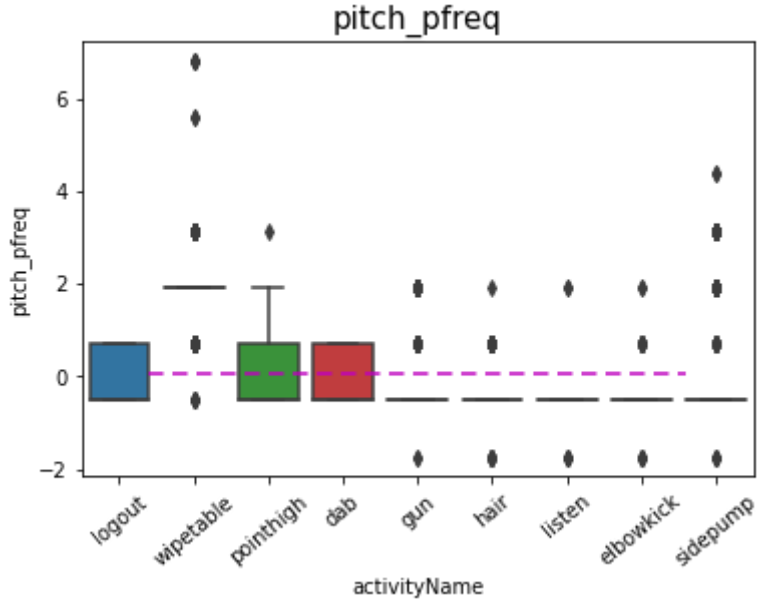


Figure 5.1.4-2 Boxplot for pitch principle frequency

Figure 5.1.4-3 shows the boxplot for min-max on z-acceleration, and it suggests that if the value is above 1, the dance move is probably dab.

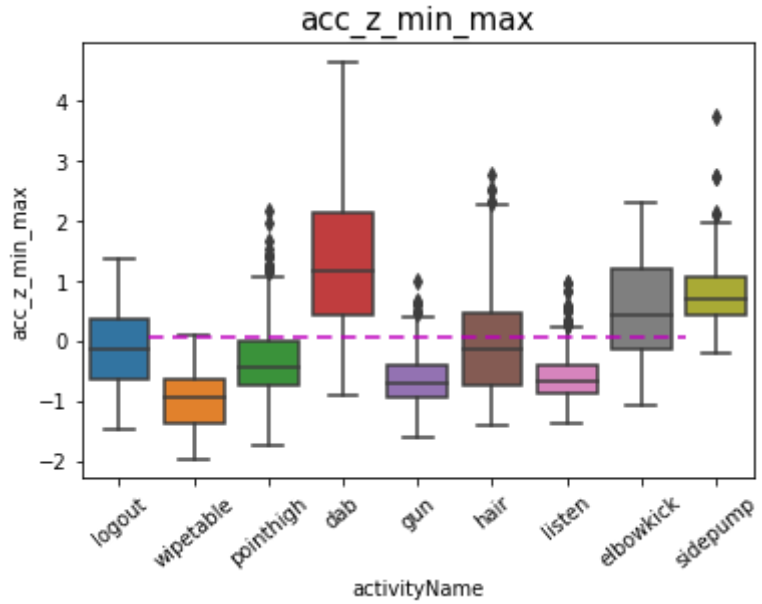


Figure 5.1.4-3 Boxplot for min-max on z-acceleration

The boxplots shown earlier demonstrated the extracted features differentiating some of the dance moves. Combining more useful features together and feeding into a deep learning model, the model can learn the features and predict the correct dance moves.

Figure 5.1.4-4 shows the T-SNE that suggests the extracted features are good in separating the classes. There are no overlapping clusters. If any cluster significantly overlaps with another, it strongly suggests more useful features are needed to distinguish the classes where their clusters overlaps in T-SNE.

Figure 5.1.4-4 T-SNE of features extraction

Section 5.1.5 Detection Of Dance Moves

The features were extracted from a window of 60 readings. The total number of features was 126. Therefore, the input layer had a size of 126. There were a total of 9 dance moves. Therefore, the output layer has a size of 9. Figure 5.1.5-1 shows the DNN's model architecture implemented. Whenever a new window of dancing data arrived, features were extracted before passing to the model to infer the dance moves.

```
DNN(
(fc1): Linear(in_features=126, out_features=64, bias=True)
```

```
(dp1): Dropout(p=0.1, inplace=False)
(fc2): Linear(in_features=64, out_features=16, bias=True)
(dp2): Dropout(p=0.1, inplace=False)
(fc3): Linear(in_features=16, out_features=9, bias=True)
)
```

Figure 5.1.5-1 DNN Architecture

Section 5.1.5.1 Training Model

The machine learning models improved in performance with more training data. The steps to collect the data for classifying dance moves are outlined below. Data was collected from 5 subjects in the team, leaving the 6th team member to test the model. The dancers were allowed to dance at their natural pace to increase the model's ability to generalize the dance moves.

1. Put on wearable sensor on the hand
2. Play the video of the dance move to follow closely
3. Dance for 1 min for 3 runs
4. Repeat step 2 and 3 for the next dance move

The total number of readings collected was 141,753 in the first and second runs for each dancer. Figure 5.1.5.1-1 shows the distribution of the data collected for each dance move. Figure 5.1.5.1-2 shows the distribution of the data collected for each dance move for each subject move. The distribution of data for each subject and for each dance move was relatively balanced to ensure no bias in the data.

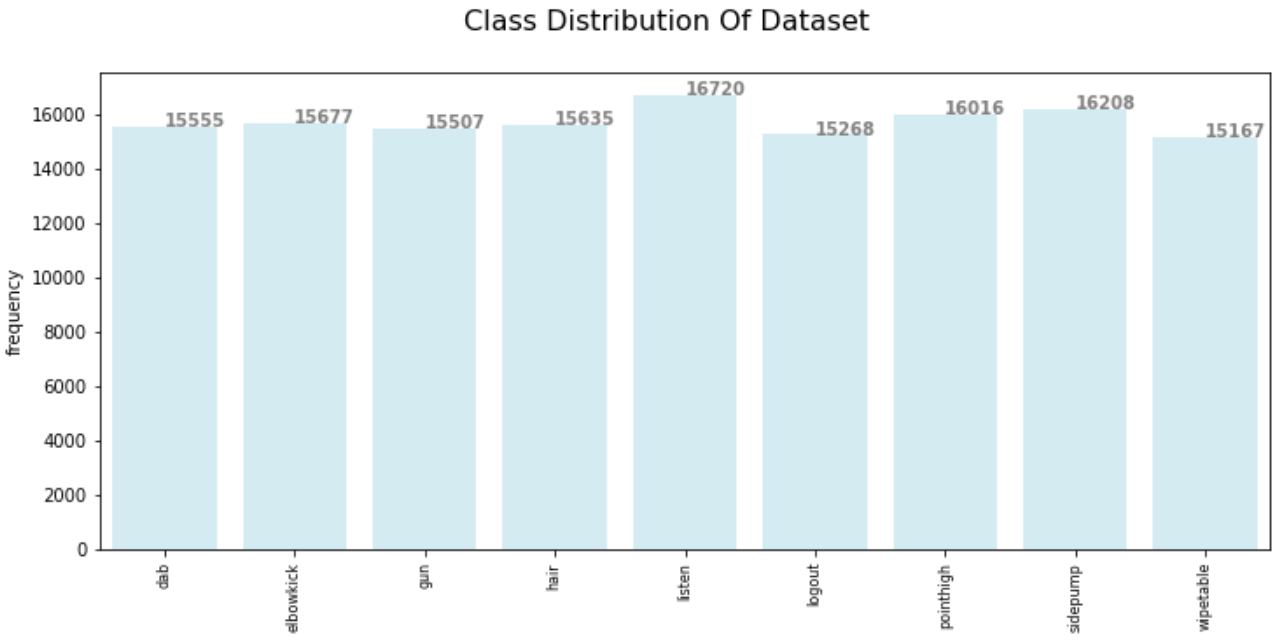


Figure 5.1.5.1-1 Distribution of data collected for each dance move

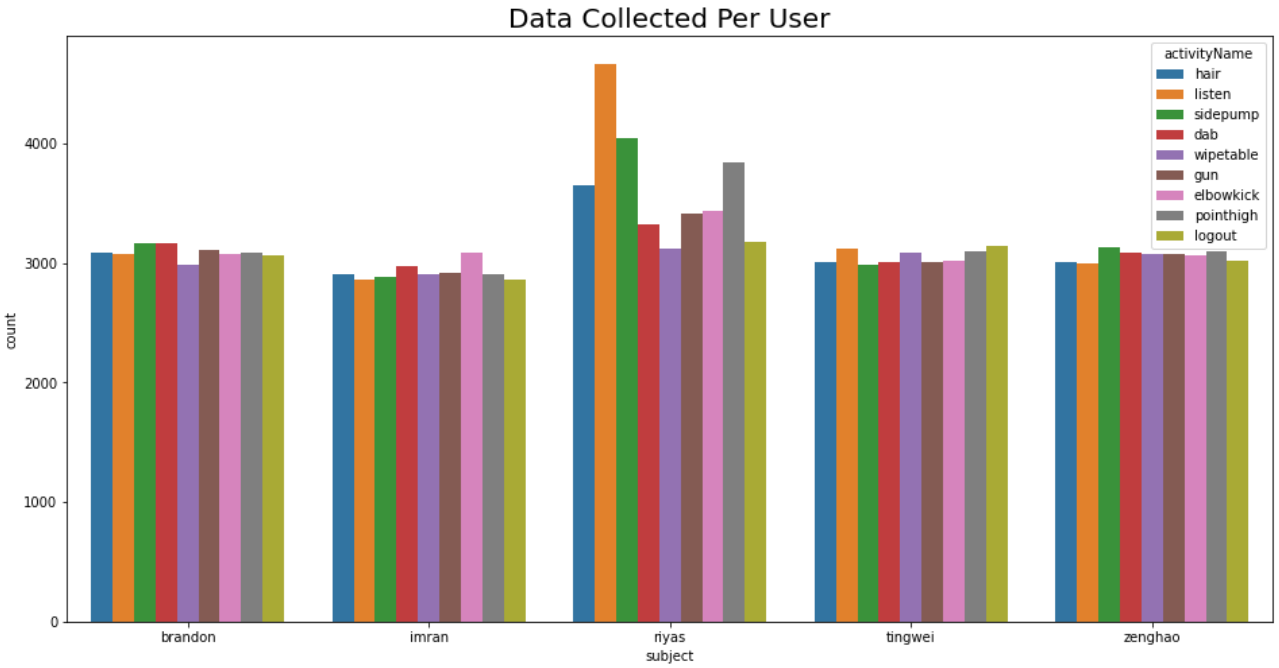


Figure 5.1.5.1-2 Distribution of data collected for each dance move for each subject

EarlyStopping was used during model training to prevent overfitting. The number of layers and its size were experimentally optimized. Figure 5.1.5.1-3 shows the model loss and accuracy against epoch on the left and right respectively. The training and validation loss and accuracy did not show signs of overfitting. The model stopped training after 198 epochs to prevent the model from learning the noise in the data.

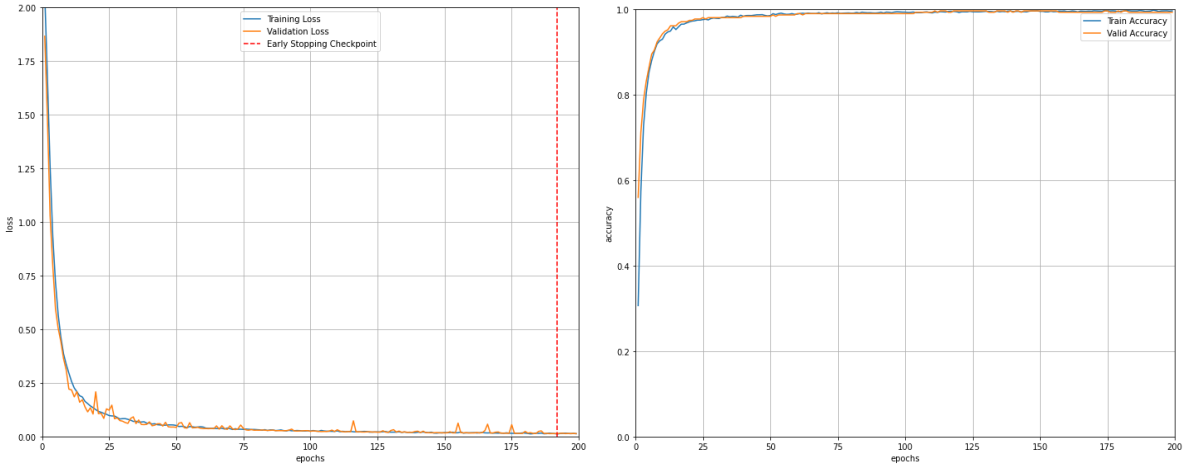


Figure 5.1.5.1-3 Model loss and accuracy with early stopping

Section 5.1.5.2 Validation And Evaluation

The dataset was split into 60% training, 20% validation, and 20% testing. DNN was trained using cross entropy as the loss function and early stopping to avoid overfitting on the training data. The metric to evaluate the model performance was the accuracy score. The formula to calculate the accuracy score is shown in Figure 5.1.5.2-1.

$$Accuracy = \frac{(correct\ predictions)}{(total\ predictions)}$$

Figure 5.1.5.2-1 Formula to calculate accuracy

To visualize the inference results, a confusion matrix was employed. A confusion matrix helps to understand which dance moves the model had difficulty in predicting and can be used as the reference to improve the model and extract features. Note that the first and second runs for each dancer were used for training, validating, and testing. The accuracy using the testing data overlaps with the training data due how the data is segmented with 50% overlap for each window; therefore, the accuracy is inflated.

To resolve accuracy inflation, the third run for each dancer was strictly used for testing the model's predictive ability instead, providing a stricter gauge of the model accuracy. Figure 5.1.5.2-1 shows two confusion matrices of the inference accuracy and the right image shows the normalized inference accuracy on the left and right respectively. There are very few mispredictions.

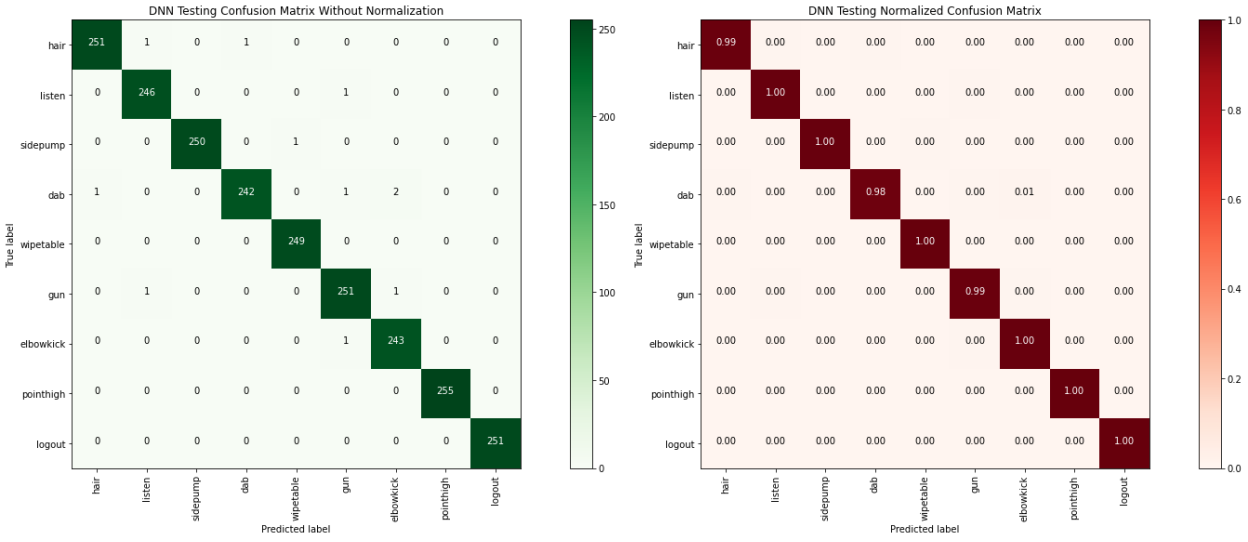


Figure 5.1.5.2-1 Confusion matrices to visualize inference accuracy

Section 5.1.6 Detection Of Relative Position

The readings from the imu sensor on the waist determined if the dancer was still, turning left, or turning right. The left and right turn used the pitch angles only. Figure 5.1.6-1 and 5.1.6-2 shows the pitch angle against reading indices for left turn and right turn respectively . Note that for left turns, the positive values of the pitch angle came before the negative values. Note that for right turns, the negative values of the pitch angle came before the position values.

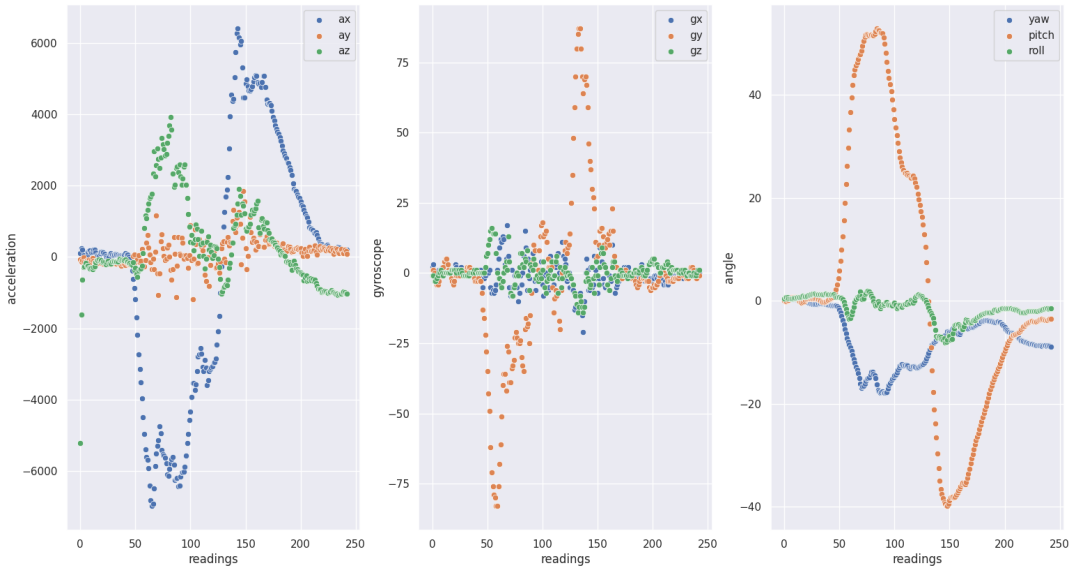


Figure 5.1.6-1 Acceleration, gyroscope, and angle plots for left turn

The indices of the pitch angle below and above the threshold of -30 and 30 in the window frame were obtained. The mean of the indices below the threshold was computed; the mean of the indices above the threshold was also computed. If the mean of the indices below the threshold was smaller than the mean of the indices above the threshold, the dancer took a left turn, else the dance dancer took a right turn. If there were no readings that exceed the threshold, the dancer was standing still.

From both Figure 5.1.6-1 and 5.1.6-2, the acceleration in the x direction changed; however, the acceleration took a long time to settle down. The acceleration values were not responsive and might result in false triggers in left and right steps if the acceleration readings from the previous window ate into the current window. Gyroscope was also not used because the values fluctuates rapidly for dance moves such as elbowkick. Setting a

high threshold did not resolve the issue because this means that the dancer needed to take an unnatural sharp turn. Therefore, pitch was used to detect the left and right turn.

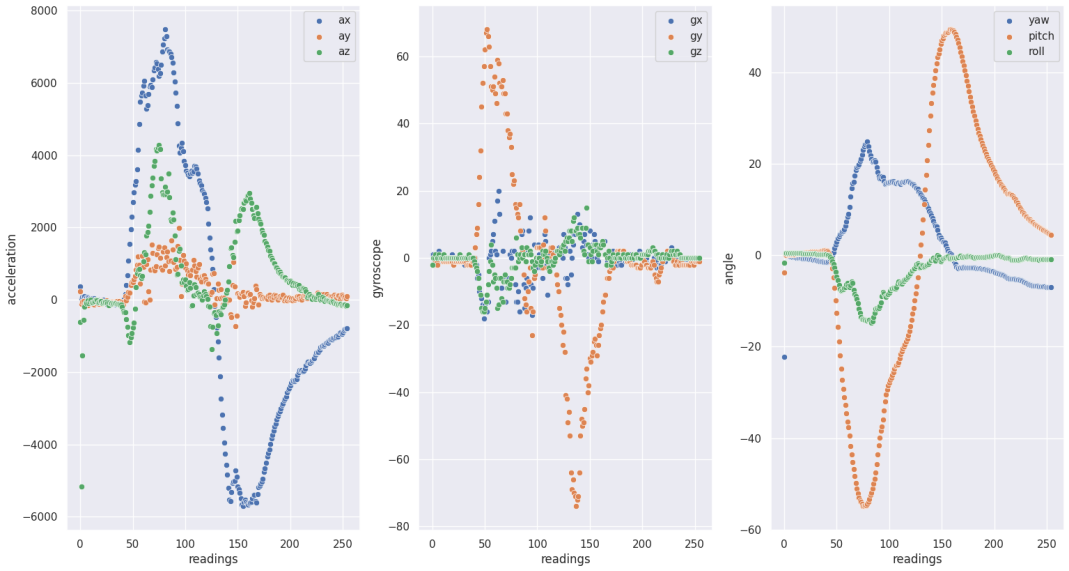


Figure 5.1.6-2 Acceleration, gyroscope, and angle plots for right turn

Section 5.1.7 Model Integration

As readings from the imu sensor on the hand arrived for each dancer, they were stored into a list for each dancer. When the length of the list exceeded 250, about 10 seconds of data was collected. The last 60 readings were used for feature extraction and for the model to infer the dance moves.

Meanwhile, as readings from the imu sensor on the waist arrived for each dancer, they were also stored into another list for each dancer. The indices that lied below and above the threshold were obtained and the means were computed. The dancer was predicted to stand still, turn left, or turn right. As there were only 6 valid permutations of the dance positions, there was a one-to-one mapping of the dancers' turns to the dancer updated positions. For example, if the turn made by the three dances from left to right were right, still, and left, this means that the dancer on the left and dancer on the right are swapping positions, while the middle dancer remains at the same spot.

The model was trained in PyTorch but it was cumbersome to install PyTorch on Ultra96 and on other laptops. The weights of the PyTorch model were loaded into a Numpy model to ease the integration and testing of the model on Ultra96. Once the model was satisfied,

the model was converted to run on FPGA for the final evaluation. Figure 5.1.7-1 shows the PyTorch model implemented in Numpy.

```
class FCLayer:
    def __init__(self, in_features, out_features, name="fclayer"):
        self.weights = np.zeros((out_features, in_features))
        self.bias = np.zeros(out_features)

    def forward(self, inputs):
        return inputs.dot(self.weights.T) + self.bias

    def __call__(self, inputs):
        return self.forward(inputs)

class DNN:
    def __init__(self):
        self.fc1 = FCLayer(126, 64)
        self.fc2 = FCLayer(64, 16)
        self.fc3 = FCLayer(16, 9)

    def forward(self, x):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        return x

    def __call__(self, inputs):
        return self.forward(inputs)
```

Figure 5.1.7 - PyTorch model in Numpy

Section 5.2 Software Dashboard

Section 5.2.1 Introduction

The dashboard system of DanceEdge aims to help tie the entire system together by giving valuable statistics and analysis to both trainees and coaches. It accepts real-time information from the wearable system. It processes the information and provides feedback in the form of graphs, charts, numbers and visually appealing visual-aids. With the help of these data, trainees will be able to understand the gaps in their performance and therefore identify the specific areas to rectify in their performance. On the other hand, coaches will be able to not only provide advice to each individual but also analyse the group's performance as a whole compared to past performances. Coaches will then be able to identify areas of weakness for the group and coach accordingly. The dashboard runs on a local server hosted on a laptop. The database is hosted on cloud. It aims to provide a smooth user experience that keeps leaving the trainee or coach wanting for more.

Section 5.2.2 Dashboard Design

Section 5.2.2.1 User Flow

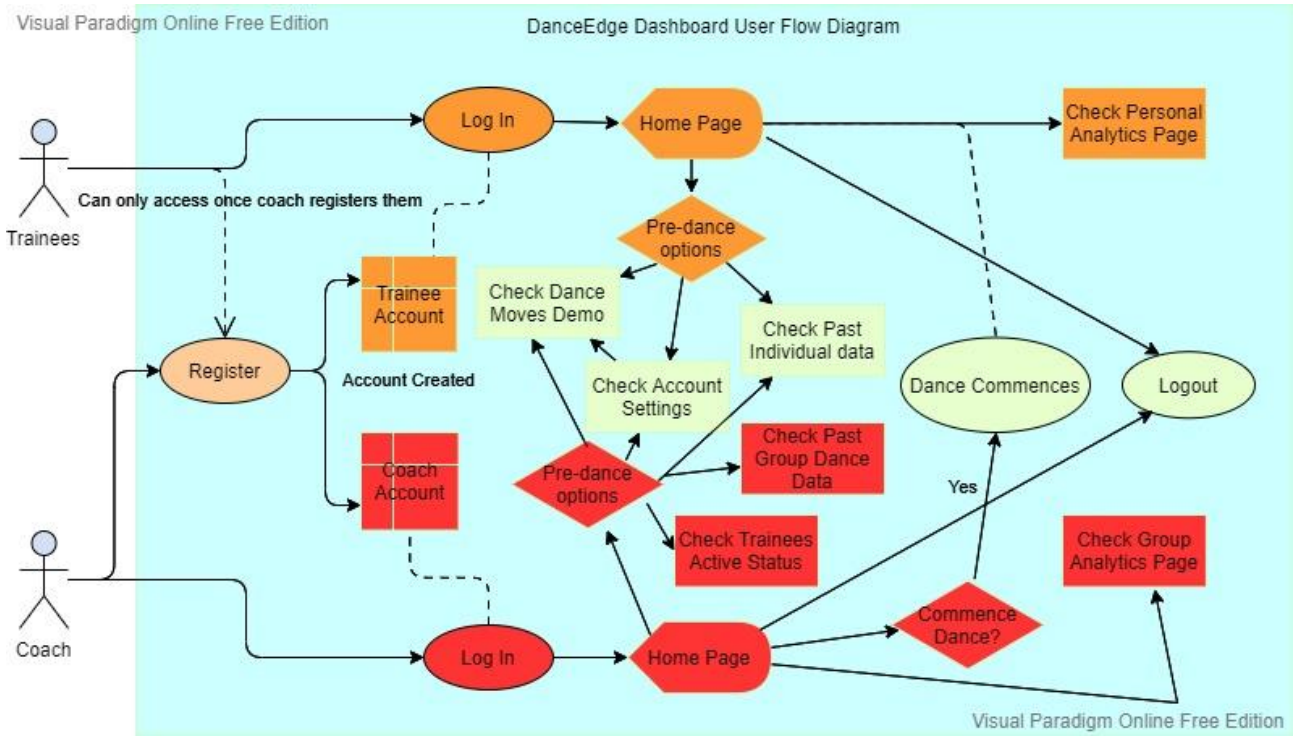


Figure 5.2.2.1-1: Initial User Flow Diagram

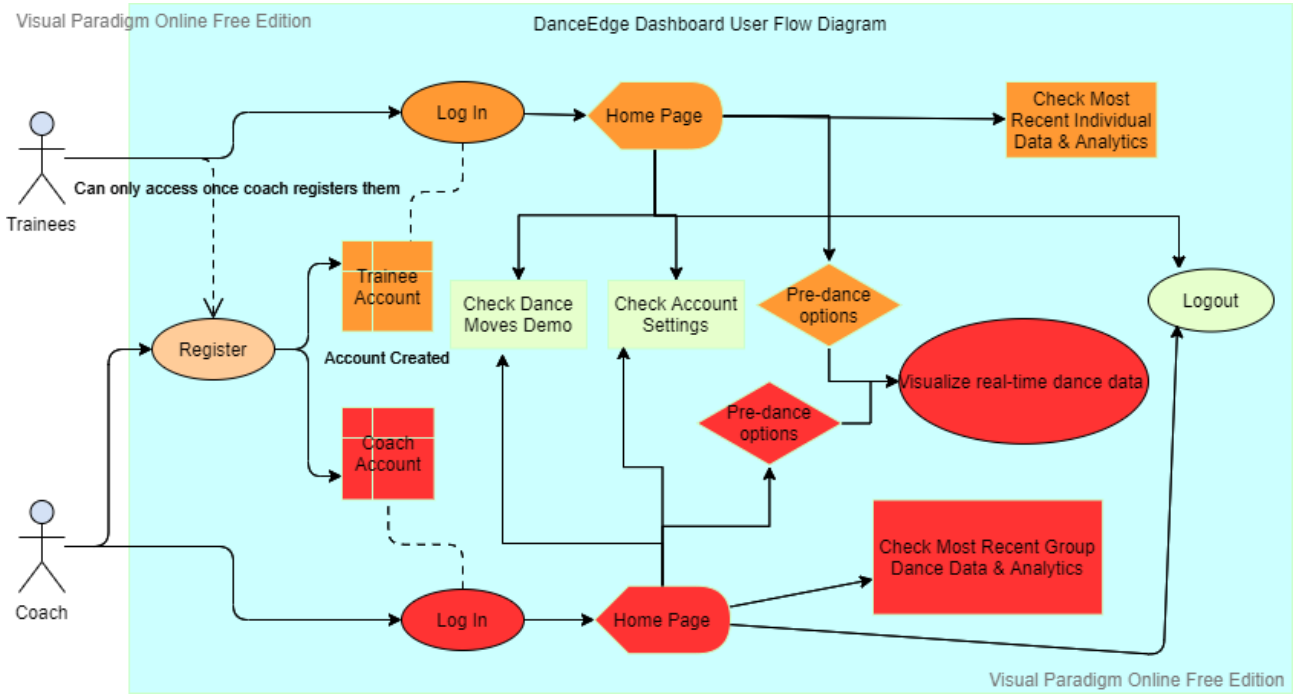


Figure 5.2.2.1-2: Current User Flow Diagram

The user flow diagram aims to provide a full overview of the different functionalities both the trainees and coaches can perform on the dashboard. The design of the dashboard aims to give a really smooth User Interface and User Experience (UI/UX). The dashboard follows a cartoon style to give users a very comfortable and welcoming feel.

In order for trainees to access their individualised, personal dashboard, they would need their coach to register for them. Coaches are allowed to register up to three trainees. They will set the usernames and passwords for their trainees.

Once logged in, coaches and trainees will be faced with a pre-dance home page. They are given the option to view their most recent dance data and analytics, view demonstrations of the dance moves themselves or view real-time data of their dance moves. The difference between the initial user flow diagram and the current one is that users can only view the data of the most recent dance moves. This feature was scaled down for simplicity for the current requirements of the project.

Coaches will be able to view data and analytics of all three dancers whilst trainees can only view their data and analytics. Once the user selects the “lets dance” option, data will seamlessly flow in real-time into the respective dashboards. After the dance, they can logout or retry. For the remainder of this dashboard section, it is important to note that the

trainee portion of the dashboard was not implemented. Only the coaches portion was implemented due to simplicity of meeting the current requirements of the project.

Section 5.2.2.2 Features to be Supported

Coaches and trainees should be able to view the following:

1. Real-time:
    a. Option to start dance routine
    b. Plots of raw values of Accelerometer sensor readings
    c. Plots of raw values of Gyroscope sensor readings
    d. Predicted position of each dancer vs correct positions
    e. Predicted dance move
    f. Synchronization delay
    g. Option to retry dance routine
    h. EMG Voltage value readings
2. Analytics:
    a. Most recent data plotted in a graph for all three trainees
    b. Position and Moves Score for each trainee. Overall score for the entire group
    c. In-depth breakdown for dance moves. Analyses which dance moves are being done wrong when trainees were supposed to be doing another dance move.
    d. In-depth breakdown for dance positions. Analyses which positions are trainees are wrong when they are supposed to be at another position.

Section 5.2.2.3 Wireframes & Actual Implementation

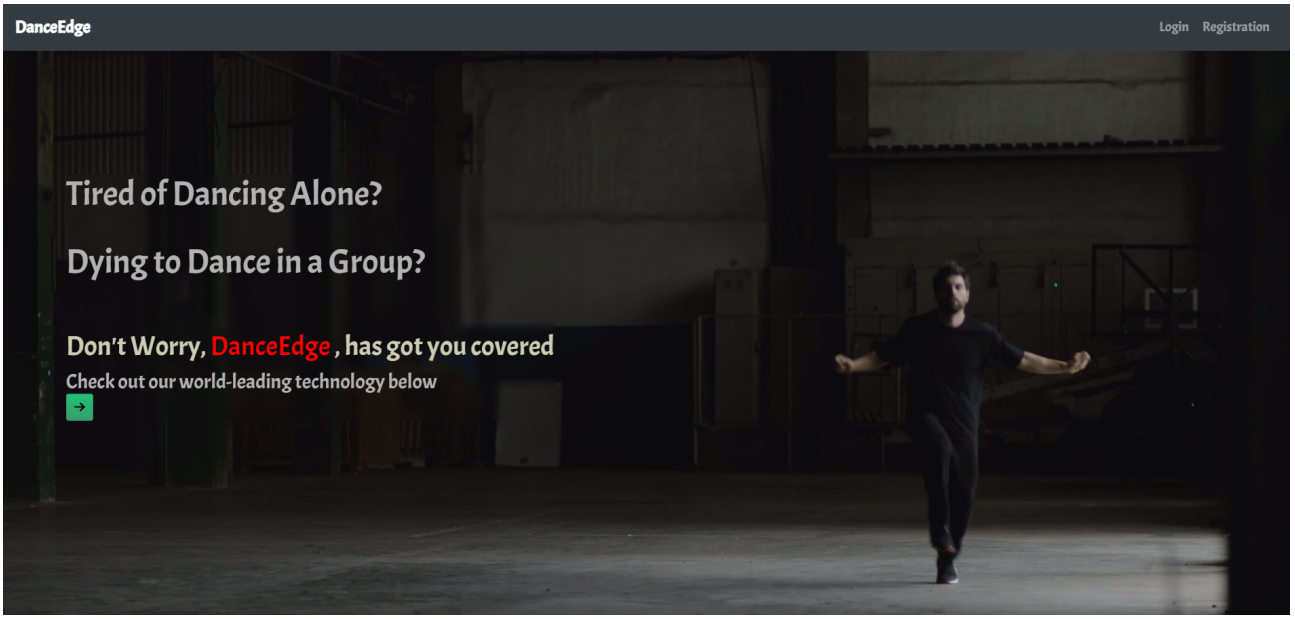


Figure 5.2.2.3-1: Home Page (1)

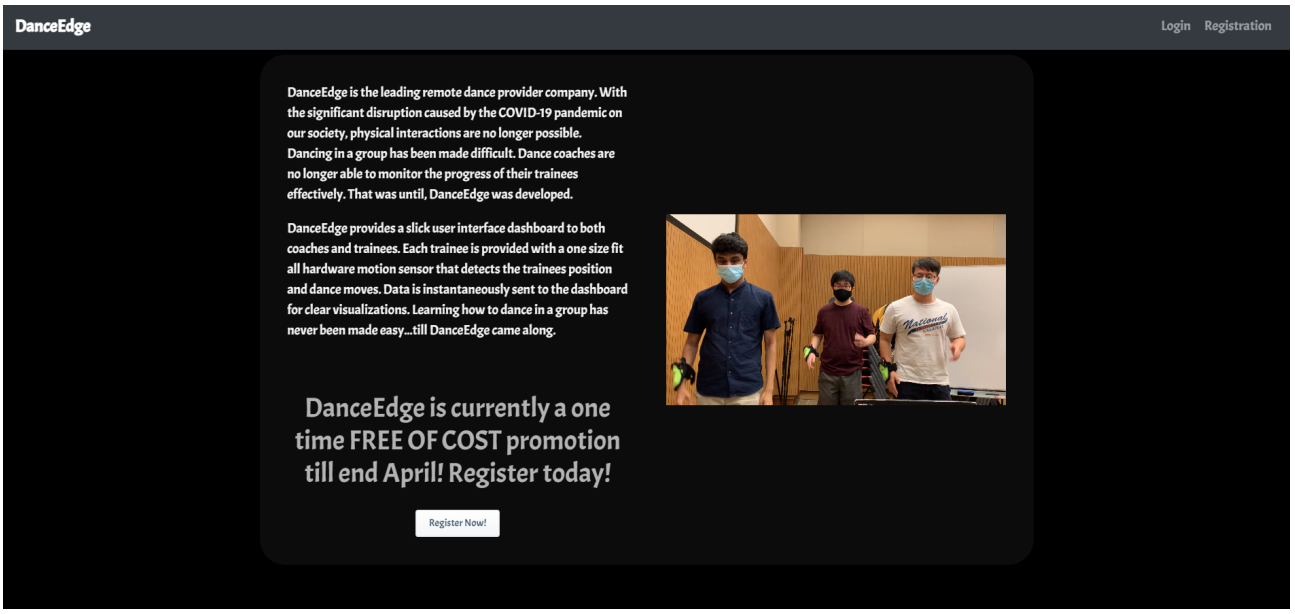


Figure 5.2.2.3-2: Home Page (2)

The home page was designed to provide a welcoming feel to new users. It provides a high level description of what DanceEdge does and how it benefits a certain target audience. At the end of it, it proclaims that there is a one time free of cost offer between now till the end of April 2021. It redirects new users to the registration page.

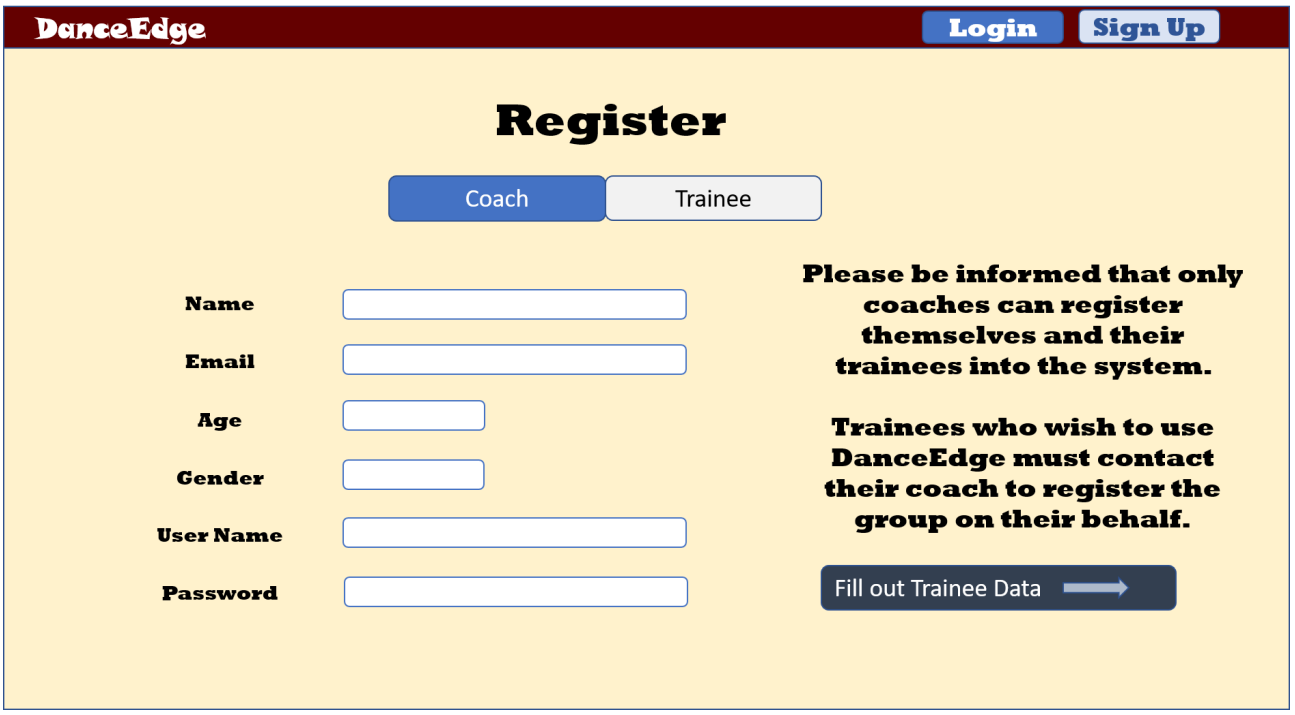


Figure 5.2.2.3-3: Register Page for Coaches (Wireframe)

Figure 5.2.2.3-4: Register Page for Trainees (Wireframe)

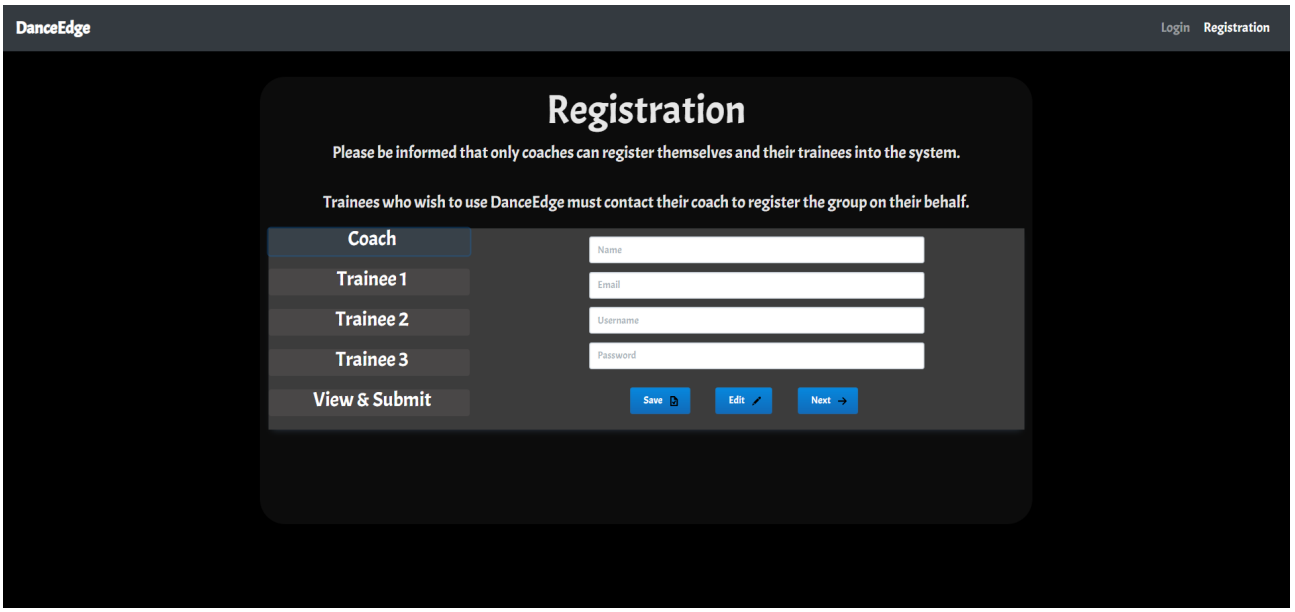


Figure 5.2.2.3-5: Register Page for Trainees (Actual)

To register as a trainee, a coach has to fill out the form for him/herself and the trainees of the group.

Figure 5.2.2.3-6: Login Page (Wireframe)

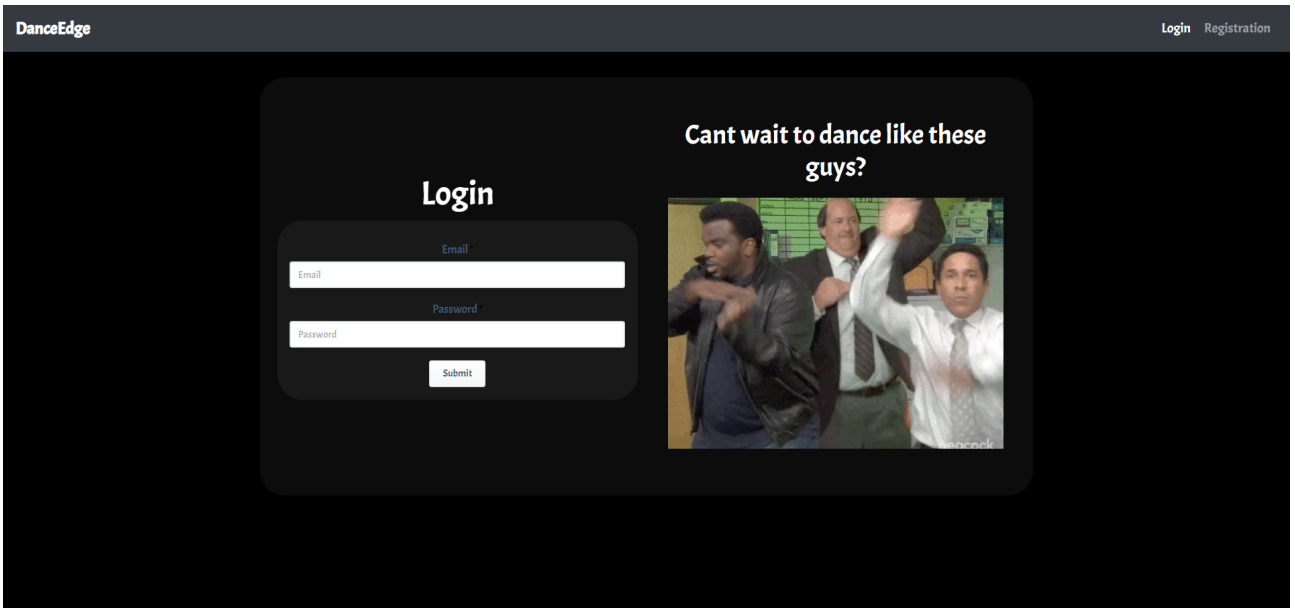


Figure 5.2.2.3-7: Login Page (Actual)

The login page prompts the user to login either as a coach or as a trainee. The prerequisite is that their accounts had to already be created.

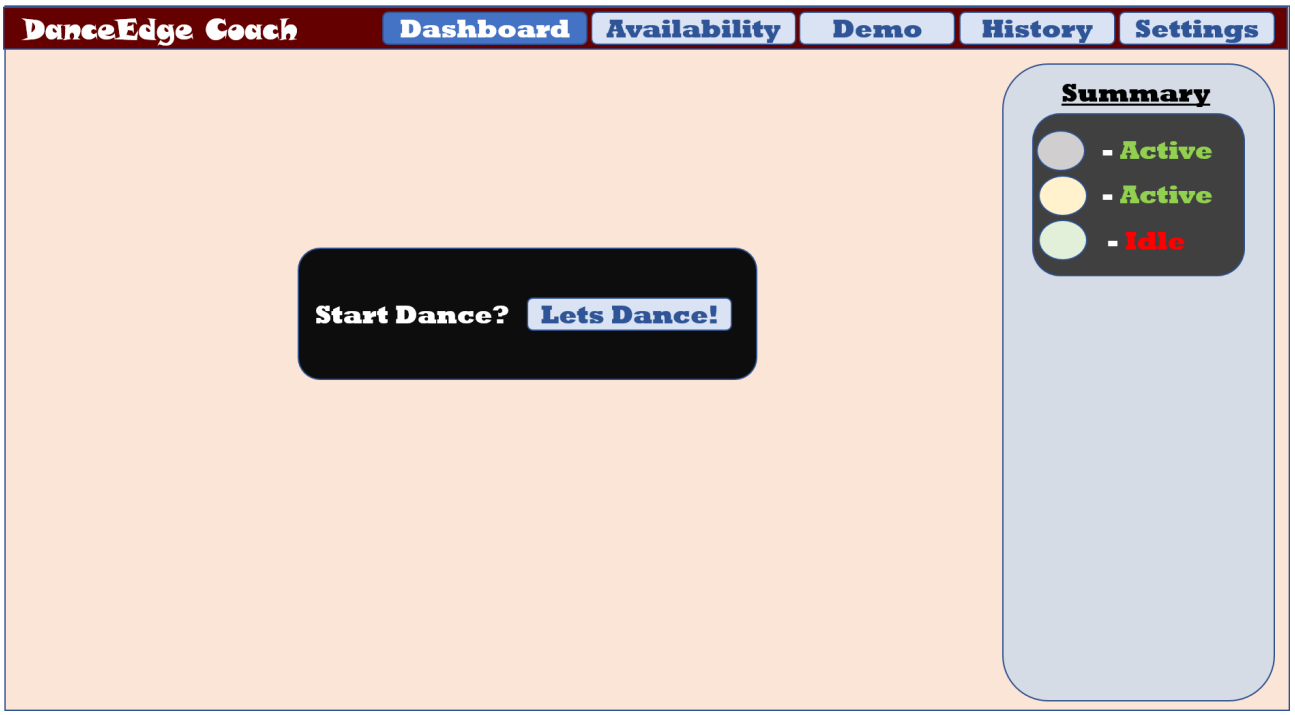


Figure 5.2.2.3-8: Pre-dance Coach's Dashboard Page (Wireframe)

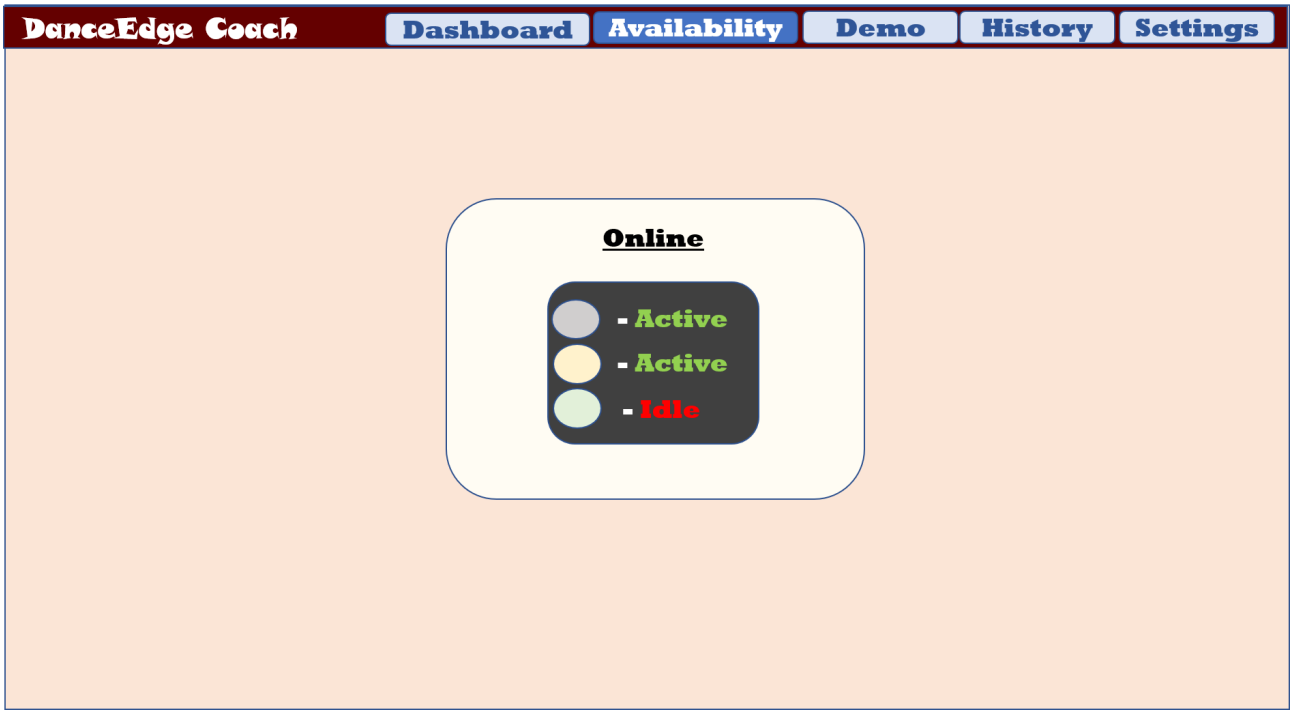


Figure 5.2.2.3-9: Availability Status of Trainees (Wireframe)

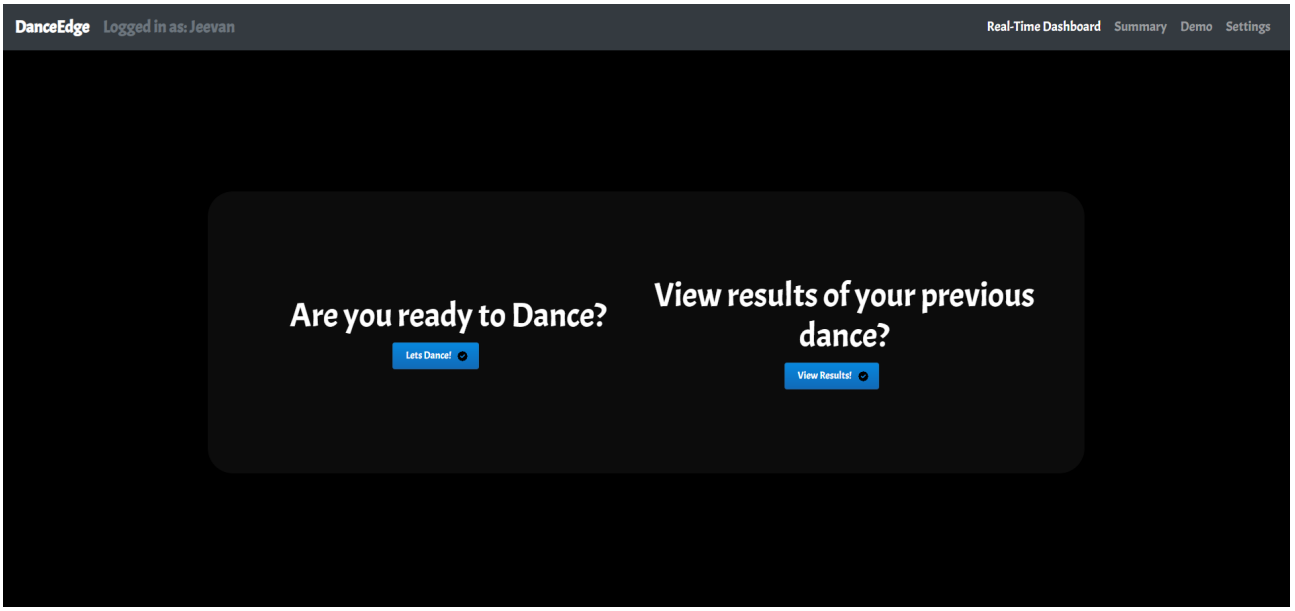


Figure 5.2.2.3-10: Pre-dance Coach's Dashboard Page (Actual)

The initial design of the coach's dashboard included a feature where coaches could check the online status of the trainees. However, due to the system design of the DanceEdge wearable, that feature was not implemented. On this page above, coaches could either start a new dance session or check the data and analytics of the most recent dance session.

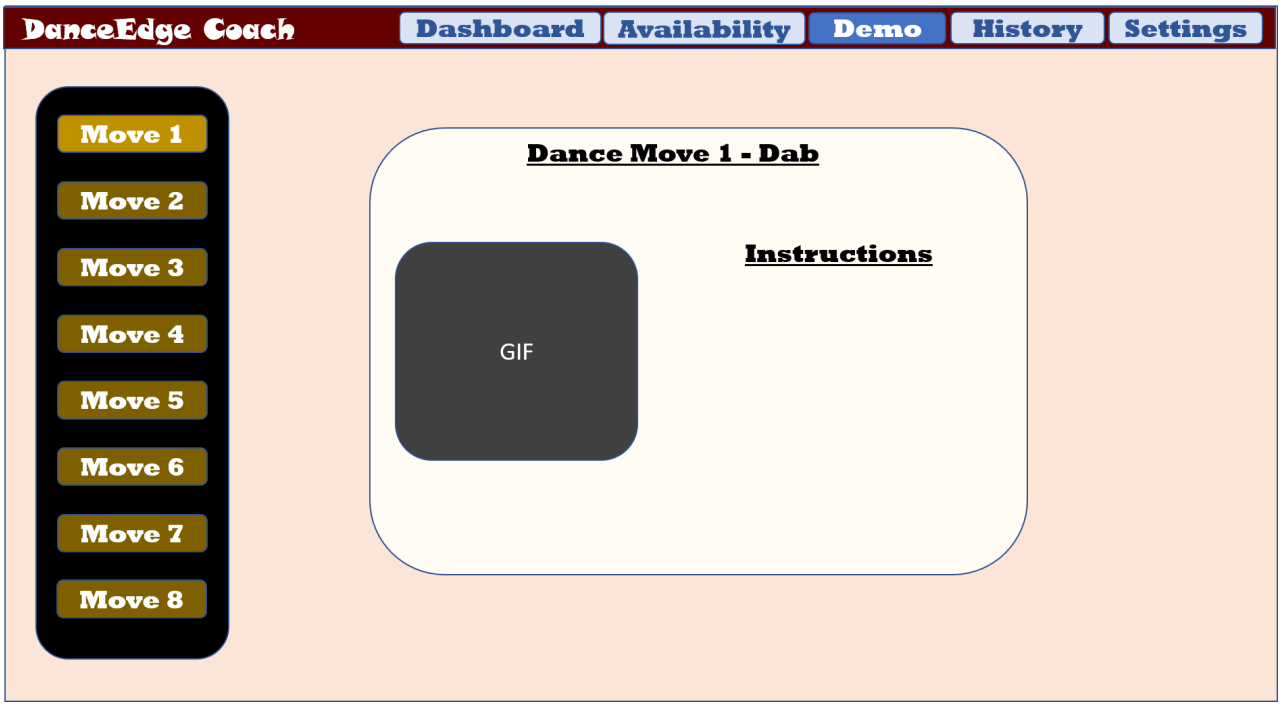


Figure 5.2.2.3-11: Demo Page (Wireframe)

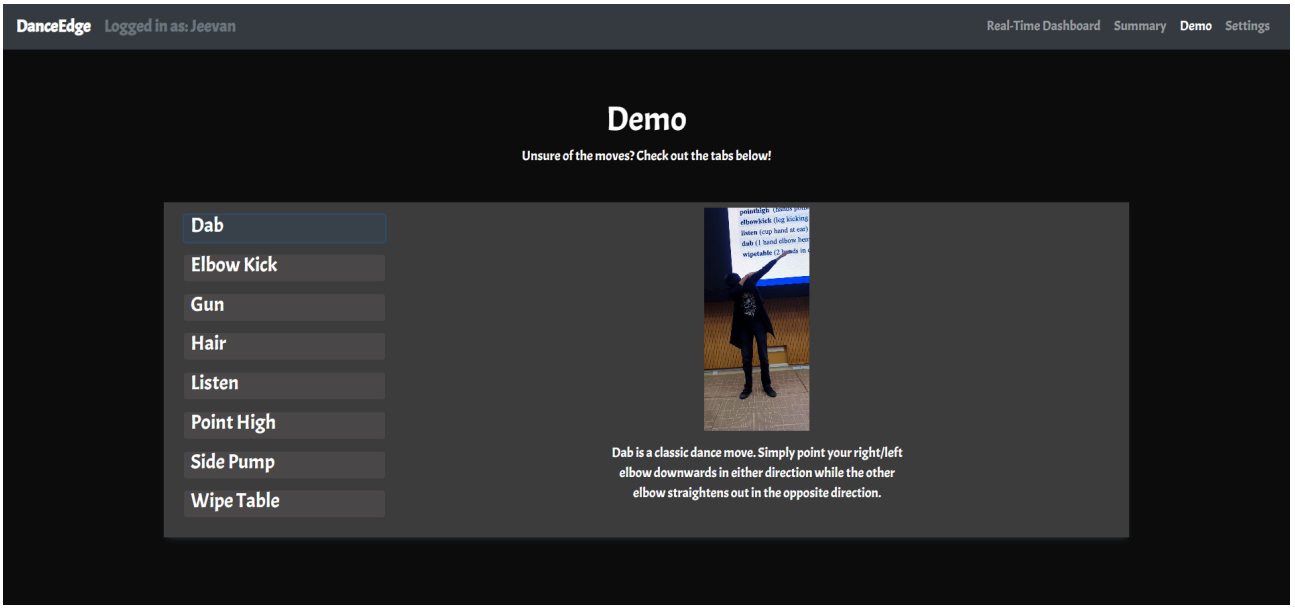


Figure 5.2.2.3-12: Demo Page (Actual)

One feature that was included to provide a holistic feel was a demonstration page. It allows coaches to watch the dance moves trainees are supposed to do over and over again.

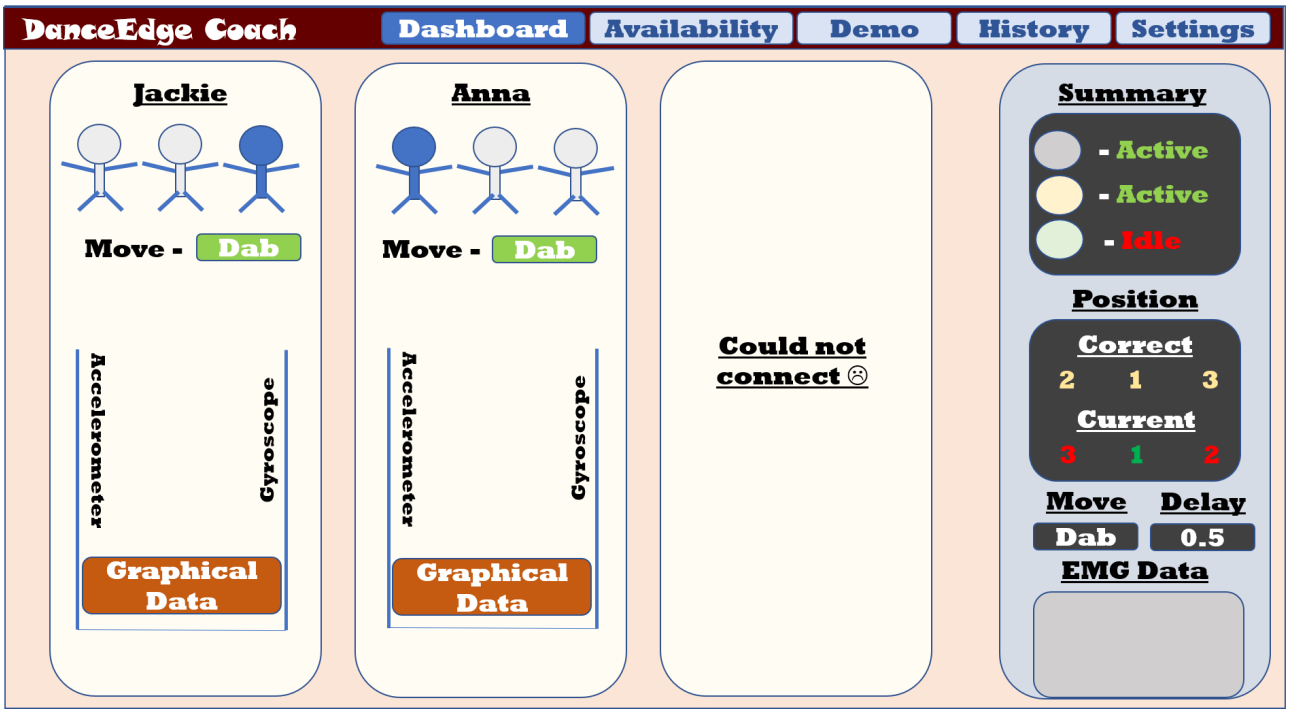


Figure 5.2.2.3-13: Coach Dashboard Page (Wireframe)

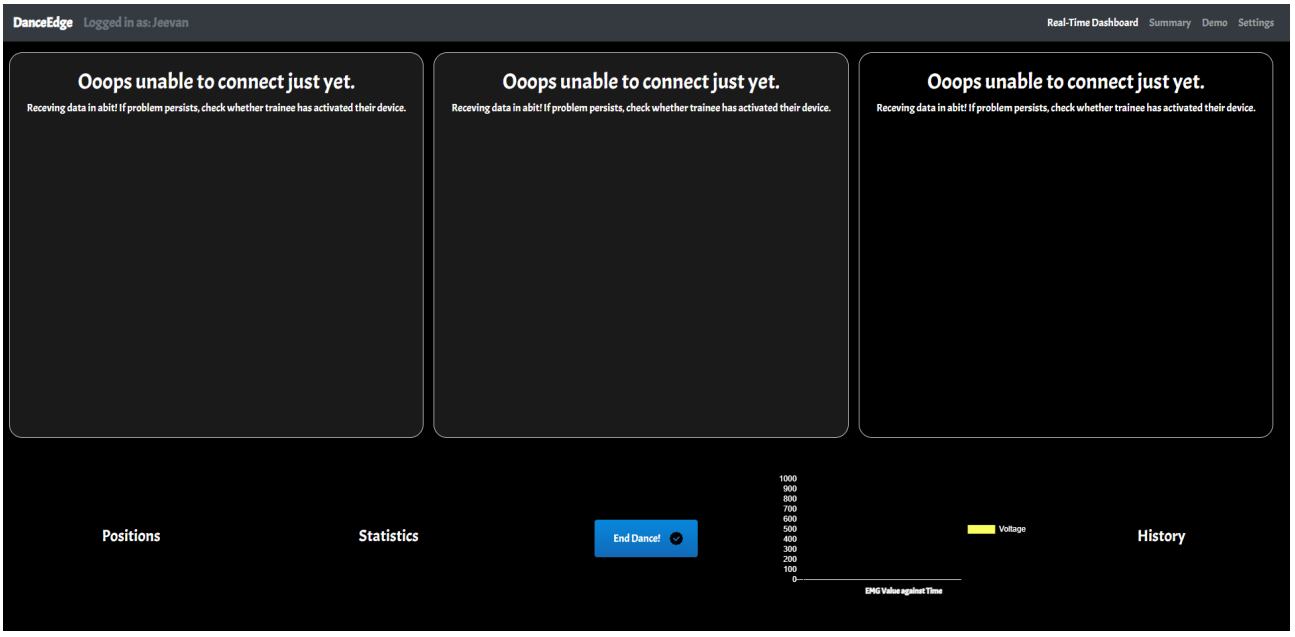


Figure 5.2.2.3-14: Coach Pre-Data Dashboard Page (Actual)

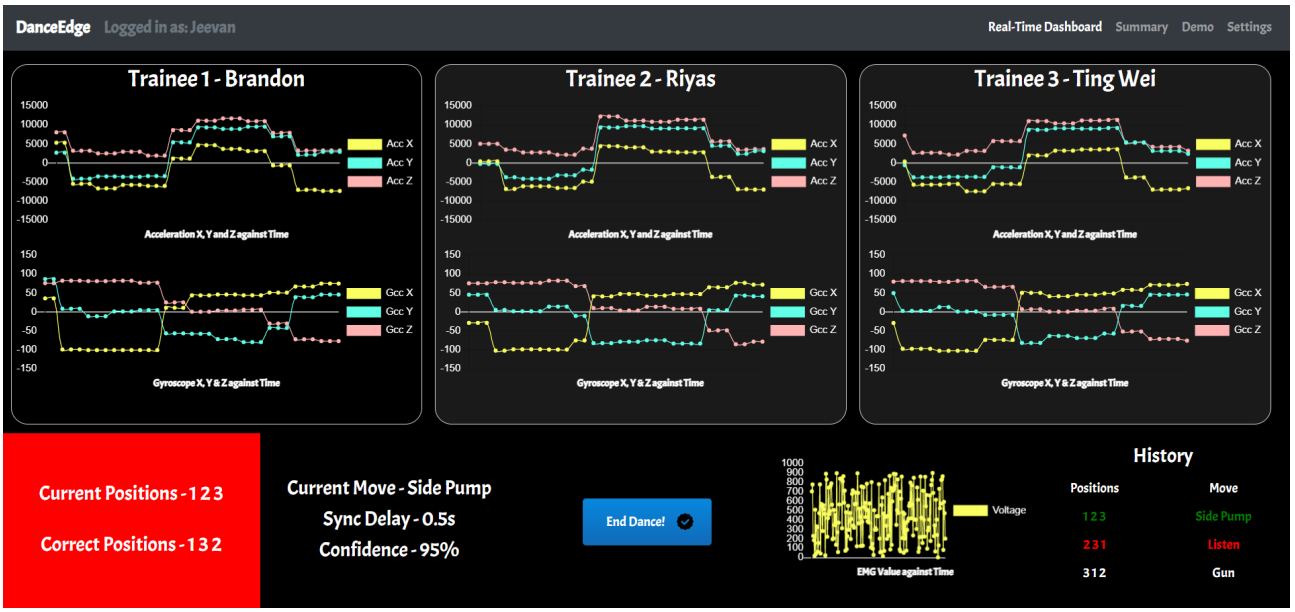


Figure 5.2.2.3-15: Coach Data Dashboard Page (Actual)

After clicking on “lets dance!”, the dashboard begins to plot the incoming data from whichever DanceEdge sensors that are sending data. In the initial design, the overall structure of the wireframes were such that the summary data were positioned on the rightmost column. With feedback obtained from the user survey, it was suggested to situate the summary data on the bottom of the screen for better readability.

In the initial design, both the accelerometer and gyroscope data points were intended to be plotted on the same graph to provide better visualization and comparison. However, that was not implemented to prevent a clutter of data that might prove difficult for coaches to view. This was another suggestion from the user survey conducted. Hence, two separate graphs were plotted.

In the initial design, a visual representation of the position of each user with respect to their other group mates was given. That was not implemented as well to improve the performance of the dashboard.

A small graph depicting the EMG voltage was also implemented. Once the dance is completed, the coach can click on “end dance” button which redirects the coach to the data and analytics page.

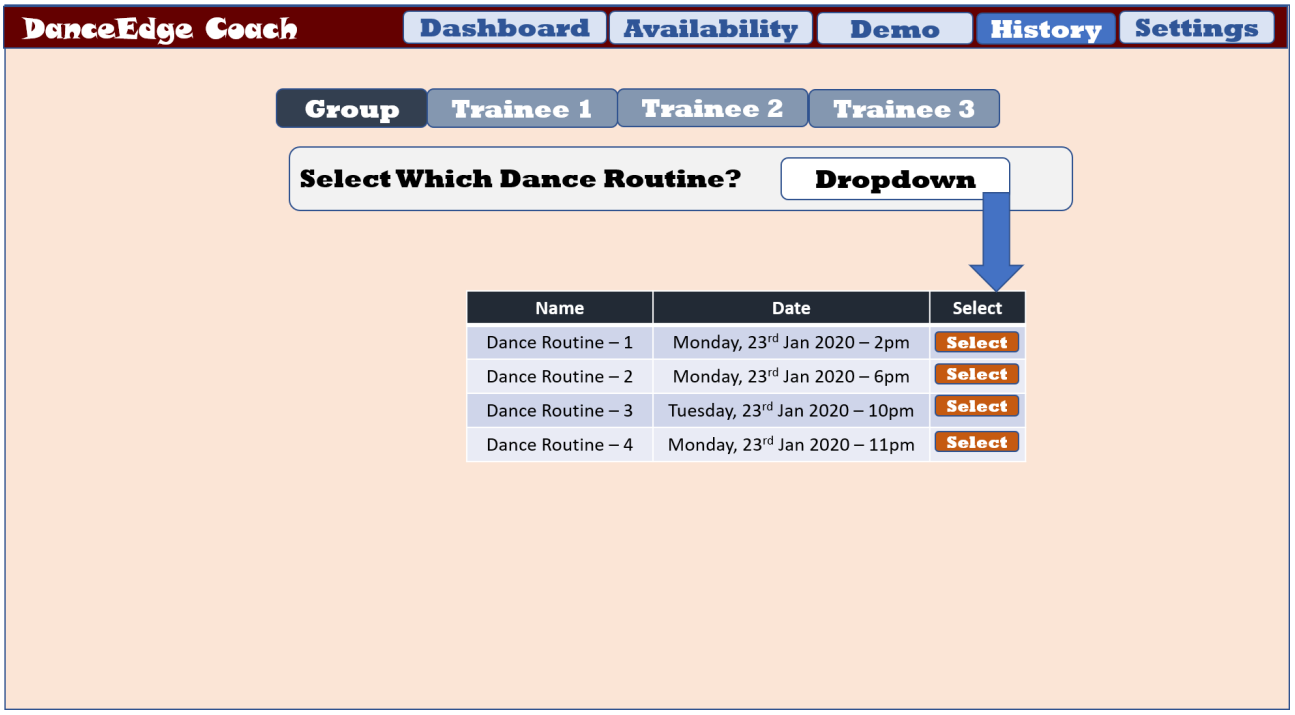


Figure 5.2.2.3-16: Coach Analytics (Wireframe)

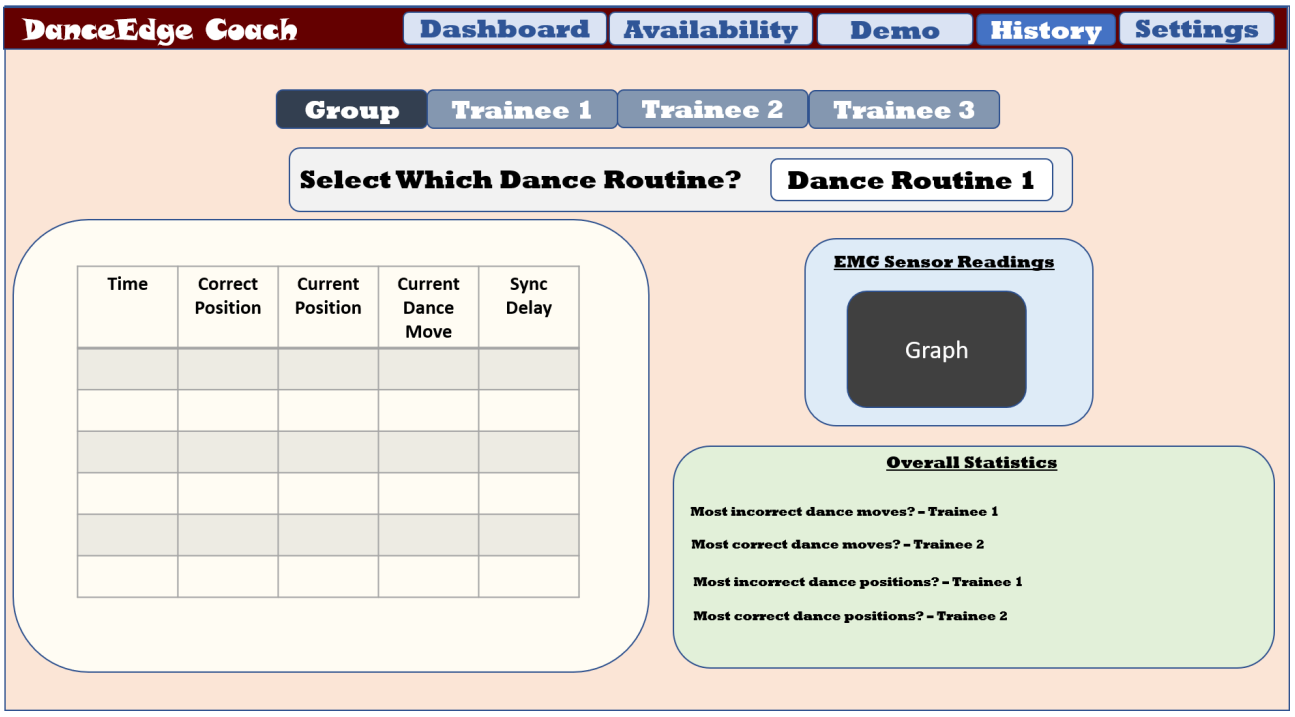


Figure 5.2.2.3-17: Coach Analytics (Wireframe)

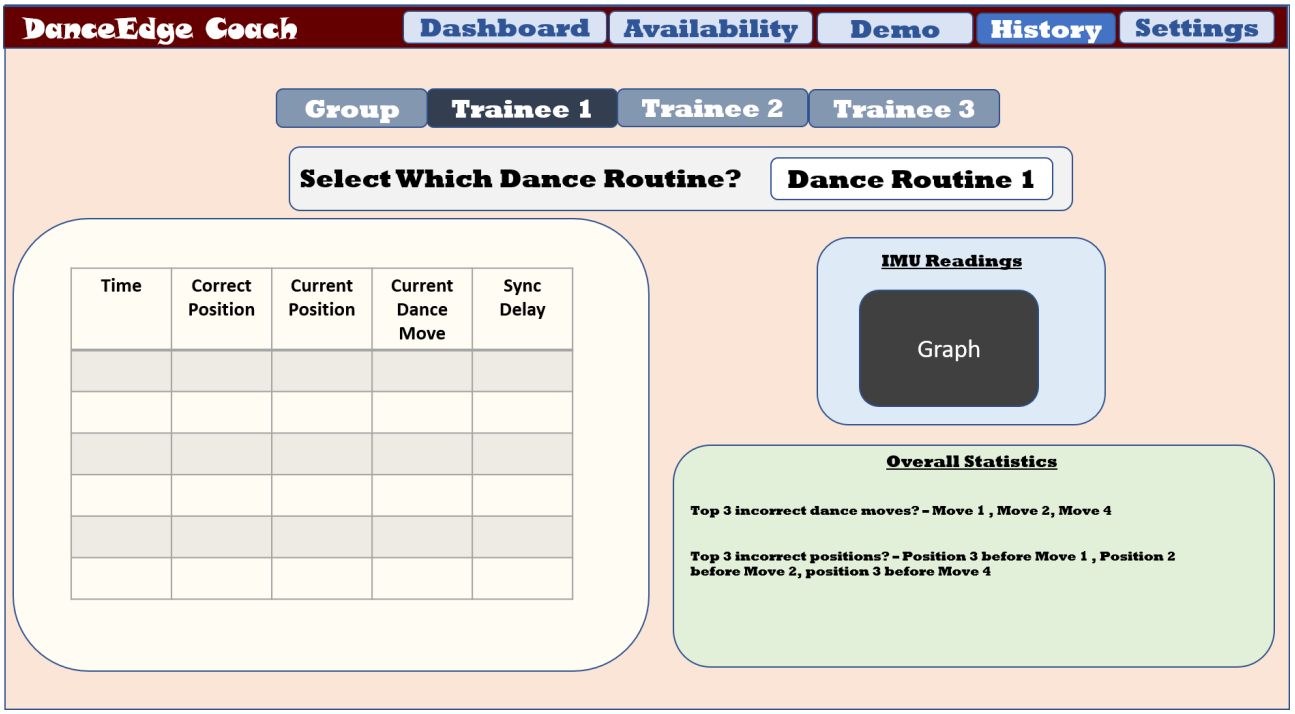


Figure 5.2.2.3-18: Trainee History Page

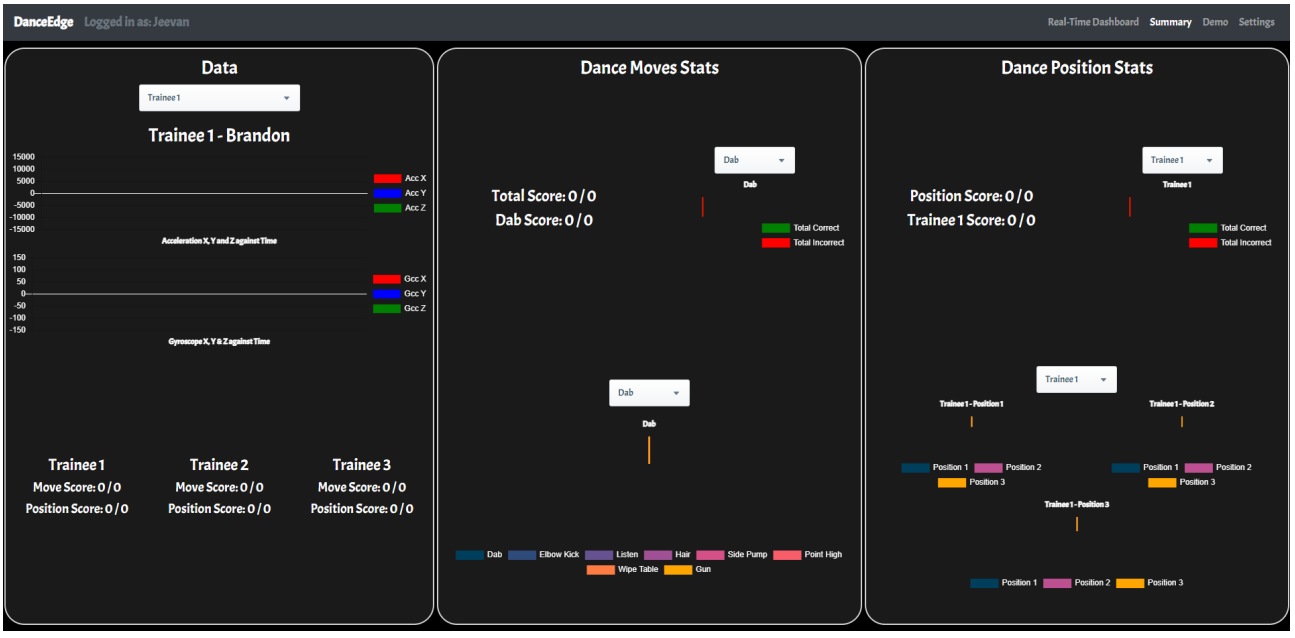


Figure 5.2.2.3-19: Coach Pre-Data Analytics (Actual)

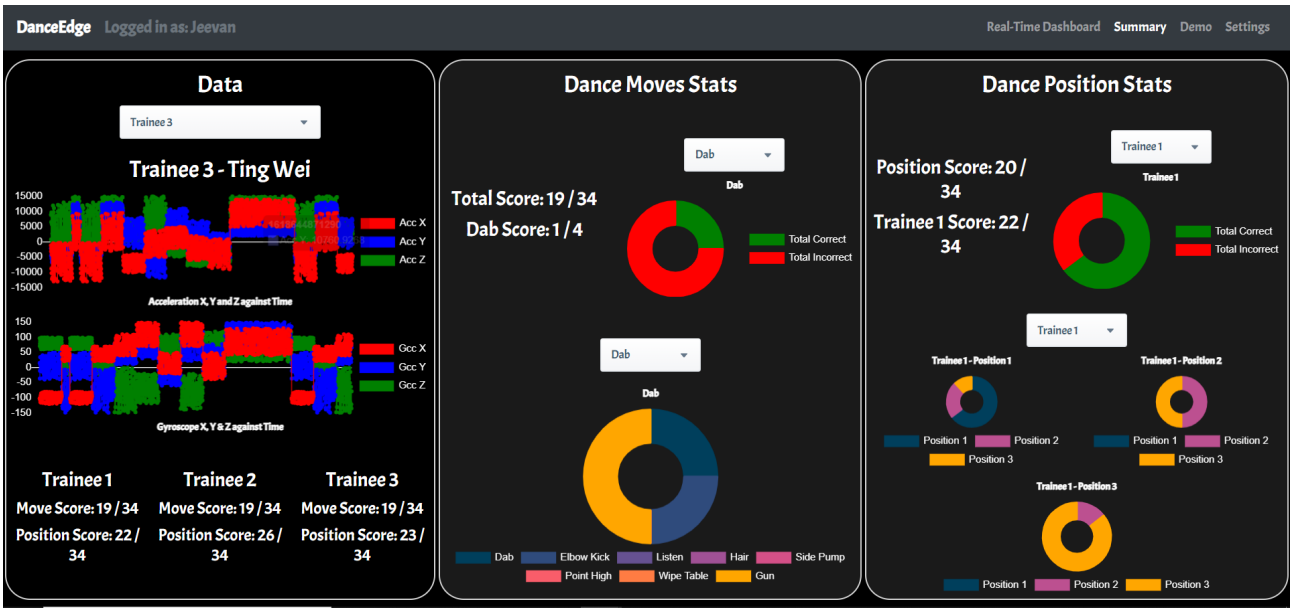


Figure 5.2.2.3-20: Coach Data Analytics (Actual)

In the initial design, there was a plan to have a section where coaches could view the history of past results. This was not implemented because the database design was scaled down to accommodate the project requirements. Instead, the summary page focused a tremendous amount on offline analytics.

On the left column in the image above, coaches could view the line graphs of the accelerometer and gyroscope values of each trainee. On the bottom of the left column, there is an individual dance and position score given to each trainee.

On the middle column, the first pie chart depicts how many of each dance move were correctly done by all the trainees. A total score is also displayed. Essentially, with reference to above, there were supposed to be 4 dab moves. However, all trainees only did 1 dab move right. What dance moves did the trainees do on the other 3 occasions? The pie chart below gives a more in depth view of which dance moves were done in place of dab. This allows coaches to understand which dance moves the trainees are mistaking for the correct dab move.

On the right column, the first pie chart describes how many positions each trainee got right. A total score is given as well. The subsequent three pie charts below describe which dance positions each trainee were correctly at. Let's say trainee 1 was supposed to be at

position 1 3 times. It turns out that he was only there 2 times. Where was he on the other occasion? The pie chart describes that very piece of detail crucial to coaches.

Section 5.2.3 System Architecture

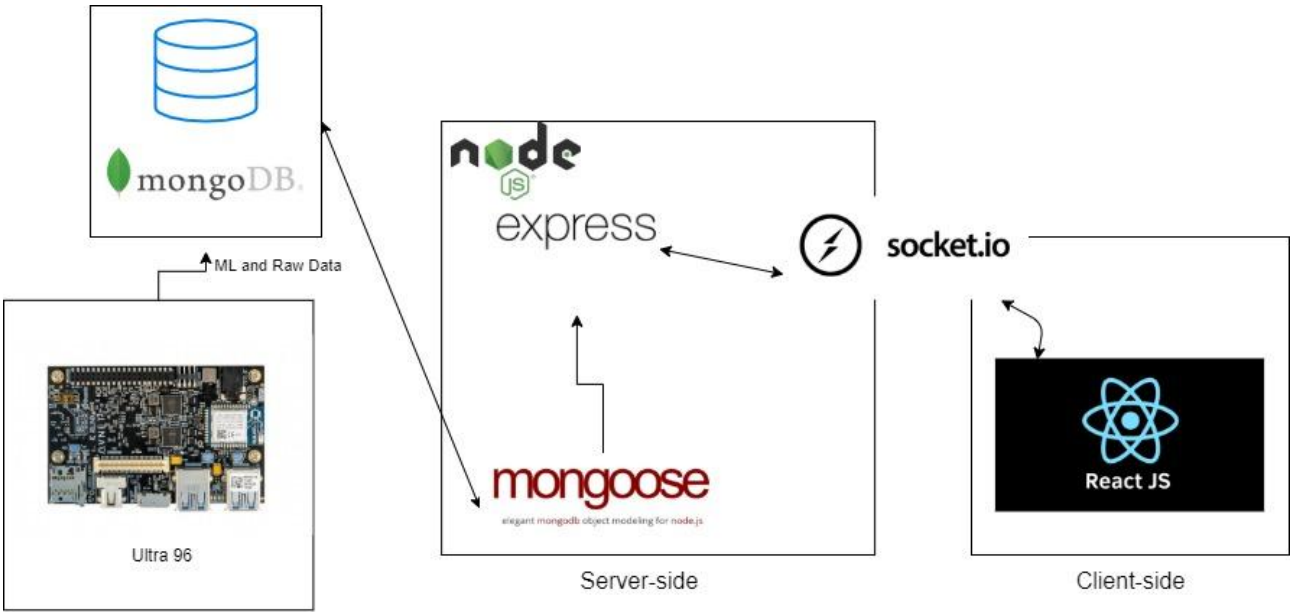


Figure 5.2.3-1: System Architecture Overview (Initial)

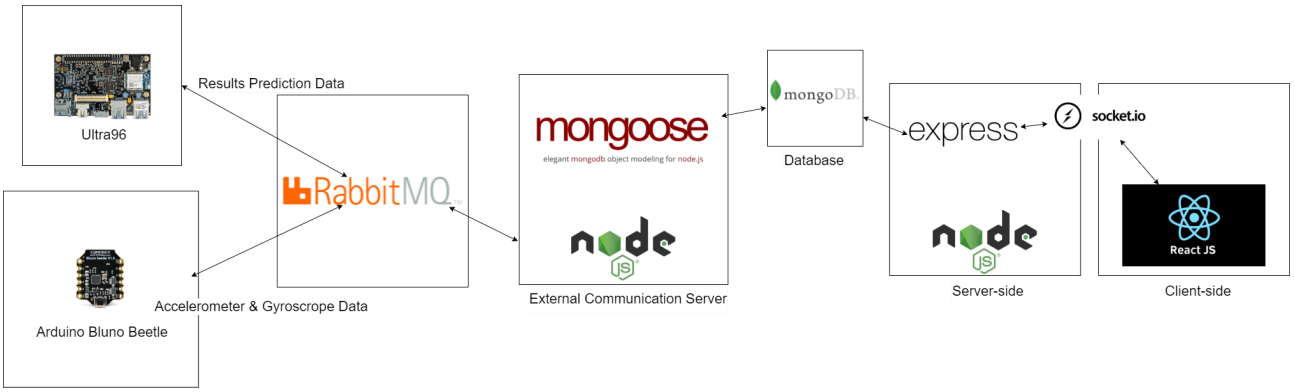


Figure 5.2.3-2: System Architecture Overview (Actual)

The system architecture used for the dashboard portion of DanceEdge revolves around the MERN stack - MongoDB, Express Framework, ReactJS library and NodeJS runtime environment. RabbitMQ was also used as a queue service to help in the transferring of data between the hardware and dashboard. This will be further explained in the next section.

Section 5.2.4 Technical Stacks

The MERN stack covers the entire development cycle from front-end to back-end using javascript (JS). The other technical stack considered was the MEAN stack - MongoDB, Express Framework, AngularJS and NodeJS. The sole difference between the two is the frontend framework AngularJS and the library ReactJS. MERN was chosen because ReactJS helps the developer code at a faster rate. In addition, the developer is more familiar with ReactJS than AngularJS.

As the developer of the dashboard is most familiar and comfortable with JS and javascript object notation (JSON), it made optimal sense to choose MERN. With one language across the client-side and server-side code, there is no need for context switching. In addition, the use of JSON data throughout the technical stack means that the data flows naturally from front to back, making it fast to build on and reasonably simple to debug.

MERN also supports the Model View Controller architecture pattern to make the development process flow smoothly.

Lastly, MERN has an extensive open-source community support with plenty of resources and documentation available for free. This significantly helps the developer in the long run with his dashboard design and implementation.

Section 5.2.4.1 Database - MongoDB

MongoDB was chosen to be the document database that stores persistent information from the Ultra96. It is one of the most popular non-relational (NoSQL) databases around. It was als worthy to note that MongoDB works extremely well with the chosen runtime environment - NodeJS.There were a couple of factors taken into consideration when choosing the appropriate database - relational vs non-relational database and data storage.

One alternative when choosing databases was PostgreSQL which is a traditional relational database management system. It has a strict schema which essentially prevents unstructured data from being inserted into the tables. In this sense, a non-relational database was preferred because it does not enforce a strict schema as compared to other relational databases. As a result of the high volume of streaming data inserted into the

database, it was preferred to have unstructured data to prevent potential errors caused by schema conflicts.

As for data storage, MongoDB was designed to store JSON data natively. It actually uses a binary version of JSON called BSON. It makes the storing, manipulating and representing of JSON data at every tier of the dashboard application easy. In the event we decided to change the way we represent data, MongoDB essentially has no downtime in schema changes.

MongoDB Atlas, a cloud database service, was used to create a database on the cloud. The database is hosted on the cloud because it allows easier access for the Ultra96 to insert data and for the backend of the dashboard to retrieve data. In addition, MongoDB Atlas provides secure end-to-end communication with the connected devices. This solves potentially another headache in terms of secure communications.

The connection to the MongoDB Atlas database will be set up via a MongoDB driver named Mongoose. The Mongoose library is an Object Data Modelling (ODM) library for MongoDB and NodeJS. It provides methods for the application to connect to the database server as well as to perform database operations. For instance, it is used to define a schema that is mapped to a MongoDB collection. Mongoose takes the schema and converts it into a model. This will be further illustrated later.

Section 5.2.4.2 Backend Framework - Express

Instead of writing full server-side code by hand on NodeJS directly such as NodeJS's HTTP module, Express was chosen as a framework to simplify the task of writing a representational state transfer (RESTful) API server code. Express is extremely flexible in the sense that it allows the developer to define routes of the application based on HTTP methods and URLs. In addition, it allows the developer to define an error handling middleware.

The framework is also known for its fast speed and minimalist structure. Express has great support from NodeJS and the Node Package Manager (NPM) packages available. These packages work great together with Express to extend its capabilities such as

authentication which will be discussed later. It has outstanding community support and plenty of resources online. Lastly, it is easy to connect to the MongoDB database.

KoaJS, which is a lightweight version of Express was also considered. However, KoaJS is still relatively new and its community support is still small. Thus, it made sense to stick to Express.

Section 5.2.4.3 Frontend Library - ReactJS

ReactJS was chosen as the javascript frontend library used for building reusable user interface components. As mentioned earlier, AngularJS was also considered. ReactJS was preferred because of the developer's familiarity with ReactJS and that AngularJS has a steeper learning curve than ReactJS. In addition, ReactJS allows the developer to have the freedom to choose any third party libraries and tools. AngularJS, being a framework, does not offer this flexibility.

Another big advantage of ReactJS over AngularJS is that ReactJS uses virtual Document Object Model (DOM) while Angular uses real DOM. DOM allows for the accessing and changing of document content, layout and structure. As the dashboard involves retrieving and displaying time series data, it adds to performance overhead if the entire page has to re-render due to changes in the data. Angular updates the entire tree structure of HTML tables until it reaches the needed data. On the other hand, ReactJS updates the changes without rewriting the entire HTML document virtually. Therefore, React.js is much faster and ensures faster performance. It is thus a good choice as it utilizes virtual DOM to re-render elements and minimize component updates whenever it's necessary.

Another key contention is data binding which is the synchronization of data between business logic and UI. Angular uses one and two way data binding. This means that changing data impacts view and changing view triggers changes in data. React, on the other hand, uses one-way binding. This means that child components are often nested within higher-order parent components. One-way binding makes the code more stable and debugging easier.

Section 5.2.4.4 Runtime Environment - NodeJS

Following the theme of using javascript as the sole language for both server and client side codes, NodeJS, a Google javascript runtime environment was chosen to run our dashboard on. It allows developers to build fast and scalable web applications.

One of the biggest advantages of NodeJS is its rich package manager, NPM that is widely available and has an increasingly heavy load of open-source tools. It is backed and actively maintained by a huge and ever-growing community. More importantly, it supports native JSON. This ties in very well with MongoDB.

Node.js operates on a single-thread, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections held in the event loop. Thus, it is well equipped to cope with multiple client requests which will be discussed later. It processes data quickly which allows for a quick data sync between client and server. In essence, NodeJS is the best environment to work on in terms of scalability and real-time data processing which is crucial to the dashboard features.

Section 5.2.4.5 Queue Service - RabbitMQ

The initial system architecture design was to send data immediately from the Ultra96 to MongoDB. However, that was not possible due to NUS' firewall. Hence, RabbitMQ was used as a workaround to send data between the hardware devices and the dashboard. RabbitMQ is an open-source message-broker. An alternative was to use Kafka. RabbitMQ was preferred because of its easy-to-use general purpose functionality.

Another tweak from the initial design was that messages were not solely sent from Ultra96. Once data was received in the Arduino Bluno Beetles, those values were sent to RabbitMQ. For prediction results, sync delay and accuracy, these were sent from Ultra96. This division of labour was done because of the respective implementations in the hardware code.

Section 5.2.5 Storing Incoming Sensor Data

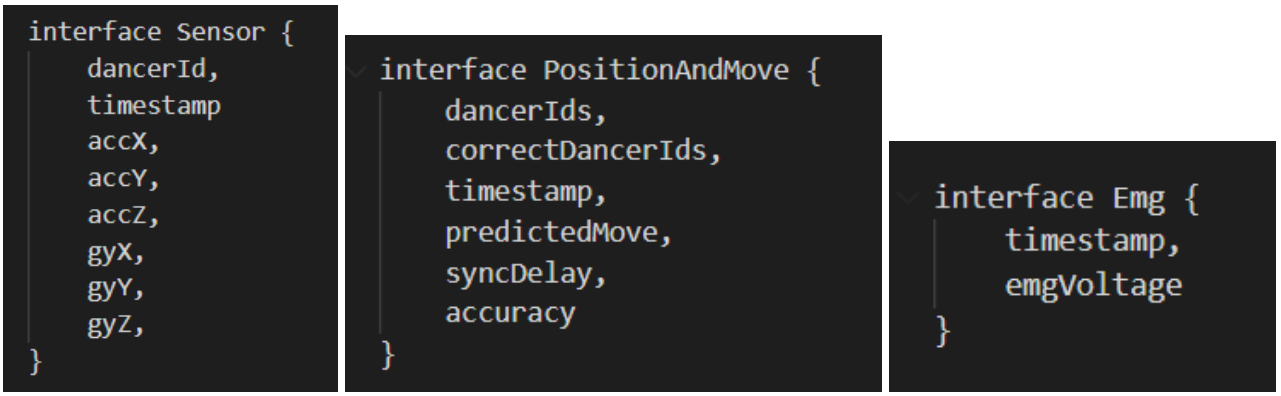


Figure 5.2.5-1: Schemas for data in MongoDB

The above are the listed schemas. They include the raw sensor readings from the accelerometer, gyroscope, dance synchronization delays, predicted positions and dance moves as well as EMG voltage values.

Section 5.2.6 Real-time Streaming

The overview of the data flows is as follows:

1. Data mentioned in the aforementioned section is pushed from the Ultra96 to MongoDB using a python library.
2. The backend server will listen to changes in the database using the Change Streams API function provided by MongoDB.
3. The frontever server will establish a connection with the backend server using socket.IO. This allows easy flow of data upon changes in the database.

Section 5.2.6.1 Data flow between Ultra96 and MongoDB

Any sensor and predicted data will be pushed from the Ultra96 to cloud database in MongoDB Atlas using a PyMongoAPI.

Section 5.2.6.2 Data flow between MongoDB and server-side

MongoDB has a Change Streams API which allows applications to subscribe to data changes within the database and react to them immediately. An alternative method for the backend to observe changes in the database is polling. However, polling is thought to be too computationally expensive for a real-time analytics application.

Section 5.2.6.3 Data flow between server-side and client-side

Socket.IO is a library that enables real-time, bi-directional and event-based communication between the client and server. It uses a web socket as a transport so that there is always a

connection open to pass data back and forth without the client requesting for it. It allows for a smooth operation whereby new data is sent directly from the backend to the frontend when Change Streams detects a change in the database.

Section 5.2.7 Authentication

In order to perform our login functionalities, JSON web token (JWT) and bcrypt libraries will be used. Jwt is one of the safest ways to authenticate HTTP requests. In a JWT flow, the token itself contains the data. The server decrypts the token to authenticate the user only. No data is stored on the backend server. JWT tokens are included in the authorization HTTP header as part of the bearer authentication scheme. JWT tokens are digitally signed by the server doing the authentication. In addition, bcrypt is used for password hashing.

Section 5.2.8 User Survey

Section 5.2.8.1 Pre-User Survey

User surveys are used to understand who users are, what do they want to achieve, what do they think of our product etc. In our regard, the user survey will be used to better understand the use cases of the hardware wearables and dashboard interface.

User Survey Methodology:

The survey will be conducted online. Due to the pandemic, we are restricted to ask within our team members. That being said, gathering the opinions of 6 different people from various backgrounds will provide sufficient results. While our ideal target audience are dance coaches and trainees, we will ask the members of the team to put themselves in the shoes of a dancer or even a friend of theirs who is an avid dancer. There are three parts to it - Understanding the user, wearable device and dashboard.

Understanding the user:

1. How has the COVID-19 pandemic affected your ability to learn dance moves?
   Purpose: Understand what is sorely lacking with the absence of physical interactions
2. What are some other alternatives that you have turned to, to learn how to dance?
   Purpose: Understand the competition and their use cases and user flows
3. What are some of the aspects you love about these alternatives?

Purpose: Understand what our competition is doing well at so that we can replicate that

4. What are some of the aspects that you crave or feel missing?
   Purpose: Understand what is lacking in the market so that we can capitalize on that

Wearable Device:

1. Accuracy, Weight, Comfortability and Durability (Power-consumption). Give a score of 1-10 with 10 being the highest on how much you value those four aspects.
   Purpose: Understand what our target audience values the most and priortize accordingly
2. Accuracy, Weight, Comfortability and Durability (Power-consumption). Rank these four aspects from 1-4 with 1 being the highest from the point of view that you would want to continue to wear the wearable device.
   Purpose: Understand what keeps our target audience engaged and fully in love with the wearable device
3. Low Accuracy, low latency, medium weight, dangling wires, uncomfortable feeling, hard to fit, high battery usage. Give a score of 1-10 with 10 being the highest on how much you do not want that feature as much as possible.
   Purpose: Understand what our target audience values the least and prioritize accordingly
4. Low Accuracy, low latency, medium weight, dangling wires, uncomfortable feeling, hard to fit, high battery usage. Rank these seven aspects from 1-7 with 1 being the highest from the point of view that you would not want to continue to wear the wearable device.
   Purpose: Understand what our target audience values the least and prioritize accordingly

Dashboard Design:

1. Would you prefer a visually appealing screen or a highly informative dashboard?
   Purpose: Understand what the users value more
2. Would you want to draw comparisons between previous dance routines?
   Purpose: Understand if that feature is necessary rather than just displaying plain history
3. What kind of analysis would you like to see added to our dashboard?
   Purpose: Open ended question to gather feedback

4. Which features of the current design do you feel are redundant and not as important?
   Purpose: Open ended question to gather feedback
5. Do you prefer graphs or numbers?
   Purpose: Understand what the user values more
6. Do you prefer many graphs or a singular graph that displays all of the information?
   Purpose: Understand if the accelerometer and gyroscope readings should be put into separate graphs or a single graph
7. What do you aim to understand from EMG readings and how do you plan to utilize them?
   Purpose: Understanding this helps to craft our analysis better and more tailored-made for the end user
8. What is the best representation of the synchronization delay? A few options are trends, graphs, statistics.
   Purpose: Understanding this helps to craft our analysis better and more tailored-made for the end user

Section 5.2.8.1 Post-User Survey

The User Survey was conducted between a small sample size of 5 people. Throughout the course of the module, the same few participants were asked about the improvements made to the product. Their responses have tremendously helped the UI/UX of the dashboard.

One improvement was the switch between light coloured background to a dark-ish mode theme. All the participants echoed the same sentiments that dark-mode aids users to look at the screen more effectively, especially with plenty of data changing rapidly.

Another improvement made was the removal of a position indicator icon. These icons were originally inside the wireframes to indicate a switch of positions for each trainee.However, after several iterations, it was clear that the positions icons made it even more confusing for users to look at.

All other improvements made were more implementation based. One final improvement that was major to the dashboard was the offline analytics. As seen in the sections above, the offline analytics in the wireframes is very different to the one that was actually implemented. This was because the user survey participants had advised that they preferred an analytics where they could immediately distinguish which positions and users were doing in place of the correct ones.

Section 6 Societal and Ethical Impact

Section 6.1 Future Use Cases

We have designed a general system that can be used for many different forms of human activity detection. This ranges from monitoring rehabilitating patients for their progress to general fitness tracking.

Other than fitness tracking and monitoring, the dance detector can be used as a low-cost game controller similar to Wii remote or Xbox kinect sensor, where the user can have a wide range of activities such as mini-games, or even fight bosses. Furthermore, this also allows the game developers to create unique games that are compatible with the dance detector.

In addition, this system can also be used in systems such as performance of athletes where data can be collected from sensors and be used to gather data with data analytics such as movements and G-force or stress that an athlete can take and also in early intervention for elderly especially in emergencies where IMUs together with other devices like camera can predict unexpected medical incidents like falling and alert the medical team to provide attention as soon as possible. In the case of the early intervention, the elderly and fall risk patients are more susceptible to falls and it is not practical to have constant supervision. Falling without detection can even be deadly. This is where technology can come in to help.

Also, another application can be a gesture recognition system. This can be used in smart homes for elderly or disabled people where similar to a dance move, a move or gesture can be detected and the machine learning model predicts what the move is. Certain gestures can be used to run certain functions like turning on and off the lights. Related to

that, our application can be used for sign language translation where based on the hand gestures, the system can detect and predict what he/she is trying to convey and then use a speaker to enable the others to understand or teach students which movements correspond to which actions or words.

Section 6.2 Privacy Concerns

The beetles uses a Linux operating system. It needs to be updated regularly. If it is not updated regularly, it may carry bugs which may be left undetected. Another potential security issue is that hackers may spoof MAC addresses of the bluno beetles. This allows them to tap onto the messages being sent from the bluetooth devices to the laptop. They maybe able to inject malicious messages as well. This ultimately poses security issues that could be solved with password protection for bluetooth connections.

The collection of data should adhere to the data protection laws like SIngapore's PDPA (Personal Data Protection Act). This will ensure that sensitive data from the people who participated in the data collection does not get leaked out and are taken after people's consent. We will have to take utmost care to ensure that we use and store their data properly. Hence, the user should not just agree to use the product but also give consent to the product having access to their data. The data should remain in local storage as much as possible and not get transmitted anyway.

In addition, manufacturers also need to ensure that the system is well-secured and there are secure pairing methods to prevent common attacks on BLE like passive eavesdropping, man-in-the-middle attacks or identity tracking. Only with stringent measures, malicious attacks like hijack can be prevented or at least reduced.

Also, it would be good to have government laws and enforcement measures to ensure that these kinds of systems are not used for nefarious purposes especially by the manufactures too. This will prevent manufactures from doing unethical practices like extracting private data as well too.

Section 6.3 Ethical

Not every product is perfect, especially in the case of ours where it has such a short production time. Trainees can come across shortcuts to dance moves that makes it seem like they are doing the dance moves when they actually are not. This defeats the purpose

of the product. With no actual life feed implemented in the product, coaches would have no idea if trainees are actually dancing. To combat this, a more thorough product testing cycle is needed. We will need good training sets as well too.

Also if the wearable is not doing its function properly, then the question is which party will take the responsibility. Would it be the fault of the manufactures or the people using the wearable? There should be an efficient way to detect and decide whether the users used the wearable wrongly or is it the designers' fault before the products can be distributed widely.

Also, if the government or entities want to make the device compulsory, should it be mandated compulsory for all the people like elderly and fall risk patients? One should ask whether is it ethically sound to force something onto someone? As such, it is important that the user agrees to wearing such a product and is not forced to wear.

Section 7 Project Management Plan

Week\Role	HW Sensors	HW FPGA	Comms Internal	Comms External	SW ML	SW Dashboard
5	Create algorithm to detect movement	Implement linear layer	Finish the connection setup and transmission of data via packet format between 1 beetle and a laptop	Finish the client and server with TCP and be able to send back the required package to the evaluation server.	Explore useful features Explore machine learning models	Get basic design of the pages up (login, main dashboard and history pages), work on login feature, create test data
6	The wirings of the wearable for one dancer are complete	Implement convolutional layer	Implement handshaking Implement stable and concurrent connection between 3 beetles and the laptop for at least 1 minute	Be able to build the multi-threads server and clients. Make sure that they can run on the Ultra96.	Train quantized network Collect training data	Continue to create test data, link components in react and continue to work on design (Dashboard and history pages)
Recess Week	Test and debug wearable	Integrate all layers and test with random data	Test and fix any issues in the transmission of the data.	Finish the week 7 individual demo part	Collect training data Train model	Test fake readings in database with dashboard

			Start implementing a reconnection function to reestablish connection in events of disconnections.			
7	The wirings of the wearable for other two dancers are complete	Test layers with actual weights but random data	Prepare, test and debug for 1st evaluation. Fix any issues or modify code based on any issues or advise during the evaluation session. Implement packet reassembly algorithm on the laptop.	Build stable TCP servers and clients which connect the laptops, Ultra96 and evaluation server.	Finetune model Integrate model with FPGA	Fine tune design and analysis of data for individual subcomponen t test
8	Fine tune the algorithm and make changes if necessary	Test layers with actual data and fine-tune FPGA implementation	Integrate with other components, especially with external comms Setup connection between 2 beetles and a laptop for 3 trainees Improve the speed and reliability of the data transmitted Verify that the reconnection functions work in events of disconnections and improve the time taken for reconnection	Integrate with the Internal comms part and be able to send the required data for the ML part. Build the server and client for the dashboard	Integrate model with FPGA Run end to end inference	Work on Demo pages, Settings and Availability
9	Configure wearable to save more power if possible	Optimize latency	Finish integration with other components especially with external comms (if not done yet) Modify the communication protocol to suit	Integrate with ML and send the learning outcome to the evaluation server with correct format. Improve the accuracy of synchronizatio	Run end to end inference Speed up inference	Test real-time data with dashboard

			any needs if required Perform rigorous testing and fix any bugs encountered. Identify any exceptions and handle them	n delay		
10	Test and debug wearable	Start to prepare for 2nd evaluation test	Carry out rigorous testing and fix any bugs encountered. Identify any exceptions and handle them. Start to prepare for 2nd evaluation test	Integrate with all other parts and make sure the information can be transferred correctly and the sync delay can be calculated properly.	Measure final accuracy and inference time Identify potential failure points	Test real-time data with dashboard
11	Test and debug wearable	Prepare, test and debug for the 2nd evaluation test (3 users with 3 moves, and tracking location).	Prepare, test and debug for the 2nd evaluation test (3 users with 3 moves, and tracking location). Fine tune the entire communication protocol if needed. Implement any improvements if any.	Test and debug for the final evaluation.	Run end to end inference Test and debug software	Test real-time data with dashboard
12	Resolve any issues that surface from the 2nd evaluation test. Test and debug wearable	Resolve any issues that surface from the 2nd evaluation test.	Resolve any issues that surface from the 2nd evaluation test. Prepare, test and debug for final evaluation Start writing final report	Resolve any issues that surface from the 2nd evaluation test. Test and debug for the final evaluation.	Resolve any issues that surface from the 2nd evaluation test. Test and debug software	Resolve any issues that surface from the 2nd evaluation test. Test and debug software
13	Final evaluation & prepare final report	Final evaluation & prepare final report	Final evaluation & prepare final report	Final evaluation & prepare final report	Final evaluation & prepare final report	Final evaluation & prepare final report

Bibliography

Alden, D. (2016, January 15). *Project Hub*. How to "Multithread" an Arduino (Protothreading Tutorial). https://create.arduino.cc/projecthub/reanimationxp/how-to-multithread-an-arduino-prototothreading-tutorial-dd2c37

Anguita, D., Ghio, A., Oneto, L., Parra, X., & Reyes-Ortiz, J. L. (2013, April 26). *A Public Domain Dataset for Human Activity Recognition Using Smartphones*. Human Activity Recognition Using Smartphones Data Set. https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones

Arduino Memory Information. (2018, February 5). Memory of Arduino. Retrieved February 6, 2021, from https://www.arduino.cc/en/tutorial/memory

Atmel. (n.d.). *8-bit Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash*. https://www.mouser.com/pdfdocs/gravitech_atmega328_datasheet.pdf

Brains, B. (n.d.). *Experiment: EMGs during Muscle Fatigue*. Backyard Brains. Retrieved February 3, 2021, from https://backyardbrains.com/experiments/fatigue

Brownlee, J. (2018, September 21). *1D Convolutional Neural Network Models for Human Activity Recognition*. Machine Learning Mastery. https://machinelearningmastery.com/cnn-models-for-human-activity-recognition-time-series-classification/

cliffgi. (2016, December). *Bluno Beetle BLE Power Consumption*. Robotshop Community. https://www.robotshop.com/community/forum/t/bluno-beetle-ble-power-consumption/27208

Condes, E. (n.d.). *arduinoFFT*. https://github.com/kosme/arduinoFFT

Conv1D. (n.d.). Pytorch. Retrieved February 6, 2021, from https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html

DFRobot. (n.d.). DFRobot Bluetooth 4.1 BLE User Guide. Retrieved February 4, 2021, from https://wiki.dfrobot.com/DFRobot_Bluetooth_4.1__BLE__User_Guide

DFRobot. (n.d.). *DFR0339 Bluno Beetle*. Wiki DFRobot. https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339

Electronic Cats. (n.d.). *MPU6050 by Electronic Cats*. https://github.com/ElectronicCats/mpu6050

Energizer. (n.d.). *Energizer E92 Product Datasheet*. https://data.energizer.com/pdfs/e92-1119.pdf

ethen8181. (n.d.). *model_selection*. Github. Retrieved February 3, 2021, from http://ethen8181.github.io/machine-learning/model_selection/model_selection.html

FreeRTOS Support. (2020, January 02). FreeRTOS FAQ – Memory Usage, Boot Times & Context Switch Times. https://www.freertos.org/FAQMem.html#RAMUse

Frumusanu, A. (2020, November 17). *Putting Apple Silicon M1 To The Test*. AnandTech. Retrieved February 6, 2021, from https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

Harvey, I. (n.d.). *BluePy*. Bluepy 0.9.11 documentation. http://ianharvey.github.io/bluepy-doc/peripheral.html#peripheral

Howard, C. (2014, January 25). *Heterogeneous computing combines benefits of CPUs, GPUs, and FPGAs*. Military Aerospace. Retrieved February 6, 2021, from https://www.militaryaerospace.com/computers/article/16718743/heterogeneous-computing-combines-benefits-of-cpus-gpus-and-fpgas

InvenSense. (n.d.). *MPU-6000 and MPU-6050 Product Specification*. https://components101.com/sites/default/files/component_datasheet/MPU6050-DataSheet.pdf

Learn, S. (n.d.). *ExtraTreesClassifier*. Scikit-Learn. Retrieved February 3, 2021, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

Linear. (n.d.). Pytorch. Retrieved February 6, 2021, from https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

Malekzadeh, Clegg, M., Cavallaro, R. G., Haddadi, A., & Hamed. (2019, 1 1). *Mobile Sensor Data AnonymizationMobile Sensor Data Anonymization*. Github. https://github.com/mmalekzadeh/motion-sense

Malyi, V. (2017, July 19). *Run or Walk A dataset containing labeled sensor data from accelerometer and gyroscope*. Kaggle. https://www.kaggle.com/vmalyi/run-or-walk

MyoWare. (n.d.). *3-lead Muscle / Electromyography Sensor for Microcontroller Applications*. https://github.com/AdvancerTechnologies/MyoWare_MuscleSensor/blob/master/Documents/AT-04-001.pdf

Orosco, E., López, N. M., & Sciascio, F. d. (2018, March 1). *Bispectrum-based features classification for myoelectric control*. ResearchGate. https://www.researchgate.net/publication/257690526_Bispectrum-based_features_classification_for_myoelectric_control

Quantization. (n.d.). Pytorch. Retrieved February 6, 2021, from https://pytorch.org/docs/stable/quantization.html

Rose, W. (2011, April 27). *Mathematics and Signal Processing for Biomechanics*. Electromyogram analysis. https://www1.udel.edu/biology/rosewc/kaap686/notes/EMG%20analysis.pdf

Seidel, I. (n.d.). *ArduinoThread*. https://github.com/ivanseidel/ArduinoThread

sparkfun. (n.d.). *Analog to Digital Conversion*. https://learn.sparkfun.com/tutorials/analog-to-digital-conversion/all

Stack Overflow. (n.d.). Maximum packet length for Bluetooth LE? Retrieved February 1, 2021, from https://stackoverflow.com/questions/38913743/maximum-packet-length-for-bluetoothle.

Sun, X. (2019, October 21). *Human Activity Recognition Using Smartphones Sensor Data*. Medium. https://medium.com/@xiaoshansun/human-activity-recognition-using-smartphones-sensor-data-fd1af142cc81

Toro, S. F. d., Santos-Cuadros, S., Olmeda, E., Álvarez-Caldas, C., Díaz, V., & Román, J. L. S. (2019, July 20). *Is the Use of a Low-Cost sEMG Sensor Valid to Measure Muscle Fatigue?* https://www.mdpi.com/1424-8220/19/14/3204

Walsh, J. (2013, June 1). *Why is the DMP yaw stable?* I2Cdevlib Forums. https://www.i2cdevlib.com/forums/topic/10-why-is-the-dmp-yaw-stable/

Xilinx. (2018). *UG902*. Xilinx User Guide. Retrieved February 6, 2021, from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf