CG4002 Computer Engineering Capstone Project

AY 2021/22 Semester 1

Group 12

"Dance Dance"

Design Report

Group 12	Name	Student #	Sub-Team	Role	Section Contribution
Member #1	Chan Hong Yi, Matthew	A0183924X	Hardware	Hw Sensors	2.2, 2.3, 2.4, 3.1, 6
Member #2	Nishanth Elango	A0184373Y	Hardware	Hw FPGA	3.2, 1.1, 2.3, 6
Member #3	Divakaran Haritha	A0187915N	Communications	Comms Internal	4.1, 2.3, 6
Member #4	Lim Hao Xiang, Sean	A0187123H	Communications	Comms External	2.1, 2.3, 2.4, 2.5 4.2, 6, 7
Member #5	Priyan Rajamohan	A0187872L	Software	Sw Machine Learning	1, 5.1, 2.3, 6
Member #6	Lim Chek Jun	A0183389M	Software	Sw Dashboard	2.3, 5.2, 6

Table of Contents

Section 1 System Functionalities

1.1 Introduction

The aim of this project is to develop a gamified system that detects dance moves and relative positions of a 3-person dance group and provides real-time feedback on the performance. Due to the COVID-19 pandemic and the resulting restrictions, it has become increasingly difficult for groups of people to meet in person. Hence, this entire system is built to function remotely, allowing its users to dance together virtually from the comfort of their homes.

Each dancer is equipped with a pair of wearable devices containing a Beetle BLE and an IMU sensor. The sensor data is transmitted to a remote processor (Ultra96), which processes the sensor data using an FPGA hardware accelerator to determine the dance move being executed, with high accuracy and low latency. The system also detects the relative position of the dancers, and the synchronization delay between the fastest and slowest dancer, which are displayed on a dashboard together with the predicted dance move. Overall, this system acts as a dance coach for its users, providing real-time feedback through an interactive dashboard.

1.2 Use Case Diagram

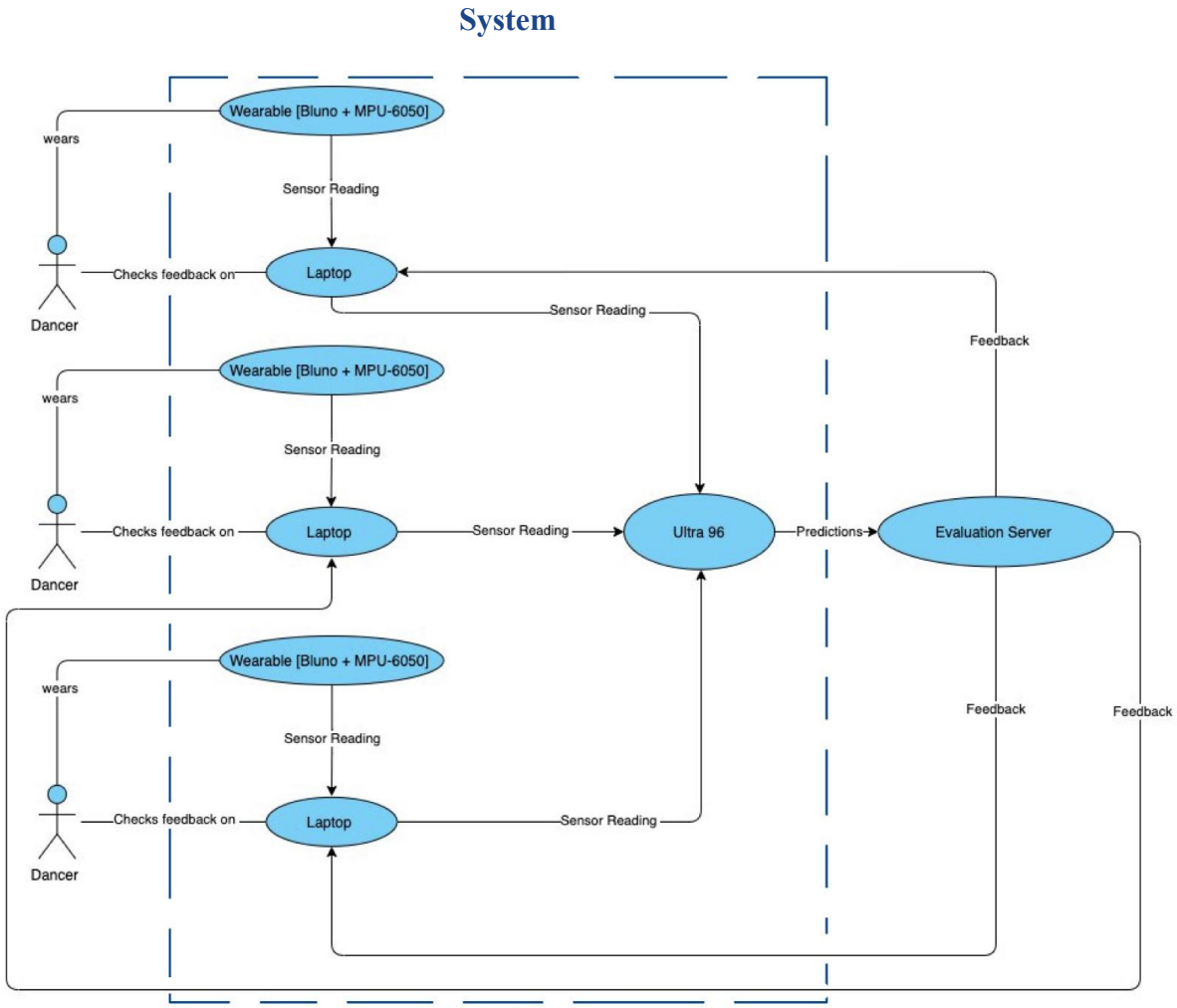


Fig. 1.2.a: Use case diagram - *system highlighted as a blue box*

This system is able to handle 3 virtual dancers who wish to know if they are performing a dance move correctly. Each dancer is equipped with a wearable that consists of a Bluno Beetle and the Inertial Measurement Unit - MPU-6050. Accelerometer and gyroscope readings from the IMU are collected and transmitted by the Bluno Beetle to each dancer's respective laptops via Bluetooth Low Energy (BLE). The laptops act as a relay channel and further transmit sensor

readings to the Ultra96. On the Ultra96, a machine learning model predicts the dance move performed by the dancer and sends the predictions to the evaluation server. The evaluation server checks if the predicted dance move matches the target dance move displayed to the dancer and returns feedback. Dancers can view their predicted moves on the interactive dashboard. The system is also capable of detecting changes in relative positions of dancers with the same flow as detection of dance moves.

1.3 System Features

Our system comes with the following unique and useful features to achieve the goal of activity detection:

Lightweight wearable

Our system provides a lightweight and compact wearable that contains all the required sensors. This makes it portable and comfortable to wear so it will not impose on the user's dancing.

Long Battery Life

The wearable is designed to have a battery life of at least 1 hour, which is sufficient to last through dance rehearsals.

Dance Move Prediction

Our system utilizes Machine Learning to find patterns within the sensor readings from the dancer's movement and identify which dance move the dancer is performing. We are able to distinguish between 8 unique dance moves.

Relative Position Detection

Our system is able to detect each dancer's position relative to the other dancers in the group. This feature helps each dancer be more aware of his surroundings to fit into the dance better.

Measure of Synchronization among dancers

Our system will measure the level of synchronization among dancers performing the same dance move. This feedback will help dancers be more coordinated and together in their dancing.

Fatigue Analysis

Our system will monitor the activity and fatigue level of our dancers over the course of the dance. This will give dancers more awareness of their physical limits and helps them set realistic goals. This will also help prevent potential injury due to excessive dancing and physical exertion.

Real-time feedback

Our system comes with a dashboard for viewing real-time feedback on the dancer's performance. This is an intuitive way for dancers to receive immediate feedback so they can make adjustments to their dance moves and coordination.

Intuitive User Experience

Our system uses a unique "logout" dance move to signify the termination of the system. This move requires the user to roll their hands. This is easy and intuitive for the user to terminate the program.

1.4 User Stories

Our system aims to provide coaching services to dancers of all levels of interest and experiences. Some of the user stories are as shown below:

As a ...	I want to ...	So that ...
Novice dancer	Have a system that provides feedback on my dance moves	I can improve on my dancing skills
Clumsy dancer	Have a lightweight wearable without lots of wiring	I can dance freely without getting entangled
Introverted dancer	Dance without feeling awkward about others watching me	I can strictly focus on improving my dance moves
Busy dance enthusiast	Have a wearable that can run on low power	I don’t have to set reminders to charge frequently
Leisure dancer	Have a system which allows for virtual dance sessions with my friends	I don't have a compulsion to be physically present

Section 2 Overall System Architecture

2.1 High-Level System Architecture

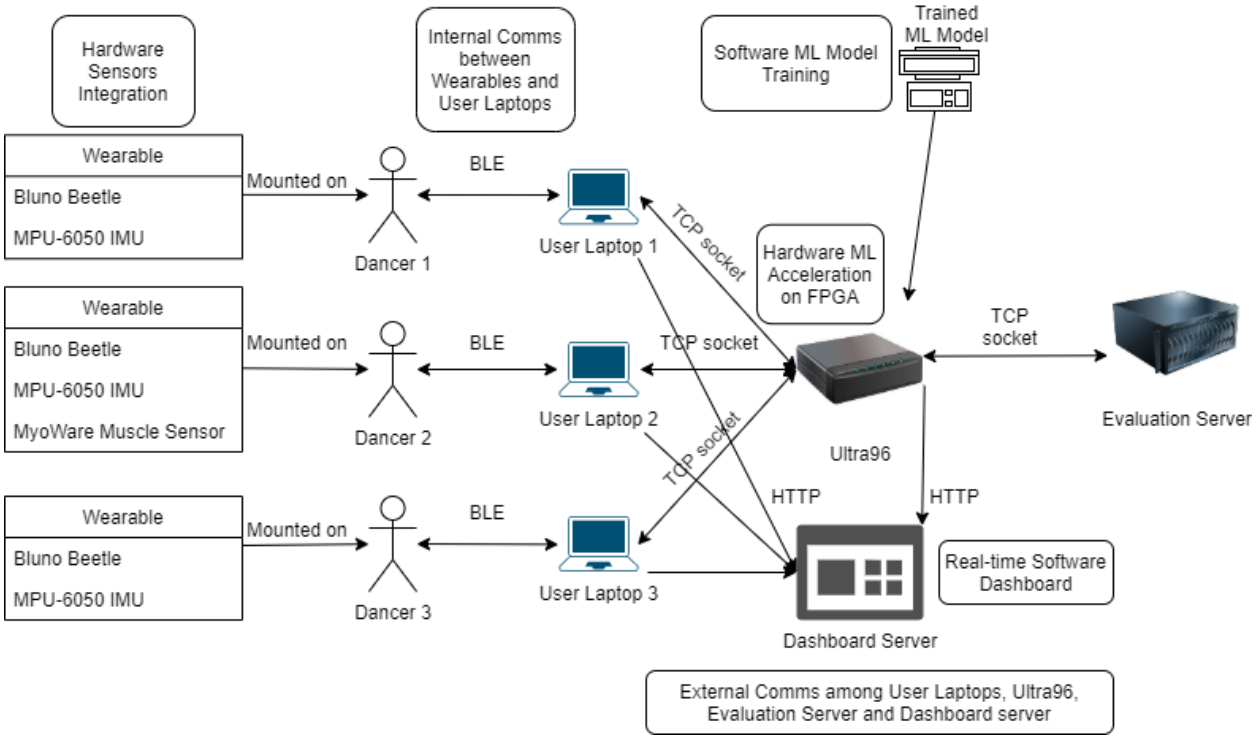


Fig. 2.1.a: Overall Architecture Diagram

Each dancer puts on a wearable device to detect the dance movements. Each wearable consists of 1 Bluno Beetle and MPU-6050 IMU sensor. The IMU sensor is connected to the Bluno and contains an accelerometer and a gyroscope for movement detection. The sensor readings are used for dance move prediction and relative position change detection. In addition, there is one wearable that is mounted with an extra MyoWare Muscle Sensor also connected to the Bluno. This is used to detect the muscle fatigue of the dancer.

The Bluno on a wearable connects to a User Laptop via Bluetooth Low Energy (BLE) for data transmission by 3-way handshake protocol. The data read by the beetle is then preprocessed into the correct packet size and sent to the laptop by periodic push.

Next, each User Laptop establishes a secure socket connection with the Ultra96 using the Transmission Control Protocol (TCP). From here, each User Laptop processes the data and forwards it to the Ultra96. The User Laptop also sends real-time sensor data to the dashboard for real-time feedback.

To detect the dance moves, we incorporated Machine Learning (ML) into our system. Prior training of the ML model is done using the sensor data collected from our dancers. Once fully trained, the model is able to predict the dance move from the live sensor values from the dancers.

To speed up the prediction of our trained ML model, the Ultra96 serves as a hardware neural network accelerator. The trained ML model is implemented on the programmable logic of the Ultra96 to accelerate the processing and computation executed by the model.

Once a prediction is returned from the ML model, the Ultra96 establishes a secure TCP socket connection with the Evaluation Server to send the predicted dance move, relative dancer positions and synchronization delay between dancers. In addition, the Ultra96 sends relevant data to the Dashboard Server as real-time feedback to the dancers.

The Dashboard displays real-time information back to the dancers for immediate feedback on their dance moves and other useful information such as their fatigue level and synchronization delay.

2.2 Final Form of System

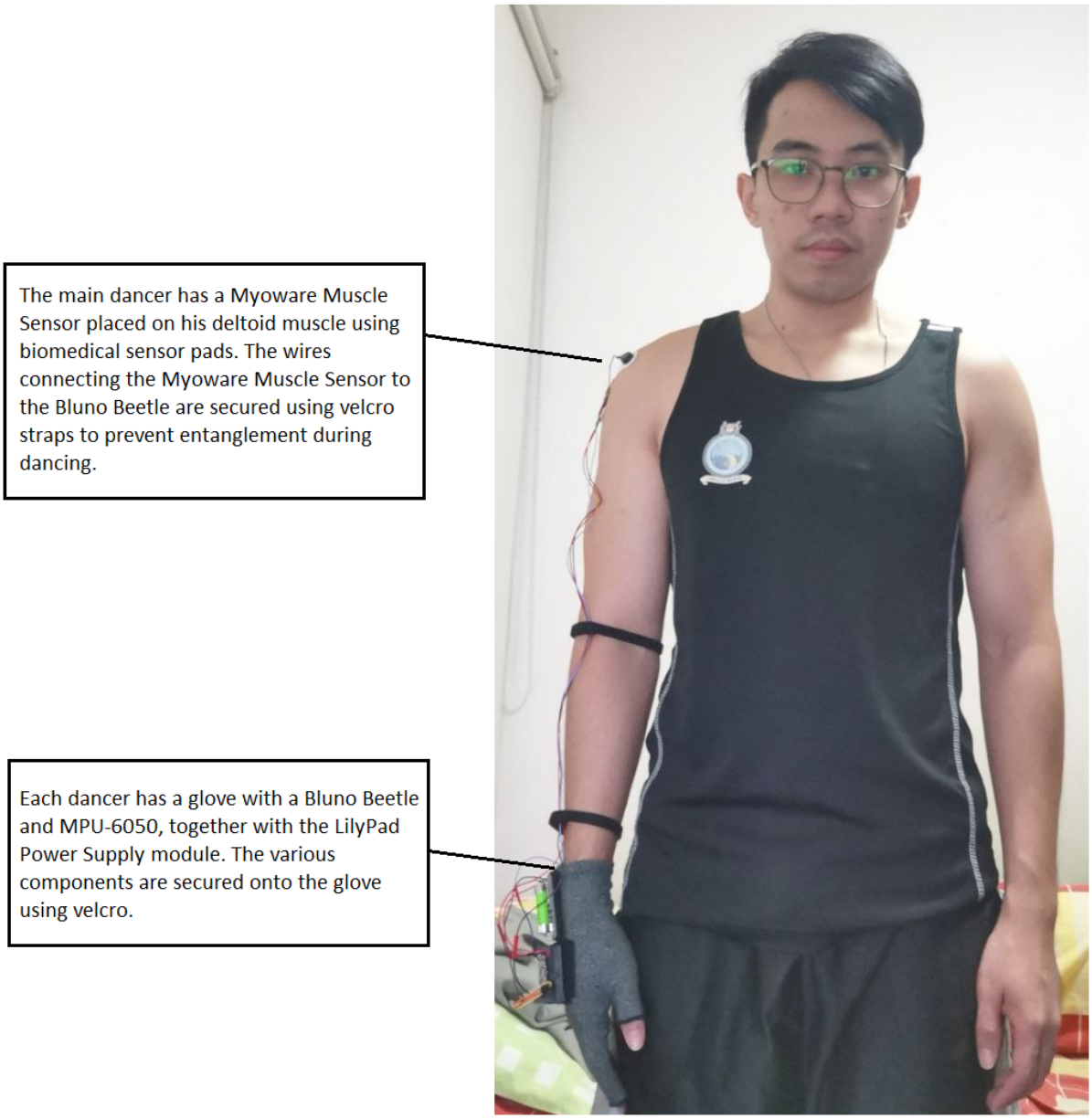


Fig. 2.2.a: Depiction of final wearable

We decided to place the Myoware Muscle Sensor on the shoulder of the main dancer as many of the dance moves involve lifting on the arms which uses the deltoid muscles. The initial design involved using a second wearable at the waist to measure lateral movement more accurately by avoiding confusion with dance moves. However, we were able to achieve high accuracy for

relative position detection with just rotational data from the wrist wearable. We also implemented algorithms to ensure that position changes are not confused with dance moves. Hence, we removed the superfluous belt wearable for the final design.

Fig. 2.2.b: Final glove wearable

Header pins were soldered onto the Bluno Beetle and it is connected to a breadboard for easy connection of the circuitry. The MPU-6050 is also connected to the breadboard. The breadboard and the LilyPad Power Supply module are secured onto the gloves using velcro to prevent them from coming off during dancing.

2.3 Main Algorithm for Dance Move Prediction

The main algorithm used for dance move prediction can be split into the following sections:

Sensors

1. Bluno collects accelerometer and gyroscope readings from the IMU sensor at 20Hz and preprocesses them.
2. When sensor readings exceed a predefined "isDancing" threshold, it means that the dancer is dancing and sets the isDancing bit to '1'. This is to detect dancing movements by the dancers.

Laptop

1. Upon successful connection, Laptop constantly receives data packets from Bluno and initializes dancingState to be False.
2. When dancingState is False, incoming IMU sensor packets are ignored.
3. If a received packet has the isDancing bit set to '1', it signifies that the dancer is dancing and sets dancingState to True.
4. While dancingState is True, incoming packets are consolidated into sets of 42 samples to be preprocessed and sent to the Ultra96.
5. When a predefined number of consecutive packets received have the isDancing bit set to '0', it means that the dancer is idle and hence deemed to have stopped dancing. Hence, dancingState is set to False .

Ultra96

1. Ultra96 receives packets of preprocessed sensor data.
2. The preprocessed data is passed into the hardware-accelerated ML model to derive a prediction of the dance move.
3. Each dancer is required to obtain 3 valid dance move predictions.
4. Once each dancer has obtained 3 predictions each, the dance move which is predicted the most is chosen as the final submission.

2.4 Main Algorithm for Relative Position Detection

For positional change of the dancers, we standardized the movement of positions to follow the following steps:

1. Dancer makes a rotational turn on the spot to the direction of the position change.
2. Dancer walks to the new position.
3. Dancer makes a rotational turn on the spot back to the front.

E.g. to move from the leftmost position to the center position, the dancer must first turn to face the right, walk to the center position, and turn back to face the front.

We also standardized that dancers perform positional change if needed before performing the dance move.

The main algorithm used for relative position detection can be split into the following sections:

Sensors

1. When the IMU sensor readings have been below the predefined "isDancing" threshold for at least 2 seconds, the sensors start to detect for turns.
2. Based on the gyrometer Y-axis readings, if the value exceeds a predefined positive threshold, it is considered a left turn. If the value exceeds a predefined negative threshold, it is considered a right turn.
3. Once a turn is detected, the direction of the turn is sent to the Laptop.
4. Subsequently, the sensor stops detecting for turns until the condition in Step 1 is met.

Laptop

The movement from the turn is likely to cause sensor readings to exceed the predefined "isDancing" threshold as well. These packets should not be seen as sensor packets for dance move prediction. Likewise, motions from performing dance moves could cause the gyrometer Y-axis readings to exceed the threshold meant to detect turns, causing a false detection of a turn. The motion of a turn is a short motion followed by no movement. Hence, the expected value of the isDancing bit for the subsequent consecutive packets is a short burst of packets with

isDancing bit set to ‘1’ with the remaining packets with isDancing bit set to ‘0’. The next few steps use this idea to verify that the received turn is indeed a real turn.

1. Upon receiving a directional change packet containing the direction of the turn, the Laptop script checks a predefined number of subsequent consecutive packets after the directional change packet as a means of confirming the validity of the turn.
2. Out of the subsequent packets, if less than a certain predefined number of packets have its isDancing bit set to ‘1’, it means that the isDancing bit of the subsequent packets follow the trend that we had expected, thus verifying the validity of the turn. If the number of packets with isDancing bit set to ‘1’ exceeds that predefined number, it is a sign that the dancer is engaging in a more physically rigorous activity i.e. performing a dance move. In this case, the turn is deemed as a false detection and hence rejected.
3. The subsequent consecutive packets are also not treated as sensor packets for dance move prediction as sensor values from a turn should not correspond to any dance move.

Ultra96

The Ultra96 receives the direction of turns from each dancer. The direction of the turn corresponds to the direction of movement that the dancer will need to shift to. For the given situation of 3 dancers, for every permutation of the starting position of the dancers, there is a unique combination of position shifts of each dancer that would map to any new position.
E.g. for starting position “1 2 3” where each number refers to the ID of a dancer and its position in the string represents their relative positions, the new position mappings are as follows:

Directional shift of dancer 1	Directional shift of dancer 2	Directional shift of dancer 3	New positions
No change	No change	No change	1 2 3
Right	Left	No change	2 1 3
No change	Right	Left	1 3 2
Right	No change	Left	3 2 1
Right	Right	Left	3 1 2
Right	Left	Left	2 3 1

The steps for handling directional change packets are as follows:

1. Ultra96 initializes the starting positions of dancers as “1 2 3” and initializes the directional shifts of each dancer to all be “No change”.
2. Upon receiving a directional change packet, Ultra96 updates the corresponding dancer’s directional shift.
3. Once sufficient dance move predictions have been obtained, the directional shifts are also confirmed as we standardized each dancer to perform positional change before performing the dance move.
4. The new positions are then obtained using the unique combination of directional shifts from each dancer.
5. In the event of errors such that no valid mapping of directional change is available, the system finds the closest match for the directional shifts and obtains the corresponding new positions of the dancers.

2.5 Main Algorithm for Sync Delay Calculation

Sync Delay is defined as the difference in time between the start of the dance move of the earliest dancer and that of the latest dancer. The main algorithm used for sync delay calculation can be split into the following sections:

Sensors

1. On the sensor level, it detects movements from the idle position.
2. It uses a threshold for the sensor values to detect if the dancer is in the idle position or is currently dancing. More details have been explained in Section 2.3 under “Sensors”.

Laptop

1. The Laptop’s clock is synchronized with the Ultra96 as its main reference clock through the Clock Synchronization protocol in Section 4.2.5 and the offset is saved on the Laptop.
2. With reference to the Laptop’s dancingState as explained in Section 2.3 under “Laptop”, we are able to keep track of when the dancer enters and exits the dancingState.

3. At the point at which the dancer starts dancing (i.e. dancingState gets set to ‘1’), the current timestamp is recorded on the Laptop and subtracted by the offset.
4. The timestamp is sent to the Ultra96 together with the sensor data.
5. The Clock Synchronization protocol will be run after each submitted dance move to maintain the accuracy of the Laptop’s clock offset relative to the Ultra96’s clock.

Ultra96

1. The Ultra96 will wait for timestamps from the Laptop which correspond to the start of the dancer’s dance move.
2. Once a timestamp is received, it will be saved in each dancer’s state.
3. Once all the timestamps are received, the Sync Delay is calculated by subtracting the latest timestamp with the earliest timestamp.
4. For each new dance move shown by the Evaluation Server, the Ultra96 clears previously stored timestamps from the dancers and waits for new timestamps.

Section 3 Hardware Details

Section 3.1: Hardware sensors

3.1.1 Introduction

In this section, we will be listing the hardware components that are required for our wearable device. The datasheets of the various components are also provided for reference.

3.1.2 Bluno Beetle DFR0339

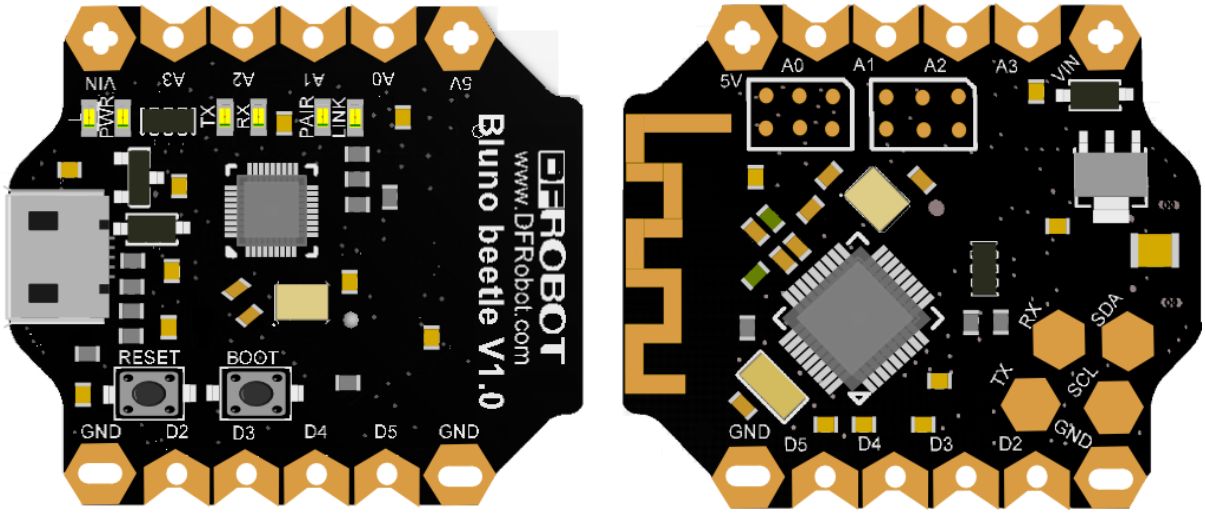


Fig. 3.1.a: Front and back view of the Bluno Beetle [1]

The Bluno Beetle is a board based on Arduino Uno with Bluetooth 4.0 (BLE) [2]. It is small and lightweight, which makes it suitable for wearables and this is the reason why we have chosen to use it for our device. We will be using the Bluno Beetle to gather and process the sensor readings before transmitting the data to the laptop via Bluetooth.

Datasheet: https://www.application-datasheet.com/pdf/dfrobot/dfr0339.pdf

3.1.3 GY-521 MPU-6050 IMU

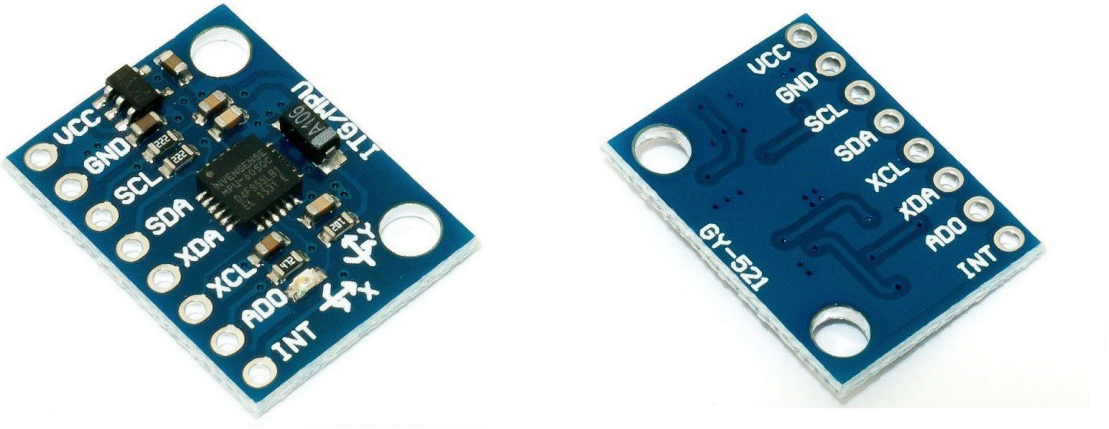


Fig. 3.1.b: Front and back view of the MPU-6050 IMU [3]

The MPU-6050 IMU is a sensor which contains a 3-axis gyroscope and a 3-axis accelerometer [3]. This allows us to measure the angular velocity and acceleration in the x, y and z axes. By analysing this data, we will be able to determine the dance moves performed by the dancers. At only 21.2mm x 16.4mm x 3.3mm and 2.1g [4], the MPU-6050 is small and lightweight, making it suitable for our wearable device.

Datasheet: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

3.1.4 MyoWare Muscle Sensor AT-04-001

Fig. 3.1.c: MyoWare Muscle Sensor [5]

The MyoWare Muscle Sensor is an electromyography (EMG) sensor which measures muscle activity [5]. This data allows us to determine the fatigue level of a dancer's muscle and analyse the performance of the dancer.

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

3.1.4 LilyPad Power Supply Module

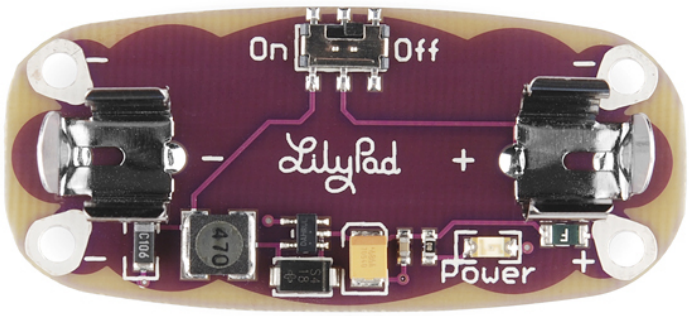


Fig. 3.1.d: Lilypad Power Supply [6]

The LilyPad Power Supply Module is a small power supply module which takes in a AAA battery and provides a 5V supply. It comes with AAA battery clips and a power switch, with short circuit protection. It is only 56 x 26mm, which makes it an ideal power supply module for our wearable device.

3.1.5 Schematics

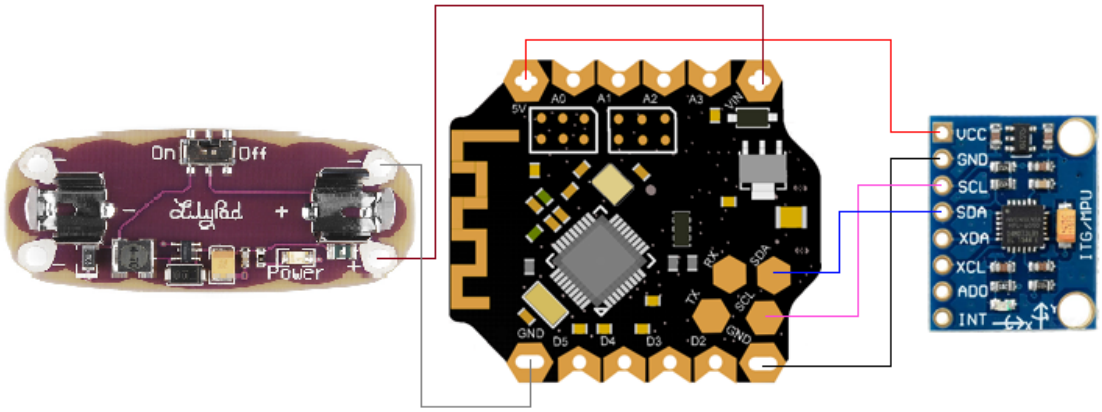


Fig. 3.1.e: Schematic of LilyPad Power Supply, Bluno Beetle and MPU-6050

Figure 3.1.D shows the connections between the Bluno Beetle and MPU-6050. The LilyPad Power Supply provides the 5V input voltage for the Bluno Beetle. The MPU-6050 is powered by

the 5V positive supply from the Bluno Beetle. The SCL and SDA pins are used to establish communication between the Bluno Beetle and MPU-6050 via I2C serial communication protocol.

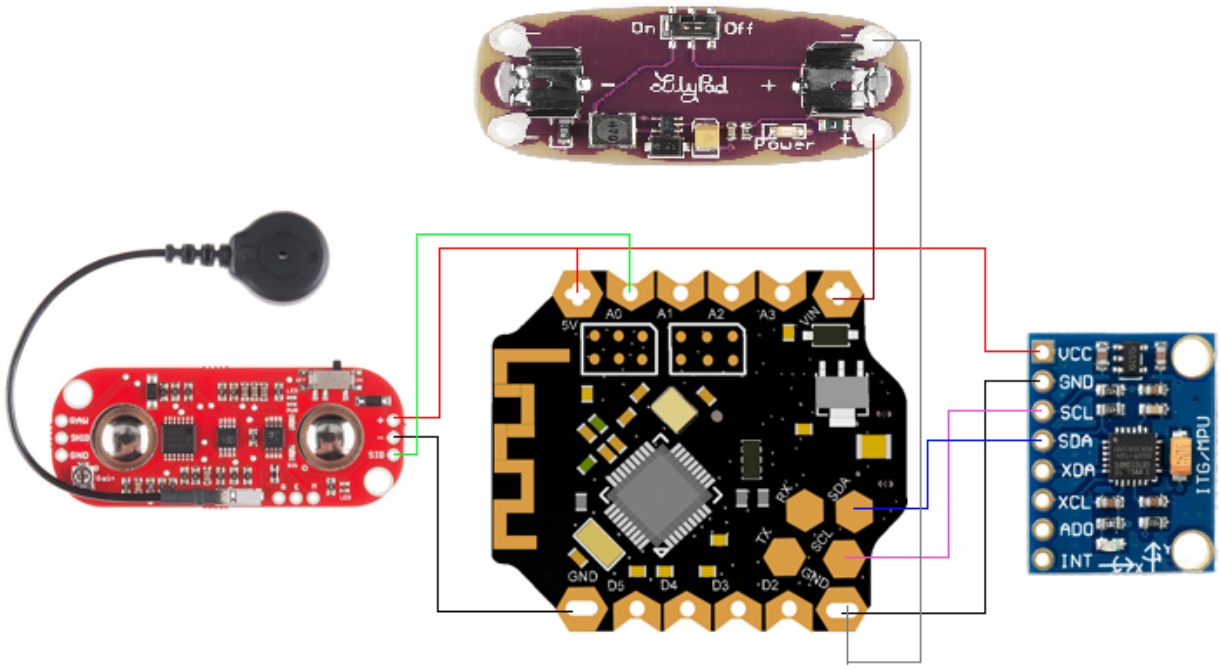


Fig. 3.1.f: Schematic of LilyPad Power Supply, Bluno Beetle, MPU-6050 and MyoWare Muscle Sensor

Figure 3.1.E is similar to Figure 3.4 but with an additional MyoWare Muscle Sensor, which is only used on one of the dancers. The MyoWare Muscle Sensor is also powered by the 5V positive supply from the Bluno Beetle. The output signal pin of the MyoWare Muscle Sensor is connected to one of the analog pins on the Bluno Beetle.

3.1.6 Operating voltage and current

Component	Operating voltage	Operating current
Bluno Beetle	5V	10mA
MPU-6050 IMU	3.3V to 5.0V	3.8mA
MyoWare Muscle Sensor	3.3V or 5V	9mA

3.1.7 Power system design

While choosing the power source for our device, we took into account the size, weight, battery life and cost. We decided to use rechargeable batteries since it will be more environmentally friendly and cost effective in the long run.

Load Current without MyoWare Muscle Sensor = 10mA + 3.8mA = 13.8mA
Load Current with MyoWare Muscle Sensor = 10mA + 3.8mA + 9mA = 22.8mA
This is used to calculate the Battery Life = Battery Capacity / Load Current.

Form Factor	Voltage	Weight	Battery Capacity	Battery Life	Voltage Regulator
NiMH AA	6 x 1.2V	162g	2100mAh	92-152h	Step down
NiMH 9V	1 x 8.4V	42.5g	200mAh	8-14h	Step down
NiMH AAA	1 x 1.2V	14g	800mAh	35-58h	Step up

After considering the various factors, we have decided to go with the NiMH AAA rechargeable battery since it is lighter and less bulky than the other batteries. Since the nominal voltage of the

battery is lower than the input voltage 5V of the Bluno Beetle, we used the LilyPad Power Supply module to step up the voltage to 5V.

From our testing, we discovered that the battery lasts around two hours before it drops below the 1.2V minimum input voltage of the LilyPad Power Supply. This is sufficient for our usage as a full run typically takes less than half an hour and we can easily replace the battery with a fully charged one when its voltage drops.

3.1.8 Algorithms and libraries

```
void setup() {
  Serial.begin(115200);           //  initialize serial communication
  Wire.begin();                   //  join I2C bus

  Wire.beginTransmission(MPU);    //  start communication with MPU6050
  Wire.write(0x6B);               //  talk to register 6B
  Wire.write(0);
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x1C);               //  talk to register 28 (ACCEL_CONFIG)
  Wire.write(0x08);               //  set AFS_SEL to 1 (+-4g)
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x1B);               //  talk to register 27 (GYRO_CONFIG)
  Wire.write(0x00);               //  set FS_SEL to 0 (+-250deg/s)
  Wire.endTransmission(true);

  Wire.beginTransmission(MPU);
  Wire.write(0x1A);               //  talk to register 26 (CONFIG)
  Wire.write(0x06);               //  set DLPF_CFG to 6 (5Hz digital low pass filter)
  Wire.endTransmission(true);

  calculate_MPU_error();
  delay(20);

  for(i = 0; i < 100; ++i) {
    myoware[i] = 0.0;
  }

  previousTimeMPU = millis();
  prevDancingTime = millis();
  currentTimeGyro = millis();
}
```

Fig. 3.1.g: Arduino setup code

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
ACCELEROMETER SENSITIVITY						
Full-Scale Range	AFS_SEL=0		±2		*g*	
	AFS_SEL=1		±4		*g*	
	AFS_SEL=2		±8		*g*	
	AFS_SEL=3		±16		*g*	
ADC Word Length	Output in two's complement format		16		bits	
Sensitivity Scale Factor	AFS_SEL=0		16,384		LSB/*g*	
	AFS_SEL=1		8,192		LSB/*g*	
	AFS_SEL=2		4,096		LSB/*g*	
	AFS_SEL=3		2,048		LSB/*g*	
Initial Calibration Tolerance			±3		%	
Sensitivity Change vs. Temperature	AFS_SEL=0, -40°C to +85°C		±0.02		%/°C	
Nonlinearity	Best Fit Straight Line		0.5		%	
Cross-Axis Sensitivity			±2		%	

Fig. 3.1.h: Accelerometer sensitivity

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
GYROSCOPE SENSITIVITY						
Full-Scale Range	FS_SEL=0		±250		°/s	
	FS_SEL=1		±500		°/s	
	FS_SEL=2		±1000		°/s	
	FS_SEL=3		±2000		°/s	
Gyroscope ADC Word Length			16		bits	
Sensitivity Scale Factor	FS_SEL=0		131		LSB/(°/s)	
	FS_SEL=1		65.5		LSB/(°/s)	
	FS_SEL=2		32.8		LSB/(°/s)	
	FS_SEL=3		16.4		LSB/(°/s)	
Sensitivity Scale Factor Tolerance	25°C	-3		+3	%	
Sensitivity Scale Factor Variation Over Temperature			±2		%	
Nonlinearity	Best fit straight line; 25°C		0.2		%	
Cross-Axis Sensitivity			±2		%	
GYROSCOPE ZERO-RATE OUTPUT (ZRO)						

Fig. 3.1.i: Gyroscope sensitivity

In the setup() function of the Arduino code, the sensitivity of the accelerometer and gyroscope is set to be +-4g and +-250deg/s respectively.

DLPF_CFG	Accelerometer (F_s = 1kHz)		Gyroscope		
	Bandwidth (Hz)	Delay (ms)	Bandwidth (Hz)	Delay (ms)	Fs (kHz)
0	260	0	256	0.98	8
1	184	2.0	188	1.9	1
2	94	3.0	98	2.8	1
3	44	4.9	42	4.8	1
4	21	8.5	20	8.3	1
5	10	13.8	10	13.4	1
6	5	19.0	5	18.6	1
7	RESERVED		RESERVED		8

Fig. 3.1.j: Digital low pass filter

We used the built-in 5Hz digital low pass filter of the MPU-6050 to filter out high frequency noise.

```
void calculate_MPU_error() {        //  Place IMU flat to calculate accelerometer and gyro data error
  while (c < 200) {                 //  Read accelerometer values 200 times
    Wire.beginTransmission(MPU);
    Wire.write(0x3B);
    Wire.endTransmission(false);
    Wire.requestFrom(MPU, 6, true);
    accX = (Wire.read() << 8 | Wire.read()) / 8192.0 ;
    accY = (Wire.read() << 8 | Wire.read()) / 8192.0 ;
    accZ = (Wire.read() << 8 | Wire.read()) / 8192.0 ;
    // Sum all readings
    accErrorX = accErrorX + ((atan((accY) / sqrt(pow((accX), 2) + pow((accZ), 2))) * 180 / PI));
    accErrorY = accErrorY + ((atan(-1 * (accX) / sqrt(pow((accY), 2) + pow((accZ), 2))) * 180 / PI));
    c++;
  }

  accErrorX = accErrorX / 200;      //  Divide the sum by 200 to get the error value
  accErrorY = accErrorY / 200;
  c = 0;

  while (c < 200) {                 // Read gyro values 200 times
    Wire.beginTransmission(MPU);
    Wire.write(0x43);
    Wire.endTransmission(false);
    Wire.requestFrom(MPU, 6, true);
    gyroX = Wire.read() << 8 | Wire.read();
    gyroY = Wire.read() << 8 | Wire.read();
    gyroZ = Wire.read() << 8 | Wire.read();
    // Sum all readings
    gyroErrorX = gyroErrorX + (gyroX / 131.0);
    gyroErrorY = gyroErrorY + (gyroY / 131.0);
    gyroErrorZ = gyroErrorZ + (gyroZ / 131.0);
    c++;
  }
  gyroErrorX = gyroErrorX / 200;  //  Divide the sum by 200 to get the error value
  gyroErrorY = gyroErrorY / 200;
  gyroErrorZ = gyroErrorZ / 200;
}
```

Fig. 3.1.k: Error calculation code

For error calculation, 200 readings are taken while the MPU-6050 is placed on a flat surface and the average error is calculated to be used for offset when the readings are taken. This is done every time the Bluno Beetle is reset to ensure accuracy. The code is adapted from an online MPU-6050 tutorial [7].

```
roll = 0.96 * gyroAngleX + 0.04 * accAngleX;        //  complimentary filter, accelerometer and gyro angle values are combined to remove noise
pitch = 0.96 * gyroAngleY + 0.04 * accAngleY;
```

Fig. 3.1.l: Complementary filter

A complementary filter was used to eliminate gyroscope drift error. Accelerometer and gyroscope angle values are combined to find roll and pitch.

For the detection of the start of dance moves, we used thresholding to determine when the dancers are in idle state, i.e. not dancing.

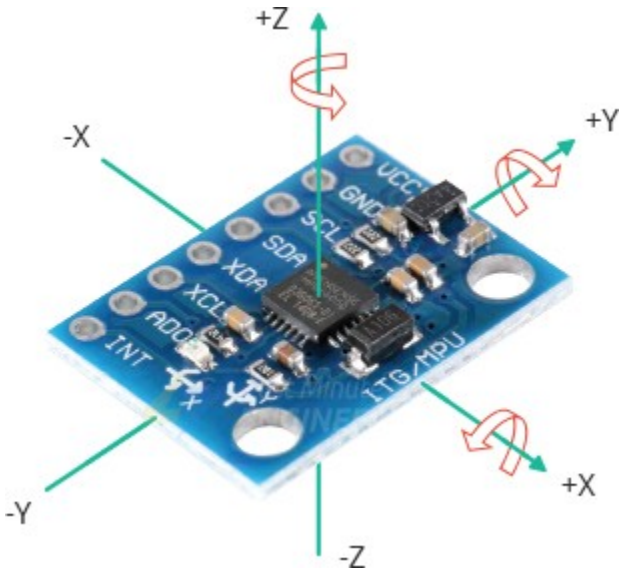


Fig. 3.1.m: MPU-6050 axes [8]

In idle state, the dancers have their arms by the sides of their bodies. The acceleration in the Y-axis should be close to 1g (acceleration due to gravity) and the acceleration in the X-axis should be close to 0g.

```
if((accX > 0.25 || accX < -0.25) && (accY > 1.10 || accY < 0.90)) {   //  detection of start of dance move
 isDancing = 1;
 dancingCount++;
 if(dancingCount > 25){
  isPositionChangeDone = false;
  dancingCount = 0;
  prevDancingTime = millis();

 }


} else {
   isDancing = 0;
   dancingCount = 0;
}
```

Fig. 3.1.n: Start of dance move detection

After some testing, we managed to choose the threshold values to detect the start of a dance move using the isDancing flag.

```
if((currentTime - previousTimeMyoware) > (1000 / MYOWARE_SAMPLING_RATE)) {
  myowareSensor = analogRead(A0);
  movingAverage = (0.9 * movingAverage) + (0.1 * myowareSensor);

  if(j > 100) {
    j = 0;
  }

  myoware[j] = myowareSensor - movingAverage;

  //only for dancer sending EMG DATA
  if(j == 100) {
    RMS = calculateRMS();
    MAV = calculateMAV();
    ZCR = calculateZCR();
    if(confirmed){
      //sendemg();
    }
  }

  previousTimeMyoware = currentTime;
  j++;
}
```

Fig. 3.1.o: EMG code

For EMG data, a buffer of 100 samples was used and an exponential moving average filter was applied to remove baseline fluctuation due to DC bias.

```
int calculateRMS() {
  float sum = 0;
  for(i = 0; i < 100; ++i) {
    sum += pow(myoware[i], 2);
  }
  float result = pow((sum / 100), 0.5) * 100;
  return result;
}

int calculateMAV() {
  float sum = 0;
  for(i = 0; i < 100; ++i) {
    sum += abs(myoware[i]);
  }
  float result = sum;
  return result;
}

int calculateZCR() {
  float crossing = 0;
  bool isPositivePrevious = false;
  bool isPositiveNext;
  for(i = 1; i < 100; ++i) {
    isPositiveNext = myoware[i] > 0;
    if(isPositiveNext != isPositivePrevious) {
      crossing++;
    }
    isPositivePrevious = isPositiveNext;
  }
  float result = crossing;
  return result;
}
```

Fig. 3.1.p: EMG feature extraction

We extracted three features for the EMG data, namely Root-mean-square (RMS), Mean absolute value (MAV) and Zero-crossing rate (ZCR). The RMS and MAV allows us to analyse the amplitude of the EMG signal and contraction of the muscle. The ZCR allows us to estimate muscle fatigue since studies show that ZCR decreases as the muscle fatigues [9].

Section 3.2: Hardware FPGA

3.2.1 Ultra96 workflow

The workflow for implementing the hardware accelerator is as follows:

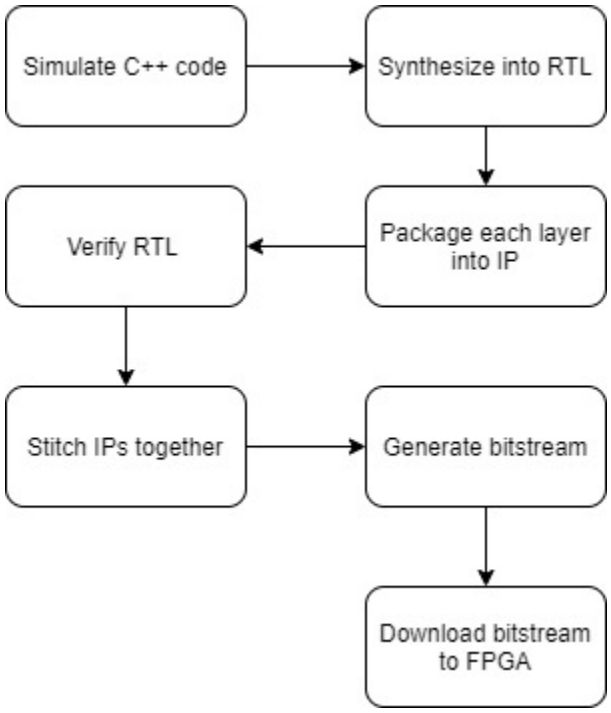
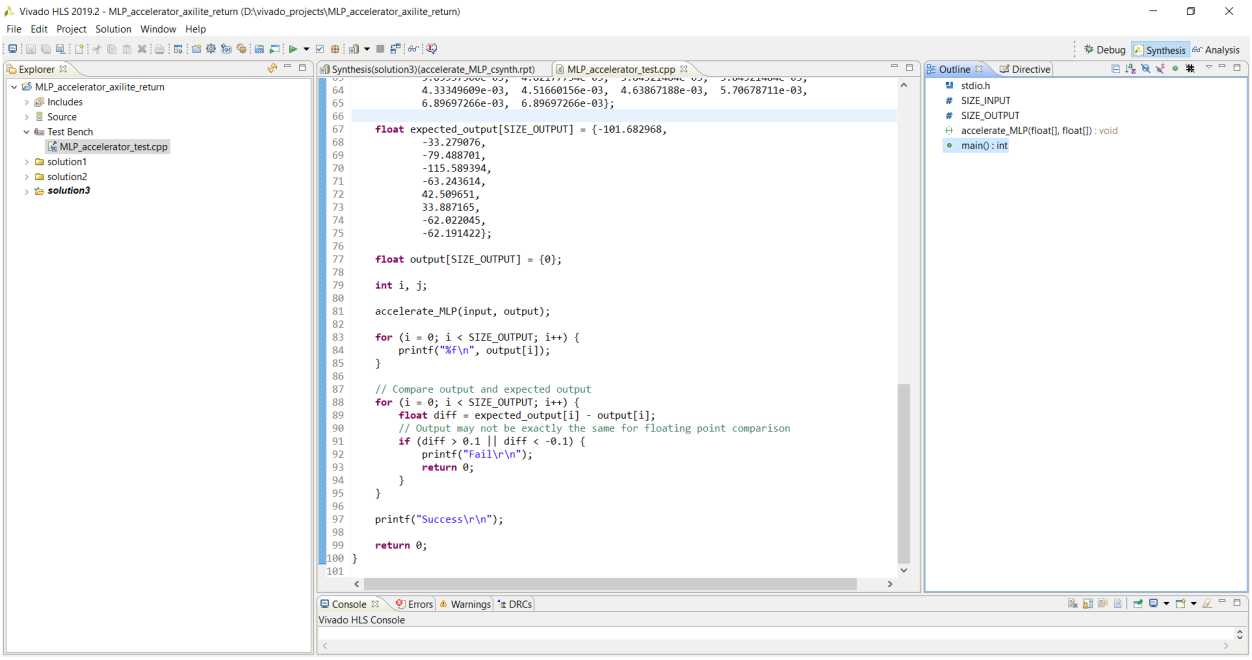


Fig. 3.2.a: FPGA workflow

Development of the neural network accelerator is done using the Vivado HLS application. The neural network is implemented in C++ HLS, which is synthesized into an RTL implementation provided in Hardware Description Language (HDL) formats, including Verilog and VHDL. C/RTL Cosimulation is used to verify that the generated RTL is functionally identical to the C++ implementation. The verified RTL is then exported as IP into Vivado Design Suite. This process is repeated for each layer in the neural network, after which all the layer IPs can be stitched together using the IP integrator tool. Finally, a PYNQ overlay is used to program the FPGA using the generated bitstream.

Final Implementation

For simulation, a C++ testbench is used. This was used to verify both the C++ algorithm as well as the RTL implementation using the C/RTL Cosimulation tool in Vivado HLS.




```
64              4.33349609e-03,  4.51660156e-03,  4.63867188e-03,  5.70678711e-03,
65              6.89697266e-03,  6.89697266e-03};
66
67      float expected_output[SIZE_OUTPUT] = {-101.682968,
68              -33.279076,
69              -79.488701,
70              -115.589394,
71              -63.243614,
72              42.509651,
73              33.887165,
74              -62.022045,
75              -62.191422};
76
77      float output[SIZE_OUTPUT] = {0};
78
79      int i, j;
80
81      accelerate_MLP(input, output);
82
83      for (i = 0; i < SIZE_OUTPUT; i++) {
84          printf("%f\n", output[i]);
85      }
86
87      // Compare output and expected output
88      for (i = 0; i < SIZE_OUTPUT; i++) {
89          float diff = expected_output[i] - output[i];
90          // Output may not be exactly the same for floating point comparison
91          if (diff > 0.1 || diff < -0.1) {
92              printf("Fail\r\n");
93              return 0;
94          }
95      }
96
97      printf("Success\r\n");
98
99      return 0;
100 }
101
```

Fig 3.2.b: C++ testbench

Due to the additional overhead and complexity involved in passing data between IPs, the RTL for the entire neural network model is exported as a single IP. An AXI4-Lite interface is used to pass data between the Processing System (PS) and Programmable Logic (PL) in the ultra96 design. This Memory Mapped Input Output (MMIO) interface is chosen over a streaming interface such as AXIS as streaming would not provide much benefit due to the system's requirement that all inputs must be received before processing can start.

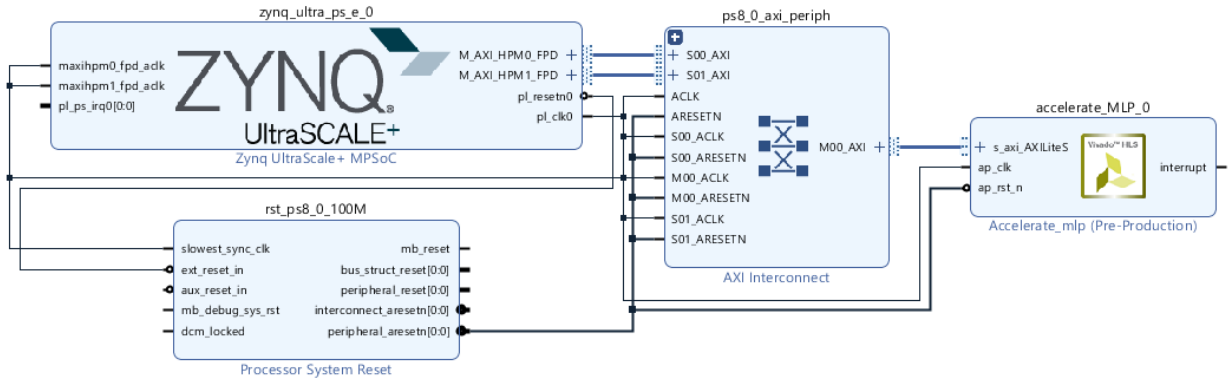


Fig 3.2.c: Vivado block design

A Pynq Driver script is used to program the FPGA using the generated bitstream and handle low level reads and writes. The neural network IP is accessed using an Overlay object that is created using the bitstream.

```
# Initialize pynq Overlay to program FPGA using bitstream
self.overlay = Overlay('MLP.bit')
self.accelerate_mlp = self.overlay.accelerate_MLP_0
```

Fig 3.2.d: Programming FPGA using bitstream

Writing and reading of data are done on specific memory addresses, obtained from vivado HLS.

```
// AXILiteS
// 0x000 : Control signals
//         bit 0  - ap_start (Read/Write/COH)
//         bit 1  - ap_done (Read/COR)
//         bit 2  - ap_idle (Read)
//         bit 3  - ap_ready (Read)
//         bit 7  - auto_restart (Read/Write)
//         others - reserved
// 0x004 : Global Interrupt Enable Register
//         bit 0  - Global Interrupt Enable (Read/Write)
//         others - reserved
// 0x008 : IP Interrupt Enable Register (Read/Write)
//         bit 0  - Channel 0 (ap_done)
//         bit 1  - Channel 1 (ap_ready)
//         others - reserved
// 0x00c : IP Interrupt Status Register (Read/TOW)
//         bit 0  - Channel 0 (ap_done)
//         bit 1  - Channel 1 (ap_ready)
//         others - reserved
// 0x400 ~
// 0x7ff : Memory 'inputs' (210 * 32b)
//         Word n : bit [31:0] - inputs[n]
// 0x800 ~
// 0x83f : Memory 'outputs' (9 * 32b)
//         Word n : bit [31:0] - outputs[n]
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

Fig 3.2.e: Programming FPGA using bitstream

The write() and read() functions are then used on the IP to access the specific addresses. Values are converted to integers before writing as MMIO does not support writing of floats.

```
# Write input values to IP
for i in range(INPUT_LENGTH):
    int_input = self.float_to_integer(inputs[i])
    self.accelerate_mlp.write(INPUT_OFFSET + i*NUM_BYTES, int_input)

# Start executing IP
self.accelerate_mlp.write(CONTROL_OFFSET, AP_START)

# Wait for AP_DONE bit to be asserted when outputs are ready
while self.accelerate_mlp.read(CONTROL_OFFSET) & AP_DONE == 0:
    continue

# Read output values from IP
result = []
for i in range(OUTPUT_LENGTH):
    res = self.accelerate_mlp.read(OUTPUT_OFFSET + i*NUM_BYTES)
    result.append(self.integer_to_float(res))
```

Fig 3.2.f: Low level reads and writes

3.2.2 Neural network model

An MLP neural network is implemented as follows:

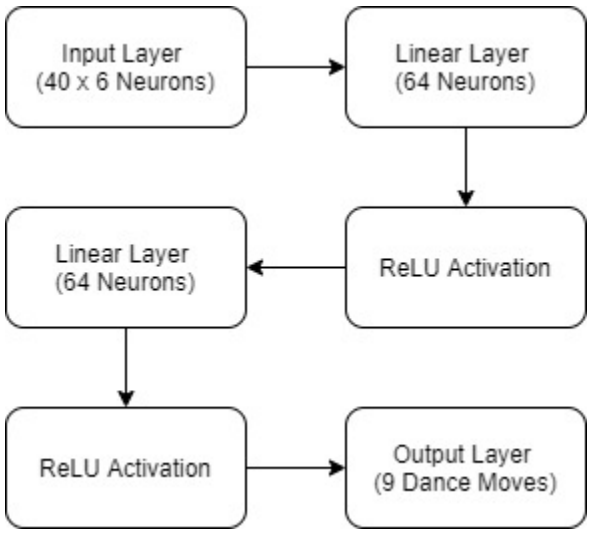


Fig. 3.2.g: MLP architecture

The parameters such as weights for the model are extracted from the trained model implemented by Software ML.

The input layer initially consisted of 40 datasets, each containing 6 axis data. The size of the input layer was reduced after feature extraction was done by software ML. The linear layers involve matrix multiplication, which are implemented using operations on 2D arrays in loops in C++. The ReLU activation layer involves a simple function that returns max(0, input), implemented using a conditional block (if else statement) in C++ code or an LUT in hardware.

Quantization of the floating point weights and inputs into 8 bit integers were considered, as hardware computation speed would be increased for integers. The tradeoff between accuracy and

performance were evaluated to determine if quantization should be used in our final model. It was not used in the final model, due to reasons explained in section 3.2.4.

Given the limited number of hardware resources, it may be important to reduce the complexity of the network. This can be achieved through neural network pruning, to remove weights or neurons from the network without affecting accuracy significantly.[20]

Final implementation

The proposed MLP design is used as it achieved a high accuracy, while not being resource intensive. Only the sizes of the layers were modified to 210, 150, and 60 for the input and 2 hidden layers respectively.

- Fully connected linear layer:

$$y = w \cdot x + b$$

where
y = output, 1 x i vector (i = number of nodes in current layer)
w = weights, i x j vector (j = number of nodes in previous layer)
x = input, 1 x j vector
b = bias, 1 x i vector

- ReLU activation function:

$$y = \max(0, x)$$

where

y = output

x = input

Fig. 3.2.h: Mathematical basis for neural network

The hidden layer and ReLU activation function logic is implemented in C++ as follows:

```
// Fully connected layer 1
loop_1: for (i = 0; i < SIZE_LAYER_1; i++) {
    sum = 0;
    loop_2: for (j = 0; j < SIZE_INPUT; j++) {
        sum += inputs[j] * weights_layer_1[i][j];
    }
    sum += bias_layer_1[i];
    //ReLU activation function
    if (sum < 0) {
        output_layer_1[i] = 0;
    } else {
        output_layer_1[i] = sum;
    }

}
```

Fig. 3.2.i: C++ code for hidden layer

This logic is replicated for the second hidden layer and output layer.

3.2.3 Neural network accelerator evaluation

Evaluation of the accelerator will be done using several metrics as follows:

1. Accuracy of inference model

The accuracy of the model can be determined by feeding a large amount of data to the model, and calculating the percentage of inference outputs that correspond to the correct classification. The accuracy of the FPGA implementation can then be compared with that of the software ML implementation.

2. Hardware utilization

The amount of FPGA resources such as BRAMs, DSPs, FFs, and LUTs can be obtained from the Vivado HLS synthesis report. Lower utilization of these resources would result in lower area usage and power consumption.

3. Latency

Latency refers to the number of clock cycles required for the function to output all values. A lower latency would correspond to a lower processing delay and hence better performance. This metric can be obtained from the synthesis report generated by Vivado HLS.

Final Implementation

The latency and utilization estimates were obtained from the synthesis report generated by Vivado HLS.

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
371526	371526	3.715 ms	3.715 ms	371526	371526	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	459	-
FIFO	-	-	-	-	-
Instance	4	5	617	598	-
Memory	81	-	32	5	0
Multiplexer	-	-	-	508	-
Register	-	-	679	-	-
Total	85	5	1328	1570	0
Available	432	360	141120	70560	0
Utilization (%)	19	1	~0	2	0

Fig. 3.2.j: Synthesis report

The latency of less than 4ms means an almost instantaneous output from human perspective. Close to 0% of FFs, LUTs and DSPs were used, with BRAMs having the highest usage at only 19%. The accuracy of the model was measured by passing in test samples to the IP using the Pynq Overlay and calculating the proportion of correct outputs. The accuracy was 94%, matching the accuracy of the software implementation.

3.2.4 Potential optimization

Vivado HLS provides several tools and directives that can be used to optimize the RTL design.[14]

1. Loop unrolling

By default, all loops are left rolled, meaning that each iteration of the loop is performed in separate clock cycles. By unrolling the loops, multiple iterations of a loop can run in parallel in

the same clock cycle, hence reducing latency. This can be achieved using the UNROLL directive provided by Vivado HLS. If a loop is fully unrolled, all iterations of the loop can run concurrently if there are sufficient resources and data dependencies allow for it. If fully unrolling is not feasible due to hardware or data dependency limitations, loops can also be partially unrolled by a specified factor. To unroll loops, we can add the following line to the loop body:

#pragma HLS unroll factor=<N>

This will unroll the loop by a factor of N. If the factor is not specified, the loop is fully unrolled.

2. Pipelining

Pipelining can be used to increase throughput. Pipelines can be implemented on a task level, allowing multiple functions or loops to execute concurrently, or at an operation level, where operations within loops are parallelized. Task level pipelining can be implemented by adding the DATAFLOW directive as follows:

#pragma HLS dataflow

Operation level pipelining can be added by using the PIPELINE directive:

#pragma HLS pipeline II=<int>

Where II refers to the initiation interval, which defaults to 1 if not specified.

3. Array Partitioning

C++ arrays are synthesized into BRAMs by default using HLS. BRAMs usually have a maximum of 2 data ports, allowing only 2 values to be read or written in a single clock cycle. This can result in bottlenecks where a large number of values need to be accessed from an array. Using the ARRAY_PARTITION directive splits up the array into multiple BRAMs, improving access to data and hence increasing performance.

Final Implementation

Several optimization methods were tested as follows:

Solution 1: No optimization

Solution 2: Loop unrolling

Solution 3: Loop pipelining + fixed point quantization

Latency

		solution1	solution2	solution3
Latency (cycles)	min	437727	196052	49146
	max	438327	196652	49146
Latency (absolute)	min	4.377 ms	2.062 ms	0.491 ms
	max	4.383 ms	2.069 ms	0.491 ms
Interval (cycles)	min	437727	196052	49146
	max	438327	196652	49146

Utilization Estimates

	solution1	solution2	solution3
BRAM_18K	97	1258	57
DSP48E	5	92	4
FF	1309	34671	986
LUT	1527	30178	3039
URAM	0	0	0

Fig. 3.2.k: Latency and resource utilization

A combination of loop pipelining and fixed point quantization was found to produce the greatest benefit for both latency and utilization. Initially, the quantization did not affect the accuracy. However, it resulted in a drop in accuracy when the model parameters changed after retraining the software model. Hence, the quantization was removed for the final implementation to avoid unexpected drops in accuracy from changes in parameters. The resulting increase in latency was not noticeable from a human perspective.

3.2.5 Power management

The Ultra96 provides support for monitoring power rails on the board using PMbus. The get_rails() function from the Python pynq package returns a dictionary of rails that can be used for power measurement. The DataRecorder class can then be used to monitor these rails. [13]

There is a tradeoff between lowering power consumption and increasing performance of the accelerator. However, performance will be prioritized to meet the real-time requirement of the system. Furthermore, as the Ultra96 board will not be mobile for the scope of this project, minimizing power consumption would be less important than reducing latency. Nevertheless, power consumption should be capped at a point where further increasing resource usage does not confer significant improvements in performance.

To reduce CPU power usage, the neural network computations will be done on the FPGA, which offers higher performance per Watt. Processes should also be killed when not in use to reduce unnecessary power usage.

To reduce power consumption of the FPGA, the complexity of the neural network could be reduced by reducing the number of layers or neurons. This would result in fewer resources being used in the hardware implementation, resulting in lower power consumption, possibly at the cost of inference accuracy.

If the methods above are not effective, lowering of clock frequency for both the CPU and FPGA could also be considered to reduce power usage.

Final Implementation

The DataRecorder class from the Pynq library was used to record power usage.

```python
# Initializes DataRecorder object to monitor power rails using PMBus
def init_power_recorder():
    rails = pynq.get_rails()
    RAIL_NAME = 'PSINT_FP'
    return pynq.DataRecorder(rails[RAIL_NAME].power)
```

```python
# Sample the power every 0.1 seconds
recorder = init_power_recorder()
with recorder.record(0.1):
    # Model execution
```

Fig. 3.2.l: Recording power usage

Several methods for reducing power consumption were tested as follows:

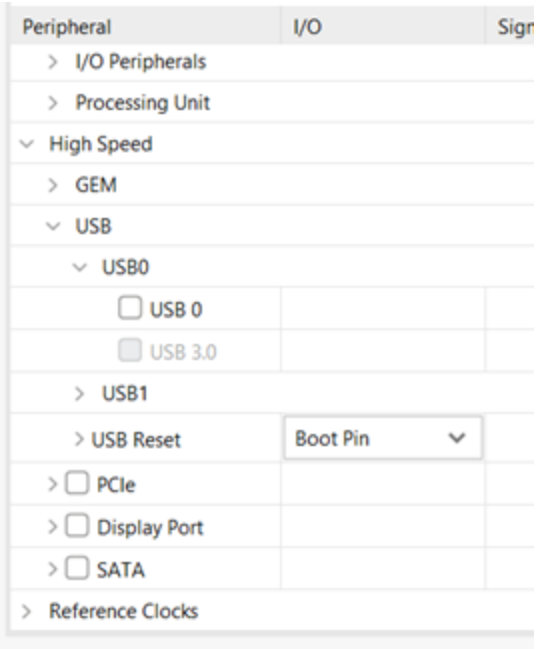


Fig. 3.2.m: Peripherals in Vivado

Disabling unused peripherals on the ultra96 such as USB and Display port did not have a noticeable effect on power consumption.

```
$ sudo cpufreq-set -u 1.1GHz
```

Fig. 3.2.m: Peripherals in Vivado

The CPU frequency can be set using the command shown above. Setting it to anything higher than 1.1 GHz did not have any noticeable effect on power consumption. At 1.1 GHz, power

consumption was significantly reduced from 0.75 to 0.65 units. However, the latency also doubled as the CPU was able to perform fewer computations per clock cycle. Hence, this optimization was not used for the final design.

```
echo 0 > /sys/devices/system/cpu/cpu3/online
echo 0 > /sys/devices/system/cpu/cpu2/online
```

Fig. 3.2.m: Peripherals in Vivado

CPU cores can be turned off using the commands shown above. Through testing, shutting down 2 cores was determined to produce the best tradeoff between latency and power consumption. Power was reduced from 0.75 to 0.71 units while having practically no effect on latency.

Section 4 Firmware & Communications Details

Section 4.1 Internal Communications

4.1.1 Introduction

The main aim of internal communications is to ensure reliable, robust and concise data transfer between individual beetles and their respective laptops. As mentioned above in section 2, there will be 1 beetle on each dancer which will be connected to one laptop. Therefore, there will be 3 laptops in total which will be collecting data from 1 beetle each.

For data transfer, it is important that the beetles are able to retrieve data from the IMU sensors and the Myoware sensors, preprocess them and send it to the laptop. While transferring data, it should handle various issues such as disconnection of beetles with the laptop, packet fragmentation and packet corruption. This section will discuss the final implementations of how the internal communications protocol was designed to handle these issues and to coordinate data transfer between beetles and laptops. At the bottom of each subsection, there will be alternative proposals discussed and the reasons for the changes made to those proposals to achieve the final design and implementation.

4.1.2 Managing Tasks on the Beetle

On the beetle, there were two main tasks handled:

1. Transmit data packets to its respective laptop
2. Receive acknowledgement packets from its respective laptop

These tasks will not be occurring concurrently as the acknowledgement packet will only be received at the beginning of each connection by the beetle during the 3-way handshaking protocol (discussed in section 4.1.6). Once connection has been established, the only task that has to be handled will be to transmit data packets to its respective laptops in the desired frequency of 20Hz (20 IMU data packets per second). It will concurrently send EMG data packets at a frequency of 0.2Hz (1 EMG data packet per 5 second). Since only one dancer will be equipped with the EMG sensor, the code specific to sending the EMG data packet (*sendemg()* function) will be commented out for dancers who do not have the sensor (code shown below). This will not affect the overall system flow as it will be considered as just another type of data packet received on the laptop's side. Frequency of IMU data packets received will not be affected.

There is no prioritization of tasks needed and therefore, no form of multithreading was done on the arduino's software. To handle the transmission of data packets in the desired frequency, a counter variable was used to send the packets periodically.

```
173         if(confirmed){
174           //sendemg();
175         }
```

Figure 4.1.a: Code snippet of EMG sensor portion commented out for non-EMG dancers

Alternative proposal:

It was assumed that three main tasks would be running concurrently at all times:

- ➔ Transmit data packets to its respective laptop
- ➔ Receive data packets from its respective laptop
- ➔ Read sensor data, preprocess and format into packets before sending to laptop

Prioritization of these tasks were needed and protothreading on arduino's software had to be done with some arduino libraries which are able to run "pseudo-background" tasks using timer interrupts and facilitates the use of multiple tasks [22].

However, we now know that acknowledgement will only be done at the beginning of the process when establishing a connection between the beetle and laptop. Moreover, there will only be simple preprocessing of data, such as scaling of floating data points to integers. Heavy processing such as using a 3rd order median filter will not be done on the beetle. We decided to do it on the machine learning side before feeding it into the model, as this is a more efficient way to ensure that the data packets are being sent without any disruption from the beetle to the laptop. This was also to ensure that we do not overload the beetle's capacity and focus on doing only one major task after establishing connection, which is to send the data packets over to the laptop.

Due to this, it was not necessary to allocate beetle's memory to perform any preprocessing of data. Therefore, our final decision was to have no form of multithreading on the arduino.

Alternative proposal:

To achieve the desired frequency of sending 20 data packets per second, we used the delay function: *delay(48)*, to halt for 48ms. Using the formula for frequency: 1/0.048s ~ 20 packets, we are able to achieve this task. However, this was disrupting the beetle's capacity to do other activities. The relative position detection algorithm relies on time where it checks if we are in the idle state for at least 2 seconds before starting to detect position change (Further explained in previous section 2.4). Having this delay of 48ms was disrupting this algorithm, therefore, a counter was used to halt the sending of packets for approximately 48ms instead of using the delay function. This decision allowed the beetle to run at maximum performance capacity at all times, allowing it to be more efficient at handling all tasks.

4.1.3 Data transfer from Beetle to Laptop

We used the periodic push approach on Arduino to transmit data from beetle to the laptop. Periodic push approach allows data to be transmitted to the laptop as soon as data is formatted into packets to be transferred. This will allow arduino to free its space for multiple tasks to run and not waste its memory. However, this posed a problem on the laptop side, as there might have been cases where too many packets were sent to the laptop, and packets got lost, fragmented or even corrupted. To ensure reliability of data packets received, we tackled this issue using different ways and algorithms mentioned in section 4.1.11.

Alternative proposal:

The periodic poll approach allows the laptop to decide when it needs the data and sends a poll packet to the arduino to request for data to be transferred. Meanwhile, data arriving from the sensors are temporarily stored in the arduino till it receives a poll packet. If polling is not done frequently by the laptop, data saved in arduino might be lost due to its small memory, furthermore, sending a poll packet everytime the laptop requires data will increase overall latency.

To avoid this issue, we used periodic push to send data packets and came up with different solutions to tackle the problems that come along with using this method.

4.1.4 Managing tasks on the laptop

Each laptop will be connected to only 1 beetle for a single dancer. Therefore, multithreading on the laptop to handle internal communications protocol was not actually necessary as there won't be concurrent connections handled by one laptop.

Alternative proposal:
In our initial design report, it was assumed that we would require 2 beetles on one dancer to deduce final dance predictions accurately. Therefore, we would have to establish 2 connections with 2 beetles concurrently on one laptop. Due to this, it was designed such that each bluno would have one thread each on the laptop, where it will establish connection through handshaking protocol and handle incoming data packets on its individual thread.

However, after testing, we realised that 1 beetle is enough to deduce dance predictions accurately as well as deduce position change well. Using just 1 beetle per laptop reduces the complexity of handling 2 different beetles on the laptop's side as well as the ultra 96 side and overall reduces the complexity of this project. Moreover, multithreading of beetles for a single dancer is not actually necessary for internal comms protocol. Nevertheless, using the code which contains multithreading will still work for a single beetle, there will just be one thread running at all times as it will only establish connection with one beetle at a time.

4.1.5 Setup and configuration for BLE interfaces

We set up BLE (Bluetooth Low Energy) on Arduino beetle using Arduino IDE with the following steps below. [10]

1. After connecting Bluno to the laptop using micro USB data cable, enter arduino IDE and choose the correct serial port in Menu->Tools->Serial Port
2. Open serial monitor and select "No line ending" and "115200 baud" in the 2 pull down menus below
3. Type "+++" and press "Send" button
4. If have successfully entered AT command mode, it will show "Enter AT Mode" on the serial monitor
5. Select "Both NL & CR" and "115200 baud" in the 2 pull down menus below.

6. To set the beetle to default peripheral settings we enter command “AT + SETTING=DEFPERIPHERAL” and press “Send” button. If “Ok” appears on serial monitor, it has been set
7. Exit AT mode by entering command “AT+EXIT”

We used setting to be “DEFPERIPHERAL” as the beetles are the peripheral devices and our laptop is the central device. After the setup above we used *Serial.read* and *Serial.write* to control bluetooth connection through serial communication. To find MAC addresses of each beetle, we can scan the devices using laptop’s in built bluetooth and find its address.

We also set up the BLE host and controller on the Ubuntu Linux system. To do this, a linux operating system is required for all 3 laptops used. This was achieved through dual booting into Ubuntu OS (Operating System) on each laptop. After setting up with the commands above, we could connect with the respective beetles using its MAC addresses with the aid of the “bluePy" library. The mac addresses of the beetles were all hard coded and not scanned every time we tried to establish connection with a specific beetle.

Alternative proposal:
We set up the BLE host and controller on the Ubuntu Linux system using the following steps below (taken from internal communications slides) using Virtualbox on each of the laptops. However, using virtualbox did not provide us with stable bluetooth connections for certain laptops, presumably due to native laptop’s OS fighting for bluetooth capabilities at the same time for these select few devices. Therefore, to ensure stable bluetooth connection at all times, and to synchronize across all group members, all 3 dancers’ laptops were dual booted in order to set up the BLE host and controller.

1. The command “hciconfig” will print information about bluetooth devices installed in the system
2. The command echo “BT_POWER_UP > /dev/wilc_bt” powers up the beetle
3. The command echo “BT_DOWNLOAD_FW>/dev/wilc_bt” downloads the firmware needed for the beetle

4. The command “BT_FW_CHIP_WAKEUP >/dev/wilc_bt” prevents the beetle from going into sleep mode
5. To attach serial UART to bluetooth stackas HCI transport interface at baud rate of 115200, run the command “hciattach /dev/ttyPS1 -t 10 any 115200 noflow nosleep”
6. To set the maximum and minimum connection interval, enter command “echo 15>conn_min_interval” and “echo 15>conn_max_interval” to have a connection interval of 15ms to maximize throughput

4.1.6 Communication protocol between beetle and laptop

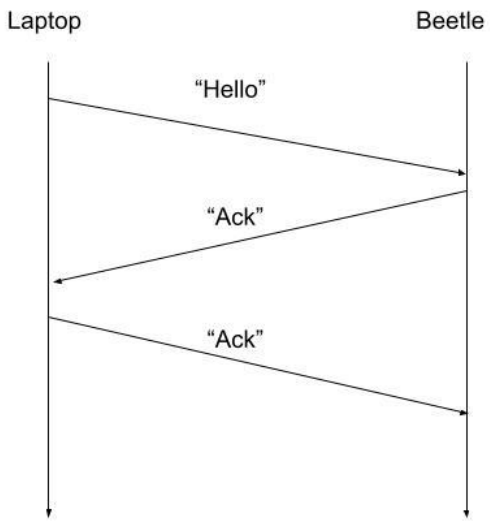


Figure 4.1.b: 3-way handshaking protocol

To establish and confirm the connection between a beetle and its laptop, we used the 3 way handshake protocol. Initiation of connection was done by the laptop, where it will first send a “hello” packet to the beetle. Upon receiving this, the beetle will send an “ack” packet to the laptop to acknowledge connection. To reaffirm this, the laptop sends an “ack” packet to the beetle to signal data transfer to begin. There will not be any request made by the laptop to send data packets as we decided to use the periodic push approach to send data packets from beetle to laptop. This is to avoid any delay in sending poll packets from laptops to beetles to request data, which may affect the overall prediction of synchronization delay.

4.1.7 Packet Types

There are 3 types of data that will be sent from the laptop to the beetle. Upon receiving these characters on the arduino side, it sends the appropriate data packet in response, indicated below Table 4.1.a.

Data Type	Character	Description
Hello	‘h’	Laptop sends beetle to initiate connection
Acknowledgment	‘a’	Acknowledges the acknowledgment packet sent by the beetle
Reset	‘r’	Before sending ‘h’ to initiate connection, it resets the beetle to ensure that calibration is done properly at the beginning

Table 4.1.a: Data types sent from laptop to beetle

There are a total 4 types of packets that were be sent from the beetle to the laptop:

Packet Type	Value encoded	Description
Acknowledgement	1	Beetle sends acknowledgement packet upon receiving ‘h’
IMU data packet	2	Beetle sends data from IMU sensors upon confirming 3-way handshake, which is after receiving ‘a’ from laptop
EMG data packet	3	Beetle sends data from EMG sensors

		upon confirming 3-way handshake, which is after receiving ‘a’ from laptop
Position data packet	4	Beetle sends position data upon confirming 3-way handshake, which is after receiving ‘a’ from the laptop. It is only sent when there is a change in direction(Left or Right) detected by the beetle.

Table 4.1.b: Data packets sent from beetle to laptop

4.1.8 Preprocessing of data

Before being assembled into a packet to be sent to the laptops, data collected from the IMU sensors have to be preprocessed. For BLE 4.0 and 4.1 the maximum ATT payload(data transfer) for one data packet is 20 bytes. For BLE 4.2 and 5.0 it is possible to use data length extensions to allow ATT payload to hold up to 244 bytes [11]. The bluno beetle at hand supports BLE 4.0. Therefore, our maximum size of the data packet should be 20 bytes. It was necessary to fit different sets of data from sensors into one packet to ensure that it is concise and clear.

After several rounds of testing of the whole system, we decided that only accelerometer and yaw, pitch and roll values were necessary for the machine learning model to make accurate dance predictions. Therefore, we did not use gyroscope values, rather, it was used in the calculation of yaw, pitch and roll values.

The accelerometer records data in separate forms of x,y and z values and yaw, pitch and roll records angular values, both in positive or negative floating type values. There are issues with these data sets if we were to package it into packets without any preprocessing. Due to it being a floating type value, to store each data point, we would require an additional byte of data.

To tackle this, these values were pre-processed to achieve a desirable format for the packet. The floating values were scaled up by a certain value for accelerometer, yaw, pitch and roll datasets to retain the precision by decimal points (scaling values for each aspect will be different and calibrated while testing). All values were scaled up by 100.

To determine byte allocation, we looked at the ranges for each of the values through rigorous testing. For accelerator values, after scaling up, ranges from $\pm$ 250 . For yaw, after scaling up, ranges from $\pm$ 200000 and for pitch and roll values, after scaling up, ranges from $\pm$ 1800[12]. After scaling them and zero offsetting, we make a deduction that the maximum number of bytes needed for x, y, z accelerator values required 2 bytes each, yaw values required 4 bytes each and pitch, roll values required 2 bytes each.

Alternative proposal:
Negative values take up an additional bit. Therefore, in order to save the full byte or two for the value to prevent overflowing, additional pre-processing of raw data values were considered. To deal with negative numbers, we ensure that all values undergo zero offsetting, for example if the maximum negative value for x, y, z of accelerometer reading is -10, then value of 10 will be added to every accelerometer value. However, after many rounds of data collection and testing, we figured out that the range of values of accelerometer and yaw, pitch, and roll values were able to fit and not overflow despite using an additional bit for signed integers. Therefore, for simplicity of preprocessing of data, we decided to use signed integers for these values.

4.1.9 Packet format

All data packets contain a byte for packet type (as mentioned in section 4.1.7). To achieve handling of packet fragmentation (discussed in section 4.1.11), all packets have a fixed size of 20 bytes. Therefore, padding is added to ensure each value meets its 20-byte value. Padding values will be all zeros. All packets will have checksum bytes allocated. The number of bytes for checksum will correspond to the highest number of bytes allocated to a certain value in the packet.

Acknowledgement(ACK) packet is shown below. Its packet type will be encoded with integer 1 :

Figure 4.1.c: ACK packet

Data packet from IMU sensors is shown below. Its packet type will be encoded with integer 2:

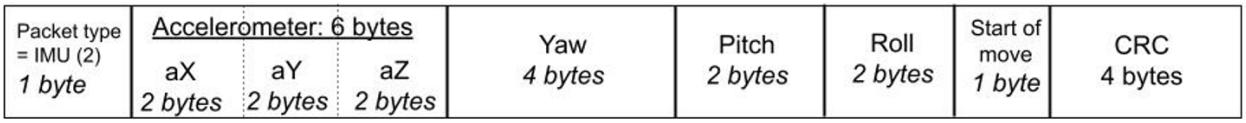


Figure 4.1.d: IMU data packet

Data packet from EMG sensors is shown below. RMS value corresponds to root means square, MAV corresponds to mean absolute value and ZCR corresponds to zero-crossing rate. Its packet type will be encoded with integer 3:

Figure 4.1.e: EMG data packet

Positional change data packet is shown below. Its packet type will be encoded with integer 4. For direction byte, a *Left* shift will be encoded with integer 1 whereas a *Right* shift will be encoded with integer 2.

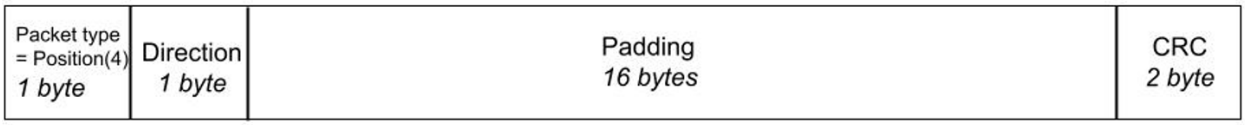


Figure 4.1.f: Relative position data packet

All the packets have a payload of 20 bytes, which will not cause overflow of data into other packets.

Alternative proposal:

An arbitrary value was discussed previously to be used for beetles to distinguish between the 2 beetles on one dancer. Packet type to take up 4 bits while arbitrary values to take up the other 4 bits. However, the mac address of the beetle can be used to distinguish on the laptop's side and also, only 1 beetle will be used per dancer. Therefore, allocating 4 additional bits for arbitrary value is not necessary and was removed from the final packet design.

4.1.10 Baud rates

We set the baud rate to 115200 bits per second in both the beetle side and the laptop side. According to the machine learning model's requirements, it required 20 datasets per second, which means 20 packets of data had to be sent to the laptop per second.

Total number of packets to send per second = 20 packets
Total number of bytes per packet = 20 bytes
Total number of bits per packet = 20 bytes * 8 bits
Total number of bits to send laptop per second = 20 packets * 20 bytes * 8 bits
= **3200 bits**

We can see that the baud rate of 115200 was more than enough to send 3200 bits per second (corresponds to 20 packets). The EMG packet will also be sent concurrently every 5 seconds, which will add another 20 bits to 3200 bits to be sent every second. This is still within the capabilities of baud rate.

4.1.11 Ensuring Reliability

To check for errors in the packets being sent, we decided to use Cyclic Redundancy Check (CRC) to check the data packet's accuracy. When a packet is being packaged with the relevant data in the beetle, CRC value is calculated and appended to the packet which is then sent over to the laptop. The laptop then checks the values of the packets received with the CRC value, if it matched it is considered a good packet. If the values do not match, the packet is considered corrupted or fragmented.

If a particular packet is lost, corrupted or fragmented, that data packet will not be retransmitted by the beetle to the laptop. Retransmission of the data packet will involve the laptop sending a request packet to the beetle to retransmit the data, and the beetle will then transfer it back. This process will incur more delay in terms of data transmission, and the main aim of this project is to predict dance moves as accurately and quickly as possible by reducing overall latency. This process will also cause a strain on beetle's small memory space, as it will be necessary for it to store certain data packets for a brief period of time, in case it needs to retransmit the packets back to the laptops. It can be noted that the machine learning model was still able to predict the dance moves even if some were corrupted data packets as we would have a considerable number of good packets that is sufficient to make a prediction. The periodic push approach ensures that each packet is assembled fully and only then will it be sent out to the laptop. Therefore, it is safe to say that if the throughput is lower than expected, we can still use them to make an accurate prediction with other full datasets.

If the data packet is received by the laptop with the correct length of the packet (20 bytes), yet CRC check fails, it is considered to be a corrupted data packet and it is discarded. In the event that the data packet is not received by the laptop with the correct length of bytes (20 bytes), it will be considered a fragmented packet. Instead of discarding the fragmented packet, we try to store it in a buffer and append it to oncoming packet and check if the crc check passes. There are 3 scenarios which will be dealt with on the laptop's side:

1. Buffer is empty and the length of the data packet is less than expected (< 20 bytes). If this is the case, store it in the buffer and wait for oncoming data to append, following which check against crc.
2. Buffer is not empty, the length of the data packet is lesser than expected (< 20 bytes). If this is the case, if length of data in buffer + length of data packet = required length, check against crc. If it passes, consider the packet as valid, else discard it. If length of data in buffer + length of data packet < required length, keep storing in buffer till next packet arrives, repeat step 2.

3. Length of data packet is more than expected(> 20 bytes). If this is the case, it is considered a corrupted packet, discard it.

If in the middle of a dance, the beetle and the laptop get disconnected, this exception was handled using the *BTLEDisconnectError()* function[23]. The laptop also checks if there are no notifications(data) being received from the beetle for more than 15 seconds, if so, it assumes there are too many error packets or it has disconnected. Therefore, it will try to reset and reconnect with the beetle by initiating handshaking protocol. After doing so, it handles incoming data packets as per usual.

To ensure that the reconnections are fast in the event that there are disconnections, we added a delay of two seconds on the laptop's script after sending the 'r'(reset) packet (further explained in section 4.1.7) to the beetle. This solution significantly improved the time taken to finish handshaking protocol and establish a reliable connection. This is due to the fact that after the beetle receives the 'r' packet, it resets the beetle and it calibrates by going into the *void setup()* function on the arduino which will take a few seconds. By adding this delay, we were able to give sufficient time for the beetle's calibration and get ready in time to receive the subsequent packets from the laptop to establish connection.

Section 4.2 External Communications

4.2.1 Overview

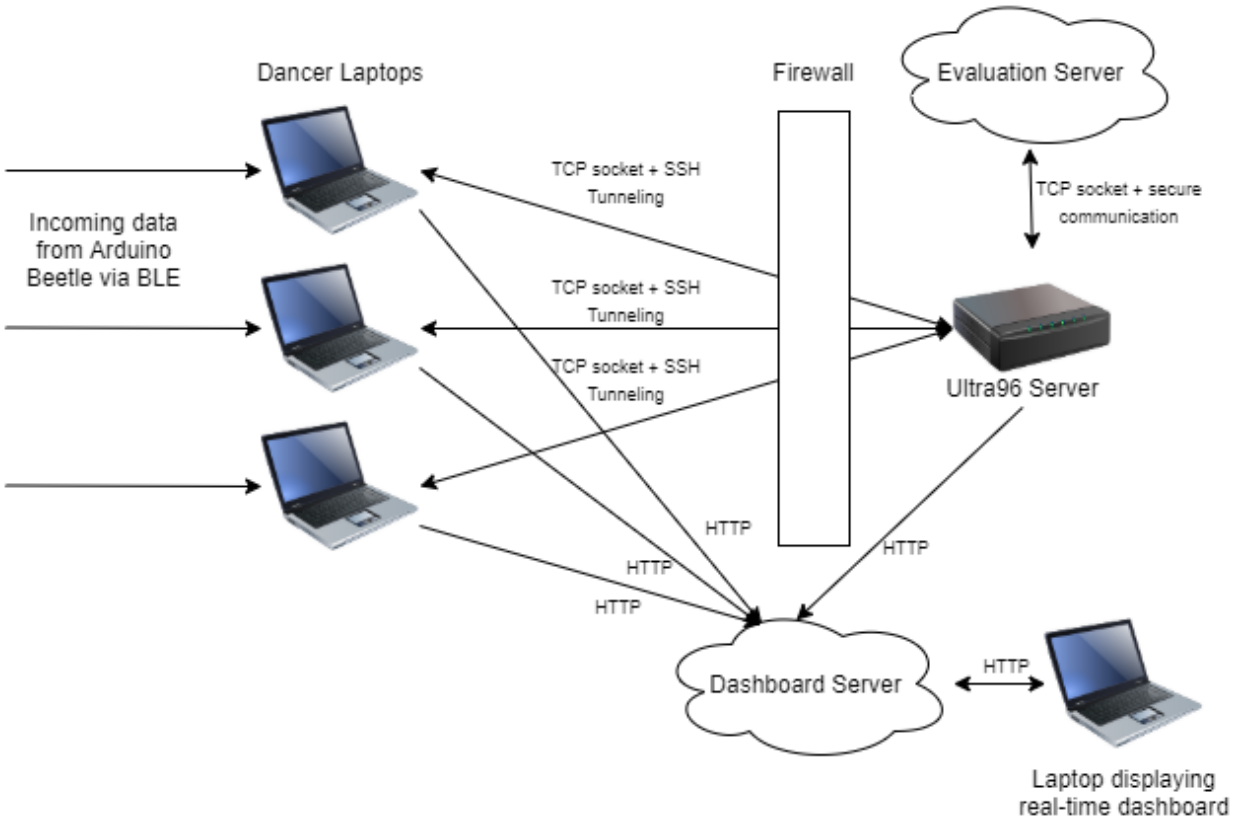


Fig. 4.2.a External Comms Architecture Diagram

The flow of data throughout the system is depicted in the architecture diagram above. Each Dancer Laptop will receive its sensor data from the Bluno via the BLE. Each Dancer Laptop will then establish a connection with the Ultra96 with the use of a TCP socket and forward the preprocessed data to the Ultra96. After further processing, the Ultra96 will send relevant data to the Evaluation Server via a secure TCP socket. The Dancer Laptops and Ultra96 will both send relevant data to the Dashboard Server via HTTP, which will be displayed on the Laptop running the real-time frontend Dashboard.

4.2.2 Dancer Laptops and Ultra96

4.2.2.1 Process Overview

1. Server script of Ultra96 will first be started to accept connections from Laptops
2. Each dancer will run the Laptop script indicating their dancer ID to establish TCP connection with Ultra96
3. Upon successful connection of each Laptop, a clock synchronization process will be started with the Ultra96 as the main reference time source
4. The Laptop will send base64-encoded data to the Ultra96 whenever received from the Bluno
5. Upon receiving the data, the Ultra96 will pass the data into the FPGA hardware accelerator to get the dance move prediction
6. After each submission to the Evaluation Server, the Ultra96 will send a cue back to the Laptops to re-synchronize the clocks
7. This cycle repeats until the Ultra96 receives a "logout" move, where it will send a message to the Laptops to terminate the connection

4.2.2.2 Handling Data Transfer

For our final implementation, we used the Python `threading` library to implement multithreading for concurrent handling of multiple tasks. This was to minimize bottlenecks between different tasks to reduce latency. Communication between threads was done using events from the Python `threading` library and queues from the Python `queue` library. We used the `queue` library as it is implemented to be thread safe using locks and ensures safe exchange of data between threads. In addition, the Event object in the threading library allows blocking of threads to wait for certain events, removing the need for busy waiting and improving latency.

As we are sending packets of variable length at a high data rate from the Laptop, we faced the issue of receiving incomplete packets on the Ultra96 and needed a way to ensure each packet is received in full before it is processed. To solve this, after encoding the data in base64 format, we used then "|" character (i.e. ASCII value 124) as a delimiter for packets sent. This character was

chosen as it is not part of the character list used for base64 encoding and could safely identify separate packets. On the Ultra96, we will ensure packets are received in full by checking for the delimiter.

Laptops

There are 2 main threads on the Laptop - the `Bluno` thread and the `Client` thread.

The `Bluno` thread is created to handle the constant receiving of data packets from the Blunos as implemented by Internal Communications. Upon receiving and confirming the validity of received packets, it will pass the data packets into a global queue `dataQueue` to be handled accordingly.

The `Client` thread is created to handle the processing and sending of packets to various destinations. After establishing a TCP socket connection with the Ultra96, this thread will retrieve data from the global `dataQueue` whenever available. Depending on the type of data packet received, the data will be processed differently and sent to either the Ultra96 or Dashboard server.

Lastly, after initialization, the `Client` thread will spawn 1 more thread to handle the receiving messages from Ultra96 as cues for clock synchronization and logging off. As the Laptop is primarily sending data to the Ultra96, the receiving is asynchronous and minimal. Hence, as a separate thread, it will be blocked while waiting for incoming messages, reducing contention of resources until a message is received.

Ultra96

For each laptop connection, there will be 2 threads created - a `dancer` thread and a `ml_handler` thread. Each `dancer` connection will also have a dedicated `dataQueue` to handle data exchange from the `dancer` thread to `ml_handler` thread.

Each `dancer` thread handles constant receiving of data packets from the laptops. Each connection uses a different port to reduce bandwidth contention. Each `dancer` thread will add each received data packet into its `dataQueue`.

Each corresponding `ml_handler` thread handles the prediction of dance move from the sensor data. They will retrieve data from their `dataQueue`, and pass it into the hardware-accelerated ML predictor. Contention of the hardware accelerator is handled using locks from the `threading` library. The predictions will then be saved to the global state for submission. From this point, the corresponding `dancer` thread will start to clear its `dataQueue` until 5 seconds after the prediction is submitted to the Evaluation Server. This is to prevent sensor packets from previous dance moves from leaking to the prediction of the next dance move. The `ml_handler` will then wait for the `submittedToEval` flag to be set to signify the submission of the previous dance move, before repeating the process for the next dance move.

4.2.2.3 SSH Tunneling

The Ultra96 is hosted within the NUS Wifi, which is behind a firewall. Hence, in order for the laptop to connect to the Ultra96, there must be tunneling done to bypass the firewall. To accomplish this, we will open an SSH connection from the local socket port to a port on Sunfire, and subsequently configure Sunfire to forward that port to the Ultra96 [15].

Final Implementation

Instead of using a separate SSH command on the terminal, we discovered the Python sshtunnel library written for SSH tunneling. This allowed us to write the tunneling logic in the Python script itself.

As the Ultra96 could only be accessed via its port 22 within the NUS network, we needed 2 SSH tunnels to bypass the NUS Firewall to establish a socket connection. The first SSH tunnel is established from a Laptop port to the Ultra96 port 22 via Sunfire as the intermediate node. The second SSH tunnel established a connection from another laptop port to the Ultra96 socket server port via the Ultra96 port 22 using the first SSH tunnel. With this, the Laptop is able to establish a socket connection with the Ultra96 socket server.

```python
def setup_connection(self):
    username = input('Enter Sunfire username: ')
    password = getpass.getpass('Enter Sunfire password: ')
    tunnel1 = sshtunnel.SSHTunnelForwarder(
        ssh_address_or_host=('sunfire.comp.nus.edu.sg', 22),
        remote_bind_address=('137.132.86.235', 22),
        ssh_username=username,
        ssh_password=password
    )
    tunnel1.start()
    print(f'Connection to tunnel1 OK: {tunnel1.tunnel_bindings}')

    tunnel2 = sshtunnel.SSHTunnelForwarder(
        ssh_address_or_host=('127.0.0.1', tunnel1.local_bind_port),
        remote_bind_address=('127.0.0.1', REMOTE_PORT),
        local_bind_address=('127.0.0.1', LOCAL_PORTS[self.dancerId-1]),
        ssh_username='xilinx',
        ssh_password='xilinx'
    )
    tunnel2.start()
    print(f'Connection to tunnel2 OK: {tunnel2.tunnel_bindings}')

    self.socket.connect(('localhost', LOCAL_PORTS[self.dancerId-1]))
    print('Successfully connected to Ultra96!')
    self.clientConnectedFlag.set()
    self.send_message(self.dancerId)
```

Fig. 4.2.b Python code implementation of SSH tunneling

4.2.2.4 Types of Data packets

There are 3 types of data packets that need to be transmitted to other parts of the system - IMU sensor data, EMG sensor data and Positional change data.

IMU Sensor Data

As this data is used by the ML model for dance move prediction, the model needed a window of 42 sensor reading samples. For our final implementation, we decided to preprocess the raw sensor readings on each Laptop to reduce the workload done on the Ultra96. The resulting preprocessed data will be a 1D array of 210 values. If that packet is the first packet of a dance move, the timestamp for the start of the dance move will be appended to the front of the

preprocessed array. This array will then be converted into a comma-separated string and sent to the Ultra96. Upon receiving, the Ultra96 will save the timestamp if any, and pass the preprocessed array into the ML model.

In addition, the Laptop will also send raw sensor readings to the Dashboard server for real-time visualization of the raw sensor values.

EMG Sensor Data

For EMG data packets received, the Laptop will send it straight to the Dashboard server for real-time update on the dancer's activity level.

Positional Change Data

Once a positional change packet is received and has its validity verified, the Laptop will send it straight to the Ultra96 for positional change update. The data for positional change will be either 'L' for left or 'R' for right.

4.2.3 Ultra96 and Evaluation Server

4.2.3.1 Process Overview

1. With the Evaluation Server waiting for connections, the Ultra96 script will establish a TCP socket connection with the Evaluation Server
2. The secret key used for AES encryption will be entered to the Evaluation Server, and it will begin waiting for submissions from the Ultra96
3. When sufficient dance move predictions are obtained, the Ultra96 will proceed to consolidate the program's state to obtain 3 values:
    a. Predicted dance move
    b. Relative positions of dancers
    c. Synchronization Delay
4. After sending the data in a suitable format, the Evaluation Server will return the ground truth of the relative positions

5. This cycle will repeat until the Ultra96 sends “logout” as the predicted dance move, in which case the Evaluation Server will terminate the socket connection

4.2.3.2 Handling submissions to Evaluation Server

In addition to the threads created on the Ultra96 as mentioned in Section 4.2.2.2, there is also an `EvalClient` thread that handles submissions to the Evaluation Server.

After establishing a connection with the Evaluation Server, the `EvalClient` thread will be blocked as it waits for the `readyForEval` flag to be set. Once all the `ml_handler` threads for each dancer have obtained the required number of predictions, the `readyForEval` event flag will be set. The `EvalClient` thread will then be unblocked and proceed to obtain the results of the dance move prediction, relative position shifts and timestamps for the start of each dancer’s dance move. Once the final submission values have been calculated, it will be sent to the Evaluation Server.

Upon submission, the Evaluation Server will reply with the ground truth relative dancer positions. The global state will then be reset to reflect the updated current dancer positions, and clear the dance move predictions and timestamps from the previous move. The `EvalClient` will then set the `submittedToEval` flag to notify the other threads to begin processing for the next dance move. Lastly, the `EvalClient` will go back to wait for the `readyToEval` flag to be set as the cycle repeats until the “logout” move is sent to the Evaluation server which will terminate the connection.

4.2.3.3 Message Format

Message packets are sent to the Evaluation Server as a concatenated string in the following format:

#position|action|syncdelay

E.g. *#2 1 3|dab|1.87*

Position refers to the relative dancer positions refer to the position of each dancer from left to right. For example, if Dancer 2 is standing on the most left, Dancer 1 is standing in the middle,

and Dancer 3 is standing on the most right, the relative dancer positions to be sent to the Evaluation server is “2 1 3”.
Action refers to the dance move that is detected from the sensor values through the machine learning model.
Syncdelay refers to the difference in time between the start of the first dancer’s movement and the start of the last dancer’s movement. It will be calculated in milliseconds.

4.2.3.4 Secure communication

To ensure secure communications, each data packet will be encrypted using the Advanced Encryption Standard (AES). This is to prevent our data from being sniffed or modified by a malicious third party. In addition, AES is a symmetric encryption scheme, which is fast and efficient for large amounts of data.

The encryption mode used is the Cipher Block Chaining (CBC) mode with a block size of 16 bytes. This mode of AES encryption is able to hide patterns in the plaintext by using a random Initialization Vector (IV). This allows the same plaintext to always be encrypted into a different ciphertext. This way, attackers are unable to infer the secret key from the ciphertext. The plaintext will first need to be padded to fit the block size. After encryption, the IV is appended to the start of the encrypted message for the server to decrypt. The encrypted message is then encoded to the base64 format to ensure the data remains intact without modification during transport. The base64-encoded encrypted message will then be sent to the Evaluation Server. The library used for the AES encryption is the `PyCryptodome` package in Python. We will also be using the `base64` module in Python for the base64 encoding.

```python
def encrypt_message(self, message):
    cipher = AES.new(self.secret_key, AES.MODE_CBC)

    # pad message
    message += ' ' * (AES.block_size - (len(message) % AES.block_size))

    encrypted_msg = cipher.encrypt(message.encode())

    encoded_msg = base64.b64encode(cipher.iv + encrypted_msg)

    return encoded_msg
```

Fig. 4.2.c Python function to pad, encrypt and encode message

4.2.4 Dashboard Server

Data that is needed as real-time feedback to the dancers is sent to the Dashboard server hosted on Mongo Realm via HTTP Post requests, either from the Laptops or the Ultra96. This was a simple interface that could be easily integrated into the system where data could be easily sent from different parts of our system at any point in time.

Data sent from the Laptops include:

- IMU Sensor data for visualization of sensor readings
- EMG Sensor data for monitoring of dancer activity level and fatigue level

Data sent from the Ultra96 include:

- Dance move prediction for each dancer so they
- Updated relative positions of each dancer as they shift positions
- Sync delay duration for dancers to know their level of synchronization
- Additional alerts and cues for dancers to take note such as:
    - “CONNECTED” or “DISCONNECTED” alerts to notify dancer of the status of their wearables
    - Cues for when dancers can start dancing
    - Feedback on the final submission that was submitted to the Evaluation Server

4.2.5 Clock Synchronization Protocol

Initial Implementation

The Ultra96 will be the main reference time source for our system as the NTP Server. The Laptops will be the clients and synchronize their internal clocks with the Ultra96's clock via NTP over the internet. On the first level, the laptops will be the clients and synchronize their internal clocks with the Ultra96's clock via NTP over the internet. On the next layer, the Arduino Beetle will synchronize its clock with the laptop.

Final Implementation

We decided to limit the clock synchronization to the Ultra96 and Laptops and remove the 2nd layer of synchronization between the Bluno and Laptop. This is because the latency of transmission between the Bluno and Laptop is negligible compared to the clock offset between the Laptop and Ultra96. In addition, the internal clock of the Bluno is relatively unreliable as it does not use a real-time clock, which might cause unnecessary noise in the timestamp.

In addition, each time the clock synchronization protocol is run, each Laptop will perform the synchronization process 10 times and take the average offset from the 10 resulting offsets. This is to reduce noise and inconsistencies in the network transmission.

The synchronization of the clocks in the Laptops with the Ultra96's internal clock via NTP, implemented as follows:

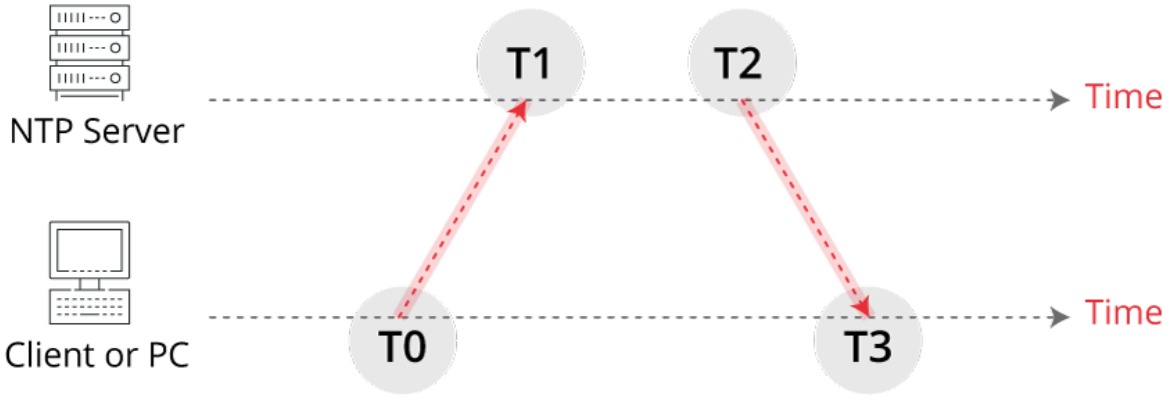


Fig.4.2.d: Depiction of NTP initialization

1. Laptop sends packet to Ultra96 and records current time when sending packet, $T0$
2. Ultra96 receives packet from laptop and records current time when receiving packet, $T1$
3. Ultra96 records current time $T2$ and sends packet containing $T1$ and $T2$ to laptop
4. Laptop receives packet containing $T1$ and $T2$, and records current time when receiving packet, $T3$
5. Laptop calculates round-trip-time between laptop and Ultra96 $RTT_{laptop-Ultra96'}$ which is $(T1 - T0) - (T2 - T3)$
6. Laptop calculates clock offset, which is $T0 - T1 - (RTT_{laptop-Ultra96}/2)$
7. Clock offset is the difference between laptop's clock and Ultra96's clock

Upon receiving a data packet indicating the start of a dance move, the Laptop will get the current timestamp and subtract it with the clock offset. This will give us the start time T_{start} of the dance move in the Ultra96's clock time. This start time will then be sent to the Ultra96 where the Ultra96 will calculate the difference between the dancers with the earliest T_{start} and the latest T_{start} to obtain the sync delay.

Section 5 Software Details

Section 5.1 Software Machine Learning

This section covers the final implementation of a machine learning model, capable of classifying different dance moves based on real-time sensor readings from accelerometer and gyroscope. Relative positioning of dancers was handled at the sensor level and has been elaborated in section 2.4. The wearable sensor-based approach for classification of dance moves is a classic application of Human Activity Recognition (*HAR*). The Activity Recognition Process (*ARP*) provides a structured workflow to generate the input vector to be fed into the learning model, which then outputs a prediction on the dance move executed.

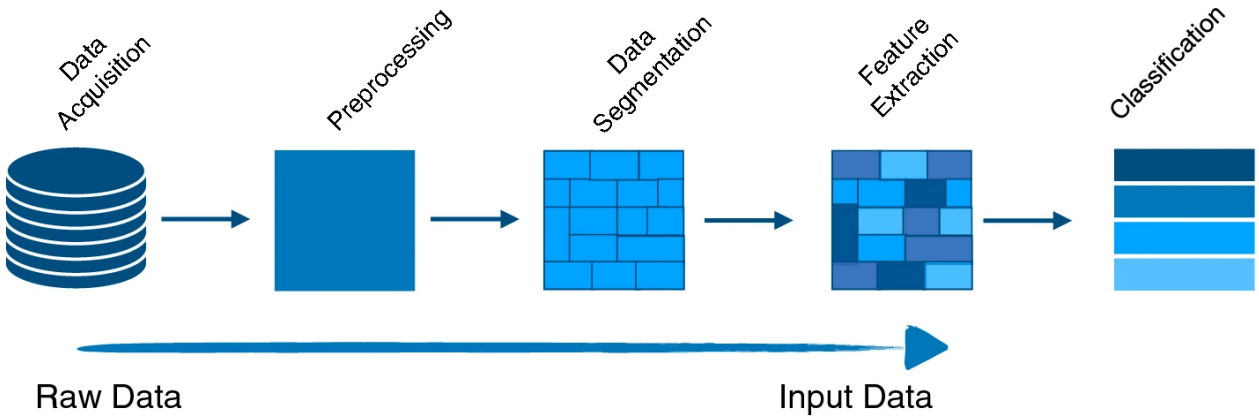


Fig. 5.1.1.a: The steps involved in Activity Recognition Process [16]

As depicted in the figure above, ARP consists of five steps - acquisition, pre-processing, segmentation, feature extraction, and classification.

5.1.1 Data Acquisition

Raw sensor data was acquired from the accelerometer and gyroscope, housed in the MPU-6050, a low cost and low energy Inertial Measurement Unit (*IMU*). The accelerometer captures 3-axial (*x,y,z*) linear acceleration while the gyroscope measures 3-axial (*roll, pitch and yaw*) angular velocity. Typical characteristic frequencies of most human activities such as hand raising,

walking and jumping are less than 10 Hz [17]. Therefore, using the Nyquist Sampling Theorem, data was sampled at 20Hz.

Each subject was made to execute a particular dance move for at least 10 trials. Each trial involved collecting data for 60 seconds. At 20Hz sampling frequency, this resulted in 1200 vectors of axial values, for each trial of a dance move, by a subject. Each data point consists of raw sensor readings and can be represented as a vector of the form : [acc_x, acc_y, acc_z, roll, pitch, yaw]

Subject-independent Cross Validation (CV) was implemented to remove the intra subject dependencies present in subject-dependent CV [18]. Hence, five members of the team generated data for the training set, with which the model was trained and validated. One member provided data for the test set. This was a crucial decision which helped to identify if the model can generalise and classify unseen data well.

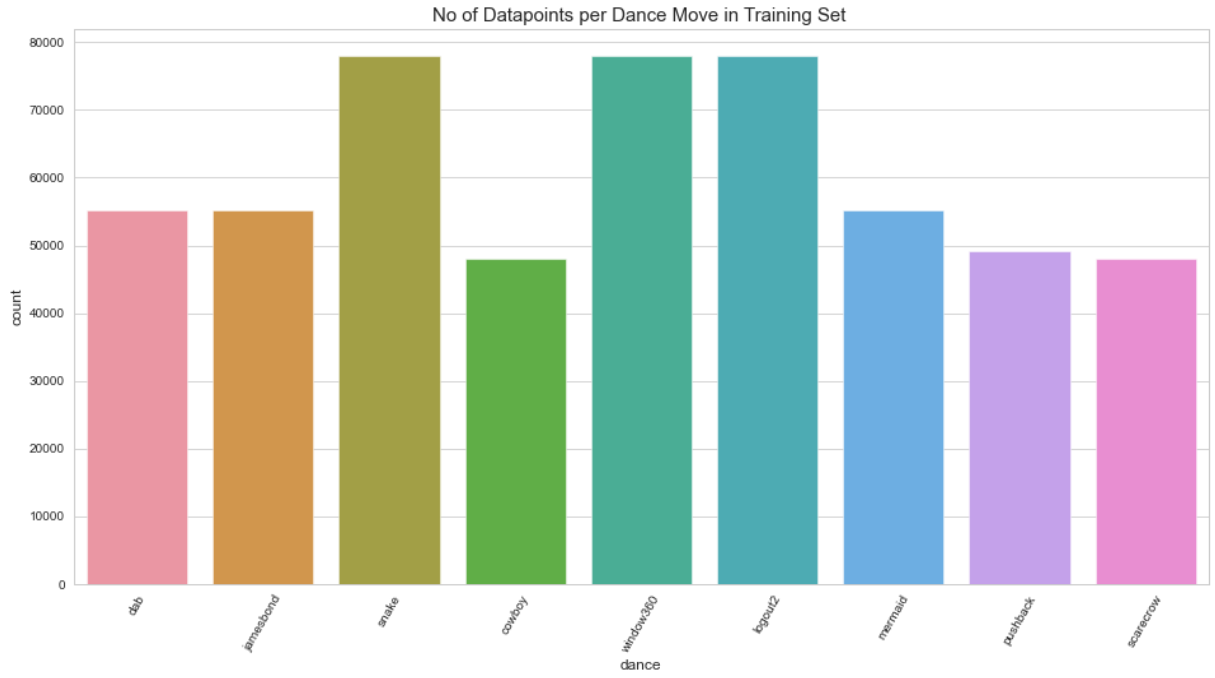


Fig. 5.1.1.a: Count of Data Points for each Dance Move in the Training Set

The bar chart in the figure above shows the count plot indicating the number of data points collected for each dance move in the training set. More data was collected for dance moves such as snake, window360 and logout which were misclassified the most. Further, subjects were made to execute dance moves with different speed and perfection to account for nervousness and fatigue during real-time evaluation. This helped to improve classification results significantly.

5.1.2 Pre-processing

Sensor data is prone to artifacts and noise as a result of unwanted movement of body segments, electronic fluctuation, etc. Hence, signal data was denoised using complementary filters, which were implemented on the Bluno beetle, using the standard Arduino IDE right at the data acquisition stage. The received sensor values were further processed using a 3rd order median filter from Scipy and normalised to achieve axial values that ranged between -1 and 1. The aforementioned pre-processing functions are detailed in the figure below.

```
def median_filter(arr):
    values = np.asarray(arr)
    med_filtered = medfilt(values, kernel_size=3)
    return np.asarray(med_filtered)

def normalise(arr):
    global MAX_VALUE
    final_vals = np.round(arr / MAX_VALUE, 16)
    return np.asarray(final_vals, dtype=np.float64)
```

Fig. 5.1.2.a: Pre-processing functions

Even after pre-processing, it was observed on inspection that yaw values tend to variate largely throughout any dance move without a pattern. This resulted in significant confusion between various dance moves. Upon dropping yaw values, classification results improved sharply and even without feature extraction, the system was able to identify different dance moves with a swift decrease in the number of confusion between the various dance moves. Thus the processed vector under consideration for each data point would be of the form : [acc_x, acc_y, acc_z, roll, pitch]

5.1.3 Segmentation

A fixed-width sliding window between 1s and 5s was intended to be explored but since the average time taken to complete a dance move once takes about 2.3 seconds, the window for experimentation was further narrowed to between 2s and 3s. The percentage overlap between any two segments was also investigated. The figure below depicts segmentation using a 2.1s sliding window with 50% overlap.

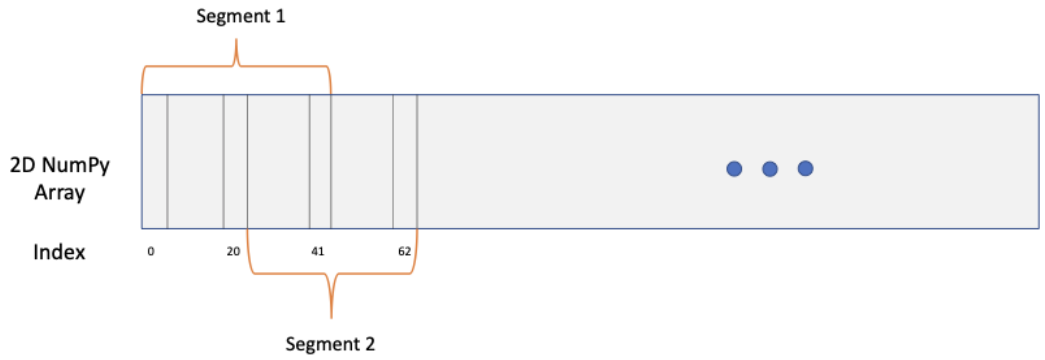


Fig. 5.1.3.a: 2.1s sliding window with 50% overlap

After much experimentation of window size and percentage overlap, best classification results were obtained with a 2.1s window that had 50% overlap between any two segments. From the training and test data sets, segmented vectors and their corresponding target values were generated using the algorithm detailed in the figure below.

```python
def segmentation(df):
    global SEGMENT_SIZE, OVERLAP, SENSOR_COLS, FREQ, WINDOW, TARGET_COL
    ncols = int(FREQ * WINDOW) * len(SENSOR_COLS)
    segments = []
    labels = []
    # In each iteration, the row jumps by the overlap size
    # grab all rows of feature column values corresponding to length of segment
    # grab corresponding mode of targetCol
    for row in range(0, len(df) - SEGMENT_SIZE, OVERLAP):
        window = []
        for col in SENSOR_COLS:
            vals = np.asarray(df[row:row+SEGMENT_SIZE][col]) # each col's raw sensor values
            filtered_vals = median_filter(vals) # filtered
            normalised_vals = normalise(filtered_vals) # normalised
            window.append(normalised_vals) # each window extracted
        segments.append(window) # 3d list of windows
        label = stats.mode(df[row:row+SEGMENT_SIZE][TARGET_COL])[0][0]
        labels.append(label)

    segments = np.asarray(segments).reshape(-1,ncols) # reshaped
    labels = np.asarray(labels)
    return segments, labels
```

Fig. 5.1.3.b: Segmentation algorithm

Although the core algorithm to generate segments remains the same, the process to handle live data during real-time evaluation differs. A high level overview of the process is presented in the figure below.

Fig. 5.1.3.c: Live data handling procedure

A buffer is used to accumulate 2.1s * 20Hz = 42 vectors of axial data from accelerometer and gyroscope. This corresponds to the data of one window. Once the buffer is full, all its constituent 42 vectors are appended to the global array, which holds data for segmentation. The buffer is emptied and continues to accumulate data until it is full. Each time the global array is appended with data, it returns the input vector to be fed into the model. The model then returns *n* predictions for *n* segments.

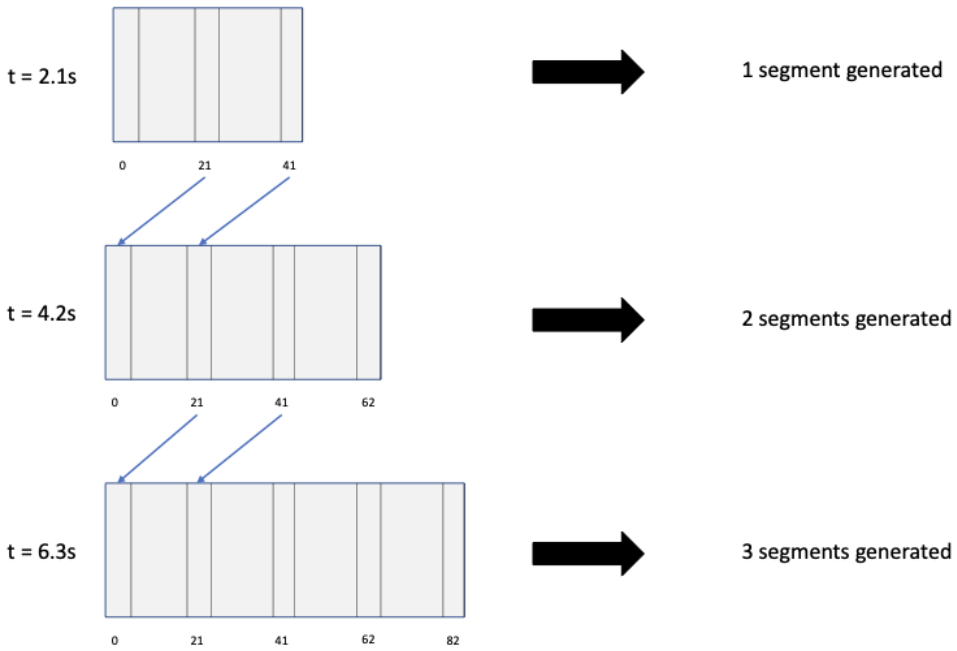


Fig. 5.1.3.d: Global Array Segmentation

As shown in the figure above, instead of maintaining a global variable corresponding to the next index position from which segmentation has to be carried out, the first 21 arrays which form the other 50% of the earliest window not used for overlap, are popped off upon segmentation. At the end of each dance move, the dancer resumes the state of idle, during which the global array is emptied.

5.1.4 Feature Extraction

From a mathematical standpoint, feature extraction can be thought of as a mapping function that leads to a new set of features where there is better data diversification [19]. The following procedures were carried out to extract features.

- L^2 norm (*Euclidean norm*) was used to obtain the magnitude of acceleration signals.
- Frequency-domain features were derived via a Fast Fourier Transform (*FFT*) on the time domain signals.
- 3 time-domain features were extracted, namely: mean, variance and median.

- 3 frequency-domain features were extracted, namely: energy spectrum, entropy spectrum and sum of the wavelet coefficients

However, the new features did not improve classification results and instead introduced new confusion between dance moves. Hence for the final implementation, extracted features were removed from the input vector fed into the model.

5.1.5 Multi-Layer Perceptron [MLP] Model

A fully connected MLP model, where each neuron shares its output with each adjacent layer's neuron was implemented using Pytorch [16]. With reference to section 5.1.3, the input vector to be fed into the model consists of one segment of data. As mentioned in section 5.1.2, only the processed vector v = [acc_x, acc_y, acc_z, roll, pitch] was considered for any data point in a window. Each 2.1s window consists of 42 vectors of 5 axial values from accelerometer and gyroscope. Hence, the input layer of the model contains 210 neurons. With 8 dance moves and 1 logout move, the number of output layer neurons was set to 9.

```
class Model(nn.Module):
    def __init__(self, in_features, h1, h2, out_features):
        super().__init__()
        self.fc1 = nn.Linear(in_features,h1)    # input layer
        self.fc2 = nn.Linear(h1, h2)            # hidden layer
        self.out = nn.Linear(h2, out_features)  # output layer

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.out(x)
        return x
```

Fig. 5.1.5.a: Multi-Layer Perceptron [MLP] class

As shown in the figure above, two hidden layers *h1* and *h2* with 150 and 60 neurons respectively were found to yield best classification results. Rectified Linear Unit (*ReLU*) was used as the

activation function to set boundaries to the output from the hidden layers. The *argmax* of the outputs from the output layer was taken since there was a need for a single predicted output rather than a probability. The resulting weights of the model were saved and passed for hardware acceleration on the Ultra96 as elaborated in section 3.2.2

5.1.6 Training the model

Stratified K-fold Cross Validation (CV) which employs stratified sampling was implemented using scikit-learn. With reference to Fig. 5.1.1.a in section 5.1.1, it is clear that there is some imbalance in the class distribution. Stratified K-fold CV preserves the class distribution in the dataset when making training and test splits. Classification results were best when $K=5$, that is a 5-fold CV. The *EPOCHS* were set to 100 and *Adam* was employed as an optimizer with the loss criterion set to *Cross Entropy Loss*. The *LEARNING_RATE* was fixed at 0.01. The detailed algorithm used for splitting the training dataset into folds along with training and validation of the model is shown in the figure below.

```
skfcv = StratifiedKFold(n_splits=K, shuffle=True, random_state=1)
training_loss = {}
val_acc = []

# per fold
for fold, (train_index, test_index) in enumerate(skfcv.split(X_train, y_train)):
    x_train_fold, x_test_fold = X_train[train_index], X_train[test_index]
    y_train_fold, y_test_fold = y_train[train_index], y_train[test_index]
    train_combined = TensorDataset(x_train_fold, y_train_fold)
    test_combined = TensorDataset(x_test_fold, y_test_fold)
    trainloader = DataLoader(train_combined, batch_size=TRAIN_BATCH_SIZE, shuffle=True)
    testloader = DataLoader(test_combined, batch_size=TEST_BATCH_SIZE, shuffle=False)
    # per epoch
    for i in range(EPOCHS):
        i+=1
        # per batch
        losses = []
        val_correct_preds = 0
        count = 0
        for batch_idx, (data, target) in enumerate(trainloader):
            mlp.train()

            # training
            y_pred = mlp.forward(data)
            loss = criterion(y_pred, target)
            losses.append(loss)

            # backtracking
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # validating
        with torch.no_grad():
            mlp.eval()
            for val_batch_idx, (val_data, val_target) in enumerate(testloader):
                y_out = mlp.forward(val_data)
                for row in range(y_out.shape[0]):
                    if y_out[row].argmax() == val_target[row]:
                        val_correct_preds += 1
                    count += 1

    # per fold
    with torch.no_grad():
        training_loss[fold] = np.array(losses).mean()
        print("-----------------------")
        print(f"fold: {fold} , training_loss: {training_loss[fold]}")
        print(f"fold: {fold}, {val_correct_preds} out of {count} = {100*val_correct_preds/count:.2f}
% correct")
        print("-----------------------")
        val_acc.append(100*val_correct_preds/count)
        count = 0

print()
print("Done Training")
print("Max Validation Accuracy: ",  np.array(val_acc).max())
```

Fig. 5.1.6.a: Pytorch algorithm for model training

5.1.7 Testing and Evaluating the model

The model was tested on unseen data from one team member inspired from Leave-one-out cross validation as elaborated earlier in section 5.1.1. The final implementation of the system capped at an accuracy of **96%** and performed well during real-time evaluation. It was robust to handling cases of fatigue and nervousness.

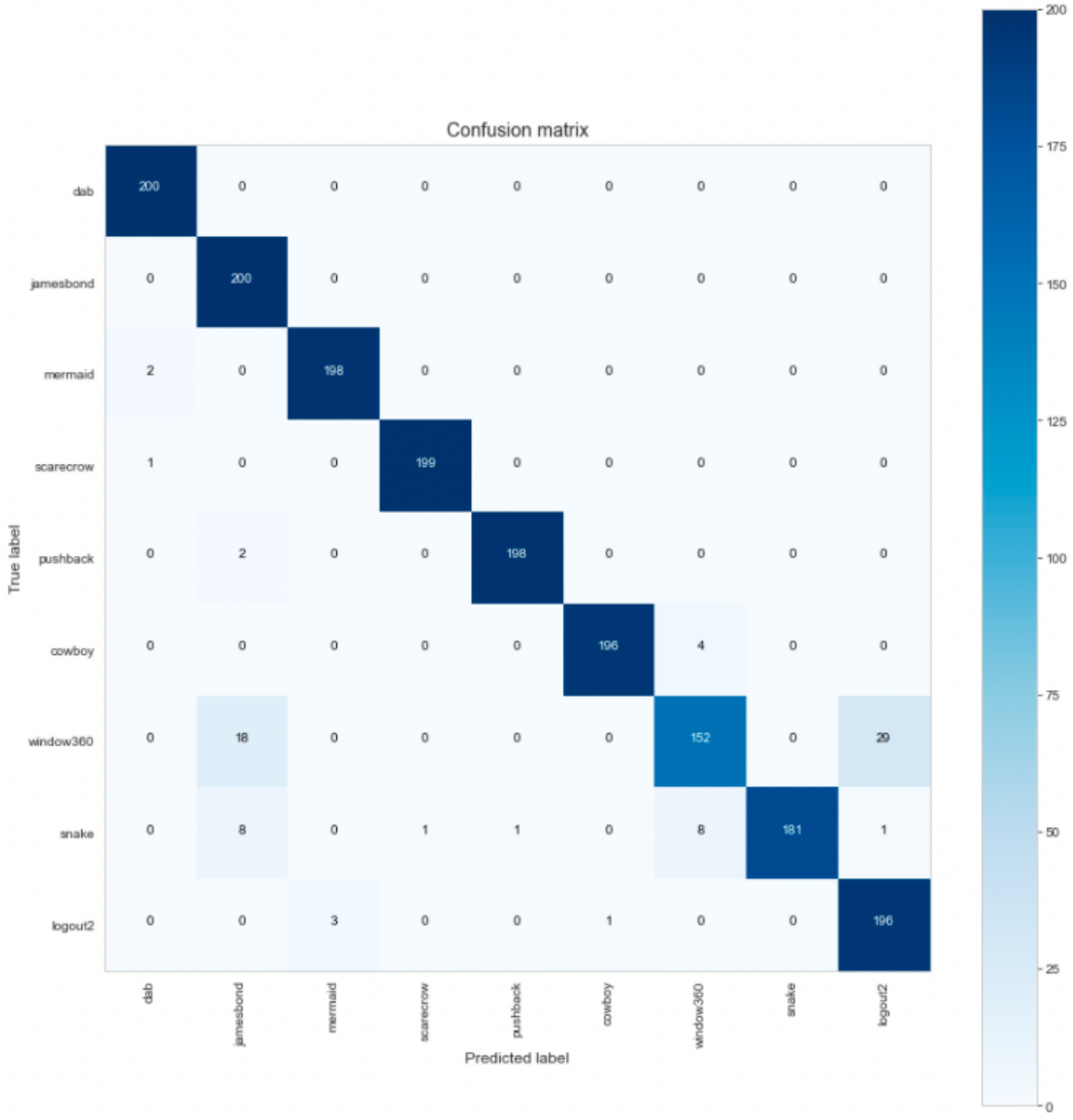


Fig. 5.1.7.a: Confusion matrix of final MLP model at 96% accuracy

As for scenarios of bluetooth disconnections, the corrupted packets were ignored and instead reconnections were made fast so that predictions are not affected. This is detailed further in section 4.1.11. During real-time evaluation, each dancer was required to obtain 3 valid dance move predictions, upon which the mode of predictions was chosen for final submission.

The diagonals of the confusion matrix in the Fig. 5.1.7.a above correspond to successful counts of prediction that match the true label of the dance move executed. Also, an ensemble learning model consisting of KNN and SVM was used as a benchmark to evaluate the accuracy of the MLP model with the same input vector. The final implementation of the MLP model was closest to the 97% accuracy achieved by the ensemble learning model.

Section 5.2 Software Dashboard

The dashboard is used to display important data such as dancer's relative position, predicted dance move, sync delay and also fatigue level of one dancer. Because setting up the dashboard involves creating both client and server side software, it is essentially a full stack solution.

5.2.1 Technology Overview

Figure 5.2.a

MongoDB is a NoSQL Database program that can be used to store data as JSON documents. The benefit of using a NoSQL database over relational databases is that it can handle large volumes of unstructured data and it is easily scalable. This makes it suitable for the purpose of our project which requires real time streaming of large amounts of data. We will be using MongoDB to store both predicted and raw data which will be accessed by the dashboard later.

MongoDB Realm is an alternative to the conventional Express and Node.js combination for a backend server. By writing realm functions the user can define and execute server side logic without hosting their own backend servers. The user can use these functions to perform CRUD operations on the collections and documents in the MongoDB database, which is required for the real time dashboard to work. Moreover, as it is very easy to create APIs for HTTP GET and POST requests, the project can be up and running very quickly if we use MongoDB Realm.

React is a javascript library used to build user interfaces. Individual components can be created and put together to create the pages required for the dashboard. React is chosen for the frontend because each of the components can be added or removed from the webpage as required which allows for rapid prototyping. Material-UI library is used as it has clear documentation and many prebuilt components which means that we do not have to create everything from scratch.

5.2.2 MongoDB Database design

The data is split into 4 collections for storage (“Prediction”, “Sensor1”, “Sensor2”, “Sensor3”).

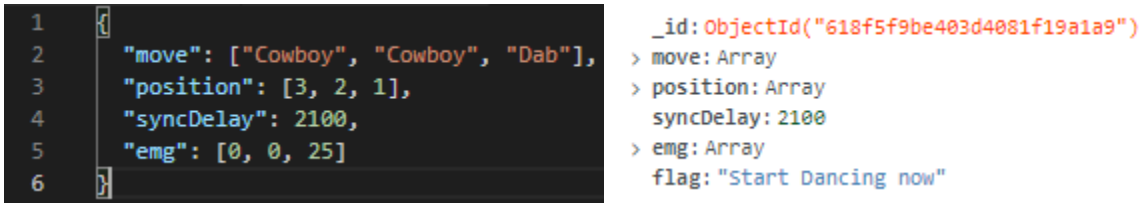


Figure 5.2.b: JSON object form of Predicted data

Prediction, as the name suggests, stores the data predicted by the ML algorithm such as position, moves and sync delay. For optimization purposes, EMG data is stored as a field under prediction data to reduce the number of collections to pull data from. When any of the predictions change, a new document with the change is inserted into the database with the above format.

Figure 5.2.c: JSON object form of Sensor data

Sensor1, 2 and 3 store the accelerometer and gyroscope data for dancers 1, 2 and 3 respectively. Each document holds 40 data points recorded over 2 seconds. If required, all the documents can be collated and the data retrieved as a single CSV file for usage by the ML specialist.

5.2.3 Mongo Realm “backend” design

For the dashboard to work, the Ultra96 will need to send processed data to the MongoDB database and the frontend must later be able to read this data. Mongo Realm has functions which can be used to expose API that allow for CRUD operations on the database.

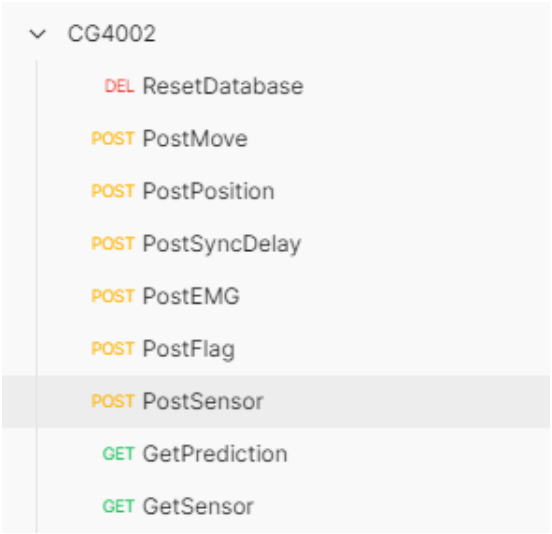


Figure 5.2.d: List of API (used by both ext comms and dashboard)

Status	Date	Runtime	Type	Name
OK	Nov 13 15:29:19+08:00	22ms	Webhook	GetSensor
OK	Nov 13 15:29:15+08:00	18ms	Webhook	GetPrediction
OK	Nov 13 15:29:12+08:00	24ms	Webhook	PostSensor
OK	Nov 13 15:29:09+08:00	19ms	Webhook	PostFlag
OK	Nov 13 15:29:05+08:00	27ms	Webhook	PostEMG
OK	Nov 13 15:29:01+08:00	17ms	Webhook	PostSyncDelay
OK	Nov 13 15:28:57+08:00	19ms	Webhook	PostPosition
OK	Nov 13 15:28:51+08:00	17ms	Webhook	PostMove
OK	Nov 13 15:28:47+08:00	20ms	Webhook	ResetDatabase

Figure 5.2.e: Tested performance of API (note the low latency)

```
exports = async function(payload, response) {
  if(payload.body) {
    const collection = context.services.get("mongodb-atlas").db("CG4002").collection("Prediction");
    const data = await collection.find().toArray();
    const body = EJSON.parse(payload.body.text());
    var newPrediction = {
      move: ["-", "-", "-"],
      position: [1, 2, 3],
      syncDelay: 0,
      emg: [0, 0, 50],
      flag: ""
    };
    if (data.length > 0) {
      newPrediction.move = data[data.length - 1].move;
      newPrediction.position = data[data.length - 1].position;
      newPrediction.syncDelay = data[data.length - 1].syncDelay;
      newPrediction.emg = data[data.length - 1].emg,
      newPrediction.flag = data[data.length - 1].flag
    }
    newPrediction.move[payload.query.dancerId - 1] = body.move;
    return await collection.insertOne(newPrediction);
  }
  return;
};
```

Figure 5.2.f: Realm function to update dancer move for one

By creating and writing simple realm functions such as the one above, Mongo Realm is able to replicate the function of a conventional Node Express backend server. The feature is hosted on an AWS server located in Singapore hence the latency is extremely low as shown in the 2nd screenshot above. As such, there is little delay between the calling of the API and the response with the data to the React dashboard.

MongoDB stores documents in EJSON format. Hence, some processing has to be done in the realm functions before storing or retrieving the data.

As an added precaution against missing or badly formatted data which can cause exceptions on the front end, some error handling is done in the realm function to ensure that the data sent to the dashboard is in the correct JSON format with the necessary fields.

5.2.4 Dashboard features

The React app forms the frontend of the dashboard component and will be used by the dancers to help guide them when practising.

5.2.4.1 Initial Dashboard Design

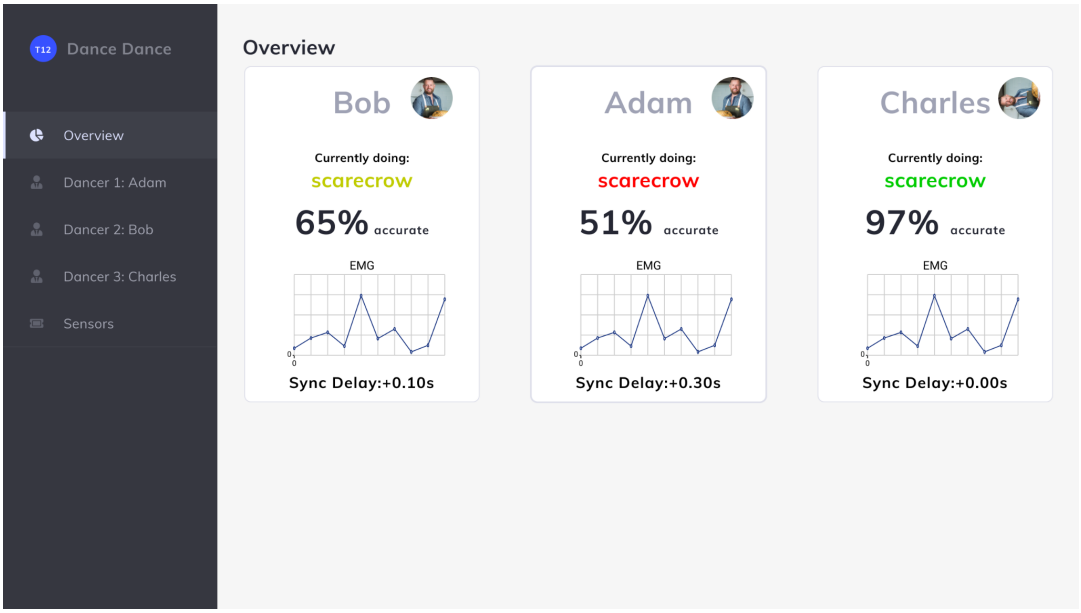


Figure 5.2.g: Overview page

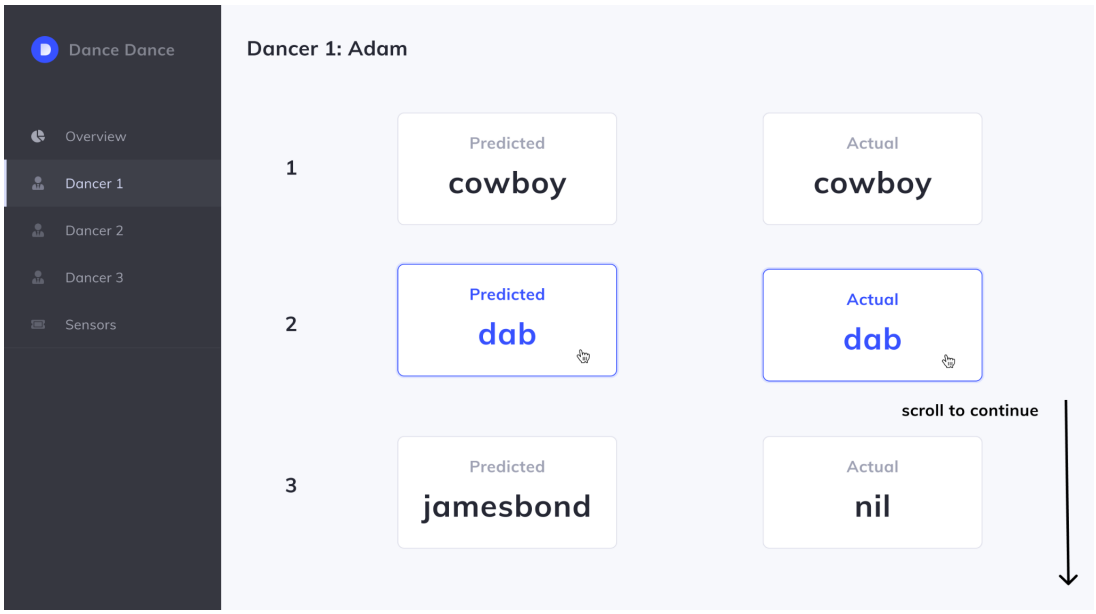


Figure 5.2.h: Prediction for each dancer

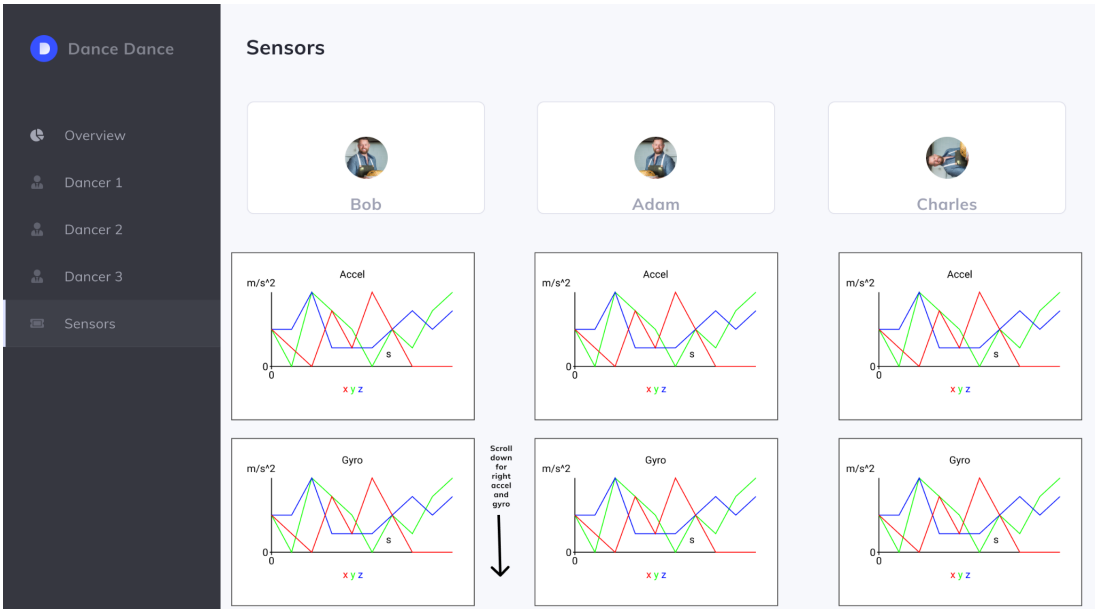


Figure 5.2.i: Sensor page

Before beginning work on the actual react app, some prototyping had to be done to ensure that the core functionality is captured and the design is pleasing to the user. Hence, the above images show the first high fidelity prototype created in Figma which was shown to the team.

Firstly, there is an overview page which allows the user to see the whole group of dancers at a glance. The move, position, sync delay and fatigue are shown on this page and these comprise the most important data for a dancing analytic application.

Next, there are individual dancer pages which allow the user to see the past and current predictions and also the ground truth for the dances. This page can be referred after the dance session to get an in depth analysis on how accurately each dancer is performing, so that they can improve on their own skills.

Lastly there is the sensor page that contains the raw sensor readings in graph form should there be a need to visualise/debug the sensor data. In the sensor page, the XYZ values of each accelerometer and gyroscope are plotted against the time axes. However, after discussion with the team, this page does not seem to fulfill the need of the user group which is made up of dancers practicing in a virtual environment. The raw values offer no significant help to the dancers. However, since there is a requirement for the project to display sensor data, this page was kept in the later prototypes.

5.2.4.2 User Survey

As with any prototyping process, there must be testing and feedback which will help the developers to improve upon the quality of the product.

The user survey should be conducted with the main user group in mind, which are dancers who will be using the dashboard to practice remotely. However, due to limitations of the project, we can only have the survey done within the project group itself.

The user survey contains questions related to the usability of the dashboard. For example, users can be given screenshots of the figma prototype and asked to identify what information they can derive from looking at the dashboard. The following are some survey questions:

Overview Page:

1. Is it clear which moves each of the dancers are performing?

2. Can you identify the relative positions of the dancers?
3. Does the EMG graph correctly relay your fatigue level?
    a. Should the scale be changed from voltage to percentage instead?
4. Are there any features that are important to the overview page not shown here?

Individual Page:

1. Is the design of the page intuitive to use?
    a. i.e. scrolling to see history
    b. Distinguish prediction from ground truth
2. Do you feel that it helps you to gauge your performance as a dancer?

Sensor Page:

1. Can you differentiate between the Gyro and Accel graphs?
    a. What about between left and right hand graphs?
    b. Are the colors helpful for differentiating the axes?
2. Are acceleration graphs useful to identify dances? Would velocity/time or displacement/time graphs be more appropriate?

There could also be other miscellaneous questions to figure out non-essential features that are good to have, such as maybe requirement for a dark/night theme, or showing the accuracy of dancers overall as a percentage.

From the initial feedback, it was mentioned that having 3 separate sensor pages was very inefficient as the user had to navigate multiple pages to look at 3 sets of data. Hence the sensor page was merged and mouse scrolling was used to browse the graphs in 1 page.

Moreover, the positional change was not very obvious which meant that the overview page UI needed some work. To distinguish the 3 dancers, they were given numbers and color coded icons to further help the dancers differentiate each other on the dashboard.

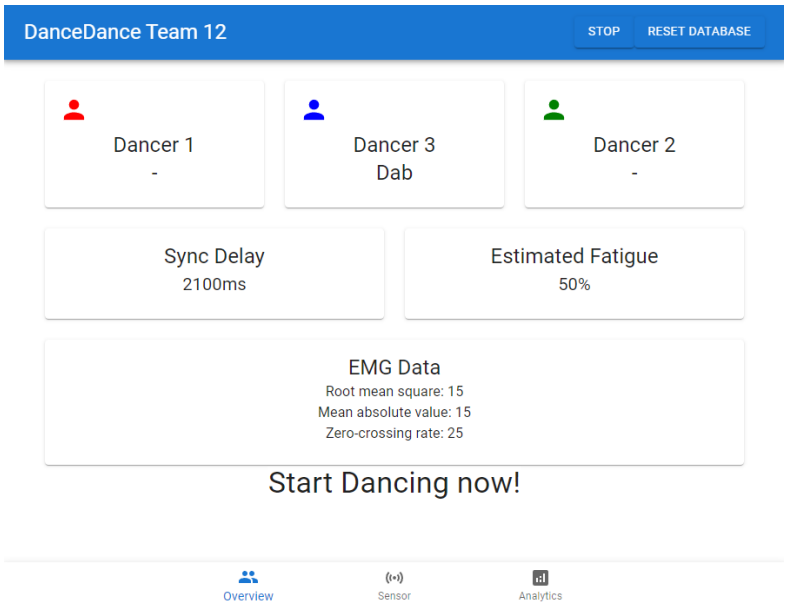


Figure 5.2.j: Overview page

Another feature added in the final design (shown above) was the addition of a message "flag". This allowed the external comms to send a text message which will show up to notify the dancers when to start dancing and when to change position, which helped the dancers know when the ML has detected their move. Unfortunately, the dancers were not allowed to view the dashboard to get this prompt in the final evaluation but we felt that this was a good feature that a coaching dashboard should have if it were not in an assessment.

5.2.4.3 Final Dashboard Design

The following section details the final dashboard design made in React after the recess week.

Each page is made up of 3 main components, namely the app bar (top), content (centre) and navigation menu (bottom). The app bar and navigation menu is constant throughout all 3 pages while the content changes based on the routing of the webpage.

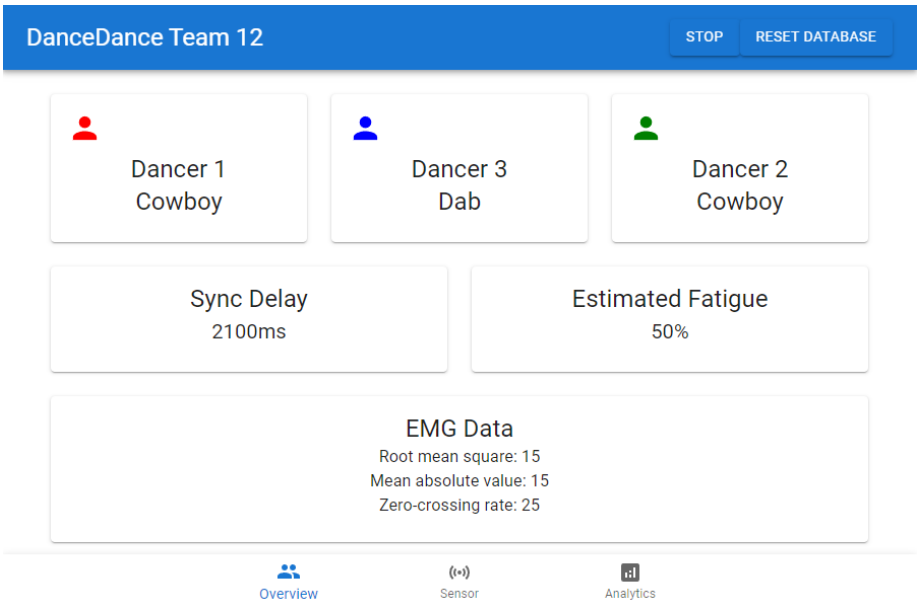


Figure 5.2.k: Overview Page

The overview page contains the most important information that a dancer needs at a glance. This includes the dance move, position, sync delay and emg/fatigue data. The dancer cards are color coded and the texts have large fonts so that they are still readable even when the dancer is standing away from the laptop/mobile device.

For the emg data, the Zero-crossing rate (ZCR) is relevant to measuring fatigue. We found through testing that ZCR values lie between 0 and 50 and are somewhat inversely proportional to muscle exertion. The threshold differs from person to person so our calculated percentage is only a rough estimate of fatigue level.

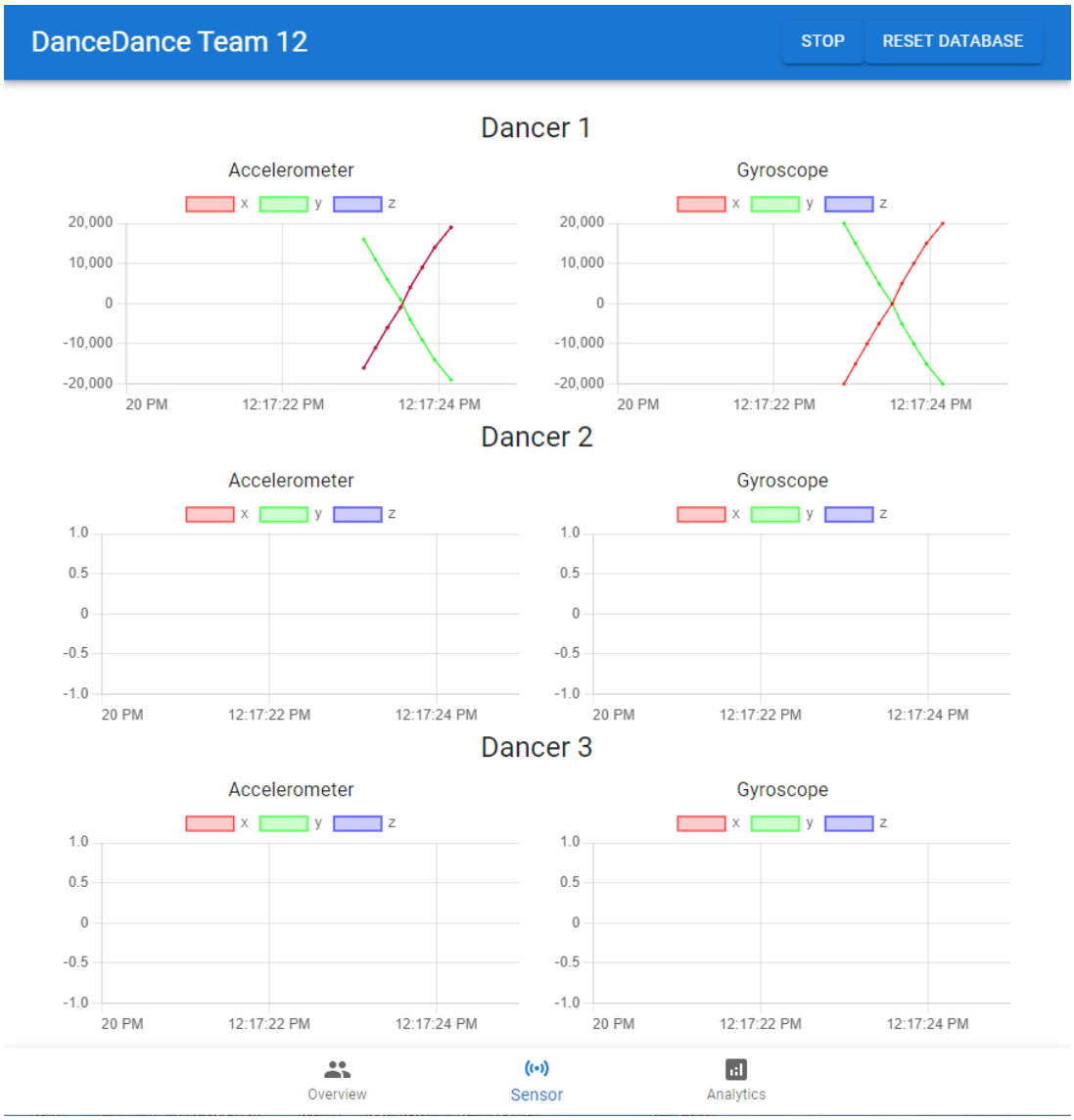


Figure 5.2.l: Sensor Page

The sensor page is made up of 2 sets of real-time graphs for each of the 3 dancers. One graph displays accelerometer data while the other displays gyroscope data. Chart.js was chosen as the library for creating the graphs as it has acceptable performance for a large number of data points. Moreover, there are tutorials with examples for creating real-time graphs in React.

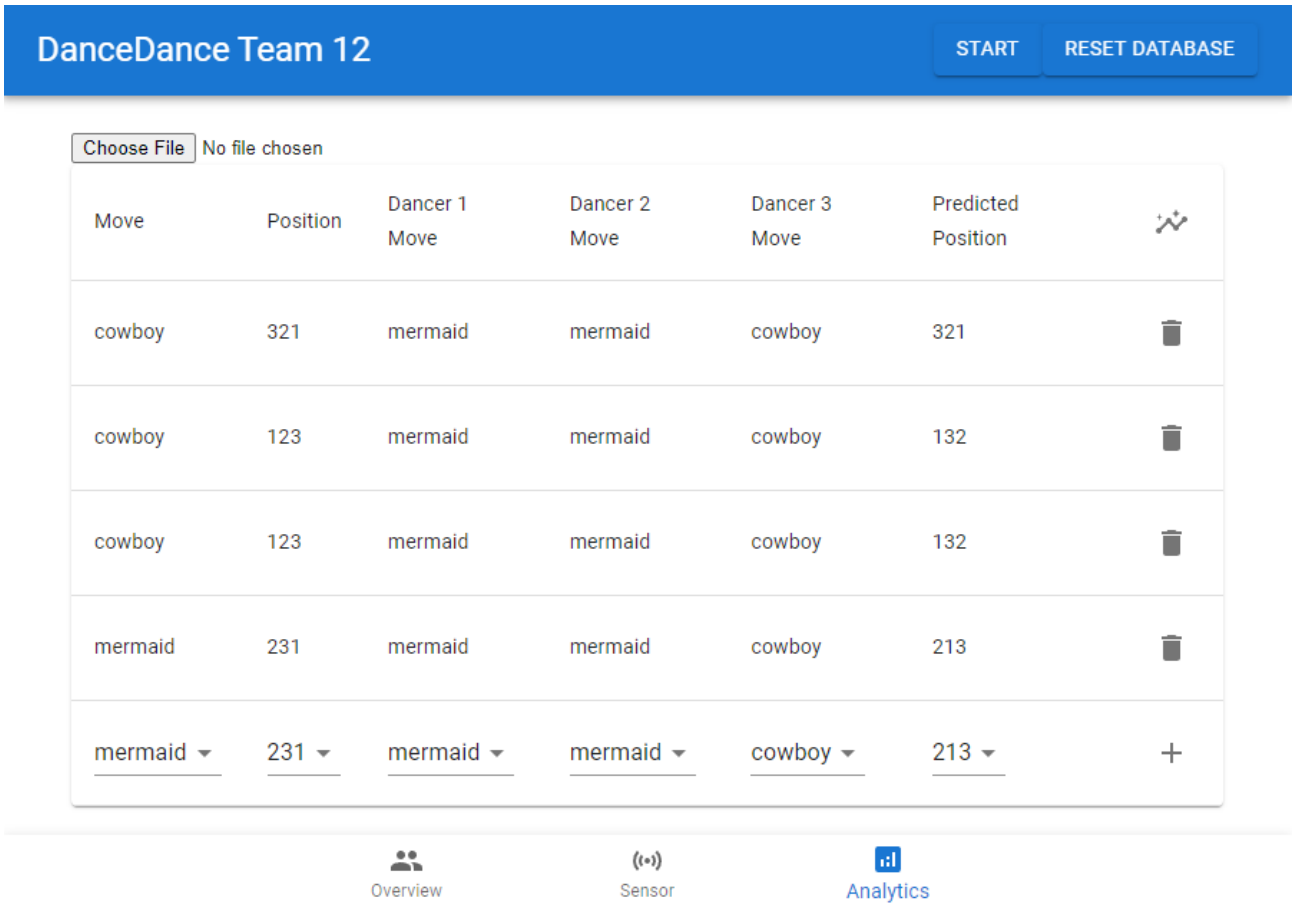


Figure 5.2.m: Offline analytics page

The offline analytics offers a way to store and calculate move and positional accuracy without connecting to the database. The user inputs both the ground truth and the user's dance move which may be visually confirmed or checked using the console output. Each set is added as a new row to the table and at the end of the session the user can click on the analytics icon on the top right to open up the calculations overlay (shown in following page of report).

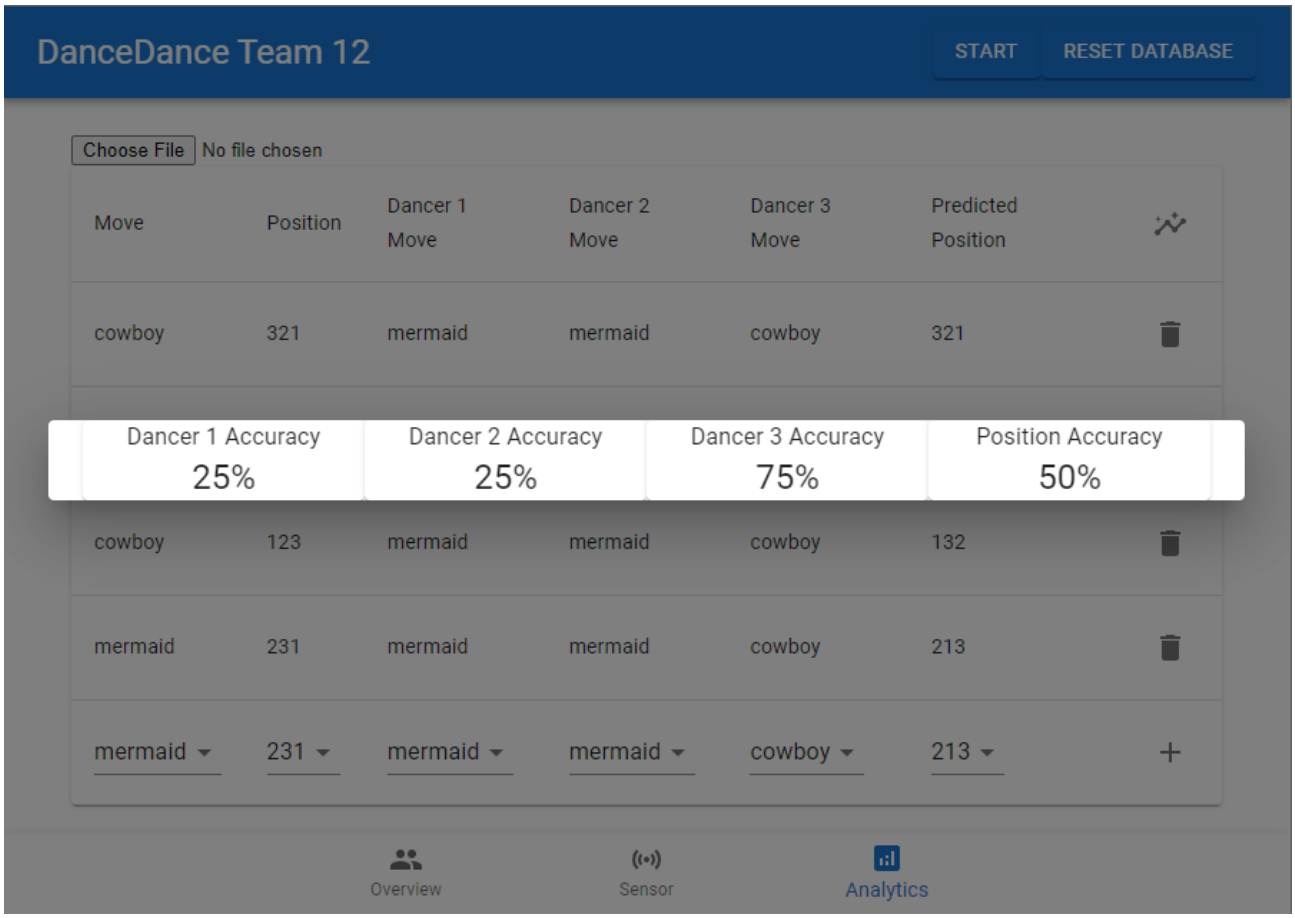


Figure 5.2.n: Offline analytics page (with calculations)

The user can then click anywhere to close this overlay and return to the application. This feature also allows users to upload their results as a json file.

5.2.5 Real-time Streaming of Data

The dashboard is updated using a polling method that makes API calls at fixed intervals. We use the setInterval function in React to call GET requests every X milliseconds and retrieve data from the database.

```
useEffect(() => {
  let interval = null;
  if (isPaused) {
    clearInterval(interval);
  } else {
    interval = setInterval(() => {
      GetPrediction();
    }, 500);
  }
  return () => {
    clearInterval(interval);
  };
}, [isPaused]);
```

Figure 5.2.o: setInterval to retrieve prediction data

The predictions on the overview page are polled every 500ms. We take 3 prediction windows of 2 seconds each which means that in the worst case scenario, there should be only a delay of slightly over 6.5s from the start of the dance to the detection showing up on the dashboard.

```
useEffect(() => {
  let interval = null;
  if (isPaused) {
    clearInterval(interval);
  } else {
    interval = setInterval(() => {
      GetSensor("1");
      GetSensor("2");
      GetSensor("3");
    }, 1000);
  }
  return () => {
    clearInterval(interval);
  };
}, [isPaused, GetSensor]);
```

Figure 5.2.p: setInterval to retrieve sensor(s) data

The sensor readings collect 40 data points over 2 seconds. Hence, in order to not miss any updates, the polling rate was set at 1 second for the dashboard side.

```
options={{
  responsive: true,
  scales: {
    x: {
      type: 'realtime',
      realtime: {
        duration: 5000,
        refresh: 50,
        delay: 1000,
        pause: false,
        ttl: 8000,
        frameRate: 30,
        onRefresh: chart => {
          var data = GetLatestData(dancerId);
          if (data) {
            chart.data.datasets[0].data.push({x: DateTime.now(), y: parseInt(data.x)});
            chart.data.datasets[1].data.push({x: DateTime.now(), y: parseInt(data.y)});
            chart.data.datasets[2].data.push({x: DateTime.now(), y: parseInt(data.z)});
          }
        }
      }
    },
```

Figure 5.2.q: options from Chart.js

The above shows the options setup for the graph in Chart.js. The data points are pulled from the database by calling the API and stored into a queue. There are 40 data points every 2 seconds hence a set of XYZ axes are de-queued every 50 milliseconds to be inserted into the graph.

For the sake of performance, data is given a time-to-live (TTL) of only 8000 milliseconds and the framerate of the graph is set to only 30 fps.

Section 6 Project Management Plan

	Week 5 - Work on individual requirement
HW Sensors	1. Wire up circuit for hardware components 2. Set up Bluno Beetle to acquire sensor readings
HW FPGA	1. Implement layer logic in HLS 2. Write C testbench
Int. Comms	1. Complete setting up connection for a beetle to interact with a laptop 2. Download required libraries on arduino IDE to read in sensor data and libraries on laptop to connect via BLE
Ext. Comms	1. Complete client socket on Ultra96 to interact with Evaluation Server 2. Implement functionality to send data to Dashboard Server
SW ML	1. Experiment implementation of MLP using kaggle dataset 2. Evaluate MLP implementation
SW Dashboard	1. Setup MongoDB Realm to replace conventional backend 2. If above fails, setup node + express server locally

	Week 6 - Complete 50% of individual requirement
HW Sensors	1. Get sensor readings using Bluno Beetle
HW FPGA	1. Synthesize into RTL 2. Verify RTL using C/RTL cosimulation 3. Export each layer into IP
Int. Comms	1. Implement handshaking protocol 2. Transmit data packets in the correct format 3. Implement stable connection between 2 beetles and the laptop for at

	least 1 min
Ext. Comms	1. Complete client script on laptop to connect to Ultra96 2. Complete server socket on Ultra96 to receive data from laptop
SW ML	1. Implement k-NN using kaggle dataset 2. Evaluate implementation of k-NN
SW Dashboard	1. Work with external comms to try and send data from Ultra96 to Database 2. Finalise mongodb schema (what data goes where and how it is stored in the JSON)

	Recess Week - Work on individual requirement
HW Sensors	1. Acquire sensor readings for dance moves
HW FPGA	1. Stitch IPs together using IP integrator 2. Create PYNQ driver script 3. Download bitstream to FPGA
Int. Comms	1. Implement reconnection function to tackle scenarios where beetles disconnects with the laptop 2. Ensure data packets transmitted are correct and no issues
Ext. Comms	1. Implement Clock Synchronization between laptop and Ultra96
SW ML	1. Experiment implementation of CNN 2. Compare against other models implemented thus far and evaluate
SW Dashboard	1. Create a react app, try to display real time graph with dummy data first

	Week 7 - Complete 100% of individual requirement
HW Sensors	1. Ensure all hardware components are working properly
HW FPGA	1. Test accelerator with dummy data
Int. Comms	1. Test and debug, fix issues for individual evaluation 2. Implement fragmented packets to be reassembled
Ext. Comms	1. Demo secure comms between laptop and Ultra96 2. Demo secure comms between Ultra96 and Evaluation Server 3. Demo Clock Synchronization Protocol locally with dummy processes
SW ML	1. Ensure completion of at least 2 ML models compatible with FPGA's NN 2. Demo models and their evaluation
SW Dashboard	1. Test full stack, getting real time data from Database to React frontend. 2. Prepare for individual component showcase

	Week 8 - Integration between parts
HW Sensors	1. Work with Int. Comms to debug any issues
HW FPGA	1. Work with SW ML to extract parameters from trained model 2. Test accelerator using updated weights 3. Compare accuracy with SW ML implementation
Int. Comms	1. Integrate with Ext. Comms to unpackage data packets and send it to ultra96 2. Finish complete setup - 3 laptops should be set up with 1 beetle connected each. 3. Improve reliability of data transfer
Ext. Comms	1. Integrate laptop BLE script from Int. Comms with laptop socket client

	script to send data to Ultra96 2. Work with HW FPGA to run ML prediction on Ultra96 3. Implement functionality to calculate relative position in the laptop script using sensor data
SW ML	1. Get team to generate data, specifically for the first 3 moves 2. Pre-process signal data and extract useful features 3. Feed extracted features as inputs to the ML model
SW Dashboard	1. Integrate working full stack with the system, data should be able to move from Ultra96 to database to display on React app in real time

	Week 9 - System works with one dancer for first 3 moves
HW Sensors	1. Gather data for SW ML to improve machine learning models
HW FPGA	1. Work with SW ML on possible neural network optimizations 2. Update FPGA implementation with new weights/layers
Int. Comms	1. Complete integration with external comms 2. Perform rigorous testing by emulating exam scenario to fix and connection bugs 3. Identify any corner cases and tackle them
Ext. Comms	1. Work with Int. Comms to implement Clock Synchronization Protocol a. Synchronize Beetle's clock with laptop clock 2. Work with HW Sensors to identify a start packet for each dance move 3. Calculate synchronization delay
SW ML	1. Model ready to handle one dancer for first 3 moves 2. Continue generating data for the remaining dance moves and relative positions

SW Dashboard	1. Improve on react app design, test with user survey. 2. Add or remove necessary pages to React app.

	Week 10 - Work on expanding to 3 dancers
HW Sensors	1. Integrate system to work with 3 dancers
HW FPGA	1. Work on optimizing hardware using HLS directives 2. Reduce latency of design
Int. Comms	1. Complete integration with external comms 2. Perform rigorous testing by emulating exam scenario to fix and connection bugs 3. Identify any corner cases and tackle them
Ext. Comms	1. Ensure Ultra96 is able to handle data incoming from 3 laptops 2. Ensure low latency in transmission of real-time data from Ultra96 to Dashboard Server
SW ML	1. Implement ML model to handle relative positions 2. Continue gathering data for all dance moves, relative positions to amplify data points 3. Evaluate models and optimise for better classification
SW Dashboard	1. Reduce latency between dashboard and Ultra96 2. Reduce latency between dashboard and database 3. Make changes on frontend to accommodate 3 dancers

	Week 11 - System works with 3 dancers for first 3 moves + relative positioning
HW Sensors	1. Fine-tune system and make improvements if any

HW FPGA	1. Prepare for evaluation 2. Continue optimizing accelerator
Int. Comms	1. Do testing and debug any issues that arise 2. Fine-tune protocol
Ext. Comms	1. Fine-tune and optimize synchronization delay
SW ML	1. Model ready to handle 3 dancers for first 3 moves and relative positioning 2. Generate data for dance moves and relative positions 3. Continue to identify optimisation techniques to improve classification results
SW Dashboard	1. Make small aesthetic changes to React app (adjust color, font, size of containers) to increase visibility

	Week 12 - Calibration and testing for additional dance moves + fine-tuning
HW Sensors	1. Final test on full system 2. Work on final report
HW FPGA	1. Prepare for final evaluation 2. Work on final report
Int. Comms	1. Prepare for final evaluation 2. Work on final report
Ext. Comms	1. Prepare for final evaluation 2. Work on final report
SW ML	1. Prepare for final evaluation 2. Identify best model to run on FPGA 3. Backup ensemble learning methods to be implemented

	4. Work on final report
SW Dashboard	1. Prepare for final evaluation 2. Make adjustments on the React app to accommodate all 3 dancers and all 8 dance moves including the logout move.

	Week 13 - Final Demo
HW Sensors	1. Finalize report 2. Final evaluation
HW FPGA	1. Finalize report 2. Final evaluation
Int. Comms	1. Finalize report 2. Final evaluation
Ext. Comms	1. Finalize report 2. Final evaluation
SW ML	1. System ready for final demo 2. Work on final report
SW Dashboard	1. Finalize report 2. Final evaluation

Section 7 Societal and Ethical Impact

7.1 Further Applications of Activity Detection Technology

While our system is specific to the use case of dance move prediction and relative positioning, the underlying hardware and software technology can be generalized and expanded to a variety of applications and domains that could have significant societal impact.

7.1.1 Active and Assisted Living applications for Smart Homes

Our activity detection wearables can be applied in the field of Active and Assisted Living. Such systems are designed to enable independent and healthy living of older or impaired people. The wearables can be combined with other ambient IoT sensors and actuators within the smart home to increase the overall ability of the system to detect and interpret actions of the user and interact seamlessly with the user. In addition, the wearable sensors could also be used for fall detection. For such individuals who may be living alone, a fall could potentially be life-threatening, and hence this system could detect such accidents and notify relevant personnel for help. With the increasing ageing population in Singapore, there is a larger group of people who require such assistance in their daily lives. This would drastically improve the quality of life of such individuals.

7.1.2 Healthcare monitoring applications

In the context of healthcare, activity detection can be extended to a patient activity monitoring system [21]. By creating wearable sensors for the patients, we are able to monitor the actions and movements of patients and their well-being at any point in time. Should the sensors detect any anomaly or irregular movement, relevant deductions could be made and assistance from healthcare professionals could be brought in immediately. With a huge increase of 60% in demand for nurses from 2008 to 2018 [21], the automation of monitoring patients would reduce the heavy load on hospitals and all other around-the-clock care givers. This is especially relevant in the current context of the COVID pandemic where hospitals struggle to cope with surges in hospital occupancy.

7.1.3 Virtual gym coach

The concepts used in our wearable device can also be applied to help people who use the gym regularly but do not have a trainer. By making changes to the ML model used, the device can be used to check if users are doing exercises in the correct posture and provide virtual guidance with a dashboard. This would further promote physical activity and exercise among people while reducing the likelihood of injuries.

7.2 Ethical Concerns

7.2.1 Data Privacy and Security

With activity detection and monitoring, large amounts of data of the user's actions and movements would be collected for the purposes of the specific application. However, without proper safeguards, this data could potentially be unethically obtained by a third-party. This would reveal the user's current activity and location among other sensitive information, infringing on the privacy of the user. This needs to be avoided with the proper security protocols and safeguards implemented as essential features of the system.

7.2.2 Health and Safety

With increasing adoption of such technology in performance-critical situations such as healthcare, there may be dire and irreversible consequences for lapses in the performance. For instance, if the sensors of a patient monitoring system malfunctions and fails to report abnormal behaviour of the patient, it could result in delayed medical attention in a critical moment, potentially causing severe complications in the patient's health. Hence, it is essential that such systems be built and implemented with the end users' safety in mind, and should only be deployed in practice when the safety of the users is guaranteed.

7.3 Conclusion

In conclusion, the benefits and applications of advanced sensor-based activity detection are huge and there is a lot of active research within this field to use and improve such technology for

positive societal impact. However, while designing and implementing such systems, it is imperative to put in place measures to safeguard the interests of the users to avoid the potential ethical dangers of the technology.

References

[1] “Bluno_beetle_sku_dfr0339,” *DFRobot*. [Online]. Available: https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339. [Accessed: 05-Sep-2021].

[2] “DFRobot beetle BLE - the Smallest board based on Arduino Uno with Bluetooth 4.0,” *DFRobot*. [Online]. Available: https://www.dfrobot.com/product-1259.html. [Accessed: 05-Sep-2021].

[3]: “Gy-521 mpu6050 6dof accelerometer + gyro,” *Cytron Technologies Singapore*. [Online]. Available: https://sg.cytron.io/p-gy-521-mpu6050-6dof-accelerometer-plus-gyro. [Accessed: 05-Sep-2021].

[4]: “Addicore GY-521 MPU-6050 3-axis gyroscope and 3-axis accelerometer IMU,” *www.addicore.com*. [Online]. Available: https://www.addicore.com/GY-521-MPU6050-p/170.htm#:~:text=Dimensions%20(excluding%20pins)%3A%2021.2,%3A%202.1g%20(0.07oz). [Accessed: 05-Sep-2021].

[5] “MyoWare muscle sensor,” *SEN-13723 - SparkFun Electronics*. [Online]. Available: https://www.sparkfun.com/products/13723. [Accessed: 05-Sep-2021].

[6] “Lilypad power supply”, *DEV-11259 - SparkFun Electronics.* [Online]. Available: https://www.sparkfun.com/products/retired/11259. [Accessed: 14-Nov-2021]

[7] “HowToMechatronics”, *Arduino and MPU6050 Accelerometer and Gyroscope Tutorial*. Available: https://howtomechatronics.com/tutorials/arduino/arduino-and-mpu6050-accelerometer-and-gyroscope-tutorial/. [Accessed: 14-Nov-2021]

[8] “Last Minute Engineers”, *In-depth: Interface MPU6050 Accelerometer & Gyroscope sensor with Arduino.* [Online]. Available: https://lastminuteengineers.com/mpu6050-accel-gyro-arduino-tutorial/. [Accessed: 14-Nov-2021]

[9] Kim, H., Lee, J., & Kim, J. (2018, July 9). Electromyography-signal-based muscle fatigue assessment for knee rehabilitation monitoring systems. Biomedical engineering letters. Retrieved November 14, 2021, from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6209086/.

[10] “Bluno Beetle Simple Tutorial,” *DFRobot*. [Online]. Available: https://www.dfrobot.com/blog-283.html. [Accessed: 05-Sep-2021].

[11] M. Afaneh, “Bluetooth 5 speed: How to achieve maximum throughput for YOUR BLE application,” *Novel Bits*, 16-Jun-2020. [Online]. Available: https://www.novelbits.io/bluetooth-5-speed-maximum-throughput/. [Accessed: 05-Sep-2021].

[12] K. G. Zoysa, “Lets work with MPU6050 (GY-521) - Part1,” *Medium*, 26-Dec-2017. [Online]. Available: https://medium.com/@kavindugimhanzoysa/lets-work-with-mpu6050-gy-521-part1-6db0d47a35e6. [Accessed: 05-Sep-2021].

[13] Avnet, “Ultra96-pynq/ultra96_pmbus.ipynb at master · avnet/ultra96-pynq,” *GitHub*. [Online]. Available: https://github.com/Avnet/Ultra96-PYNQ/blob/master/Ultra96/notebooks/common/ultra96_pmbus.ipynb. [Accessed: 05-Sep-2021].

[14] H.-L. Synthesis, “Vivado Design Suite User Guide,” *Xilinx.com*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf. [Accessed: 05-Sep-2021].

[15]“SSH port forwarding/tunneling use cases and concrete examples. Client COMMAND, server configuration. Firewall considerations.,” SSH port forwarding/tunneling use cases and concrete examples. Client command, server configuration. Firewall considerations. [Online]. Available: https://www.ssh.com/academy/ssh/tunneling/example. [Accessed: 05-Sep-2021].

[16] A. Ferrari, D. Micucci, M. Mobilio and P. Napoletano, "Trends in human activity recognition using smartphones", 2021. [Online]. Available: https://link.springer.com/article/10.1007/s40860-021-00147-0#Sec4 .[Accessed: 01-Sep-2021]

[17] R. Khusainov, D. Azzi, I. E. Achumba, and S. D. Bersch, “Real-time human ambulation, activity, and physiological monitoring: taxonomy of issues, techniques, applications, challenges and limitations,” Sensors (Basel), vol. 13, no. 10, pp. 12852–12902, 2013, doi: 10.3390/s131012852. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3859040/ .[Accessed: 02-Sep-2021]

[18] A. Dehghani, O. Sarbishei, T. Glatard and E. Shihab, "A Quantitative Comparison of Overlapping and Non-Overlapping Sliding Windows for Human Activity Recognition Using Inertial Sensors", 2021. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6891351/ .[Accessed: 03-Sep-2021]

[19] Researchgate.net. [Online]. Available: https://www.researchgate.net/publication/337514687_Hand-crafted_Features_vs_Residual_Networks_for_Human_Activities_Recognition_using_Accelerometer .[Accessed: 03-Sep-2021]

[20]R. Bandaru, “Pruning neural networks - towards data science,” *Towards Data Science*, 01-Sep-2020. [Online]. Available: https://towardsdatascience.com/pruning-neural-networks-1bb3ab5791f9. [Accessed: 05-Sep-2021]

[21]“Activity Recognition Using Wearable Sensors for Healthcare,” *thinkmind.org*. [Online]. Available: https://thinkmind.org/articles/sensorcomm_2013_7_40_10165.pdf . [Accessed: 14-Nov-2021].

[22] D. Alden, “How to ‘Multithread’ an Arduino (Protothreading Tutorial),” *Arduino Project Hub*, 15-Jan-2016. [Online]. Available: https://create.arduino.cc/projecthub/reanimationxp/how-to-multithread-an-arduino-protothreading-tutorial-dd2c37. [Accessed: 05-Sep-2021].

[23] “Python bluepy.btle module,” *Python Module: bluepy.btle*. [Online]. Available: https://www.programcreek.com/python/index/8887/bluepy.btle. [Accessed: 14-Nov-2021].