CG4002 Computer Engineering Capstone Project
AY2020/21 Semester 2

"Dance Dance"
Design Report

Group **13**	Name	Student #	Role	Specific Contribution
Member #1	Sean Tan Rui Xiang	A0164209H	Hardware Sensors	Sections 2, 3.1 and 6.2
Member #2	Lim Cheng Yin, Ryan	A0184507B	Hardware FPGA	Sections 2, 3.2 and 6.2
Member #3	Karnati Sai Abhishek	A0184397M	Comms Internal	Sections 1, 4.1 and 6.2
Member #4	Chok Xin Yan	A0171272L	Comms External	Sections 1, 4.2, 6.2, 6.3 and 6.4
Member #5	Yeap Chun Lik	A0185876H	Software Machine Learning	Sections 5.1, 6.1 and 6.2
Member #6	Liu Chaojie	A0177842U	Software Dashboard	Sections 5.2 and 6.2

Section 1 System Functionalities– *Written by Chok Xin Yan, Karnati Sai Abhishek*

1.1 Introduction

Aim: To detect movements of a 3-person dance group and provide real-time feedback of the performance.

This system should consist of a portable, lightweight, and user-friendly wearable that is able to sense the user movements. There can be up to three dancers equipped with the wearable, and the data from these wearables should be transferred to a remote computer for processing. The results of the processing should be displayed in a meaningful manner to provide feedback to the user.

1.2 Use Case Diagram

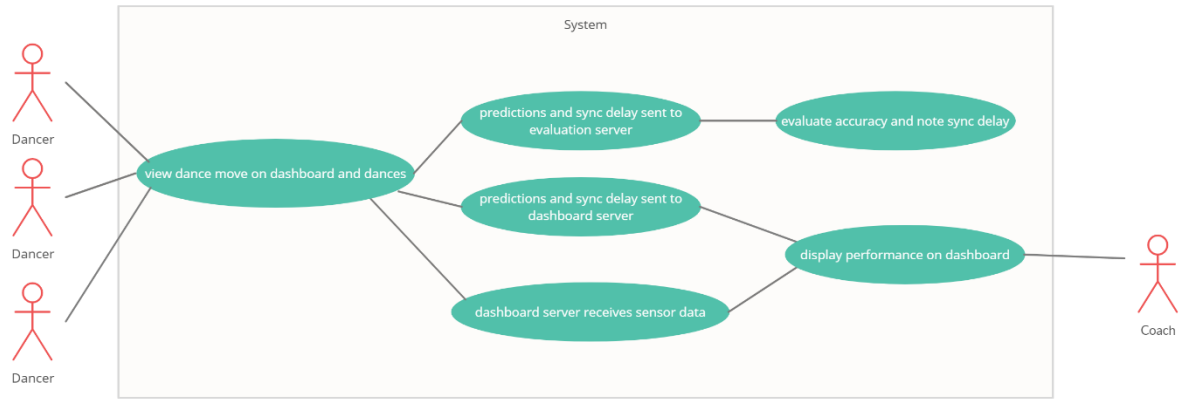


Fig 1.2-A
Use Case Diagram

1.3 Features

- Detects the start of dance moves
- Accurately differentiates between 8 different dance moves (+ 1 end move) quickly
- Detects the relative position of dancers within the group (Members of the group need not be physically present; they can dance remotely in their own homes)
- Determines the time delay between the quickest dancer and the slowest dancer
- Displays meaningful feedback to the users through an intuitive web dashboard
- Sensors packaged in a simple. user-friendly and lightweight wearable that has a battery life of at least 1 hour
- Track the progress and statistics of the dance group over time

1.4 User Stories

As a...	I want to...	So that...
Frequent dancer	Have wearables which are lightweight, easy to put on and remove	It is convenient for me to use and set up.
Clumsy dancer	Have a neat wearable that does not have any wires sticking out	I can dance freely and not get entangled.
Dancer who has a tight schedule but wants to practice dancing	Have wearables which run on low power	I can charge the device fast to use it whenever I want.

Lazy dancer	Be able to dance with my other friends virtually	I can play with my friends at the comfort of home.
Professional dancer	Have a system that measures the timing between dance moves across dancers	The group can be in synchrony and no one lags.
Intermediate dancer	Obtain real-time feedback of the performance	I can keep track and improve myself
Novice dancer	Have a system that tracks if my moves are correct	I can learn the dance moves.
Dance enthusiast	Have an intuitive web dashboard	I can review my performance easily to see how I did.
Anti-social dancer	Learn moves as accurately as possible from the machine	I can learn the dance and moves without a real-life instructor.
Dance coach	View the synchrony and accuracy of the dancers' moves	I can know if the dancers performed the right move.

Section 2 Overall System Architecture – *Written by Sean Tan, Ryan Lim*

2.1 High Level Architecture

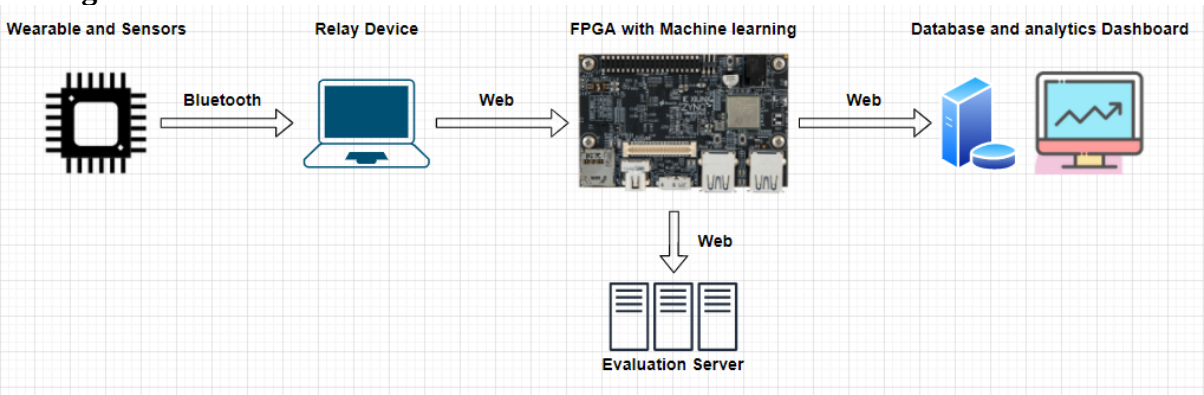


Fig 2.1-A
High Level Overview

Each Dancer will only have one wearable responsible for gathering data for dance move identification. This wearable contains one Bluno Beetle with one hardware sensor, the MPU-6050. The following is the overall flow of data within our system:

1. There will be one beetle on each dancer's right arm. The beetles will be connected to two sensors, one IMU and one proximity sensor.
2. The beetles will interpret data from the two sensors, process and/or extract some features
3. This information will be sent to the users' laptop over BLE. The laptop will then pass the information to the Ultra96 board over the Internet for classification. In the case where we find that we are unable to establish decent Bluetooth connection using a laptop, we will fall back to using an android device as the BLE adapter for our relay laptop. The use of the android device as a Bluetooth adapter should be transparent to the rest of our system.
4. The Ultra96 will perform more complex feature extraction and processing, then run the data through a neural network classifier.
5. Classified dance moves will be sent to the evaluation server and the dashboard database.
6. The dashboard database will notify dashboard UI and push the results and real time data from the database for display.

2.2 System Communication

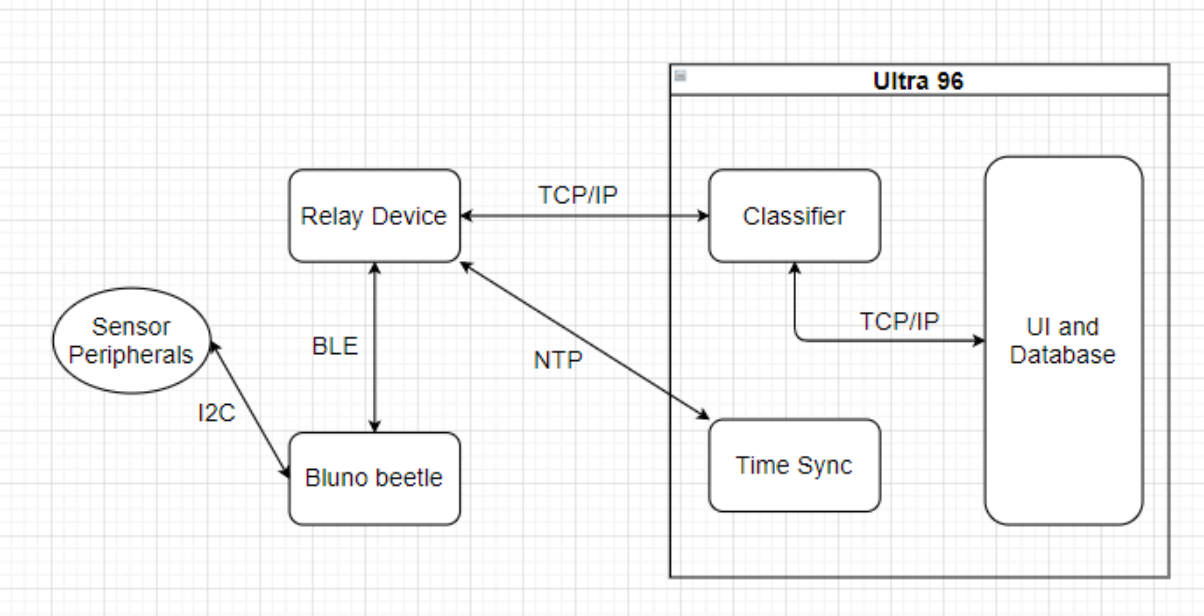


Fig 2.2-A
Communications Overview

There are two broad communication categories for this project: Internal and External. The internal communications refer to the communication between our wearable and our relay laptop. External communications refer to the communication between our relay laptop and the Ultra96, as well from the Ultra96 to the evaluation and dashboard servers. Additionally, the communications between sensor peripheral (MPU-6050) and the Bluno is achieved through a shared I2C bus.

Internal communications will be implemented via Bluetooth, specifically Bluetooth Low Energy. The Bluno has a CC2540 that acts as a Serial-BLE bridge. Once configured and connected, the Bluno will be able to send data to the relay laptop as if over a Serial Connection. This helps hide the complexity of the Bluetooth stack and allows us to easily establish communication with the Laptop. That said, we still have to be cognizant that the underlying communications protocol is BLE, and therefore the size of packets we design and the rate at which we send data should remain within the constraints of BLE connections. Internal communications will be detailed in 4.1 Internal Communications.

External communications will be implemented via TCP/IP over WiFi/Ethernet. The relay laptop and the Ultra96 will communicate via TCP sockets. Similarly, the Ultra96 will communicate with the evaluation server and dashboard database using TCP sockets. Furthermore, the Ultra96 will use NTP to achieve time synchronization with relay laptops, and by extension, the Bluno Beetles. All TCP connections will be encrypted via AES. Furthermore, each TCP connection on the Ultra96 will have its own dedicated thread. External communications will be detailed in *Section 4.2 External Communications*.

2.3 Main Algorithm Overview

To achieve classification of dance moves, we implement the following algorithm:

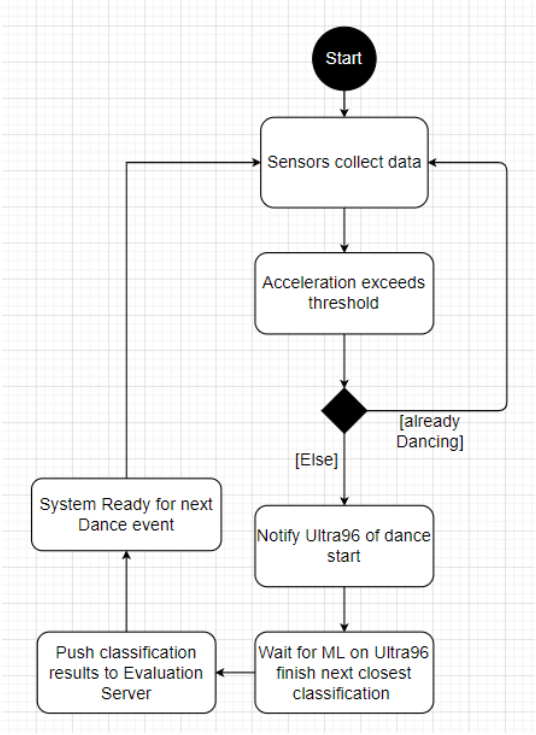


Fig 2.3-A
Main Algorithm Overview

Briefly, this algorithm can be separated into 4 general parts:

1. On system start, the Bluno and its sensors will start to collect and process data. This data processing includes filtering and thresholding to increase the signal-to-noise ratio. This data is streamed via the relay device to the Ultra96.
2. The start of a dance move is defined as the time at which the MPU-6050 on the Bluno detects a large movement of the dancer's hand. This can be accomplished simply by setting a threshold on the acceleration vector produced by the MPU-6050. To prevent multiple consecutive false detection, the Bluno will only detect the start of a dance move if the system is not currently analyzing a previous dance move.
3. On detection of dance move start, the Bluno will notify the Ultra96 using the BLE connection. This notification can be achieved by setting a flag within the data payload of Bluetooth communication packet.
4. On receipt of the start notification, the machine learning pipeline collect a window of a number of samples after the start flag to be used for classification of the current dance move. More information regarding the machine learning architecture can be found in *Section 5.1.1*.
5. Once this classification is complete, the system is ready for the next dance move. On top of the fixed length lockout period after the start of each dance move, the Bluno will also only be able to consider the start of the next dance move after the acceleration stabilizes to a low value below the threshold.

Final Implementation

While the general algorithm remains mostly unchanged, we have implemented a state machine to handle our main loop. Please refer to Section 2.5.

2.4 Intended Final Form

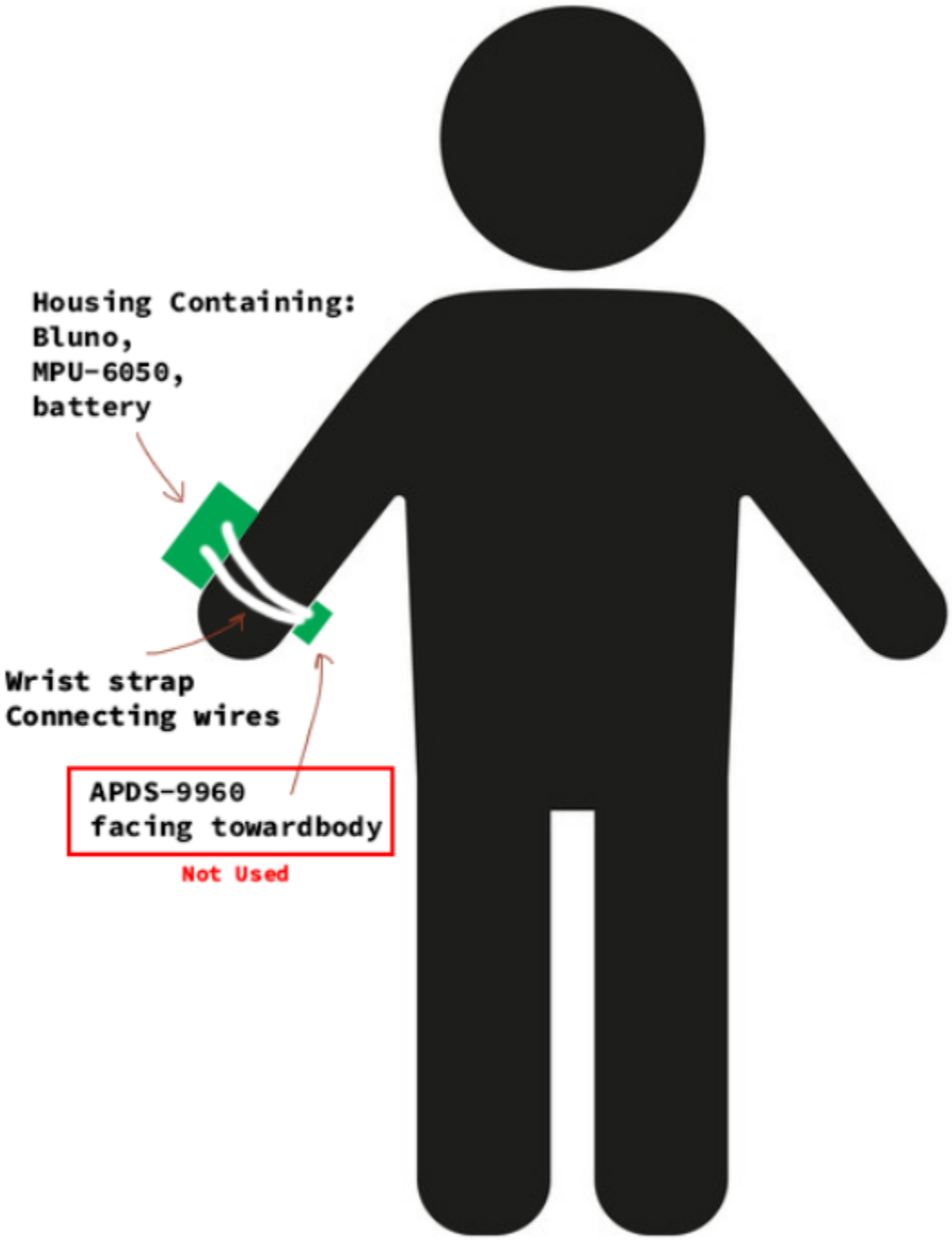


Figure 2.4-A
Wearable form factor

We intend to have the beetles, sensors and power source integrated into a self-contained unit that we be placed on the right wrist of each dancer, much like a smartwatch. The electrical components will be mounted onto a 3D-printed frame to ensure that parts will not fly off during the performance of dance moves, with the proximity sensor positioned on the inside face of the wrist. Since our system is relatively low powered, and we do not expect to have this system running for extended periods of time, we will use small form factor coin cell batteries so that the users will not be weighed down during dancing while still allowing for sufficient battery life. Refer to section 3.1.1 for details regarding the final implementation of the wearable.

2.5 State Machine

Our system features a finite state machine that manages and controls the state of our system. This state machine can be found in *Test/relayMain.py*. Each dancer, and thus Bluno, has its own states and is managed by a self-contained state machine. Therefore, each Bluno handles its own state and lifecycle via its own state machine hosted on the Ultra96. This state machine corresponds to the *DeviceDataBuffer* class. To govern our overall system lifecycle and state, there is additionally a Master state machine, which corresponds to the *liveDataEngine* class.

The following diagram illustrates broadly illustrates the state machine that handles each Bluno. Since there are many flags and counters within the state machine, the following diagram is simplified to show only the most important components of the state management for each Bluno.

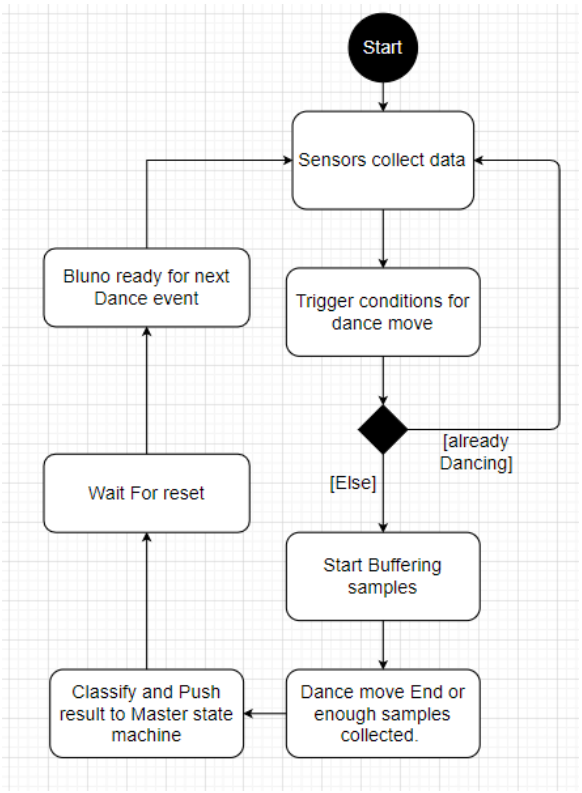


Figure 2.5-A
Flow Diagram for DeviceDataBuffer

During normal operation, once connected, each Bluno state machine will generally transition between three states, *idle*, *dancing*, and *post-dance*. When in the *idle* state, the Bluno state machine waits for the detection of the start of a dance, in which case it will transition from the *idle* state to the *dancing* state. In the *dancing* state, the Bluno state machine will explicitly be buffering samples received from its corresponding Bluno. The Bluno state machine will wait for either 100 samples to be buffered, or the detection of the end of a dance. When either of these events occur, the Bluno will submit buffered samples for feature extraction and subsequent prediction. Upon the result of the prediction, the Bluno state machine will transition to a *post-dance* state, where it will ignore subsequent start of a dance detection events until the Bluno state machine is reset to the *idle* state by the Master state machine upon reply of the evaluation server. In addition, as part of our algorithm to determine dancer positions, this module also implements the turn detection for each Bluno. Turn detection events can only occur in the idle state. The result of each Turn detection will be passed to the Master state machine.

A Master state machine governs the overall state of our system. It is responsible for keeping track of the both the results of the prediction of each Bluno state machine, as well as the results of any turn detection events that occur for each Bluno. The Master state machine contains two main modules, the *positionTrackingSystem* and the *predictionEnsembleManager*. The master machine itself is *liveDataEngine.*

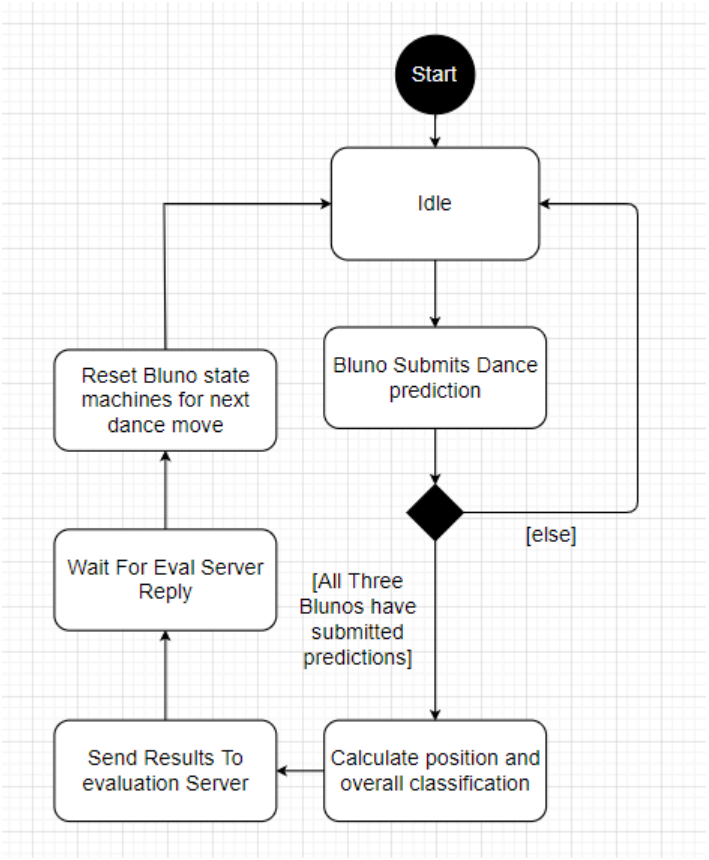


Figure 2.5-B
Flow Diagram for liveDataEngine

The *positionTrackingSystem* keeps track of the current position of each dancer, as well as registers turn events from the Bluno and infers the new positions of the dancers depending on which direction each dancer turned. The *positionTrackingSystem* will either attempt to calculate a new set of dancer positions when it receives a turn event from each Bluno, or the *predictionEnsembleManager* receives a prediction result from all three Bluno's. The algorithm for the inference will be further discussed in Section 3.1.6.3.

The *predictionEnsembleManager* is responsible for keeping track of the predictions submitted by each Bluno state machine. Upon receipt of a dance move prediction from each of the three Bluno state machines corresponding to each dancer, the *predictionEnsembleManager* will submit the result with the highest occurrence within the three predictions to the evaluation server along with the dancer positions obtained from the *positionTrackingSystem*.

Section 3 Hardware Details

3.1 Wearable Hardware - *Written by Sean Tan*

This section will detail the hardware architecture of the wearable sensor system, as well as electrical characteristics of individual components. Hardware components highlighted in Red were part of the original design but were not included in the final implementation. Where relevant and appropriate, the initial design choices and changes from the original design report will be explained.

Throughout Section 3.1, these explanations will have the sub header **Final implementation.**

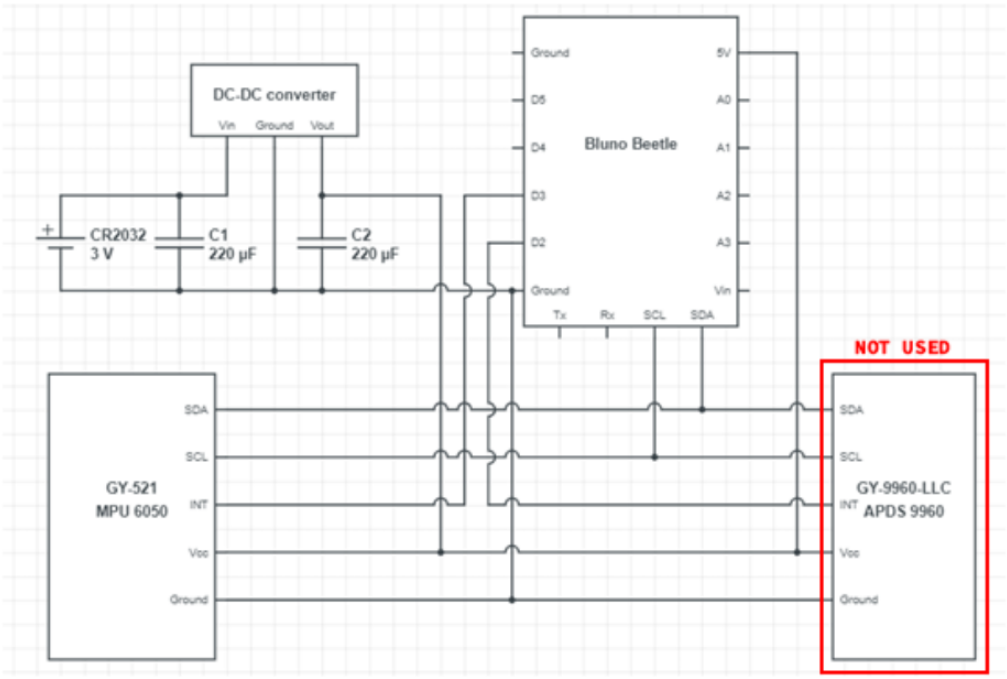


Figure 3.1-A
Wearable Schematic

As seen in Figure 3.1-A, each wearable will consist of:

1. a DFR0339 Bluno Beetle Microcontroller
2. an MPU-6050 Inertial Measurement Unit (IMU) in the GY-521 package
3. an APDS-9960 Digital Proximity, Ambient Light, RGB and Gesture Sensor in the GY-9960-LLC package (Unimplemented)
4. a 0.9-3V to 5V DC-DC Converter (switching boost converter)
5. CR2032 coin cell battery

For communications between the Bluno and peripherals, we will be using a singular I2C bus via the SDA/SCL pins. Additionally, both sensor peripherals generate interrupts, and therefore will be connected to the Bluno D2 and D3 Pins. All other data Pins are unused.

Final Implementation

Our initial design included an APDS-9960 sensor. This sensor was at its core, a proximity sensor. The concern was that certain dance moves might be too similar with just the accelerometer and gyroscope readings. Therefore, we initially thought to use it as an additional source of information to discriminate between dance moves, since the dancers' body would occlude the sensor differently for each dance move. However, after empirical testing, we discovered that the sensor readings from the IMU were sufficient to reasonably discriminate between dance moves (refer to *Section 5.x.x* for details), and therefore the

APDS-9960 was left out the final implementation. However, the documentation for the APDS-9960 sensor is left in this report as potential improvement to our system.

3.1.1 Final Hardware Form Factor and Design Philosophy

Final implementation

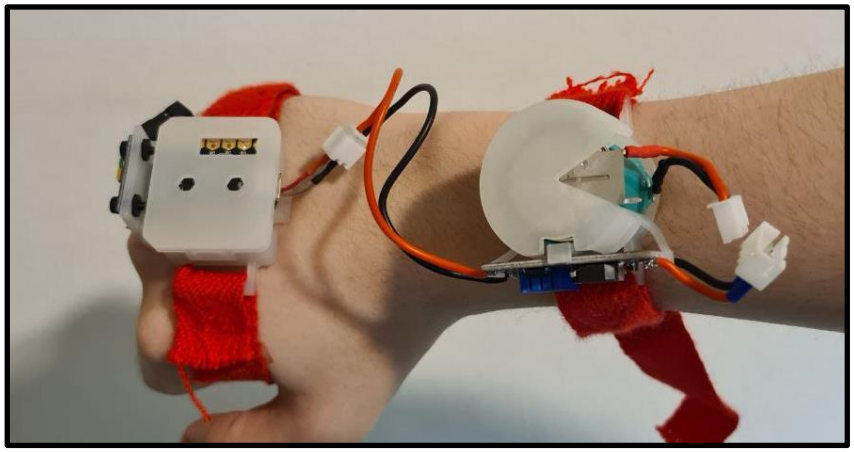


Figure 3.1.1-A
Mounted Sensor Unit and Battery

Our design Philosophy revolves around the following core concepts:

- The wearable must not be bulky. It must be easy to mount and unmount, and easy to dance and move around in while mounted. It also must be sturdy enough to withstand vigorous dancing.
- The wearable must be modular. Individual components should be able to be swapped out without much trouble. Different electronic components should not be permanently affixed to each other but connected by easily removed but reliable connectors. This facilitates quick and easy ad-hoc replacement of failing or failed parts, even by other hardware components that fulfil the similar function. This also allows for rapid prototyping.
- The mounting system should be flexible enough to accommodate changes to hardware. This also allows for rapid prototyping.

Our hardware design thus focuses heavily on the modularity of hardware components to facilitate rapid prototyping. With this in mind, we designed the mounting systems for our wearable.

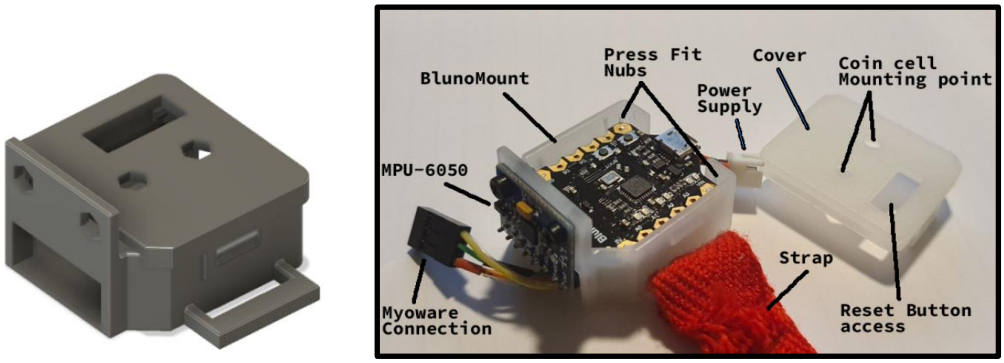


Figure 3.1.1-B
Bluno Holder module CAD and Diagram

The Bluno/IMU module is strapped around the palm. It consists of the Bluno and the MPU-6050 IMU. It is imperative that the IMU is held securely in the mount. Therefore, it is secured to the mount by a pair of M2.5 Hex nuts and bolts. Internally, it is connected to the Bluno via DuPont connectors.

The Bluno is connected to the mount via press/friction fit as well as 2mm nubs in each corner to prevent it from being dislodged during dancing. Even without the cover Bluno does not come loose from its case when dancing. The Bluno can be connected to the external battery unit via 2 pin JTAG connector.

The cover comes with additional mounting points for a planned small coin cell battery module that would be secured to the cover via the M2.5 Hex nuts and bolts. However, this module was not implemented due to shipping and time constraints. Refer to section 3.1.5.2 for more information.

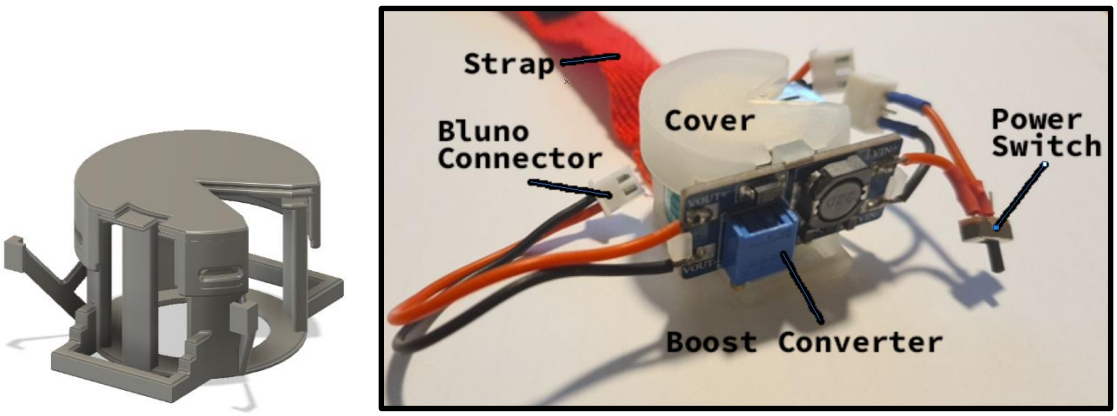


Figure 3.1.1-C
Batter Holder module CAD and Diagram

The Battery holder is strapped around the wrist. The battery can be removed and replaced by removing the cover. The Boost converter is held in place by a snap fit holder and can also be easily removed and replaced. The battery and boost converter are connected by a 3 pin JTAC connector to avoid confusion as to which connectors are for the Bluno and which are for the battery.

3.1.2 DFR0339 Bluno Beetle

As a cautionary opening statement, the correctness of some of the specifications detailed in this subsection regarding the Bluno Beetle is not guaranteed. The official datasheet regarding the Bluno Beetle is very sparse and does not contain much specific information that one would expect to find in a traditional datasheet, much of the information presented below will be inferred based on official datasheets of the Bluno's individual components based on the Bluno's schematic.

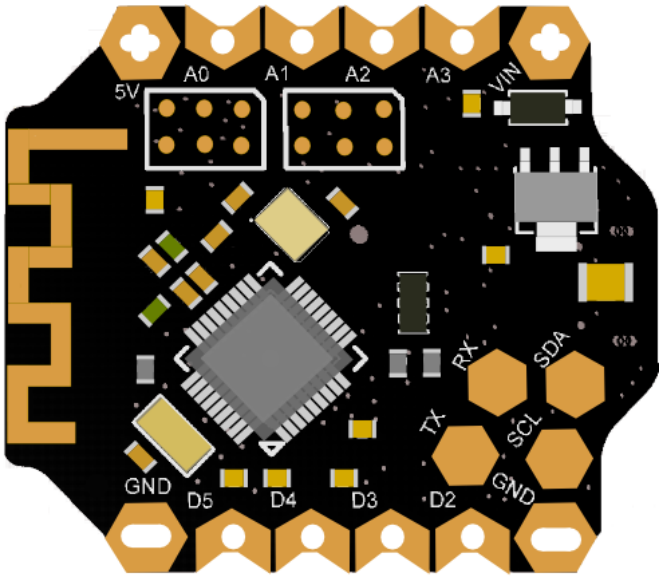


Figure 3.1.2-A
Bluno Beetle

The Bluno Beetle[1][2] is an ATMEGA328P[3] based board that is compatible with Arduino platform with an integrated CC2540[4] Soc that provides Bluetooth 4.0 and Bluetooth Low Energy (BLE) capability. The Bluno Beetle will be the heart of the wearable device, serving as the primary microcontroller for the wearable.

The Beetle will have three main responsibilities:

- Manage, Configure, and interrogate the sensor peripherals (MPU-6050, APDS-9960)
- Parse, filter, and pre-process sensor data in preparation for handoff to dance-move classification pipeline
- Communicate wirelessly over BLE with relay device, and ultimately Ultra96 to transmit and receive data and control signals.

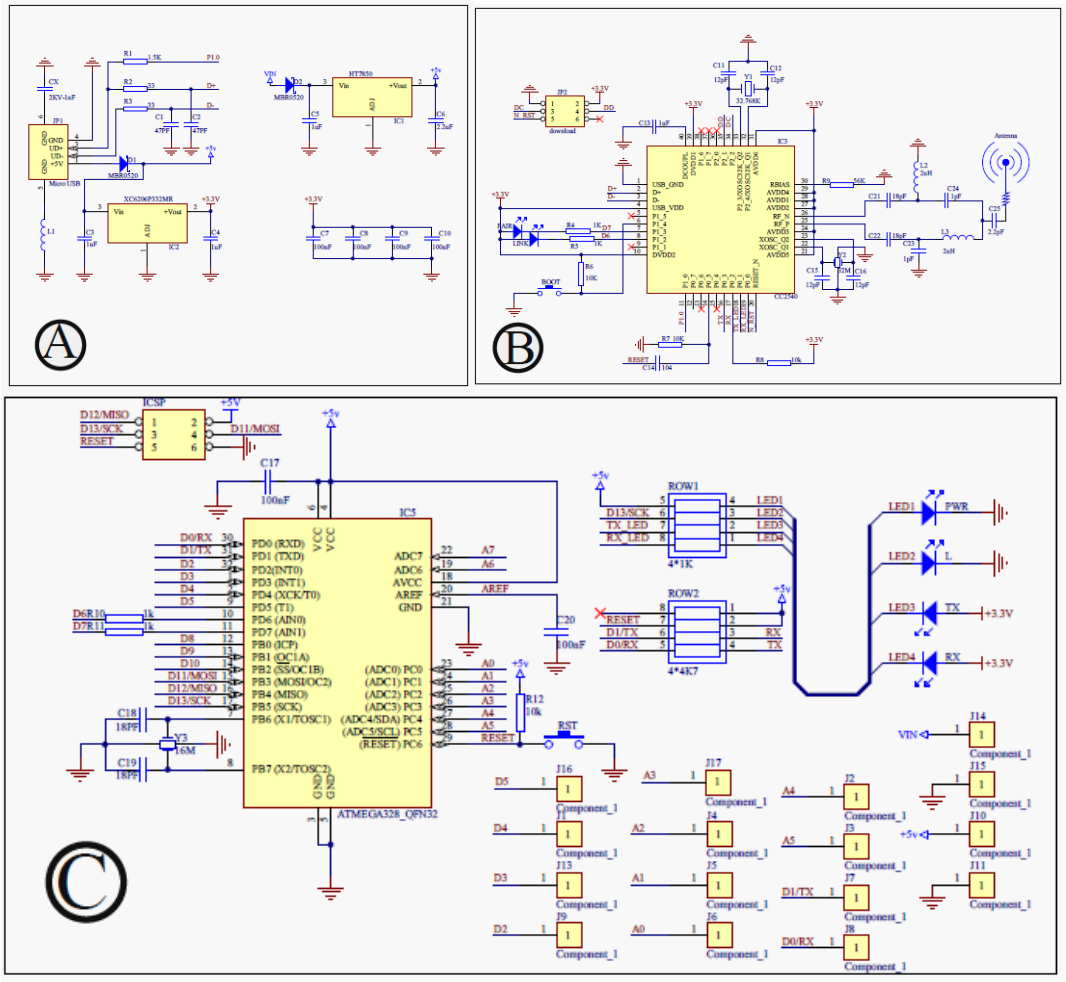


Figure 3.1.2-B
Bluno Beetle Schematic

12 data pins of the ATMEGA328P are broken out for use. This consists of PD0 to PD5 of PORT D and PC0 to PC5 of PORT C. Therefore, for GPIO functionality, we have access to 6 Digital Pins, 2 of which are PWM capable and attached to timer0(PD5) and timer2(PD3) respectively, and 6 Analog Pins.

In our wearable, communications relay device will be implemented via the CC2540 Bluetooth module through UART TX/RX internal connections within the Bluno package. It should be noted here that the internal connections to the CC2540 are also connected UART TX/RX breakout pins, which might present a problem for UART communication with sensor peripherals, should the need for such sensors arise. All communications with sensor peripherals will be implemented using the I2C pins SDA/SCL (PC4/PC5). Additionally, we have access to INT0(PD2) and INT1(PD3) external interrupts which will be used by interrupt generating peripherals to signal the ATMEGA328P, this will be further elaborated upon in subsections for individual sensor peripherals.

3.1.2.1 Bluetooth

The Bluno Bluetooth capabilities are provided by Texas Instruments CC2540 Soc. The CC2540 is a BLE wireless MCU with USB that will run the BLE protocol Stack for the Bluno. While a fully capable MCU, the CC2540 is flashed with firmware provided by Bluno manufacturer DFROBOT to act as a UART BLE bridge which allows for easy and convenient BLE communications through a UART Serial interface. Practically, this allows the ATMEGA328P to establish serial UART communications easily with other devices

though the UART BLE bridge implemented by CC2540. Additionally, the CC2540's behavior can be configured via AT commands over UART, which will be further elaborated upon in the *Section 4.1 Internal Communications*.

While using the DFROBOT firmware provides an easy and convenient abstraction for BLE based communication, using the CC2540 in this way limits the capabilities of our sensor to the configuration options and functionality provided by the DFROBOT firmware. While this firmware is sufficiently robust for this project, working directly with the CC2540 will allows to both optimize the Bluetooth communication but also will give us access to CC2540's more advanced features, such as its AES Security Coprocessor for full end-to-end encryption. While this would certainly be an optimal solution an interesting challenge, it is likely that it is too ambitious for the scope and timeframe of this project.

3.1.2.2 Electrical Characteristics

The ATMEGA328P can operate over a versatile range of voltages, from 2.7V to 5.5V. When operating at lower voltages however, the ATMEGA328P will have be run at a lower clock speed to remain stable. The CC2540 can operated between 2V and 3.9V. The Bluno beetle operates the CC2540 at 3.3V and is not 5V tolerant. For this project, we plan to operate the ATMEGA328P at 5V @ 16Mhz, however we will consider 3.3V @ 8Mhz operation to reduce circuit complexity should it become apparent that the reduced 8Mhz clock is more than enough for the Bluno's tasks.

The Bluno Vin Pin is connected to a HT7850[5] low dropout linear regulator to regulate a 5V Vcc for the ATMEGA328P. The HT7850 is tolerant to a maximum of 8V input voltage, which means that the Bluno Vin pin can handle up to 8V. A XC6206P332MR[6] low dropout linear regulator to further regulate a 3.3V Vcc from the HT7850's regulated output for the CC2540. Both these regulators have exceptionally low quiescent power consumption and high efficiency. We plan to provide pre-regulated 5V to the Bluno via its 5V pin, bypassing the HT7850. A full breakdown of the wearable power supply and regulation design and design considerations will be provided in Section 3.1.5.

3.1.3 GY-521 MPU-6050 IMU

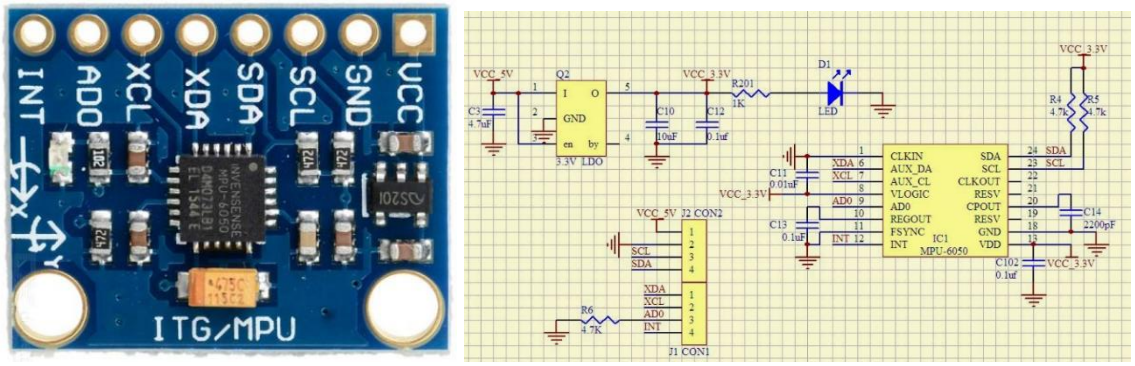


Fig 3.1.3-A
GY-521 MPU-6050 IMU and Schematic

The MPU-6050[7] SoC is a 6 Degree of Freedom (DoF) IMU that combines a 3-axis accelerometer and a 3-axis Gyroscope. The GY-521[8] breakout board integrates a low dropout linear regulator to regulate input voltage to 3.3V and breaks out the SDA/SCL, AUX_DA/AUX_CL, AD0 and INT pins for external connections. The GY-521 MPU-6050 will be the main source for our classification, measuring linear and rotational acceleration in three dimensions. Communications with the Bluno Beetle with me implemented via I2C using the SDA/SCL pins. The MPU-6050 also has an onboard Digital Motion Processor (DMP) (unused) that is capable running motion processing algorithms to process raw accelerometer/gyroscope data and notifying the parent controller of events by generating interrupts on the interrupt pin.

3.1.3.1 Peripheral Configuration and Use

Arduino Platform's inbuilt Wire Library [9] can be used to easily handle I2C communications between the Bluno and the MPU-6050. For convenience, jrowberg's i2cdevlib's MPU-6050 library[10] will be used to facilitate easy configuration and interrogation of the MPU-6050. Using i2cdevlib with the DMP requires the interrupt pin to be connected. The i2cdevlib library exposes functions and macros that allow us to easily configure the DMP and obtain the following useful data:

- A Quaternion representative of the MPU-6050's orientation.
- The Gravity Vector.
- The acceleration Vector.
- The acceleration Vector with gravity components removed.
- The acceleration Vector with gravity components removed in world coordinate space, defined coordinates relative to the MPU-6050's initial pose.

On top of configuring the DMP, the MPU-6050 also has many other different configuration registers that will provide different functionality to the MPU-6050. For the purposes of this project, we are particularly interested in Digital Low Pass Filter as well as the Sampling Rate and Sensor Resolution.

Final Implementation

Unfortunately, while the DMP would have greatly simplified the post-processing of the IMU data, after testing, we realized that the DMP processed sensor readings proved to be too unreliable for use with our system. Therefore, we made the design decision to bypass the DMP and directly access the sensor readings.

3.1.3.2 Digital Low Pass Filter

Raw accelerometer and gyroscope data contain a significant amount of noise. Since we are only interested in extracting data related to dance moves, we can safely filter out any high frequency signals via the MPU-6050 inbuilt Digital Low Pass Filter. The Digital Low Pass Filter (DLPF) can be configured by writing to the Register 26 or *CONFIG*.

CONFIG

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1A	26	-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		

Bit 2 –Bit 0 of *CONFIG* configures the DLPF according to the table below:

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

The exact frequency threshold that separates useful information from noise for our dance move detection is not known at the time of writing, and the pass band will have to be empirically determined iteratively over the development of our system. However, we can make an educated guess that, with the position of the MPU-6050 on the wrist, dance moves should mostly consist of low frequency sweeping motions(<10Hz). Therefore, we can use DLPF_CFG mode 6 as a starting point for our implementation. It should be noted that the digital low pass filter will introduce some delay between the actual sampling of acceleration/gyroscopic acceleration data and the availability of low pass filtered data to the Bluno.

This delay is acceptable because of the following reasons:

- The framerate of a typical recording device is 30 Frames per Second. This translates to a frame period of 33.33 milliseconds. Since the maximum delay of the DLPF is 19 milliseconds, it can only influence the perceived synchronization difference between multiple dancers in a 30fps video recording by at most one frame.
- More importantly, all MPU-6050 on our wearables should be configured with same DLPF configuration, which means that the delay induced by the DLPF should be consistent across multiple wearables, and therefore the synchronization time delta between each dancer should not be affected by the DLPF delay.
-

Final Implementation

Though empirical testing, we found that we are only really interested in very low frequency changes in the IMU readings. Therefore, to provide an increased signal-to-noise ratio, we have applied DLPF_CFG mode 6 as our Low Pass Filter Setting.

3.1.3.3 Sampling Rate and Sensor Resolution

Depending on requirements and design of the final dance move classifier, we may find that a lower sampling rate and resolution also provides sufficient discriminative features to classify dance move. Should this be the case, we can reduce change these parameters to reduce the minimum bandwidth required to transfer data from the Bluno to the relay device.

The sampling Rate can be configured by writing to Register 25 or *SMPRT_DIV*.

SMPRT_DIV

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
19	25	SMPLRT_DIV[7:0]							

The Sampling rate will be set with the following formula:

$$\text{Sample Rate} = \text{Gyroscope Output Rate} / (1 + \text{SMPLRT\_DIV})$$

The Gyroscope Output Rate will be either 8KHz or 1KHz depending on the configuration of the DLPF. For our final implementation, we have set the Sample Rate for the MPU-6050 to 1kHz , since we configure DLPF_CFG to mode 6. However, we will only be requesting samples from the sensor at 20Hz to facilitate smooth Bluetooth communications and streaming.

The sensor resolution of the accelerometer and gyroscope can be configured can be configured by writing to Register 27 *GYRO_CONFIG* and Register 28 *ACCEL_CONFIG*. These registers set the range scale for each sensor. Since the accelerometer and gyroscope of the MPU-6050 operate using 16-bit ADCs, each sensor reading is a 16-bit number. Therefore, by changing the range scale of each sensor, we can adjust the granularity of the sensor reading.

GYRO_CONFIG

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1B	27	XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]		-	-	-

ACCEL_CONFIG

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1C	28	XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]		-		

FS_SEL and AFS_SEL configures the resolution of the gyroscope and accelerometer according to their respective tables below:

AFS_SEL	Full Scale Range
0	± 2*g*
1	± 4*g*
2	± 8*g*
3	± 16*g*

FS_SEL	Full Scale Range
0	± 250 °/s
1	± 500 °/s
2	± 1000 °/s
3	± 2000 °/s

In this case, from empirical observations and experimentation with the dance moves, we can expect the acceleration and rotation rate to remain in the lower range of these scales. Therefore, it is likely that we will be able to keep both the AFS_SEL and FS_SEL in mode 0, which is the default configuration for the i2cdevlib library.

Final Implementation

Through testing, we also realise that our typical range of acceleration is well within ±2g. Similarly, we are not do see rotational rates that exceed 250°/s. Therefore, we set both AFS_SEL and FS_SEL to mode 0, which is the default setting.

The sample rate for the MPU-6050 itself is kept at the default of 1Khz by setting *SMPRT_DIV* to 0. To achieve our sampling rate of 20hz, we query the device only at 20Hz.

3.1.3.4 Electrical Characteristics

The MPU-6050 can operate over a narrow range of operating voltages (Vdd), from 2.375V to 3.46V. The digital logic pins are also not 5V tolerant, and digital logic must always be below operating voltage. The I2C pins SDA/SCL have minimum Logic-1/HIGH voltage of 0.7*Vdd, and the maximum Logic-0/LOW voltage of 0.3*Vdd.

Despite different operating voltages, there is no need for logic level conversion for I2C communication with the Bluno Beetle, since I2C operates on open-drain principle. Instead of driving pins HIGH or LOW, open drain pins can only pull the pin's voltage down to ground. This means that to express a logic HIGH, the I2C communication lines need to have pull up resistors to pull the voltage up to express a logic HIGH. These pullup resistors are included in the GY-521 board as 4.7K Ohm resistors R4 and R5(*On visual inspection these resistors appear to be 2.2K Ohm eac*h), which will pull the I2C bus to 3.3V. Since 3.3V exceeds the ATMEGA328P@5V's minimum voltage threshold for a logic HIGH, the Bluno and MPU-6050 will be able to communicate without any additional logic level shifters. This also applies to the interrupt line that will be used by the MPU-6050 to signal the ATMEGA328P.

The GY-521 board also has an unspecified 3.3V low dropout linear regulator connected to its Vcc breakout pin. We plan to supply pre-regulated 5V to the GY-521 MPU-6050 via the Vcc breakout pin. A full breakdown of the wearable power supply and regulation design and design considerations will be provided in Section 3.1.5.

3.1.4 GY-9960-LLC APDS-9960 (unused)

Another caveat: upon inspection, the schematics provided for the GY-9960 seem to differ slightly from the board. These differences will be pointed out when it becomes relevant. Secondly, it seems to be difficult to find the schematics for the GY-9960-LLC, instead, we will use the schematics for the GY-9960, the base board, and augment that with inspection of the board components. Even so, the schematics provided for the non-LLC GY-9960 still does not correspond 1-to-1 non-LLC GY-9960.

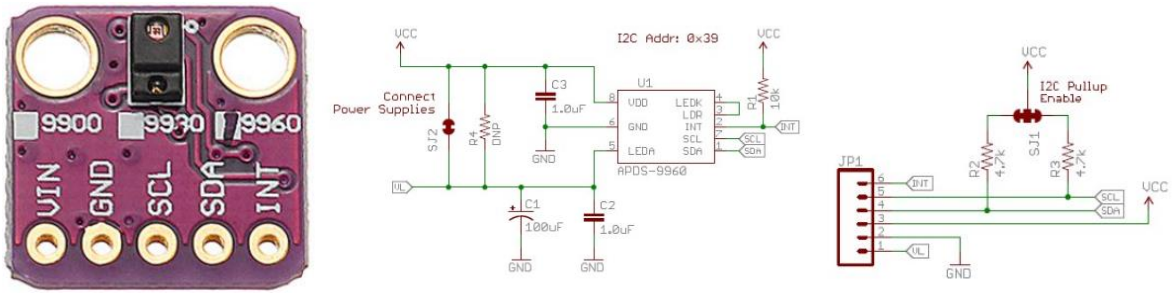


Fig 3.1.4-A
GY-9960-LLC APDS-9960 and Schematic

The APDS-9960[11] is a Digital Proximity, Ambient Light, RGB and Gesture Sensor. The GY-9960-LLC[12] is a breakout board that includes a 5V to 3V low Dropout regulator (*missing from the GY-9960 schematic*) to provide 3.3V to the APDS-9960. The GY-9960-LLC breaks out the I2C SDA/SCL and Interrupt pins for development. The APDS-9960 can generate interrupts on the interrupt pin to signal the parent controller on certain events. The APDS-9960 will be used mainly for proximity detection, to provide discriminatory information that indicates when dance moves bring the dancer's wrist into proximity with another body part.

Final Implementation

For our final implementation, we have excluded this sensor peripheral since the IMU already provides enough data for our system to achieve the desired project specifications of dance move classification and position detection. However, both our hardware and software architecture can accommodate this sensor without much modification, and this section has been left in the report as a potential improvement to our system, or as an expansion should the system be required to discriminate between much more similar actions/activities.

3.1.4.1 Peripheral Configuration and Use

Arduino Platform's inbuilt Wire Library[9] can be used to easily handle I2C communications between the Bluno and the APDS-9960. For convenience, sparkfun's APDS-9960_RGB_and_Gesture_Sensor library[13] will be used to facilitate easy configuration and interrogation of the APDS-9960, with slight modifications to make sure that it remains compatible with the library that is driving I2C communications for the MPU-6050, since both libraries use Wire for underlying I2C communication. This library exposes functions and macros that allow us to detect the proximity of the object, if any, in line of sight of the APDS-9960's LED aperture.

3.1.4.2 Proximity Engine

While the APDS-9960 has registers that configure a multitude of configuration options ambient light, RGB and gesture Sensing, for the purposes of this project, we are mainly only interested in the Proximity Engine of the APDS-9960.

The Proximity Engine allows us to configure the APDS-9960 to trigger a hardware interrupt when the proximity detected is beyond a certain threshold. The main registers of concern are the registers that configure the threshold values, and the register that configures the interrupt behavior. There are two thresholds, high and low, that can be set by writing to the Proximity Interrupt Threshold Registers (offset: 0x89/0x8B) PILT and PIHT respectively.

Field	Address	Bits	Description
PILT	0x89	7:0	This register provides the low interrupt threshold.
PIHT	0x8B	7:0	This register provides the high interrupt threshold.

The Proximity Engine can provide a pseudo low pass filtering effect to reduce the chance of false firing of the proximity interrupt due to momentary spikes in the LEDs returns. It does this by requiring a certain number of outside-threshold proximity sensor results consecutively before triggering the interrupt. The number of consecutive outside-threshold proximity sensor results required can be set by writing to PPERS, or bit-7 to bit-4 of the Persistence Register (offset: 0x8C).

Field	Bits	Description	
PPERS	7:4	Proximity Interrupt Persistence. Controls rate of proximity interrupt to the host processor.	
		FIELD VALUE	**INTERRUPT GENERATED WHEN…**
		0	Every proximity cycle
		1	Any proximity value outside of threshold range
		2	2 consecutive proximity values out of range
		3	3 consecutive proximity values out of range
		…	…
		15	15 consecutive proximity values out of range

At this time, prior to experimentation and iteration with the APDS-9960, we cannot determine the exact threshold and count required to achieve our objectives.

3.1.4.3 Electrical Characteristics

The APDS-9960 can operate over a narrow range of operating voltages (Vdd), from 2.4V to 3.6V. The digital logic pins are also not 5V tolerant, and digital logic must always be below operating voltage. The I2C pins SDA/SCL have minimum Logic-1/HIGH voltage of 1.26, and the maximum Logic-0/LOW voltage of 0.54V.

We will be able to connect the APDS-9960 to the same I2C bus used for communication between the Bluno and the MPU-6050. For similar reasons to that of the MPU-6050's I2C connections, there is no need for logic level shifting even though there is a difference in operating voltage between the Bluno's ATMEGA328P and the APDS-9960. It is worth noting that the schematics for the GY-9960 indicates that there also on-board 4.7K Ohm SMD pull-up resistors on the SDA/SCL pins as well as a jumper pad to disable these pull-ups. On visual inspection of the GY-9960-LLC, it seems that the resistance values for these pull-up resistors are actually 10K Ohm.

The GY-9960-LLC also includes a 3.3V linear regulator, which on visual inspection appears to be a XC6206P332MR, which is the same as the regulator used to provide 3.3V for the CC2540 in the Bluno. We will be providing the GY-9960-LLC's Vcc with regulated 5V.

3.1.5 Power System Design

Our wearable must operate for at least the time duration it takes for dancers to complete one full set of 8 dance moves and position changes. Taking reference from the dance move videos, each dance move lasts anywhere from 2 to 5 seconds. From preliminary testing, we know that moving between positions will around the same amount of time. Until the full implementation of the Machine Learning algorithm on the Ultra96, we will not be able determine the time needed for classification of dance moves. However, we shall use a VERY generous estimate of 2 minutes per combined dance move execution and classification. This means that our wearable must last at least 16 minutes of continuous operation. To provide a healthy margin for error, as well as to make testing and iteration more convenient we shall set a soft target of 20 minutes of continuous operation.

Final Implementation

Our final implementation averages about 13 to 15 seconds per dance move and classification cycle. However, the evaluation is on 24 dance moves, not 8. Therefore, by this new specification, our wearable should last for at the very minimum 15 * 24 seconds, which approximates to 6 minutes of constant operation. Our overly cautions original power budget estimate of 20 minutes provides at least three times as much battery life, well within this new requirement. Additionally, the project evaluation requires the wearable to undergo two such evaluations, with the potential for these evaluations to be arrange consecutively. Since our wearable can last for about 2 hours of consecutive operation, its power budget is much more than required for the project specifications and provides a healthy margin for error.

3.1.5.1 Estimating Power Consumption

Calculating the accurate power consumption of in the context so semi-reliable datasheets is tenuous at best, therefore the numbers presented in this section can only serve as a very rough estimate of the power requirements of our wearable. To compensate for this, our design will include very generous margins for error to make sure that we exceed our minimum runtime requirement. We estimate the typical power consumption of each device based on the datasheet. We make the following assumptions and concessions:

- We assume that all components are active not in any low power or power saving mode.
- We disregard the micro Ampere level current consumptions of various small chips and components
- We disregard current consumption of the I2C bus
- We will account for disregarded power consumption by including a large margin for error in final battery choice
- The current draw for APDS-9960 is highly dependent configuration settings, due to fact that it is an active sensor. The current estimate is a ballpark figure for periodic driving of its sensor LED.
- The linear regulator efficiency is modeled as Vin/Vout[14]

The calculated total power consumption is approximately 0.24W. The following table presents the power consumption of each component:

IC	Vcc (V)	I (mA)	I (A)	Power Needed (W)	Regulator Efficiency	Regulated Power Needed (W)
ATMEGA328P	5	9.2	0.0092	0.046	bypassed	0.046
CC2250	3.3	24	0.024	0.0792	0.66	0.12
MPU-6050	3.3	3.9	0.0039	0.01287	0.66	0.0195
APDS-9960	3.3	10	0.01	0.033	0.66	0.05

Final Implementation

While we do not use the APDS-9960 in our final system, we purchased the batteries and voltage regulators early in the implementation pipeline. Therefore, our power estimation is and subsequent power budget includes the APDS-9960, while our final implementation does not include this sensor peripheral. Our actual final power estimate is approximately 0.19W instead of 0.24W.

3.1.5.2 Battery Selection and Battery Voltage Regulation

Since our wearable must be worn on an extended limb that is expected to undergo significant movement, our solution to provide power to the components of our wearable must be small and light enough to pose no significant risk of blunt force injury should our wearable fail to adhere to our dancers.

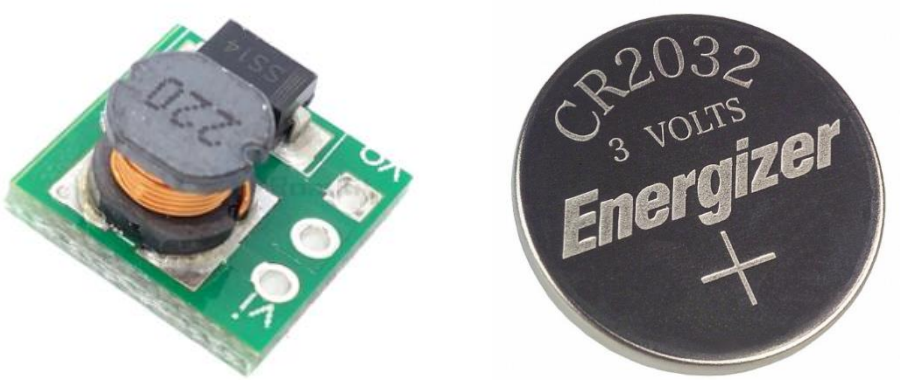


Fig 3.1.5.2-A
DC-DC Boost converter and CR2032

The Energizer CR2032[15] is a 3V 235mAh coin cell battery with small 2mm by 2mm form factor and suitable for our purposes. To provide regulated 5V to all our components, we will be using an OEM 0.9-3V to 5V DC-DC Boost Converter[16]. This DC-DC Converter has an efficiency of 85%. Therefore, the battery must be able to supply at least 2.82W for 20 minutes of continuous operation. At 3V, 2.82W translates to about 92mA. Rated at 235mAh until the battery voltage drops to 2V, the CR2032 can last for approximately at least 2.5h, which a healthy margin for error. Since our DC-DC converter is rated for a minimum input voltage of 0.9V, the CR2032 should be able to continue providing power even after its nominal cut-off voltage of 2V. To make sure that the voltage supplied to our sensor are clean,

we will include a pair of 220 microfarad capacitors between Vin and ground, and Vout and Ground for the DC-DC converter.

Final Implementation

Over the course of development, we realised two things:

- While a single use battery for a production version of this system would be acceptable. A version of the system with a rechargeable battery was needed for testing and iteration.
- Purchasing components online with uncertain shipping times is hazardous for projects on a tight timeline since none of the 5V versions of the DC-DC Boost converters purchased online arrived in time for the final implementation. Fortunately, we ordered multiple variations of different boost converters as a contingency plan.

Since our system was designed from the ground up to be a modular and conducive for quick prototyping. Therefore, we were able to quickly switch out the missing components for other similar components with little to no modification to the rest of our system.

Fig 3.1.5.2-B

MT3608 Boost converter, NIMH rechargeable Battery, Combined Battery Module

For the boost regulation, we used a MT3608[22] based variable step-up regulator to generate the 5V input for our wearable. While larger, this regulator boasts an efficiency of up to 98%, which outperforms our original Boost converter. However, it has a cut-off voltage of only 2V, which is lower than our original Boost converter. Since we provide the regulator with at least 3V, this restriction does not greatly affect us.

For our battery, we opted for a VARTA 3.6V 240mAh NiMH battery [23], providing more than enough capacity to satisfy our power requirements. Unlike Lithium polymer or Lithium-Ion batteries, NiMH batteries have much lower tendency to explode on overcharge or physical deformation, which makes it a much safer choice to use on our wearable.

From empirical testing, and general use of our wearable though the development process, we have found that at full charge, this regulator and battery combination lasts anywhere from 2 to 2.5 hours of continuous operation, well above our minimum requirements for the project specifications.

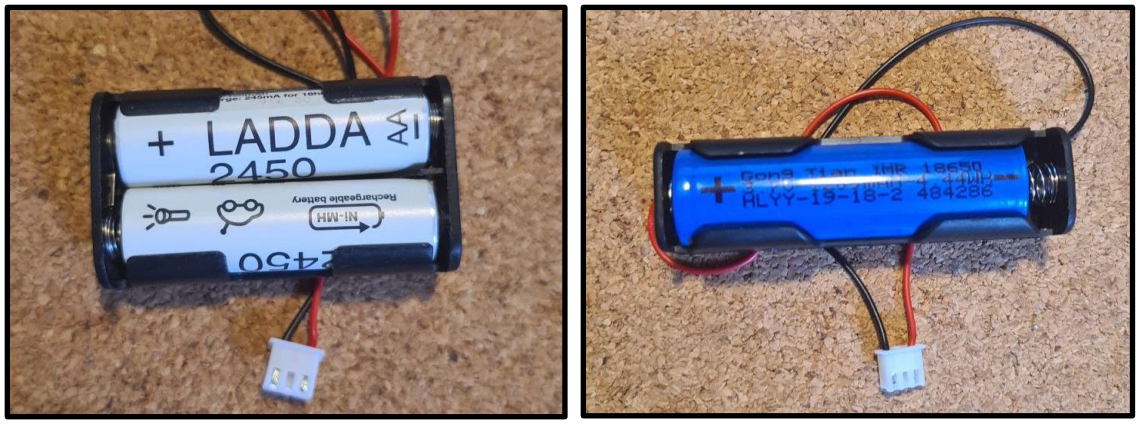


Fig 3.1.5.2-C
AA Backup Battery Module and 18650 Backup Battery Module

As an added advantage to the modularity and robustness of our system, we have also prepared 2 sets backup battery packs that can contain either any pair AA sized batteries or standard 18650 sized batteries. We mainly used a pair of 1.2V 2450mAh LADDA rechargeable NiMH AA batteries in series from as our backup batteries, as well as batteries during the long sessions of data collection. Both versions of these battery packs have been implemented with compatible JTAG connectors to our MT3608 Boost converters. The MT3608 provides consistent voltage for our wearable as along as our battery voltage is above 2V, providing a highly modular and power supply system.

Unimplemented Content

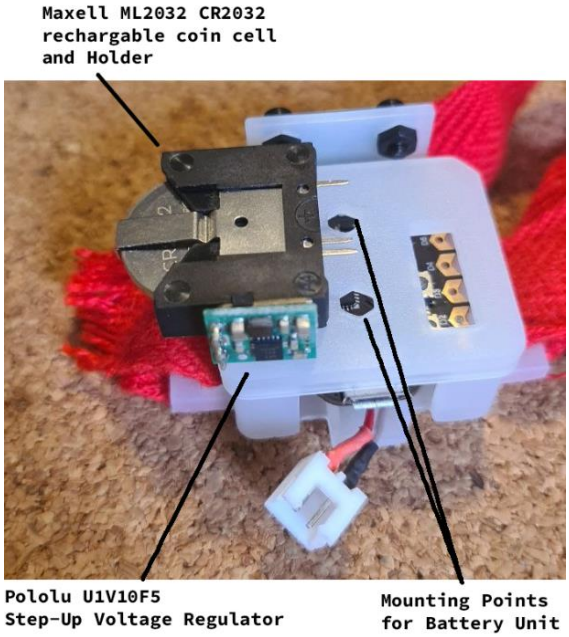


Fig 3.1.5.2-D
Design for final wearable

Due to Shipping constraints, we were unable to implement the full extent of our planned power supply system. The combined Battery Unit in Fig 3.1.5.2-B was only supposed to act as a secondary power supply system for longer dance sessions. Our original battery solution for normal dance sessions ideally would have consisted of a Maxwell ML2032[24] CR2032 65mAh 3V battery as well as a TI TPS6120[25] based Pololu 5V U1V10F5.

Step-Up Voltage Regulator, which would be mounted directly unto the main wearable via M2.5 Hex screws.

Although it has much smaller capacity that our current implementation, it would have provided more than enough power to last at least one evaluation of 24 moves and would have allowed us to have a much sleeker single unit, instead of separate battery and sensor units. All components arrived before the date of the final evaluation, however, they arrived too late for this design solution to be implemented and thoroughly tested before the evaluation.

3.1.6 Preprocessing and Feature extraction on Main Wearable

There a total of 3 broad streams of data generated by our wearable:

- Acceleration Data
- Rotational Data
- Proximity Data

As with most sensor applications, before any raw noisy sensor information can be used, they must first be processed to increase the signal-to-noise ratio as well as to convert the information into a format suitable for analysis. In our case, most of the noise reduction and thresholding is done by the peripheral sensors. The MPU-6050 handles the low-pass filtering of the accelerometer and gyroscope data. The proximity sensor is also configured with to only trigger past a certain threshold. Furthermore, the library used to drive MPU-6050 is able to provide relatively stable acceleration vectors in world coordinate space. Since most of the signal processing can be done outside the Bluno, the ATMEGA328P is free to attempt to make sense of the data through preliminary feature extraction.

Final Implementation

Since we have excluded the APDS-9960 from our design, we will no longer be obtaining Proximity Data from our wearable. Secondly, as earlier mentioned, we discovered that the processed data from the DMP was unreliable, therefore we will not be using the processed world coordinates produced by the DMP.

Instead, we will be streaming raw acceleration and rotation data from each Bluno to our ultra96 at a rate of 20Hz. Since we are streaming the data anyway, we decided to implement the feature extraction on the comparably less resource constrained Ultra96 instead of directly on the Bluno. This has several advantages:

- We can take advantage of more powerful signal processing techniques and at lower latency by taking advantage of the greater processing speed and memory of the Ultra96.
- We can buffer more samples on the Ultra96 therefore having the potential larger window sizes.
- We can take advantages of much more readable and user-friendly programming languages like python to implement these aforenoted signal processing techniques.

Therefore, the wearable's main responsibility now is to make sure that data has a high signal-to-noise ratio and is sent to the Ultra96 in stable and reliable manner. Since signal processing can also be done on the Ultra96, the pre-processing on the Bluno is minimized to give priority to the communications for a stable 20Hz. Surprisingly, on top of the Digital Low Pass filter provided by the MPU-6050, very little else had to be done to improve the signal-to-noise ratio to a point where our machine learning model can successfully discriminate between each dance move. Therefore, we only implemented an additional simple moving average filter on the Ultra96 to further reduce noise caused by small movement or fidgeting of each dancer. This moving average filter is defined by:

$$Filtered\_value = Sensor\_reading * \alpha + old\_Filtered\_value * (\alpha - 1)$$

This moving/weighted average takes inspiration from a technique to estimate Round Trip Times for a network connection, and functions to reduce the impact of small and isolated high intensity movements. Our empirically chosen alpha is 0.4.

Additionally, the 16-bit raw reading of each IMU axis trivially quantized to 8-bits by taking the leftmost 8-bits of each reading. Then rescaled to ± 2g for the acceleration data, and ±250°/s for the gyroscope data.

3.1.6.1 Feature extraction

The information required to discriminate between however, is contained within a subset of each data stream. In other words, not all the information gathered by the wearable is useful in discriminating between dance moves. To reduce the minimum bandwidth requirement on the BLE communications, these data streams can be converted into more compact forms that are retain the information required to discriminate between different dance moves but discard any extraneous information. This is the essence of feature extraction. Our goal in feature engineering here is to look out for obvious patterns in the data that can easily discriminate between dance moves and extract that pattern as a feature.

In our case, we propose to implement a rudimentary localization system. Since acceleration, speed and distance travelled are all mathematically linked, one can estimate the distance travelled by the sensor based on the acceleration data and the time taken for acceleration. Therefore, it is possible to estimate the current location of the sensor relative to its starting location. However accurate localization based solely on accelerometer and gyroscope data is a very difficult problem; small errors in the either the data or the distance estimation add-up very quickly. However, for short periods of time, this system can be fairly accurate in determining the wearables location in relation to its starting position, which we can constrain to be on the wrist while the arm is in a resting position by the side of the body. Therefore, our localization system would be able to tell us approximately what position the dancer's wrist/arm is in over time. This position can be categorized into a set of pre-determined common arm positions or regions. Therefore, at any point in time, this system would be able to provide the region that the dancers wrist is in. This can serve as an additional potential discriminatory feature for our machine learning model. This system also has the added benefit of being able to determine if the dancer has moved to position of another dancer, and therefore aid in determining the order of the dancers.

Final Implementation

Since the data produced by the DMP proved to be unreliable, implementation of a dead reckoning based features based solely on accelerometer and gyroscope IMU data proved to be too difficult to implement given the time frame of our project, and much of a risky time investment. Therefore, we decided to pursue a much safer and traditional option of extracting statistical features over windows of samples. While the details of this windowing and feature extraction will be explained in detail in Section 5.1, we will provide a brief overview here.

Statistical Features extracted per IMU axis over a window of N samples:

- Mean
- Standard Deviation
- Variance
- Kurtosis
- Skew
- Peak-to-Peak

Statistical Features extracted from combined axis or state-machine information over a window of N samples:

- Mean total acceleration / Mean magnitude of acceleration vector.

- Percentage of samples with low energy (where total acceleration < threshold)
- Percentage of samples where Z-axis is acceleration is approximately 1g (indicates hand slack to the sides of the body)

3.1.6.2 Characterizing the Start of a Move

Solving the problem of determining when a dance move has started is like that of feature engineering; we are looking for patterns in the sensor reading that can characterize the start of a move. To construct our algorithm, we make the following assumptions:

- There is some time between dance moves where dancers do not move much.
- The magnitude of the world space acceleration vector is reliably and stably bounded beneath a certain threshold when the dancer is not dancing. When the dancer is dancing, it exceeds the threshold.
- It takes a minimum amount of time to perform a dance move, therefore dance moves starts are only valid if they last long enough.
- Dance moves cannot start if the dancer is currently performing a dance move.

With these assumptions, we propose a simple set of conditions to detect the start of the dance move:

- The magnitude of the world space acceleration vector exceeds an empirically determined threshold.
- There has been a minimum time interval since the end of the previous dance move.
- The dancer is not currently dancing.

Additionally, we propose a condition to detect the end of a valid dance move:

- The moving average of the magnitude of the world space acceleration vector is below the aforenoted threshold.

Although we do attempt to determine both the start and end of a dance move, the detection of the end of the dance move just serves to prevent subsequent start of dance move detection events form occurring until we can be reasonably certain that the dancer has stopped dancing.

Final Implementation

The basic principle of our start of move detection algorithm has not changed much from our original idea in the initial design report. However, because we are no longer using world space coordinates provided by the MPU-6050, we use data from each individual axis of the IMU instead. The revised assumptions are as follows:

- Dancer will pause for a moment after each dance move before either beginning the next dance move or moving to another position.
- The magnitude of the combined acceleration vector is reliably and stably bounded beneath a certain threshold when the dancer is not dancing. When the dancer is dancing, it exceeds the threshold.
- It takes a minimum amount of time to perform a dance move, therefore dance moves starts are only valid if they last long enough.
- Dance moves cannot start if the dancer is currently performing a dance move.
- A dance move can only start if the Dancer starts the dancing from a neutral/idle position, where both hands are held loosely to the side of the body.

With these updated assumptions, these conditions characterise the start of the dance move:

- The magnitude of the combined acceleration vector exceeds an empirically determined threshold after accounting for gravity.
- The dancer is currently or was very recently in the neutral/idle position.
- There has been a minimum time interval since the end of the previous dance move.
- The dancer is not currently dancing.
- The dance move is considered invalid if the number of samples that have been collected while the dancer is dancing is below a threshold when the end of a valid dance move is detected.

We also update the condition to detect the end of a valid dance move:

- The magnitude of the combined acceleration vector is consistently below a threshold for several samples, signifying the ceasing of vigorous motion.
- The dancer is remains in the neutral/idle position for several samples.

To detect if the dancer is in the neutral position, we check the Z axis of the acceleration data. Since we know the orientation in relation to gravity that the sensor should be in when the dancer is in the neutral/idle position, we also know the which axis the gravity vector would be aligned with when the dancer is in the neutral/idle position. In our case acceleration due to gravity should be mostly aligned with the negative Z axis of the IMU. Therefore, a dancer in in neutral position when the negative Z axis is approximately to 1g of acceleration (gravity).

For more information on the specific numbers for the thresholds and number of samples used in our algorithm, please refer to *Test/relayMain.py*

3.1.6.3 Calculating Dancer Positions

Final Implementation

This module was not present in the initial design report. However, the initial design report did refer to the using the world coordinates provided by the DMP for dancer position calculation. Since the DMP proved unreliable, we made use of an inference-based system instead.

Since we have three dancers, it is possible infer and calculate changes in dancer position if we have the following information:

- The original positions of each dancer relative to one another.
- The direction which each dancer turned to walk to the next position.

There are only six valid ways in which the three dancers can turn to produce a legal and valid new position. We therefore pre-compute the results of these movements to produce a lookup table following lookup table:

Turn Directions	**New position indices**
('left', 'left', 'right')	(2, 0, 1)
('left', 'right', 'right')	(1, 2, 0)
('left', 'still', 'right')	(2, 1, 0)
('left', 'right', 'still')	(1, 0, 2)
('still', 'left', 'right')	(0, 2, 1)
('still', 'still', 'still')	(0, 1 ,2)

In the table *Turn Direction* is a three-element vector of turn directions made by each dancer. It is assumed that these directions are arranged this order:

(*Dancer in leftmost position,*
Dancer in middle position,
Dancer in rightmost position)

whereby leftmost and rightmost are from the perspective of an onlooker standing in front of the dancers and facing the dancers. Aside from this turn vector, we generate a corresponding vector that corresponds to each dancer's ID in the same order as the Turn Direction vector. We call this vector the position vector.

New position indices is another three-element vector as follows:

(*Index of Dancer in leftmost position,*
Index of Dancer in middle position,
Index of Dancer in rightmost position)

The index refers to the index of the dancer from the previous position vector. Therefore, by selecting the dancers from the previous position vector using New position indices, we can generate a new position vector with:

(*Dancer ID in leftmost position,*
Dancer ID in middle position,
Dancer ID in rightmost position)

In the case where the Turn Direction vector generated by the three dancers do not match any vector within the table, we consider the dancers not to have moved.

To detect the event where a dancer turns, we will look at the gyroscopic rotation rate of the IMU. We make the following assumptions regarding a dancer's turn:

- Dancer is not dancing.
- Dancer keeps his hands close to the neutral position while turning.
- Dancer makes a natural right left turn at a leisurely speed.

Therefore, the conditions to detect a dancer's turn are as follows:

- The dancer is not currently dancing.
- The gyroscopic rotation rate of the Z rotational axis exceeds a positive or negative threshold for a small number of samples.
- No turns have been detected for this dancer recently.
- The gyroscopic rotation rate does not exceed both the positive and negative thresholds within a small number of samples. This condition prevents vigorous shaking from triggering a dance move.

Of all the components of our system, the timing, samples, and thresholds for this turn detection is the most sensitive, with highest probability of failing for dancers who move too quickly or too slowly. Since there is no reference for this sort of position change movement, we can only try our best to calibrate this system based what feels natural to as many people as possible.

3.1.7 MyoWare Muscle Sensor Wearable for Muscle Fatigue

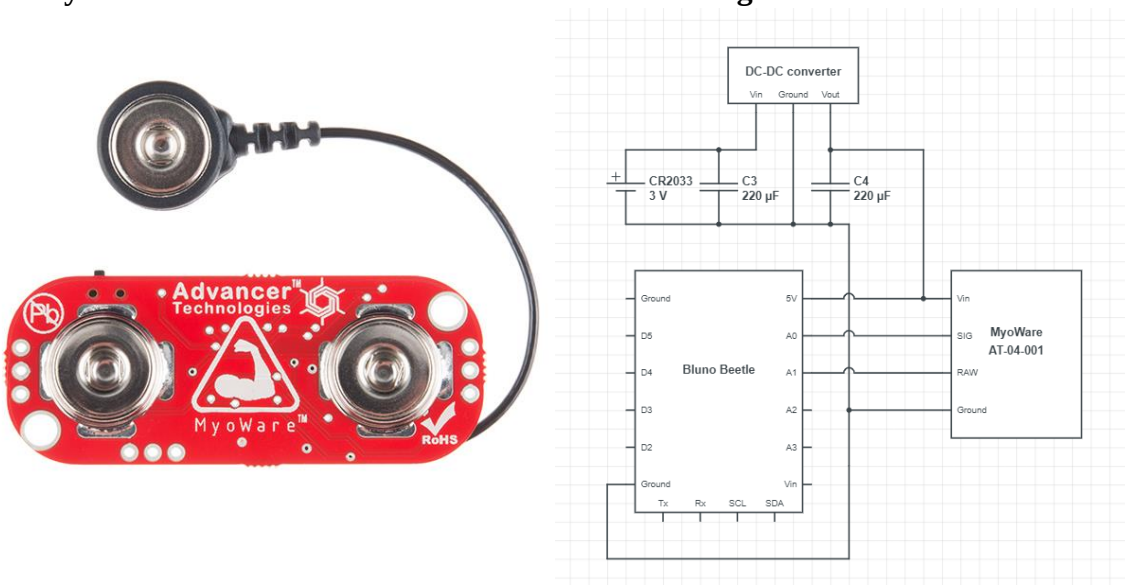


Fig 3.1.7-A
AT-04-001 *and Wearable Schematic*

The MyoWare Muscle Sensor AT-04-001[17] performs electromyography (EMG) to sense muscle activations via electric potential. This sensor will not be used in any way in the detection of dance moves. Instead, it will be responsible for providing data of analytics on a physical/health related characteristic of a dancer. To that end, we will be attempting to use the EMG signal to roughly approximate muscle fatigue of the dancer. Since the AT-04-001 is not related to the dance detection, and we only have one such sensor, we propose to provide it with its own dedicated Bluno to form an auxiliary wearable without the MPU-6050 or the APDS-9960. The AT-04-001 can operate over a wide voltage range, from 3.1V to 6.3V. We will be supplying the AT-04-001 regulated 5V, like that of the main wearable. This Muscle Sensor Wearable can make use of the same BLE communications protocol as our baseline wearable.

3.1.7.1 Peripheral Configuration and Use

The Muscle Sensor is able to provide both raw and processed signals. This processed signal SIG is the envelope signal of the rectified raw EMG signal. The GAIN of the processed signal can be adjusted via the gain potentiometer, located on the side of the board. We will be using the Analog-to-Digital Converter (ADC) on the Bluno's ATMEGA328P for analog signal quantization. The ATMEGA328P's ADC is capable of up to 15K Hz sampling rate at 10-bit quantization rate. This sampling rate can be improved is less quantization bits are needed.

3.1.7.2 Signal Processing and Feature extraction

In isolation, without any processing, the RAW and SIG EMG signals tell us little regarding the state of muscle fatigue in a dancer. Therefore, these EMG signals must first be processed before it can be used to characterize muscle fatigue. The following table is an overview of the signal processing pipeline for EMG signals:

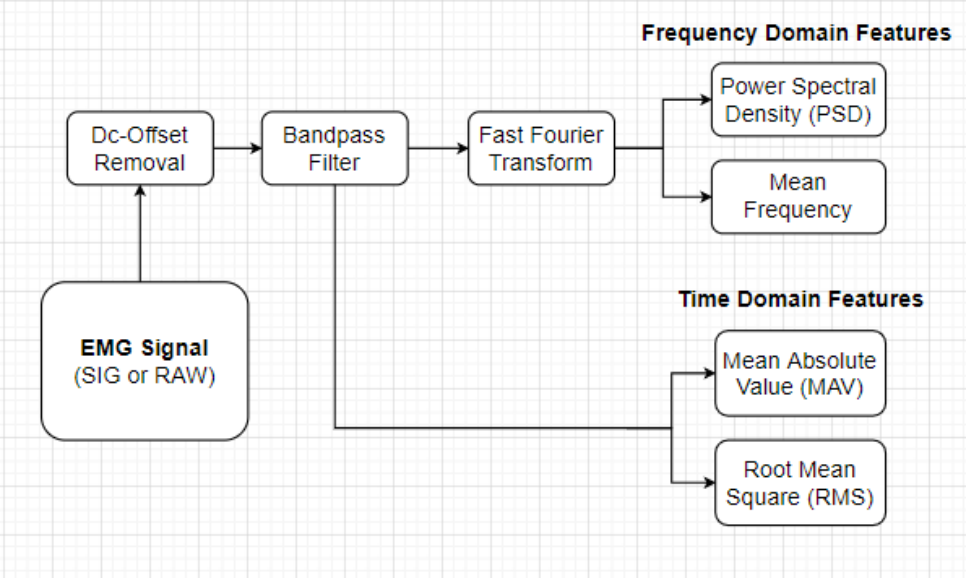


Fig 3.1.7.2 -A
EMG Signal processing pipeline

Generally, it seems that EMG signals bandwidth is 0-500Hz with the dominant frequencies between 50-150 Hz [18]. Since, our sampling rate of 15K is most definitely above the Nyquist rate of 1KHz, therefore, we should not expect to see any aliasing if we sample above 1KHz.

As is common with most signal processing pipelines, the first stages of any signal processing pipeline are the DC-offset removal. However, in this case, the pre-processed SIG signal is already rectified, therefore it is likely that SIG will not require this stage of the pipeline. The RAW signal however is centered about Vdd/2, half of the Muscle Sensor's operating voltage, which should be approximately 2.5V or 1024/2 = 518 ADC quantization units. This DC-offset would have to be removed.

These EMG signals are almost definitely guaranteed to be noisy, especially the RAW signal. Since we know the bandwidth of the typical EMG signals, we apply filters to filter out frequencies that we know for sure are noise. In this case we can apply a band-pass filter with a passband of 20 – 500 Hz. The 500Hz cutoff will remove high frequency noise, frequently generated from CPU or ADC hardware, while the 20Hz cutoff will remove any low frequency noise, including any drifts or shifts in the DC-offset over time. The high pass filter component of this bandpass filter might not be required on the pre-processed SIG signal, since the signal envelope essentially removes much of the high frequency information. However, without documentation, effect of the MyoWare Muscle Sensor's envelope detector pre-processing step can only be determined empirically. Moving forward, until these tests can be done, we will assume that the SIG signal does not attenuate frequencies within our bandwidth of interest of 0-500Hz (*or at least 50-150Hz for the dominant frequencies*). These filters can be either implemented using libFilter by MartinBloedorn [19] or with RC circuits in hardware.

Generally, there are two information domains contained within an analog signal: Time-domain and Frequency Domain. In our case, muscle fatigue can be characterized by and is associated with Power Spectral Density (PSD) and the Mean Frequency of the signal [20], both of which are frequency domain features. The Power Spectral Density (PSD) is the amount of power per frequency interval. An estimation for the PSD can be obtained from a

Fast Fourier transform over a sampling window by squaring the magnitude of each frequency bin. The power of each at frequency should be decreasing as the muscles get more fatigued [20]. The Mean Frequency is average of the frequencies of all the bins in the FFT window weighted by the power at each frequency bin. The Mean frequency should decrease as muscle fatigue increases [20]. Aside from these frequency domain features, we can also explore analysis of some time-domain features, such as the mean absolute value (MAV) or root mean square (RMS) for each window. FFT on Bluno can be implemented using ArduinoFFT[21], from which the frequency domain features can be derived.

Final Implementation

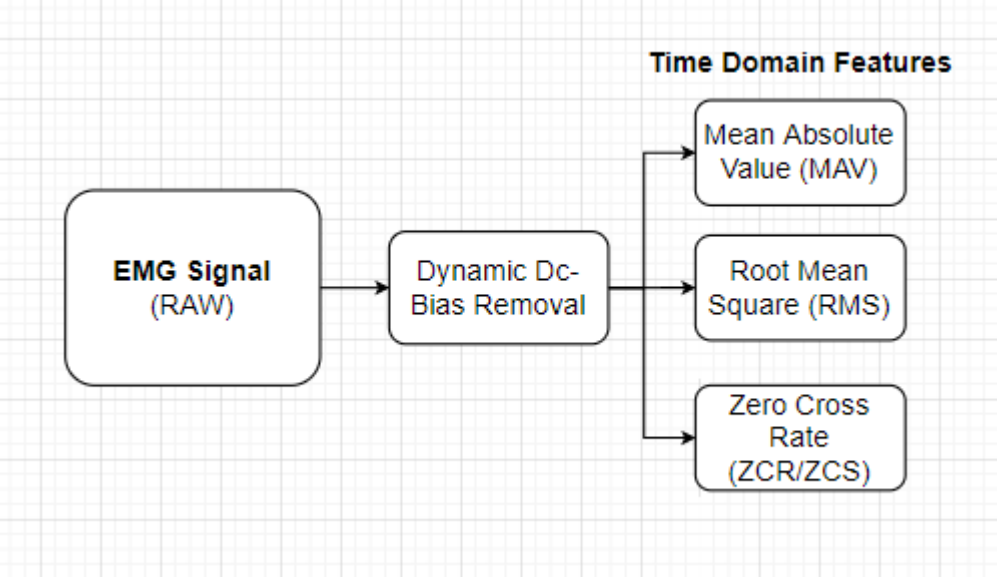


Fig 3.1.7.2 -B
Revised EMG Signal processing pipeline

In our final implementation, the Bluno samples RAW EMG signals the MyoWare at 500Hz. These samples are inserted into size 100 circular buffer. Before insertion, each sample is subject to dynamic DC-bias removal. During development, we found that the DC-bias drifts over time as the MyoWare sensor shifts and moves slightly. To account for this, we keep track of the moving average of the Raw EMG sensor data, and we subtract this local moving average from each data point before adding it to the buffer. This removes the local DC-bias, and acts like a pseudo High-pass filter.

The Bluno attempts to calculate the following features for the samples present within the buffer at a rate of 10Hz:

- Mean Absolute Value (MAV)
- Root Mean Square (RMS)
- Zero Cross Rate (ZCR)

Mean Absolute Value is calculated by taking the sum of absolute values of all the EMG samples in the circular buffer divided by the size of the buffer (100) given by the following equation:

$$MAV = (\sum_{k=0}^{n} abs(\text{Buffer}_k))/n$$

The Root Mean Square is calculated by taking the sum of the square of all the EMG samples in the circular buffer divided by the size of the buffer (100) given by the following. This result is then rooted to produce the RMS value. This is given by the following equation:

$$RMS = sqrt(\frac{\sum_{k=0}^{n}(\text{Buffer}_k)^2}{n})$$

The Zero Cross Rate is calculated by counting the number of times the EMG signal cross the x-axis. We iterate through the buffer and count the number of times the sign of the EMG signal flips from positive to negative and vice-versa

The implementation of ArduinoFFT on the Bluno proved to slow to implement at our desired sampling rates and interfered with the stability of our streaming 20Hz IMU DataStream. Therefore, we decided to stick with time-domain features for Myoware EMG data.

Section 3.2: Hardware FPGA - *Written by Ryan Lim*

This section will describe the design and use of the FPGA, which will be used to accelerate the processing of data through a neural network, so to provide timely dance move predictions.

3.2.1 Ultra96 Synthesis and Simulation Setup

Initially, we were exploring the use of a convolutional neural network (CNN) for the classification of real time data because of its accuracy and its ability to extract features. In order to translate the software model into hardware circuits, there was FINN, a framework developed by Xilinx Research Labs, to generate the neural network. FINN itself is a complier that can create accelerators to classify data and comes with a high-level synthesis (HLS library) written in C++ to implement the layers within the neural network [1].

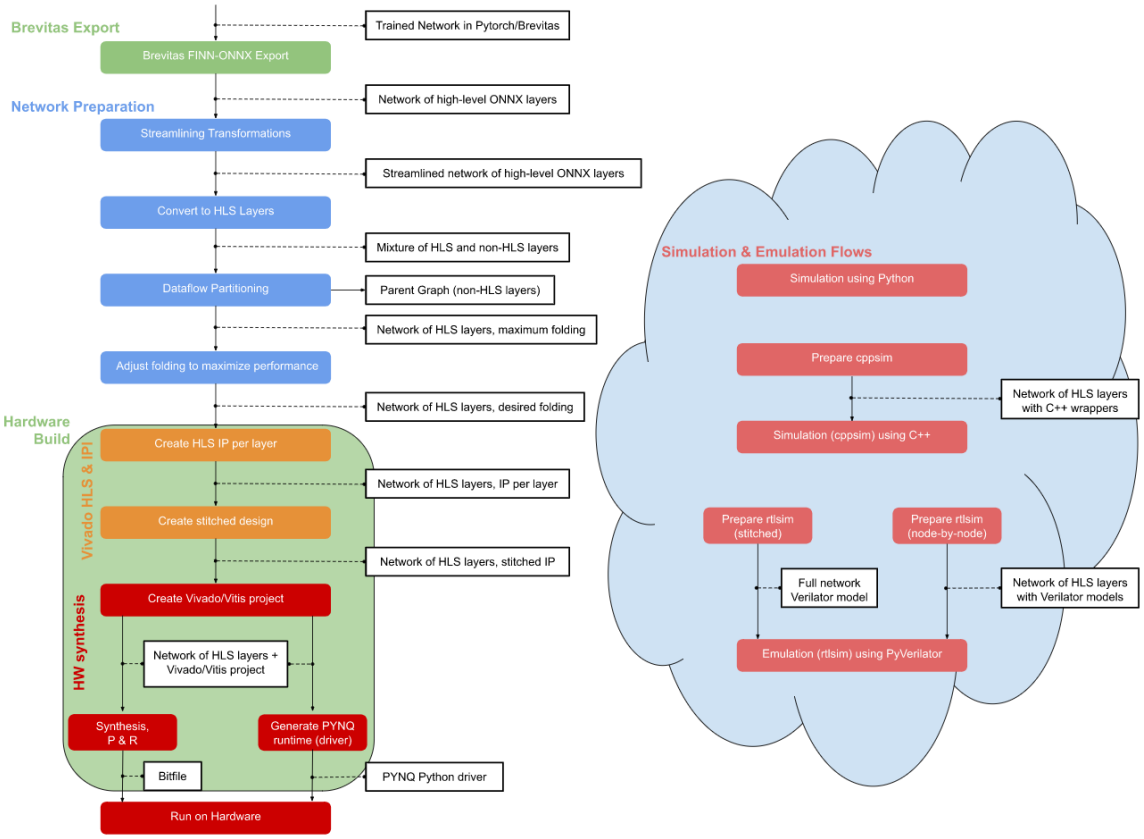


Figure 3.2.1-A
FINN end-to-end workflow

Figure 3.2.1-A lays out the workflow in FINN [2], with the green step the start (using the Brevitas export), followed by the network preparation steps in blue, the generation of IPs using Vivado in orange, and lastly the synthesis steps in red. Using the whole workflow from end-to-end is not required. The output bitfile and driver files from this workflow can then be executed on the Ultra96.

However, this integrated workflow was deemed not suitable for our needs. We needed an implementation that was flexible and customizable, with minimal overheads. Additionally, feedback from the teaching team suggested that this experimental framework would be troublesome to work with due to stability issues and bugs.

This meant that to synthesize the neural network onto the FPGA, we would have to translate the architecture of the model into C++ code ourselves, and use the Vivado HLS program to

export an IP, which could then be synthesized and generate bitstream [3]. Figure 3.2.1-B shows a block design in Vivado with an IP connected to the processor system via an interconnect which handles streams of data. There is also an IP to handle the resetting of the system.

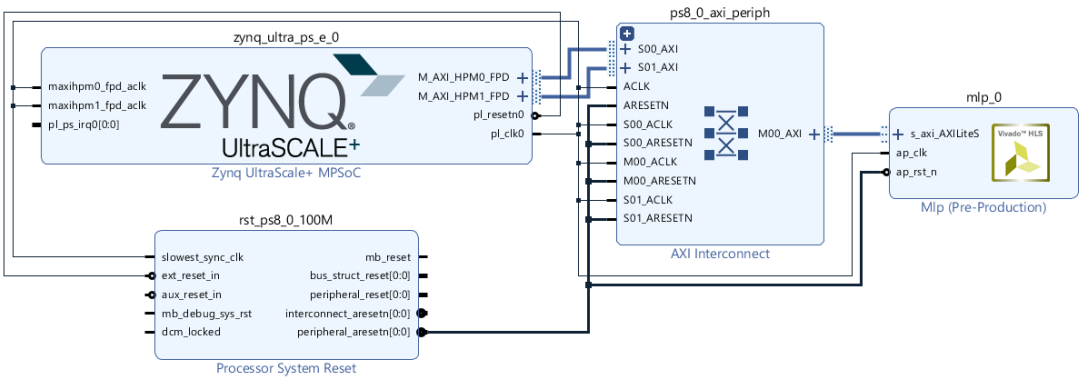


Figure 3.2.1-B
Imported IP in Vivado block design

Once the bitstream was generated, to program the board and feed data into the FPGA, a Python overlay was required, as shown in Figure 3.2.1-C. This called Pynq libraries to first program the FPGA with the specified bitstream [4], convert the input data from Python float into C-style representation, then write to the address locations that serve as the input to the FPGA. The overlay would then wait for the 'done' signal from the FPGA (accomplished by polling), before reading the output from another set of addresses.

```
overlay = Overlay('/home/xilinx/pynq/overlays/mlp3/mlp3.bit')

add_ip = overlay.mlp_0

def float_to_decimal(num):
    return int(''.join('{:0>8b}'.format(c) for c in struct.pack('!f',
    num)),2)

def decimal_to_float(num):
    return struct.unpack('f', struct.pack('I', num))[0]

for i in range (0, 150):
    add_ip.write(0x400 + i*0x4, float_to_decimal(input0[i]))
print('Example 0')
for i in range (0, 3):
    result = add_ip.read(0x800 + i*0x04)
    print(str(i) + ": " + str(decimal_to_float(result)))
```

Figure 3.2.1-C
Overlay code used for verification of model design

This overlay would be integrated into our team's main code as a class, and as the data streams into the Ultra96, the state machine running on the CPU would decide the start of the dance move, and send the appropriate data to the FPGA. From the output of the FPGA, the state machine would compare the numbers to determine the most likely dance move performed.

Note that in our implementation, only the 8 dance moves will be predicted by the FPGA. The CPU of the Ultra96 will perform the detection of the start of the dance moves, dancer position changes, and logout move detection.

3.2.2 Neural Network Design

We considered two main design approaches: CNN and multi-layer perceptron (MLP). For CNN, our research suggested that it will be able to give more accurate results for this application, as the convolution method will be able to extract features automatically during training. Seen in Figure 3.2.2-A, a CNN will first apply a set of filters to the input data in the convolution layer [5]. Then a max pooling layer will be applied, to further reduce size and minimise noise. This can be done multiple times, before feeding into fully connected layers, which is similar to an MLP. As our data is '1 dimensional', just a stream of numbers, we would use Conv1D layers in keras. MLP would be a simpler neural network, with just a few fully connected layers (such as the one depicted in Figure 3.2.2-B [6]), but it comes with a more brittle nature, as the data is directly fed into the model.

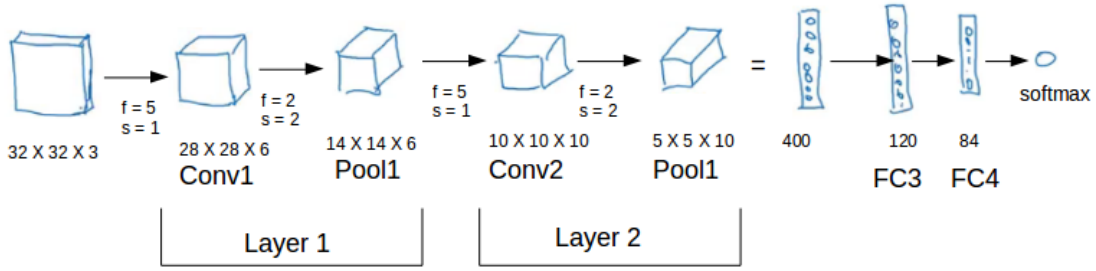


Figure 3.2.2-A
Visualization of a CNN

Overall, CNN would be more complex to translate into C++ for HLS and possibly use more FPGA resources. We needed to consider the limited resources on the Ultra96 board, such as 2GB RAM [7]. Even if we decided to use MLP, we needed to be cognizant of the size of the model as the memory capacity will limit the number of nodes (or parameters) in the network.

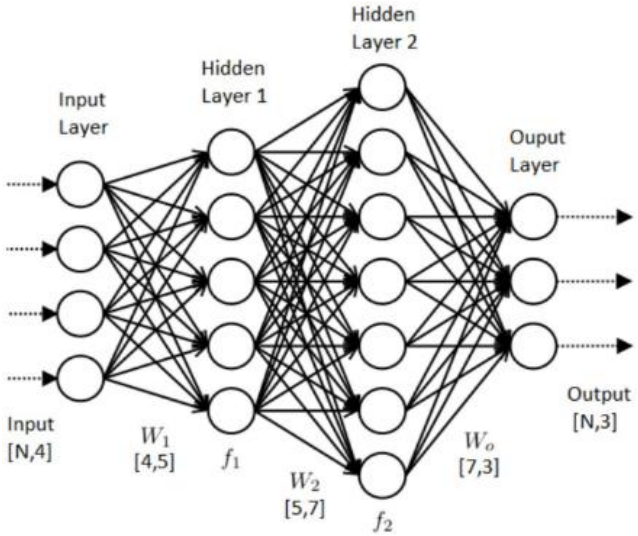


Figure 3.2.2-B
Sample MLP schematic

Due to its simplicity, we decided to implement an MLP model in our system first. The models that we tested had 4 layers in total (similar to Figure 3.2.2-B): an input layer, 2 hidden layers, and an output layer. We experimented with different shapes of the layers, ranging from 8 to 128 nodes. Initially our model would take in 100 samples from 6 sensors (3 axes of accelerometers and 3 axes of gyroscopes) for a total of 600 input values. The results from these early models were quite promising; they were able to classify dance moves correctly for

all team members for the majority of the time. We decided to improve the model by performing feature extraction first, then feeding 39 features into the model, allowing our final model to have two extremely compact hidden layers of 16 neurons each, followed by an output layer with a size of 8 representing the 8 dance moves.

We also considered how the neural network will have to be optimised for an FPGA, as the data straight from the sensors will be in floating point, for which calculations are computationally intensive. One option was to create a fully quantised network, where all values are converted to integers. FINN is specifically designed for this application, and can generate a quantised neural network. As we decided not to use FINN, we decided to convert the floating point input to fixed point before putting it into the model, then convert the fixed point outputs back to floating point. This conversion will be done within the FPGA itself, before any calculations are performed, and after the data is processed.

Further elaboration on neural network design can be found in Section 5.3 Neural Networks.

3.2.3 Mapping onto the FPGA Board, and Evaluation of Design

As per the workflow mentioned in Section 3.2.1, the design of the neural network will be mapped to the FPGA board using Vivado, with the Vivado HLS tool translating the C++ code to Verilog, then Vivado will generate a bitstream to place the circuits in the FPGA. If we decided to use FINN, the FINN framework has a C++ HLS library to aid this process [8]. We made a decision to go with translating the architecture into C++ code, and using the HLS tool to generate Verilog code, which can be seen in the code snippets in the figures below.

```
void mlp(float inputData[1][INPUT_ROW], float result[1][OUTPUT_ROW]) {
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS INTERFACE s_axilite port=inputData
#pragma HLS INTERFACE s_axilite port=result
#pragma HLS INTERFACE s_axilite port=return

    ap_fixed<32,16> convertIn[1][INPUT_ROW];
    ap_fixed<32,16> convertOut[1][OUTPUT_ROW];

    for (int i = 0; i < INPUT_ROW; i++) {
        convertIn[0][i] = (ap_fixed<32,16>)inputData[0][i];
    }
```

Figure 3.2.3-A
Initialization of main function

From Figure 3.2.3-A, the inputs and outputs of the function can be seen in the first line. The following lines are HLS pragmas that create interfaces so that values can be passed in by writing to memory addresses, and output and control values can be read. The for loop at the bottom is to convert the float input values to fixed point representation, with 16 bits on both the left and right of the decimal point.

```
    for(int i = 0; i < 1; i++) {
        for(int j = 0; j < HIDDEN_ROW1; j++) {
            hidden1[i][j] = 0;
            for(int k = 0; k < INPUT_ROW; k++) {
                hidden1[i][j] += convertIn[i][k] * weightsHidden1[k][j];
            }
        }
    }
```

Figure 3.2.3-B
Fully connected layer

The nested for loops in Figure 3.2.3-B is the implementation of a fully connected layer. The value of a node in this layer is the sum of all nodes in the previous layer multiplied by the weight specified from training. Thus, this layer is essentially a matrix multiplier.

```
for(int i = 0; i < 1; i++) {
    for(int j = 0; j < HIDDEN_ROW1; j++) {
        hidden1[i][j] += biasHidden1[j];
        if (hidden1[i][j] < 0) {
            hidden1[i][j] = 0;
        }
    }
}
```

Figure 3.2.3-C
Bias and ReLU

The code in Figure 3.2.3-C applies the bias to each node in the layer and performs rectified linear unit (ReLU) activation on each node. Other types of activation functions can be implemented, such as sigmoid and tanh. The code in Figures 3.2.3-B and 3.2.3-C can be repeated for more layers.

```
double m = -9999, sum = 0.0, constant;
for (ap_uint<8> i = 0; i < OUTPUT_ROW; ++i) {
    if (m < (float)convertOut[0][i]) {
        m = (float)convertOut[0][i];
    }
}
for (ap_uint<8> i = 0; i < OUTPUT_ROW; ++i) {
    sum += exp((float)convertOut[0][i] - m);
}
constant = m + log(sum);
for (ap_uint<8> i = 0; i < OUTPUT_ROW; ++i) {
    convertOut[0][i] = (ap_fixed<16,8>)exp((float)convertOut[0][i] - constant);
}
```

Figure 3.2.3-D
Softmax layer

Most neural networks normalize their outputs to values between 0 and 1, indicating a rough probability of what it thinks the classification should be. This can be done with a softmax layer, with the code seen in Figure 3.2.3-D [9]. We decided to do away with this layer as the raw result from the fully connected layers could be easily interpreted by the state machine running on the CPU.

```
for (int i = 0; i < OUTPUT_ROW; i++) {
    result[0][i] = (float)convertOut[0][i];
}
```

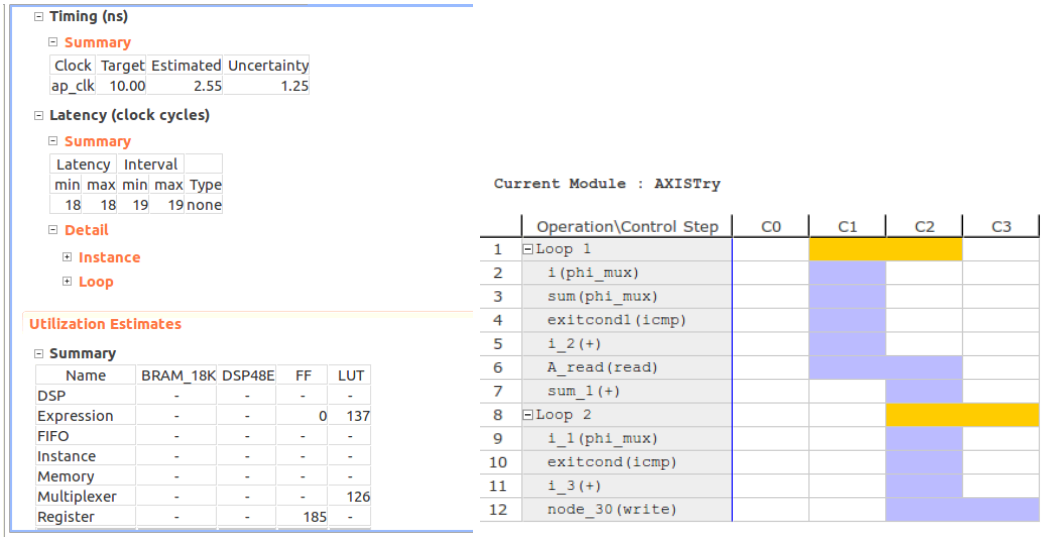
Figure 3.2.3-E
Output

The code in Figure 3.2.3-E is the reverse of the for loop in Figure 3.2.3-A; it converts the result from fixed point representation into floating point.

The code above could be reused in the implementation of CNN for the fully connected layers at the end. The convolution and pooling layers can be implemented in C++ as well [10].

The Vivado HLS program has several tools that allows for easy analysis of the design. After synthesis, it will generate a report containing statistics for the number of units used (area utilization), seen in the Figure 3.2.3-F below, and timing diagrams to show the number of clock cycles required to execute a function [11]. Using these tools will allow us to identify

areas that require substantial resources and target our efforts to improve the design, such as improving the speed.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.55	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
18	18	19	19	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	137
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	126
Register	-	-	185	-

Current Module : AXISTry

	Operation\Control Step	C0	C1	C2	C3
1	Loop 1				
2	i(phi_mux)				
3	sum(phi_mux)				
4	exitcond1(icmp)				
5	i_2(+)				
6	A_read(read)				
7	sum_1(+)				
8	Loop 2				
9	i_1(phi_mux)				
10	exitcond(icmp)				
11	i_3(+)				
12	node_30(write)				

Figure 3.2.3-F
Vivado HLS analysis interface

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.621 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
22592	22592	0.226 ms	0.226 ms	22592	22592	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	6	160	5917	-
FIFO	-	-	-	-	-
Instance	4	90	3684	5454	-
Memory	14	-	172	36	0
Multiplexer	-	-	-	968	-
Register	-	-	1938	-	-
Total	18	96	5954	12375	0
Available	432	360	141120	70560	0
Utilization (%)	4	26	4	17	0

Figure 3.2.3-G
Performance and resource utilization of an early design (600-16-16-3)

We can then use information from these tools to determine the impact of any optimizations that we did. From Figure 3.2.3-G, we used the latency of 0.226 ms and lookup table

utilization of 17% from one of our first models as a baseline to compare with changes to this design or a whole new model architecture.

Following synthesis and exporting the RTL from Vivado HLS, the IP can then be loaded into a Vivado block design, as was seen in Figure 3.2.1-B. The individual IPs can then be prepared for synthesis as a whole system, after which Vivado can run its implementation process and generate the bitstream.

3.2.4 Potential Optimizations

As FINN prepares the graph into HLS layers, it automatically does some optimising to improve performance [12]. It can perform a streamlining transformation, which works to reduce the number of floating point operations, such as by removing add with zero and multiply by one operations, or collapsing repeated operations. The FINN framework also does folding, which can be adjusted by changing the values for PE (Processing Elements) and SIMD (Single Instruction Multiple Data). Increasing the values would increase the amount of parallelism in the design.

While we did not use FINN, Vivado HLS has a number of built-in pragmas that are able to improve the efficiency of the output from synthesis [13]. One of these is the UNROLL pragma, which does loop unrolling where there are multiple instances of the loop created, allowing more than one loop iterations to execute in parallel and improving throughput. Others include the DATAFLOW pragma, which improves concurrency of the design, and the PIPELINE pragma, which also will improve the throughput of the design by allowing multiple inputs to be processed at the same time. The figures below demonstrate the effect of the DATAFLOW (Figure 3.2.4-A) and PIPELINE (Figure 3.2.4-B) pragmas.

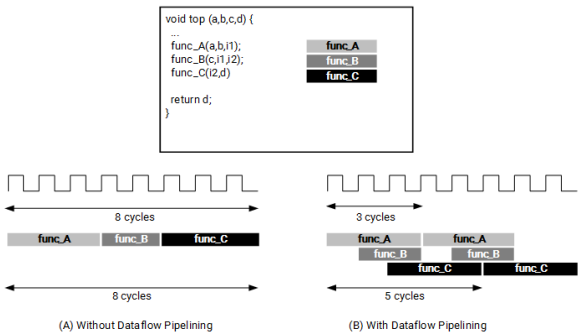


Figure 3.2.4-A
Dataflow pragma

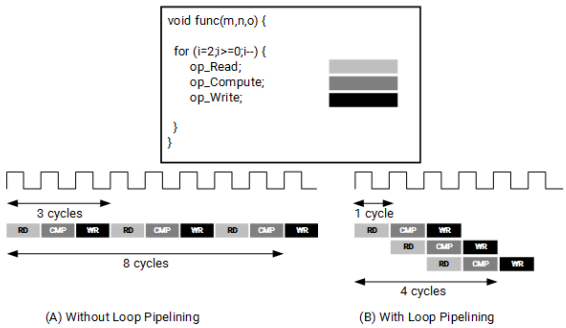


Figure 3.2.4-B
Pipeline pragma

These optimizations can improve the latency, but it can result in a negative impact on the area utilization and power usage. As seen in Figure 3.2.4-C, when the PIPELINE pragma was applied to all loops in the code, the latency improved by more than 5 times (see Figure 3.2.3-G for statistics of the same design but without PIPELINE pragma). However, the resources required far exceeds the capabilities of the FPGA. Even when only half of the loops are optimized with the PIPELINE pragma, there is still a significant jump in the resource utilization while there is a considerable fall in the latency. When we considered this performance boost from a macro perspective – that we are trying to detect human movements – this optimization which results in a reduction of less than 1 ms in prediction time makes virtually no difference as the human reaction time is much slower. Therefore, we decided not to implement pragmas in our final design. Perhaps, if there were a future need to be constantly running predictions on a large number of dancers, it would be worthwhile to explore this as a method to improve performance.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.621 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
3092	3092	30.920 us	30.920 us	3092	3092	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	1296	160	56136	-
FIFO	-	-	-	-	-
Instance	4	90	3684	5679	-
Memory	3306	-	262	73	0
Multiplexer	-	-	-	5138	-
Register	0	-	76644	5242	-
Total	3310	1386	80750	72268	0
Available	432	360	141120	70560	0
Utilization (%)	766	385	57	102	0

Figure 3.2.4-C

Performance and resource utilization of early design with PIPELINE pragma (600-16-16-3)

We have also explored the adjustment of the amount of quantization, but there is a trade-off. A higher amount of quantization could mean a faster speed but a loss of accuracy. Faraone *et al.*, however, notes that QNNs are able to achieve "state-of-the-art accuracies under low-precision weights and activations" [14]. In our final implementation, we decided to go for a 'middle of the road' approach, by using fixed point values of 32 bits (16 bits on each side of the decimal point). While not as efficient as when using integers only, there is a vast reduction in the amount of resources required when compared with floating point calculations.

Another optimization would be to use a simpler architecture, as a smaller size with fewer (or less complex) calculations would allow for data to be processed faster. From the start, we tried experimenting with models with around 10 000 parameters. Even as we were using raw data from the sensors (100 samples of 6 streams of data), we managed to make a model with relatively good accuracy with 16 neurons in two hidden layers each. The next technique was to perform feature extraction. If a large amount of feature extraction can be done, this will allow us to simplify the neural network design. We initially tried doing feature extraction at the sensor level, but the chips on the Beetles were not powerful enough. So the feature extraction would have to be implemented on the CPU of the Ultra96. With feature extraction, the model only had 39 inputs, reducing the size of the model greatly with only about 1000 parameters (refer to 3.2.4-D for latency and utilization statistics). At this size, we realized that

other optimizations on the FPGA (such as the use of pragmas and quantization mentioned above) would have limited positive impact on the overall performance of the system.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.562 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
2364	2364	23.640 us	23.640 us	2364	2364	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	6	40	2347	-
FIFO	-	-	-	-	-
Instance	4	0	296	319	-
Memory	5	-	241	31	0
Multiplexer	-	-	-	404	-
Register	-	-	413	-	-
Total	9	6	990	3101	0
Available	432	360	141120	70560	0
Utilization (%)	2	1	~0	4	0

Figure 3.2.4-D

Performance and resource utilization of final design (39-16-16-8)

3.2.5 Power Management of Ultra96 CPU+FPGA

The Ultra96 has an ARM Cortex-A53 processor as its CPU [7]. This chip has the ability to minimise power consumption by shutting down cores and power domains. Another method to reduce power consumption is to slow the clock. A lower frequency will result in fewer computations per second and therefore result in slower response. These settings can be changed through the Pynq OS with a command, as seen in Figure 3.2.5-A. Our team decided that we would prefer not to throttle the CPU, as this could have unexpected effects on the streaming of data or the state machine, impacting our prediction accuracy.

```
xilinx@pynq:~/pynq$ sudo cpufreq-set -u 1200Mhz
xilinx@pynq:~/pynq$ echo 1 | sudo tee /sys/devices/system/cpu/cpu1/online
```

Figure 3.2.5-A

System commands to change clock speed and turn on/off cores

Another way to save power would be to use a smaller and simpler neural network, so that fewer FPGA resources are required. To create a smaller network we could have experimented with training a lower number of nodes, or by performing feature extraction. These methods of improving performance while minimising power consumption were discussed in Section 3.2.4. While feature extraction may reduce the workload on the FPGA, it could cause the

CPU of the Ultra96 to work more (hence consuming more power), as it has to compile the data and calculate statistics such as the mean, variance, skew, kurtosis of each stream of data.

For the FPGA, power consumption can be reduced by optimising use of the circuits themselves; minimising the area used on the board will result in lower power usage. However, this would likely have a negative impact on the processing speed. There are some pragmas available that would reduce the number of resources used by the design, such as the ALLOCATION pragma to limit multiple instances of a function to be implemented as one RTL block and the ARRAY_MAP pragma to combine multiple small arrays into a single array [13]. However, due to our extremely small final model, there was minimal difference between using these pragmas in terms of performance and power savings. Thus, we decided not to include these pragmas in our final design.

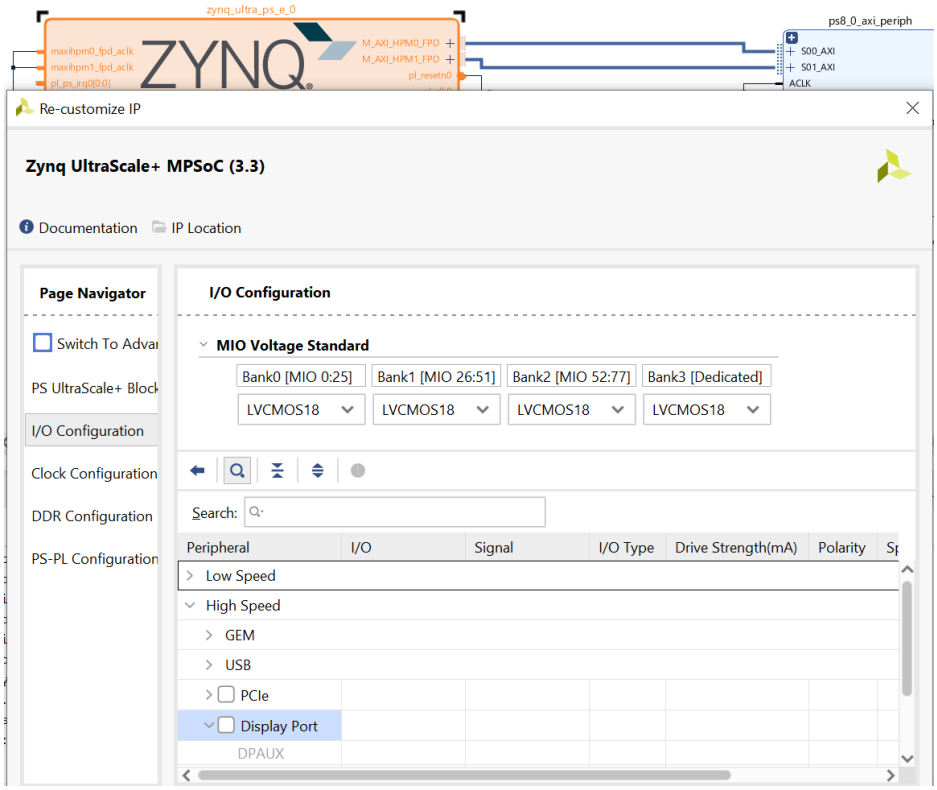


Figure 3.2.5-B
DisplayPort turned off in Vivado

Further reduction of power consumption may come from the other components. We explored the possibility of turning off the peripherals on the Ultra96. Some of these were the ports on the board, such as the DisplayPort. This could be switched off using Vivado, as seen in Figure 3.2.5-B above. We tried to measure any power savings from such measures using the power monitoring features in the Pynq OS, which takes readings from the Ultra 96 power monitoring rails [15] (see code in Figure 3.2.5-C), but the data collected from the program indicated minimal or no difference.

```python
import pynq
import time

rails = pynq.get_rails()
#print(rails)

if 'VSYS' in rails.keys():
    print("Recordign Ultra96 v1 power...")
    rail_name = 'VSYS'
elif 'PSINT_FP' in rails.keys():
    print("Recording Ultra96 v2 power...")
    rail_name = 'PSINT_FP'
else:
    raise RuntimeError("Cannot determine Ultra96 board version.")
recorder = pynq.DataRecorder(rails[rail_name].voltage, rails[rail_name].current)

recorder.reset()
with recorder.record(0.5):
    time.sleep(5)
    recorder.mark()
    for _ in range (10000000):
        pass
    recorder.mark()
    time.sleep(5)

print(recorder.frame)

```

Figure 3.2.5-C
Power monitoring code

Section 4 Firmware & Communications Details

4.1 Internal Communications - *Written by Karnati Sai Abhishek*

The primary objective of internal communications was to provide a reliable medium to transfer sensor data from the Beetles to the laptop. The first subsection will give an overview of how the internal communications subcomponent fits into the project. The second subsection will elaborate upon the tasks which ran on the Beetles and how they were managed efficiently. The next subsection will cover details about how a robust and efficient communication link was ensured between the Beetles and the laptop. The final subsection will elaborate upon the various challenges we faced during the implementation phase.

4.1.1 Internal Communications Architecture

The following diagram provides an overview of the internal communications subcomponent architecture:

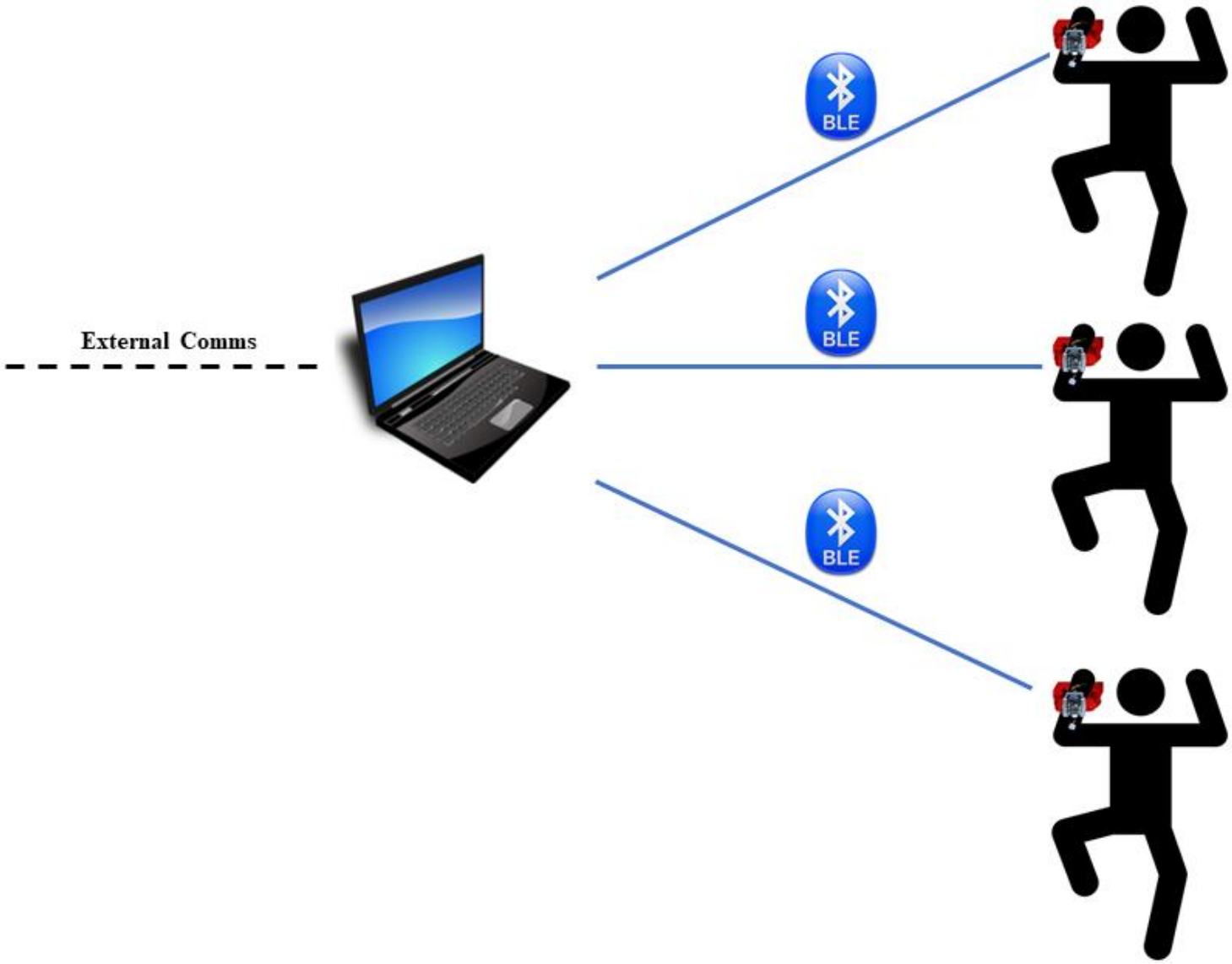


Fig 4.1.1
Internal Communications Overview

Firstly, the three Beetles on the dancers gathered sensor data. This data was pre-processed and packaged into data packets to be sent to the laptop over a BLE (Bluetooth Low Energy) connection. Finally, these packets were deconstructed and the relevant data was relayed over to the Ultra96 over the external communication link.

4.1.2 Task Management on Beetles

The Beetles served to read data from the sensors, process this data and transmit the data over a BLE connection to the laptop. These tasks can be broken down further as shown below:

1. Read sensor data
   The accelerometer and gyroscope values were read from the IMU on the dancer's wearable. This data was sampled at a frequency of 20Hz.

2. Process sensor data
   The noise was filtered out from the sensor data to ensure that it did not skew the results of the machine learning model's predictions.

3. Transmit sensor data
   The processed data was packaged into structured packets (detailed in section 4.1.3) which were then sent to the laptop over a BLE connection at a frequency of 20Hz.

At the beginning of the project, there were two approaches to implementing the tasks:

Super-Loops

The first option was to use super-loops. Super-loops involve the use of an infinite loop within which all the system tasks are contained. Given that the tasks of reading, processing followed by transmitting make sense in that order, super-loops would allow for the continuous sequential execution of these tasks. The diagram below depicts what a typical super-loop with the three tasks looks like:

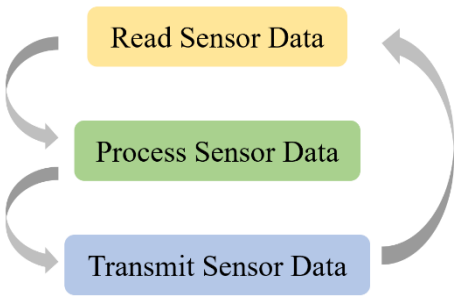


Fig 4.1.2-A
Super-Loop with tasks

Real-time Scheduling

The second option facing us was to use real-time scheduling with the help of the FreeRTOS library. RTOS (Real-Time Operating System) separates the program functions into self-contained tasks and implements on-demand scheduling of their execution [1].

Instead of sequential execution, RTOS allows the scheduling and prioritization of tasks to ensure that all three tasks run smoothly at any given point in time. Task scheduling would ensure better program flow and event response while context switching gives the illusion of executing multiple tasks simultaneously. The diagram below depicts real-time scheduling between the 3 tasks:

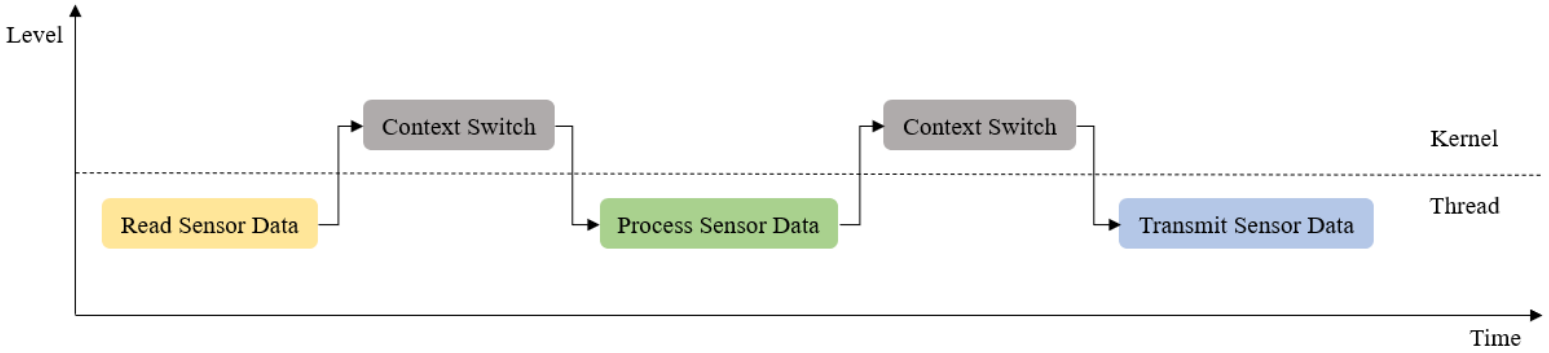


Fig 4.1.2-B
Real-Time Scheduling of tasks

Over the course of the project, we came to realize that the tasks of reading, processing and transmitting data were equally important to the system functionality. After testing out the system with both super loops and RTOS, we observed that there was no significant difference in the system performance in terms of latency between the two options. Since the priorities of the tasks were equal, there was no tangible benefit to using RTOS. As such, we chose to use super-loops for task management to prevent adding unnecessary complexity to the system.

4.1.3 Communication Protocol

Communication protocols specify a common packet format so that all nodes in a communication network know how to construct and parse data packets from others [2]. In our use case, such protocols were quintessential to coordinate between the Beetles and the laptop.

Packet Types

Following are the different types of packets that were used in our communication protocol:

Packet Type	Packet Code
HELLO	‘H’
ACK	‘A’
DATA	‘D’
TIME	‘T’

Fig 4.1.3-A
Packet codes

1. HELLO: This packet was sent from the laptop to the respective Beetle to initiate a 3-way handshake
2. ACK: This packet served as a form of positive acknowledgement that the data had been successfully received
3. DATA: This packet contained the processed sensor data which was sent from the Beetles to the laptop for feature extraction in the machine learning model
4. TIME: This packet served to fulfil the purpose of clock synchronization between the laptop and the Beetles

Packet formats

The HELLO and ACK packets did not have any payload associated with them. The following figure depicts the format of a typical TIME packet intended for clock synchronization:

Packet Code	Timestamp	Checksum
1 byte	4 bytes	1 byte
Total Size: 6 bytes		

Fig 4.1.3-B
TIME packet format

1. Packet Code: Determined the type of the packet which in this case is ‘T’
2. Timestamp: Specified the timestamp at the point of sending the packet
3. Checksum: XOR checksum which helped in error detection

The following figure depicts the format for a typical DATA packet:

Packet Code	**Accelerometer data (x, y, z)**	**Gyroscope data (x, y, z)**	**Timestamp**	**Checksum**
1 byte	3 bytes	3 bytes	4 bytes	1 byte
Total Size: 12 bytes				

Fig 4.1.3-C
DATA packet format

1. Packet Code: Determined the type of the packet which in this case is 'D'
2. Accelerometer/Gyroscope Data: Contained the pre-processed sensor data read from the IMU
3. Timestamp: Specified the timestamp at the point of sending the packet
4. Checksum: XOR checksum which helped in error detection

Our initial design had an additional ID field that was intended to identify the Beetle sending the data. However, during our system integration, we found that the Beetle transmitting data could be identified based on the BLE connection which mitigated the need for ID. Furthermore, we were able to shrink the allocated size of the sensor data from the initial 12 bytes to 6 bytes due to efficiencies in the machine learning model's predictions. All this together helped to significantly bring down the total packet size from 19 bytes to a compact 12 bytes. A small packet size that was well below the BLE connection limit meant that there was an increase in the reliability of the data transmission.

4.1.4 BLE (Bluetooth Low Energy) Communication

As mentioned earlier, the communication link between the Beetles and the laptop was over a BLE connection. BLE is a wireless communication technology that is essentially a Bluetooth connection optimized for applications with considerably low power consumption targeted at low power devices, wearables, etc. It is ideal for applications that only need to exchange small amounts of data periodically [3]. Given that our system matched these specifications perfectly, BLE was adopted as the main communication link between the Beetles and the laptop.

This subsection aims to answer the following questions:

1. How were the Beetles configured for communication with the laptop?
2. How did the laptop connect to the Beetles?
3. How did the laptop communicate with the Beetles?
4. How was a reliable data transfer ensured between the laptop and the Beetles?

Configuration of Beetles

Baud rate determines the speed of communication over a serial data channel. A high baud rate is needed to make sure there is low latency in transmission and to maintain a "real-time" transfer of data. If the baud rate is too high, it could potentially cause an increase in the number of errors due to the inability of clock and sampling rates to maintain high speeds. Hence, for the laptop and the Beetles to send and receive data at the same speed, the recommended and reliably fast baud rate of 115200 bps was chosen.

The following set of AT configuration commands were run in the Arduino IDE to configure the Beetles [4]:

AT Command	Description
AT+UART=115200	Sets the baud rate to 115200 bps
AT+FSM=FSM_HID_USB_COM_BLE_AT	Changes the working mode of the Beetle to USB-UART BLE HID mode
AT+ROLE=ROLE_PERIPHERAL	Changes the CENTRAL-PERIPHERAL configuration of the Beetle to BLE PERIPHERAL mode
AT+MIN_INTERVAL=10	Sets the minimum connection interval to the recommended 10 ms
AT+MAX_INTERVAL=40	Sets the maximum connection interval to the recommended 40 ms

Connecting to the Beetles

We chose to use the bluepy library running on Linux to establish BLE connections between the laptop and Beetles. The following code snippet depicts how the laptop connected with the Beetles:

```
devices = Setup.scanDevices()

for d in devices:
    if d.addr in addr_arr:
        start = time()
        print("Connecting to Beetle %s..." % (addr_map[d.addr]))
        beetle = Setup.formPeripheral(d.addr)
        print("Successfully formed peripheral with Beetle %s!" %(addr_map[d.addr]))
        CommsThread(beetle).start()
```

Fig 4.1.4-A
Laptop-Beetle connection code

As shown above, initially, the laptop scans for all the available Bluetooth devices in its vicinity. If the MAC addresses of the Beetles are found, the laptop proceeds to connect to them in each iteration of the loop. Following this, once a successful peripheral connection has been made with the Beetle, a new thread is created for every Beetle. Using multiple threads allow for concurrent communication between the laptop and Beetles.

Handshaking

Before the Beetles and the laptop start transmitting sensor data between each other, all the communicating parties must be ready to receive/transmit data. This is to ensure that the communicating parties have a reliable connection between them and that no data is lost. Hence, to initiate the communication link, we used the 3-way handshake protocol [5] so that the Beetles and the laptop were aware of each other. The diagram below depicts a successful 3-way handshake between a Beetle and laptop:

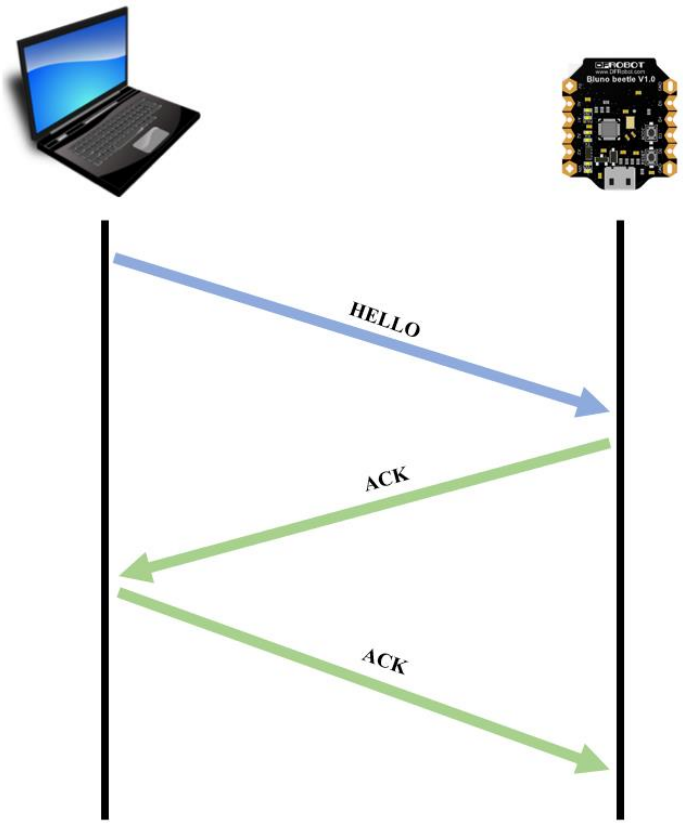


Fig 4.1.4-B
3-way handshake between the Laptop and a Beetle

As can be seen in the diagram, the laptop initiates the 3-way handshake by sending a HELLO packet. Following this, it waits for an ACK packet to be received as a form of acknowledgement from the Beetle. Finally, the 3-way handshake ends with the laptop sending the same Beetle an ACK packet as a form of acknowledgement from the laptop. Successful execution of these steps marks the end of the handshake, following which, the two respective devices can begin their communication. Similarly, the laptop repeats the same 3-way handshake protocol to establish connections with the rest of the Beetles in the system.

Following is a laptop code for initiating the handshake:

```
def initHandshake(self):
    print('Initializing handshake with Beetle %s...'%addr_map[self.beetle.addr])
    count = 1
    while not handshake_status_map[self.beetle.addr]:
        self.serial_char.write(bytes(HANDSHAKE,'utf-8'),withResponse = False)
        print(str(count) + " HANDSHAKE packet(s) sent to Beetle %s"% addr_map[self.beetle.addr])

        if self.beetle.waitForNotifications(2.0):
            print("Successfully connected to Beetle %s!" % (addr_map[self.beetle.addr]))
            self.serial_char.write(bytes(ACK,'utf-8'),withResponse = False)
            return True
        count += 1
        if count == 3:
            print('Unsuccessful handshake with Beetle %s'%addr_map[self.beetle.addr])
            return False
```

Fig 4.1.4-C
Laptop handshake code

Following is the Beetle code for handling the handshake:

```
if (Serial.available()) {
  byte cmd = Serial.read();
  switch (cmd) {
    case HANDSHAKE:
      start_time = millis();
      handshake = true;
      handshake_confirmed = false;
      Serial.write(ACK);
      start_time = 0;
      break;

    case ACK:
      if (handshake) {
        handshake = false;
        handshake_confirmed = true;
      }
      break;
  }
```

Fig 4.1.4-D
Beetle handshake code

As shown above, the laptop initiates the handshake by sending a HANDSHAKE packet. When the Beetle receives a HANDSHAKE packet, it sends back an ACK packet to the laptop. Upon receiving this ACK packet, the laptop sends back an ACK packet to the Beetle as a form of acknowledgement that the handshake operation was successful.

In the case when the laptop does not receive an acknowledgement for the HANDSHAKE packet, it repeatedly sends the HANDSHAKE packet three times before which the operation terminates as a sign of an unsuccessful handshake.

Clock Synchronization

It is important to do clock synchronization between the laptop and Beetles to establish a common ground for the timestamps sent as part of the DATA packets. For this purpose, we chose to use Network Time Protocol (NTP) [6]. The following diagram showcases how clock synchronization between the laptop and Beetles takes place:

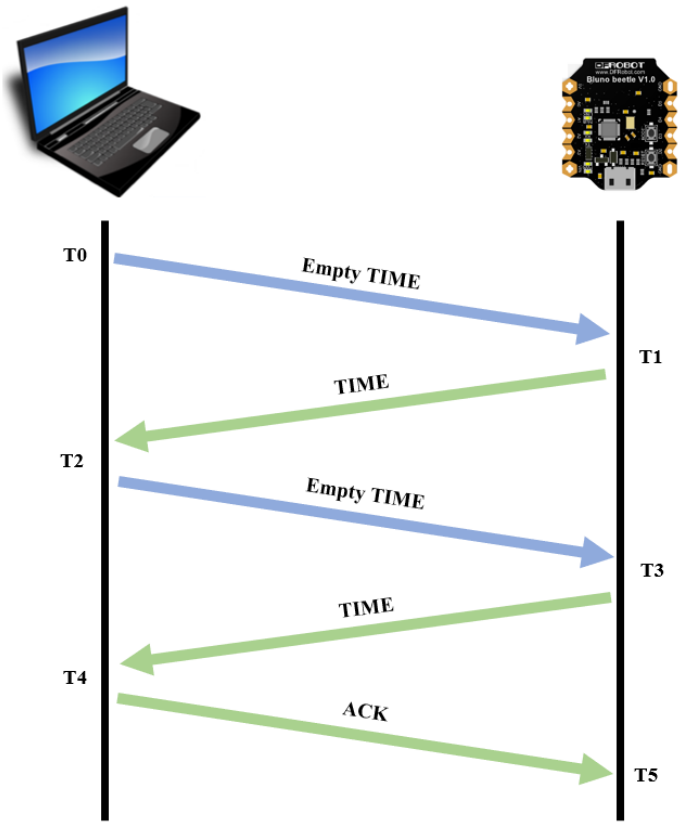


Fig 4.1.4-E
Clock synchronization between the Laptop and a Beetle

As shown above, the laptop initiates the clock synchronization by sending an empty TIME packet to the Beetle. The Beetle responds with a TIME packet containing the timestamp at T1. When this packet is received by the laptop at T2, the laptop time is recorded. Following that, the laptop sends another empty TIME packet to the Beetle that responds with a TIME packet containing the timestamp at T3. When this packet is received by the laptop at T4, the offset is calculated in milliseconds as follows:

$$offset = \frac{(T3 - T2) + (T2 - T1)}{2}$$

Once the offset is computed, the laptop sends an ACK packet to the Beetle as a sign of acknowledgement that the clock synchronization was successful. This clock synchronization protocol took place periodically between the laptops and Beetles to keep the sync delay between the dancers as low as possible.

Data Transmission

Once the 3-way handshake and clock synchronization are successful, the Beetle is ready to transmit sensor data while the laptop is ready to receive sensor data.

Following is the Beetle code for sending sensor data:

```
void sendData(){
  unsigned long actual_time = millis() - start_time;
  int8_t chksum = 0;

  // packet code
  Serial.write(DATA);
  chksum = computeChecksum(chkSum, DATA);

  // accelerometer data
  Serial.write((int8_t)((int16_t) ax >> 8));
  chksum = computeChecksum(chkSum, ax >> 8);
  Serial.write((int8_t)((int16_t) ay >> 8));
  chksum = computeChecksum(chkSum, ay >> 8);
  Serial.write((int8_t)((int16_t) az >> 8));
  chksum = computeChecksum(chkSum, az >> 8);

  // gyroscope data
  Serial.write((int8_t)((int16_t) gx >> 8));
  chksum = computeChecksum(chkSum, gx >> 8);
  Serial.write((int8_t)((int16_t) gy >> 8));
  chksum = computeChecksum(chkSum, gy >> 8);
  Serial.write((int8_t)((int16_t) gz >> 8));
  chksum = computeChecksum(chkSum, gz >> 8);

  // timestamp
  byte timestamp[4];
  timestamp[0] = (actual_time >> 24);
  timestamp[1] = (actual_time >> 16);
  timestamp[2] = (actual_time >> 8);
  timestamp[3] = actual_time;
  Serial.write(timestamp, sizeof(timestamp));
  chkSum = computeChecksum(chkSum, timestamp);

  // checksum
  Serial.write((int8_t) chksum);
}
```

Fig 4.1.4-F
Beetle code for sending DATA packets

The data transmitted by the Beetle follows the format of a DATA packet. As shown above, the 16-bit accelerometer and gyroscope integer values read from the IMU were right-shifted by 8 bits. As such, we were able to preserve the sensitivity and accuracy of the data while also keeping the packet size small thus allowing for faster and more reliable transmissions. The sensor values are followed by the timestamp. Finally, the XOR checksum appended to the end of the packet serves the purpose of detecting errors during transmission.

Following is the laptop code for receiving the sensor data:

```
def handleNotification(self, cHandle, fragment):
    if handshake_status_map[self.addr]:
        self.buffer += fragment
        if len(fragment) >= 12:
            current = self.buffer[0:12]
            self.processData(current)
            self.buffer = self.buffer[12:]
```

Fig 4.1.4-G
Laptop code for receiving DATA packets

When there is a BLE notification, the above function is called. This indicates the transmission of new data from the Beetle to the laptop.

At times, there was packet fragmentation during transmission. To deal with this, the received data was initially stored in a buffer. Since the length of a DATA packet was 12 bytes, if the length of the buffer exceeded 12, then it implied that the entire packet had been received. As such, the packet was sent for processing and this data was cleared from the buffer.

```
def processData(self, data):
    if data[0] == 68:
        b = struct.unpack('!cbbbbbbLb', data)
        if checkChkSum(b):
            for i, x in enumerate(b):
                if i == len(b) - 2:
                    x = clock_sync_map[self.addr] + x
                if isinstance(x, bytes):
                    x = x.decode('utf-8') + "|" + str(addr_map[self.addr])
                self.packet += str(x) + "|"
            self.sendData()
            self.packet = ""
```

Fig 4.1.4-H
Laptop code for processing DATA packets

The bytes retrieved from the buffer needed to be decoded to do any further processing. As shown above, the struct.unpack() function is used to decode the data based on the standardized DATA packet format. If the computed checksum does not match the checksum at the end of the DATA packet, the packet is dropped. If the checksum checks out, the data is formatted and sent to the Ultra96 by the call of the external communications function self.sendData().

Handling Reliability Issues

XOR Checksum

Having a BLE communication link between the Beetles and laptop meant that there was room for issues like loss of packets or corruption of data in the packets. This is why it was important to check for errors upon receiving packets to ensure that the data received was as intended.

To keep this error detection process simple, a simple XOR checksum was used. An XOR checksum is essentially a longitudinal parity check which computes the exclusive OR (XOR) of all the bytes in a packet. Once this checksum byte is computed, it is appended to the end of the DATA packet before being sent out by the Beetles.

```
def checkChkSum(self, data):
    try:
        received_chksum = data[-1]
        data = data[0:len(data)-1]
        chksum = 0
        for i in range(len(data)):
            chksum ^= ord(data[i])
        if (chksum != received_chksum): return False
        return True
    except ValueError:
        return False
    except Exception as e:
        print(self.packet)
        print(e)
        error_other_count_map[self.addr] += 1
        return False
```

Fig 4.1.4-I
Laptop code for verifying the checksum

Upon calling the above function, the received checksum is stored and the XOR checksum is computed using the remaining bytes in the packet. If there is a disparity between the computed checksum and the received checksum, the system can conclude that an error has been detected. In such a scenario, the corrupted packet is dropped. If there is no error detected, the data is relayed over to the Ultra96 over the external communication link.

Unreliable BLE link

At certain times, when the established BLE link was not reliable, we observed that the Beetles got disconnected from the laptop. To deal with this, we needed a way for the laptop to quickly reconnect to the Beetles without human intervention.

```
except BTLEDisconnectError:
    handshake_status_map[self.beetle.addr] = False
    print('Error! Beetle %s got disconnected!'%addr_map[self.beetle.addr])
    self.reconnect()
```

Fig 4.1.4-J
Laptop exception handling code for BLE disconnections

The above exception handler helps to detect Beetle disconnections from the laptop and consecutively attempts to reconnect to the Beetle.

```
def reconnect(self):
    sleep(5)
    print('Attempting to reconnect with Beetle %s...'%addr_map[self.beetle.addr])
    try:
        self.beetle.connect(self.beetle.addr)
        sleep(3)
        print('Reconnected to Beetle %s'%addr_map[self.beetle.addr])
        self.run()
    except Exception as e:
        print(e)
        print('Error! Unable to connect to Beetle %s'%addr_map[self.beetle.addr])
        self.reconnect()
```

Fig 4.1.4-K
Laptop code for reconnecting to the Beetles

The above function showcases how the laptop continuously makes attempts to reconnect to the Beetle until a successful connection has been established.

There were other times of unreliable connections when we observed that many of the received packets were corrupted. To deal with this, a counter was implemented to take note of the number of packets that were dropped due to a bad checksum.

```
if (error_current_count_map[self.addr] >= ERROR_THRESHOLD):
    error_current_count_map[self.addr] = 0
    reset_status_map[self.addr] = True
```

Fig 4.1.4-L
Laptop code for error count

If this counter value exceeds the specified error threshold, the laptop will deliberately disconnect from the Beetle. After a certain delay, the laptop will reconnect with the Beetle. Through extensive testing, we observed that this process of disconnection and reconnection helped to establish a more reliable connection thus significantly reducing the error rate in transmissions.

4.1.5 Challenges Faced

We faced quite a few challenges with internal communications during the system integration phase. This section aims to detail some of these challenges and how we overcame them.

Random Disconnections

Despite our efforts at keeping the packet size down to a minimal 12 bytes and the data rate at a relatively low 20 Hz, we observed that there were times when the Beetles would disconnect from the laptop. Since these disconnections happened randomly, we had to extensively test the system to narrow down the actual issue.

We initially thought that this could have been a result of loose connections in the power circuit of the wearables. Upon close inspection, we found that there indeed were a few wires which came loose as a result of our rigorous dancing. After resoldering the wires and ensuring that the circuits were properly connected, we observed that the disconnections still occurred. Having checked that the hardware components were all working as intended, we were sure that these disconnections were occurring as a result of a software issue.

After several hours of tweaking AT configuration settings and reanalyzing the laptop and Beetle code, we finally concluded that the BLE connection was unstable because of poor Linux Bluetooth drivers. We were using a Linux virtual machine to run the internal communications laptop program then, so the idea was to instead use a native Linux based system to run the program. As such, we dual booted the laptop to install a native version of Linux and manually updated all the Bluetooth drivers. We also created an android app that we could fall back on in case the solution did not work out. To our fortune, using a native Linux installation worked out in improving the reliability of the communication link and disconnections became less frequent thereafter.

Loss of Data during Disconnections

Despite being rare, disconnections still occurred and we had to account for them. During our tests, we observed that when the Beetles got disconnected from the laptop, the laptop was aware of the disconnection but the Beetles were not. This meant that they continued to transmit sensor data even after the BLE link was broken and we were effectively losing all that data.

As such, we implemented a buffering mechanism on the Beetles. As and when DATA packets were sent out by the Beetles, the packets were also added to a temporary buffer. On the laptop end, when the BLE connection was stable, we periodically sent an ACK packet as a form of acknowledgement that the Beetles were still connected to the laptop. When this ACK was received by the Beetles, the buffer was cleared as it meant that the laptop had successfully received the previously sent out DATA packets.

If the Beetles got disconnected from the laptop, they did not receive the ACK and so the DATA packets accumulated in the buffer. Upon reconnection with the disconnected Beetles, after a successful 3-way handshake, all the packets in the buffer were resent before clearing the buffer. Following that, the Beetles resumed transmitting the real-time sensor data as per normal.

In such a manner, we were able to ensure that even when the Beetles got unexpectedly disconnected from the laptop, we did not lose any of the essential sensor data.

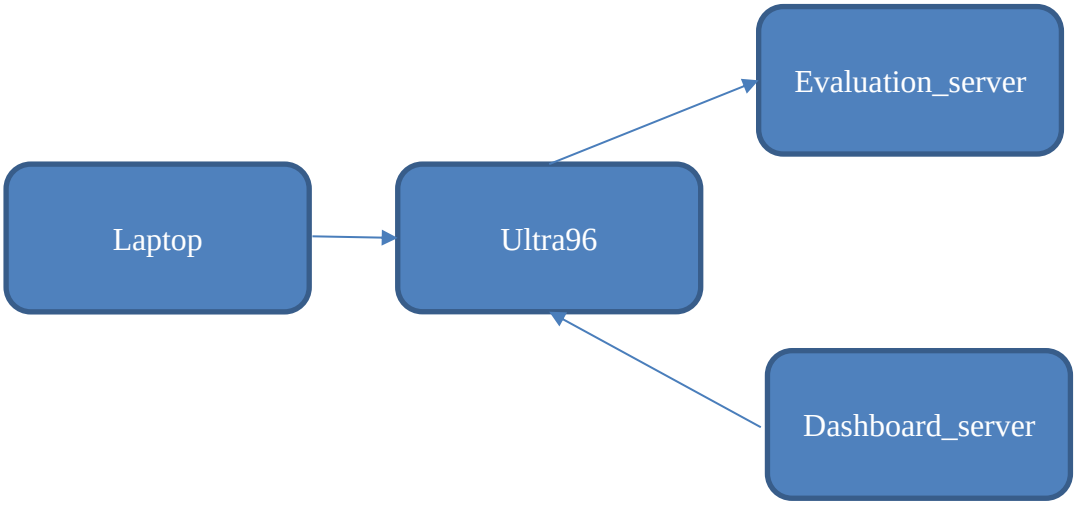


Figure 4.2-A
Overview of External Communications between laptops, dashboard and evaluation server

A main.py file will be in charged to run the evaluation_client, dashboard_server, ultra96_server files that will be receiving or sending messages over to the evalution_server, dashboard_client and laptop_client. The following shows how the files interact with each other:

- Laptop client will send bluno data to the ultra96 server.
- Ultra96 client will send bluno data with predicted data to the evaluation server.
- Dashboard server will send bluno data and predicted data to dashboard client.

TCP

External communications will be implemented via TCP/IP over WiFI/Ethernet. Sockets and the socket API are used to send messages across a network. They provide a form of inter-process communication (IPC). The network can be a logical, local network to the computer, or one that is physically connected to an external network, with its own connections to other networks. The obvious example is the Internet, which one connects to via ISP.

Transmission Control Protocol (TCP) will be utilized as our main protocol as it is reliable (packets dropped in the network are detected and retransmitted by the sender) and it has in-order data delivery (data is read by the application in the order it was written by the sender). On the other hand, User Datagram Protocol (UDP) sockets created with are not reliable and data read by the receiver can be out-of-order from the sender's writes.

Networks are a best-effort delivery system. There is no guarantee that one's data will reach its destination due to potential packet drops or that one will receive everything that is sent. Network devices (for example, routers and switches), have finite bandwidth available and their own inherent system limitations. They have CPUs, memory, buses, and interface packet buffers, just like our clients and servers. TCP relieves one from having to worry about packet loss, data arriving out-of-order, and many other things that invariably happen when one is communicating across a network [1].

The TCP socket flow can be seen in the below diagram:

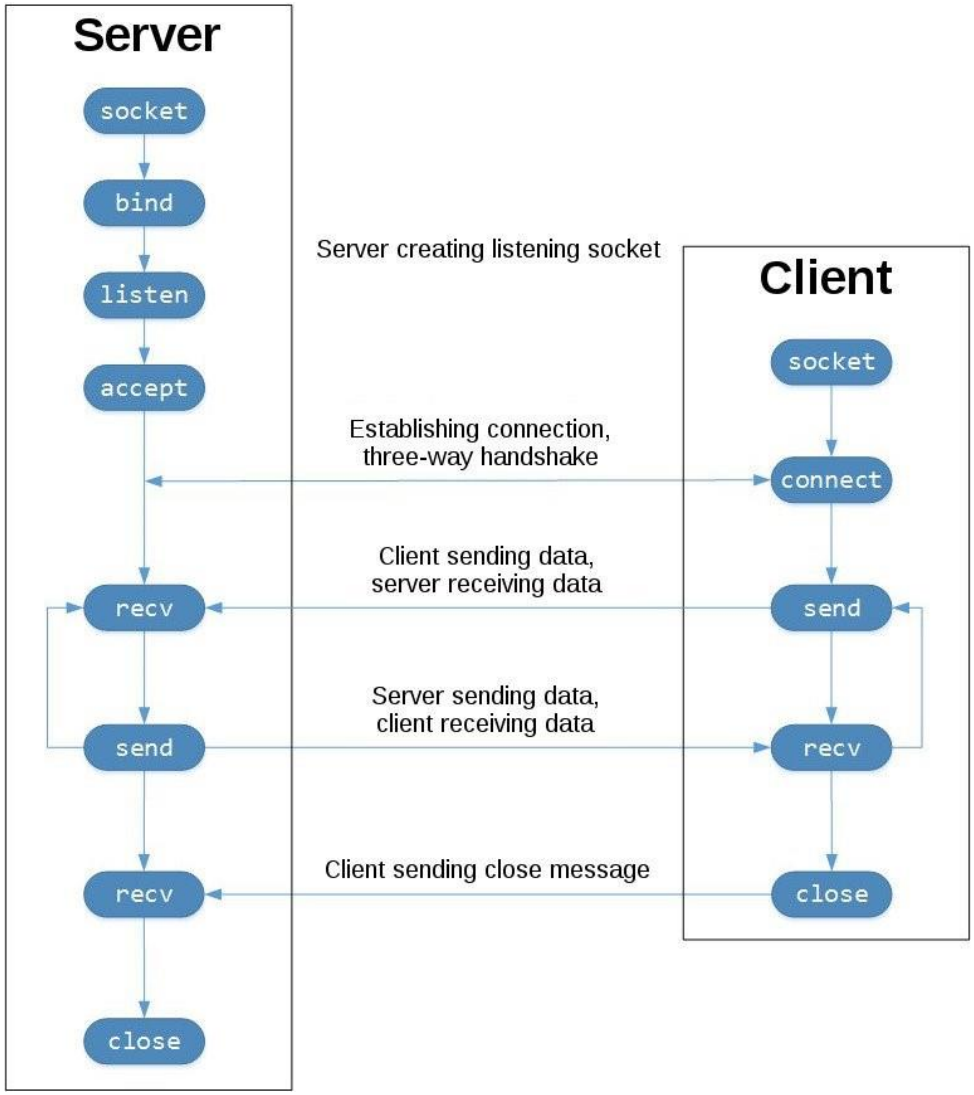


Figure 4.2-B
Overall TCP socket flow between server and client

Starting in the top left column, note the API calls the server makes to setup a "listening" socket:

- socket()
- bind()
- listen()
- accept()

A listening socket listens for connections from clients. When a client connects, the server calls accept() to accept, or complete, the connection.

The client calls connect() to establish a connection to the server and initiate the three-way handshake. The handshake step is important since it ensures that each side of the connection is reachable in the network, in other words that the client can reach the server and vice-versa. Data is exchanged between the client and server using calls to send() and recv(). At the bottom, the client and server close() their respective sockets.

Our implementation will follow the above protocol for all communications, which will be further discussed in each of the sections below.

Furthermore, the Ultra96 will use NTP estimation to achieve time synchronization with relay laptop, and by extension, the Bluno Beetles. All TCP connections will be encrypted via AES protocol (further explained in Section 4.2.4). Furthermore, each TCP connection on the Ultra96 will have its own dedicated thread.

4.2.1 Communication between laptops and Ultra96

Connection between laptops and the Ultra96 will be achieved through Transmission Control Protocol (TCP) using the Socket library in Python. The Ultra96 server file, where the socket is created using the Ultra96's IP address and a particular port number, will be run first to wait for connections.

SSH port forwarding is a mechanism in SSH for tunnelling application ports from the client machine to the server machine, or vice versa. [2]. It can be used to tunnel through the NUS firewall and establish communication between dancers' laptops and Ultra96. Port forwarding will be created from the laptop to Sunfire, and then from Sunfire to Ultra96 using the sshtunnel library in Python. Hence, it is required to tunnel twice [3].

The laptops will then create client-side TCP socket with localhost and a port number to bind to the Ultra96 server. This establishes the communication between laptops and Ultra96.

```
def start_tunnel(self, user, password):
    tunnel1 = sshtunnel.open_tunnel(
        ('sunfire.comp.nus.edu.sg', 22),  # host address for ssh, 22
        # xilinx address to bind to localhost port
        remote_bind_address=('137.132.86.236', 22),
        ssh_username=user,
        ssh_password=password,
        block_on_close=False
    )
    tunnel1.start()
    print('Tunnel into Sunfire opened ' +
          str(tunnel1.local_bind_port))
    tunnel2 = sshtunnel.open_tunnel(
        ssh_address_or_host=(
            'localhost', tunnel1.local_bind_port),
        remote_bind_address=('127.0.0.1', 8081),
        ssh_username='xilinx',
        ssh_password='xilinx',
        local_bind_address=('127.0.0.1', 8081),
        block_on_close=False
    )
    tunnel2.start()
    print('Tunnel into Xilinx opened')
    print('TUNNEL SUCCESS!')
```

Figure 4.2.1-A
Code implementation of SSH Tunneling

Data are encrypted before being exchanged by the client and server using AES protocol. Encrypted messages will then be sent to the Ultra96 server by the send_message() function.

```python
def sendSample(self):
    if self.connectionToUltra and (not self.isUltra96) and self.transmitToUltra:
        a_x = self.aX[-1]
        a_y = self.aY[-1]
        a_z = self.aZ[-1]

        g_x = self.gX[-1]
        g_y = self.gY[-1]
        g_z = self.gZ[-1]
        # message = "!D|" + f"{addr_map[self.addr]}" + "|" + "|".join(data[1].split(",")) + "|"
        msg = "!D|{id}|{a_x}|{a_y}|{a_z}|{g_x}|{g_y}|{g_z}|{ts}|".format(
            id = self.DEV_NAME_TO_EXT_COMMS_ID[self.deviceName] ,
            a_x = a_x,
            a_y = a_y,
            a_z = a_z,
            g_x = g_x,
            g_y = g_y,
            g_z = g_z,
            ts = int(round(time.time() * 10000))
        )
        self.connectionToUltra.send_message(msg)
```

Figure 4.2.1-B
Code implementation of sending data

Final Implementation/Change:
As all the dancers are connected to one laptop through a phone application, only one laptop client will be required to run on one laptop instead of one client file on each of the 3 different dancer laptops.

Overall:

1. Internal comms code will call functions from the laptop_client file.
2. Creation of portforwarding from laptop to sunfire and then the Ultra96.
3. Ultra96_server file creates socket for laptop to bind to.
4. laptop_client waits for Start packet from ultra96_server.
5. Upon start, sends raw sensor and EMG data to ultra96_server.
6. Ultra96_server file receives messages.
7. Performs timely clock synchronisation.

Message formats:

- Real-time streaming of sensor data:
    - !D | Device id | ax | ay | az | gx | gy | gz | timestamp
- EMG data:
    - !E | Device id | rms | mav | zcs | timestamp
- Predictions:
    - !M | position | action | sync delay 1 | sync delay 2 | sync delay 3

Encryption: AES (Cryptodome library in Python)

4.2.2 Communication between Ultra96 and evaluation server

Connection between the Ultra96 and evaluation server will be achieved through Transmission Control Protocol (TCP) using the Socket library in Python. The evaluation server file provided creates a socket with IP address and a particular port number. The IP address and port number will be required as input arguments (i.e., python eval_server [IP Address] [Port #] [Group #]) to bind to the socket. The server listens for the Ultra96 client connection.

The Ultra96 client creates a socket (Socket library in Python) as well with the same IP Address and port number as the socket of the evaluation server. Once the connection is established, a secret key is prompted by evaluation server, which is also used in AES Encryption Scheme to ensure a secure communication of data sending from client and receiving by server. This protocol will be further explained in section below (Section 4.2.4).

Ultra96 will determine the position and action through machine learning and the synchronization delay is calculated from timestamp data of the 3 Beetles. The client then presents the data in the plaintext format which will be encrypted and sent to the evaluation server. Upon receiving, the server decrypts the encrypted message with the same secret key and split the data up into position, action, and sync delay accordingly. If the action is 'logout', the TCP connection will be closed.

Message format: # position | action | syncdelay | (e.g., # 3 2 1 | gun | 1.2 |)
Encryption: AES (Cryptodome library in Python)

4.2.3 Communication between Ultra96 and dashboard server

Connection between the Ultra96 and dashboard server will be achieved through Transmission Control Protocol (TCP) using the Socket library in Python. The dashboard server file will create a socket with IP address and a particular port number. Like the communication between Ultra96 and evaluation server, the data will be encrypted by Ultra96 (Section 4.2.4) and decrypted by dashboard server upon receiving. The data will then be processed accordingly on the dashboard's end.

SSH Port forwarding, same as the communication between laptop and Ultra96 but with a unique port number, is also created on the dashboard client's end to connect to the dashboard server on Ultra96.

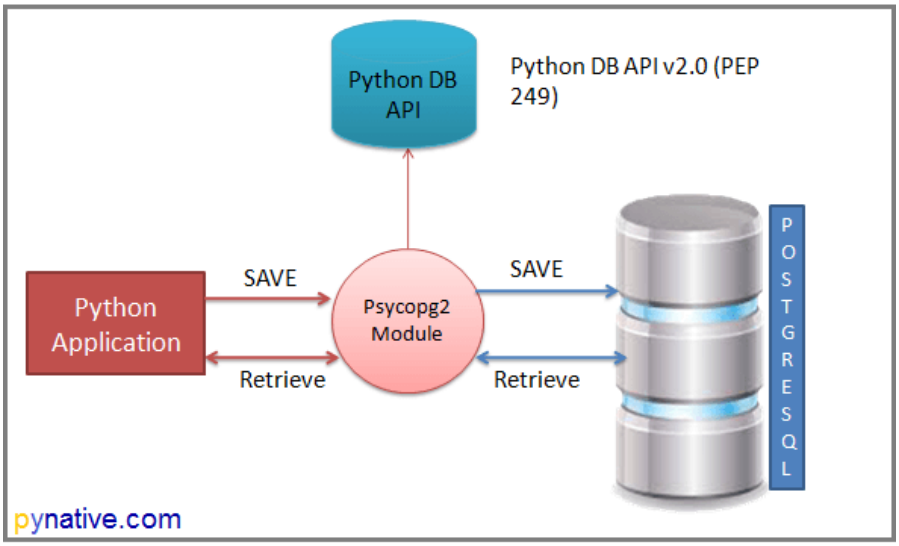


Figure 4.2.3-A
Using Psycopg2 to send and query PostSQL database

Our dashboard server, which uses a PostgreSQL database (Section 5.2.3.2), will connect with the Ultra96 through the Psycopg2 library in Python. Ultra96 will send encrypted message to the server for the server to decrypt and query for [4].

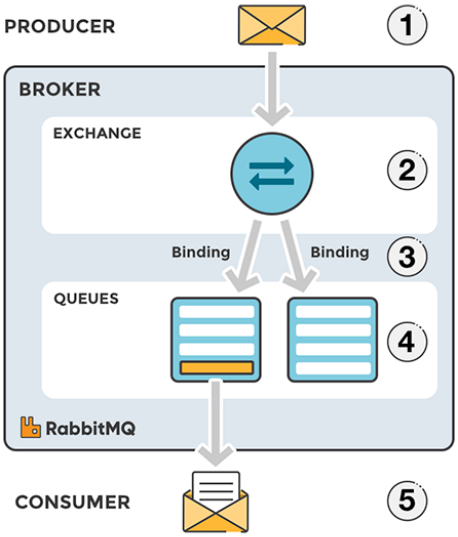


Figure 4.2.3-B
RabbitMQ messaging protocol

Another way to establish communication between the dashboard server and Ultra96 will be through a message broker, for example RabbitMQ, which is lightweight and easy to deploy on premises and in the cloud. Hence, RabbitMQ was our other option to send the message [5] to the dashboard server.

Final Implementation
We compared and tested the 2 messaging protocols: TCP and RabbitMQ.

Our main priority in the messaging protocols is that we want the transmission to be fast since we want the "coach" to be able to see the dashboard as soon as possible to assess the dancers. RabbitMQ is slower and has higher latency compared to TCP. We also found it easier to implement TCP since we can simply use the Psycopg2 library to connect to the database instead of installing and maintaining an external message broker system. While there may be packet loss in the case of TCP, we saw it as a benefit in the sense that it can help us prevent potential congestion in the streaming of real-time data.

```
class DB():
    def __init__(self):
        #Establish the connection
        self.conn = psycopg2.connect(
            database="cg4002",
            user="postgres",
            password="cg4002",
            host="localhost",
            port="5433")

        self.cur = self.conn.cursor()
```

Figure 4.2.3-C
Code implementation on connecting to Postgres database

Using TCP, we were able to ensure secure messaging with AES Encryption. The messages will be decrypted and read into the database with the psycopg2 library.

```
def insertBeetle(self, dev, time, ax, ay, az, gx, gy, gz, activation=1):
    query = "insert into Beetle(uid, time, yaw, pitch, roll, x, y, z, activation) values ('{}', {}, {}, {}, {}, {}, {}, {}, '{}')".format(dev, time, gx, gy, gz, ax, ay, az, activat
    self.cur.execute(query)
    self.conn.commit()
    print(query)

def insertEMG(self, time, rms, mav, zcs):
    query = "insert into EMG(time, rms, mav, zcs) values ({}, {}, {}, {})".format(time, rms, mav, zcs)
    self.cur.execute(query)
    self.conn.commit()
    print(query)

def insertMove(self, dev, start_time, prediction):
    query = "insert into DanceMove(uid, start_time, prediction) values('{}', {}, {})".format(dev, start_time, prediction)
    self.cur.execute(query)
    self.conn.commit()
    print(query)
```

Figure 4.2.3-D
Code implementation on inserting data into database

Message Formats:

- Real-time streaming of sensor data:
  - !D | Device id | ax | ay | az | gx | gy | gz | timestamp
- EMG data:
  - !E | Device id | rms | mav | zcs | timestamp
- Predictions:
  - !M | position | action | sync delay 1 | sync delay 2 | sync delay 3

Encryption: AES (Cryptodome library in Python)

4.2.4 Encryption Protocol

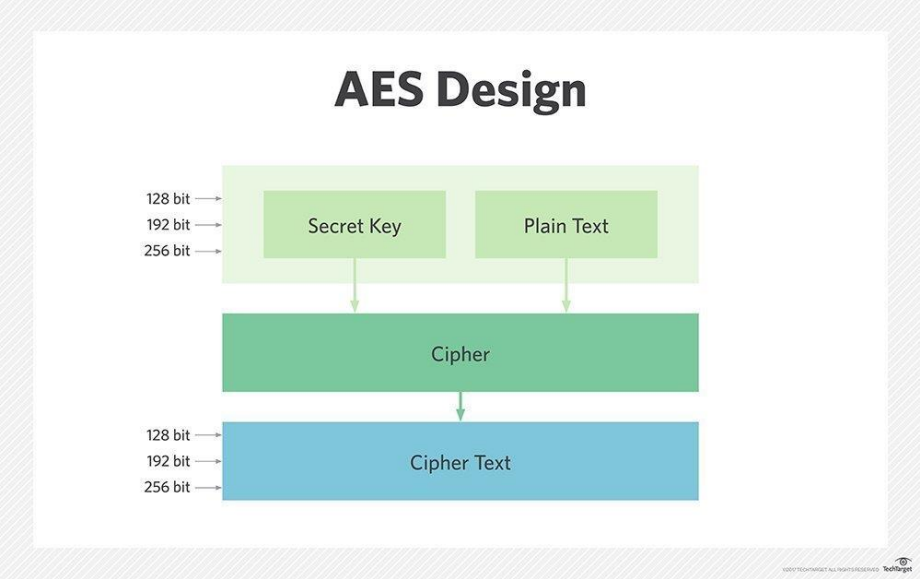


Figure 4.2.4-A
General flow of AES Implementation

AES (Advanced Encryption Standard) is a symmetric block cipher standardized by National Institute of Standards and Technology (NIST). It has a fixed data block size of 16 bytes. Its keys can be 128, 192, or 256 bits long. As AES is very fast and secure, and it is the de facto standard for symmetric encryption, it will be used to encrypt the messages exchanged in the TCP sockets [6]. We will use the Cryptodome library in Python to encrypt our messages sent and decrypt messages received. Our key will be 128 bits long even though a longer key will be more secure. This is because we want the encryption and decryption process to be fast.

An example of encryption in the Ultra96 client file and how it is achieved is presented below: The client file will encrypt the message according to what the evaluation server code (provided to us) expects to receive and decrypt.
Server expects:

- Secret key
- Message string of format – # position | action | syncdelay
- Sync delay is in milliseconds
- Base encoded message of 128-bits initial value with the message
- padding

Initial Implementation of Encryption function:

```
def encrypt_message(self, plain_text):
    plain_text = plain_text.ljust(AES.block_size," ")
    secret_key = bytes(str(self.secret_key), encoding="utf8")
    iv = Random.new().read(AES.block_size)
    cipher = AES.new(secret_key, AES.MODE_CBC, iv)
    msg = plain_text.encode("utf8")
    encoded = base64.b64encode(iv + cipher.encrypt(msg))
    # pad to buffer size
    encrypted_msg = encoded.ljust(1024, b' ')
    return encrypted_msg
```

Figure 4.2.4-B
Initial Encryption Function in eval_client.py

1. The plaintext in the format will be padded to its fixed data block size of 16 bytes.
2. Message is then encrypted with the secret key
3. IV (initialization vector or initial value) is added to the encrypted message (uses the Random library from Cryptodome as well to generate a fixed size input for additional security)
4. Message is encoded through a base64 encoding
5. Final encrypted message is padded to 1024 buffer size.

Final Implementation of Encryption function:

```
# encryption function
def encrypt_message(self, plain_text):
    plain_text = plain_text.ljust((int(len(plain_text)/AES.block_size) + 1) * AES.block_size," ")
    secret_key = bytes(str(self.secret_key), encoding="utf8")
    iv = Random.new().read(AES.block_size)
    cipher = AES.new(secret_key, AES.MODE_CBC, iv)
    msg = plain_text.encode("utf8")
    encoded = base64.b64encode(iv + cipher.encrypt(msg))
    # pad to buffer size
    encrypted_msg = encoded.ljust(1024, b' ')
    return encrypted_msg
```

Figure 4.2.4-C
Modified Encryption Function in eval_client.py

The main difference lies in the padding of the plain text. Initially, the plain text was padded to its fixed data block size of 16 bytes. However, when the length of the text exceeds 16, for example, #312|elbowkick|3.5, an error would occur as the method encrypt() (and likewise decrypt()) of a CBC cipher object expects data to have length multiple of 16 bytes for AES [7].

4.2.5 List of Ultra96 Processes

Priority	Process	Purpose	Synchronization Mechanisms
1	send_to_eval_server	Sends the position, actions and sync delay	Run a loop to constantly check for new predictions,

		predicted by the machine learning to the evaluation server.	then send to the evaluation server.
2	send_to_database	Sends the position, actions and sync delay predicted by the machine learning to dashboard database.	When new predictions are made, they will be added to a queue. This new data will be checked by a thread (thread 1) before being sent to the database.
3	recv_from_laptops	Receives Beetles' real time data of sensors and send to the database.	The data from the laptops will be added to the queue as well. This new data will be checked by another thread (thread 2) before being sent to the database.
4	time_sync	For time synchronization handshake between Ultra96 and the laptops.	It is run on another thread (thread 3) to ensure the time is synchronized across the laptops.

Since threads 2 and 3 will be accessing the queue, a lock to check the queue (and add and delete from queue) can be set in place to prevent concurrency. Lock Based Protocols is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. It helps to eliminate the concurrency problem for simultaneous transactions by locking or isolating a particular transaction to a single user. A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks help synchronize access to the database items by concurrent transactions [8].

Using the Threading library in Python, we can eliminate data corruption by using threading.Lock() to access one thread at a time.

Final Implementation
Time synchronization (Priority 4) is no longer required as an Ultra96 process since we found another way to measure the synchronization delay between the dancers. This will be further elaborated in the final implementation of Section 4.2.6.1 below. Having fewer processes also implies that there will be less overhead.

4.2.6 Clock Synchronization Protocol

Network Time Protocol (NTP) is used to synchronize time across an IP network. The NTP network generally uses a time source such as a radio or atomic clock attached to the main time server, then the NTP server distributes the time across the network [9]. Our implementation will adopt the NTP estimation of clock offset and round-trip time, with the Ultra96 as the reference clock.

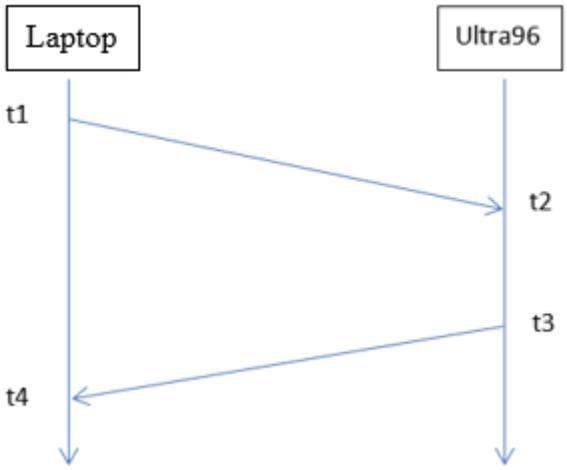


Figure 4.2.6-A
2-way handshake between laptop and Ultra96

The Laptop sends a handshake to the Ultra96, the Laptop time at t1 is recorded.
The Ultra96 receives the handshake from Laptop at t2, which is recorded.
The Ultra96 then sends back the handshake to Laptop at t3, which is recorded.
Finally, the Laptop receives back the handshake at t4, which is recorded.
These timestamps will be used to calculate the round-trip time and the clock offset.
The Round-Trip Time (RTT) is calculated as follows:

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

The clock offset (O) is calculated as follows:

$$O = t2 - t1 - (RTT/2)$$

Hence, the Laptop synchronized time will be at $t1 + RTT/2$
To minimize errors and maximize the performance of this clock synchrony, the laptops will be prompted after a certain time (to be determined) to synchronize itself.

The code snippets below show how the clock synchronization protocol is implemented from the handshake between the Ultra96 server and laptop client:

```
def start_clock_sync_all(self):
    for conn in self.socket_connection:
        self.send_message(conn, "!T|")

# Replies the clock synchronisation protocol
def clock_sync(self, conn, msg, dancer_id, recv_time):
    msg += f"{recv_time}|"
    send_time = time.time()
    msg += f"{send_time}|"
    self.send_message(conn, msg, dancer_id=dancer_id)
```

Figure 4.2.6-B
Sending of Clock Synchronization Protocol from Ultra96

```python
def clock_sync(self):
    t1 = time.time()
    clock_msg = f"!T|{t1}|"
    self.send_message(clock_msg)

    data = self.client.recv(self.buff_size)
    t4 = time.time()
    msg = self.decrypt_message(data)
    split_message = msg.split("|")
    t1 = float(split_message[1])
    t2 = float(split_message[2])
    t3 = float(split_message[3])
    self.one_way_trip_delay = ((t4 - t1) - (t3 - t2)) / 2

    self.send_message(f"O|{self.one_way_trip_delay}|")
    self.server_offset = t4 - t3 - self.one_way_trip_delay
    print(f"Offset calculated: {self.server_offset}")
```

Figure 4.2.6-C

Receiving of Clock Synchronization Protocol on laptop and calculation of server offset

4.2.6.1 Estimating Synchronization Delay

The Beetles set a flag at the start of a move, then sends flag packet with local timestamp to Ultra96 (t_1, t_2, t_3). Using the above protocol, we can calculate the clock offset of each Beetle (o_1, o_2, o_3). Hence, the synchronization delay is essentially the difference between the last timing recorded and first timing recorded, which is:

$$sync\ delay = largest\ (tm + om) - smallest\ (tn + on)$$

```python
# Calculates the sync delay, average of the first 5 time stamps
def get_sync_delay(self):
    self.sync_evaluated = True
    start_of_each_dancer = [mean(timestamps)
                            for timestamps in self.first_x_timestamp.values()]
    earliest_dancer = min(start_of_each_dancer)
    latest_dancer = max(start_of_each_dancer)
    self.sync_delay = int((latest_dancer - earliest_dancer) * 1000)
```

To account for packet drops, the average of the first 5 timestamps received of each dancer will be used to calculate the synchronization delay.

Final Implementation

While we came together to integrate the different subcomponents, we found another method of estimating synchronization delay. Once the first dance move is detected, it will start counting the number of samples until the next dancers start dancing. As we have set the sampling rate to 20Hz, the time delay can be calculated accordingly.

$$sync\ delay = (max\ stop\ time - min\ start\ time) * sampling\ period$$

We were able to get the same results for both methods. Comparing the two methods, we noticed that the former method is unstable especially when the Bluno restarts itself. Furthermore, not only does the latter method require less packets overall, it creates less overhead with fewer processes on the Ultra96, rendering it more effective.

Hence, our synchronization delay estimation is implemented based on the sampling rate.

Section 5 Software Details

Section 5.1 Software Machine Learning - *Written by Yeap Chun Lik*

5.1.1 Machine Learning Workflow

Solving problems with machine learning – in this case, classifying dance moves – begins with data representation which transforms signal data in ways that are comprehensible by the computer and learning model. The choice of input data and the type of learning algorithms would determine the model's accuracy and robustness. This is followed by a suitable evaluation method to quantify the performance of the learning models. A good classifier with high accuracy score means that it is capable of making a prediction that matches the correct true label a high percentage of the time. Hence, the evaluation stage allows us to compare the effectiveness of different learning algorithms or classifiers. Lastly, based on the prior evaluation processes, we can continually refine or omit features, and fine-tune the the learning model's hyperparameter to assess their effect on accuracy [1].

Figure 5.1.1-A
A typical machine learning workflow involving feature representation, model evaluation and optimization.

Essentially, the machine learning workflow is a cycle or iterative process in order to perfect the learning process, as illustrated in Figure 5.1.1-A. With good planning, however, we can arrive at an optimal machine learning model with minimal iterations. Such planning begins with understanding the problems and requirements, which is crucial in ensuring a well-designed and robust learning model.

The following sections outline the planning and the underlying learning process in greater detail.

5.1.2 Segmentation

Segmentation involves segregating the data into segments that share the same characteristics. An important objective of such process is for each segment to contain sufficient traits such that a complete dance move can be captured at an appropriate time interval to prevent crucial details from being omitted.

Accelerometer-based human activity recognition system with wearable sensors that uses suboptimal sampling rates would introduce high degree of inaccuracies and omitting essential movement data. Research has shown that the typical characteristic frequencies of most human activities, such as hand raising, walking and jumping are less than 10 Hertz [1]. Hence, the optimal sampling rate employed in this project would be twice of that i.e., 20 Hertz, as based on Nyquist Sampling Theorem, in order to adequately capture information to distinguish dance moves.

The signal data captured by accelerometer during the dance movements may contain noises caused by environmental, motion and dancer's behavioural factors. Hence, the signal data has to be pre-processed and de-noised in order to extract features with good fidelity, which would in turn improve the learning model's accuracy. Common pre-processing and analytical methods to be used include digital Low-Pass filter (for gravitational acceleration) and Butterworth Filter (with a cut-off frequency at 20 Hertz, as reasoned above), all of which are supported in the vast array of libraries in scikit-learn.

While a small sliding window size allows for swifter activity detection, as well as being resource and energy lite; a large sliding window size is generally used for recognising more complex motion. A research in 2014 shows that a window size of 2-5 seconds is the optimal period in terms of speed and accuracy in human activity recognition [3]. Hence, this project would consider a sliding window with a fixed length of 3 seconds as well as a 50% overlap between successive windows, as shown in Figure 5.1.2-A, for the first few stages of the project.

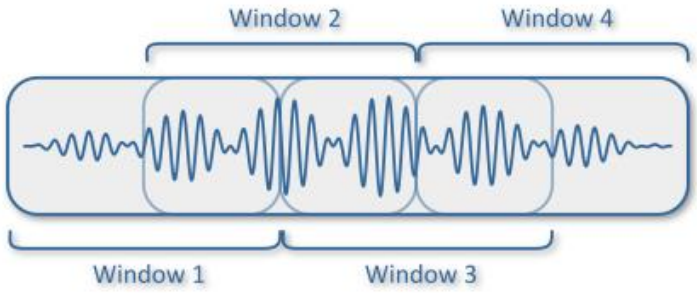


Figure 5.1.2-A
Fixed-width sliding windows technique with 50% overlap between successive windows to segregate signals.

However, should the sliding window technique prove inaccurate or computationally too expensive, this project would also explore the use of dynamic and sensor-based windowing, as well as adjusting the window width and degree of overlapping to optimise model accuracy and efficiency.

We conducted empirical study and experimentation on the available data and gathered the following information.

Through repeated trial-and-error with reference to the dancing videos, the duration of each dance move takes approximately 2 to 5 seconds to complete. Hence, a sampling rate of 20 Hertz was used, with 100 samples (5 seconds) as the sampling window to capture as much useful traits from each dance move as possible.

Each window (consisting of 100 samples) begins with an *x* number of samples (default at 20 samples) *before* the detection of a dance move, achieved by buffering the data streaming from each Bluno device. The buffer size becomes irrelevant as long as it is greater than *x*.

Our system also accommodates a sliding window with arbitrary step or arbitrary overlap. However, empirically, we noticed that a single window already produces good classification results. Subsequent windowing techniques with varying step or overlap produced no significantly better classification results, hence, our final system adopted to use only the first window, captured after a dance move has been detected.

5.1.3 Features Extraction

Selecting and extracting features that contain characteristics that are representative of the dance moves is a crucial step in the machine learning process, as they serve as inputs to the training models that would impact the accuracy of the classifier. Feature selection (or dimensionality reduction), if chosen well, is also useful if the dataset contains high-dimensional feature space. Figure 5.1.3-A illustrates a typical activity classification process based on activity signals from accelerometer.

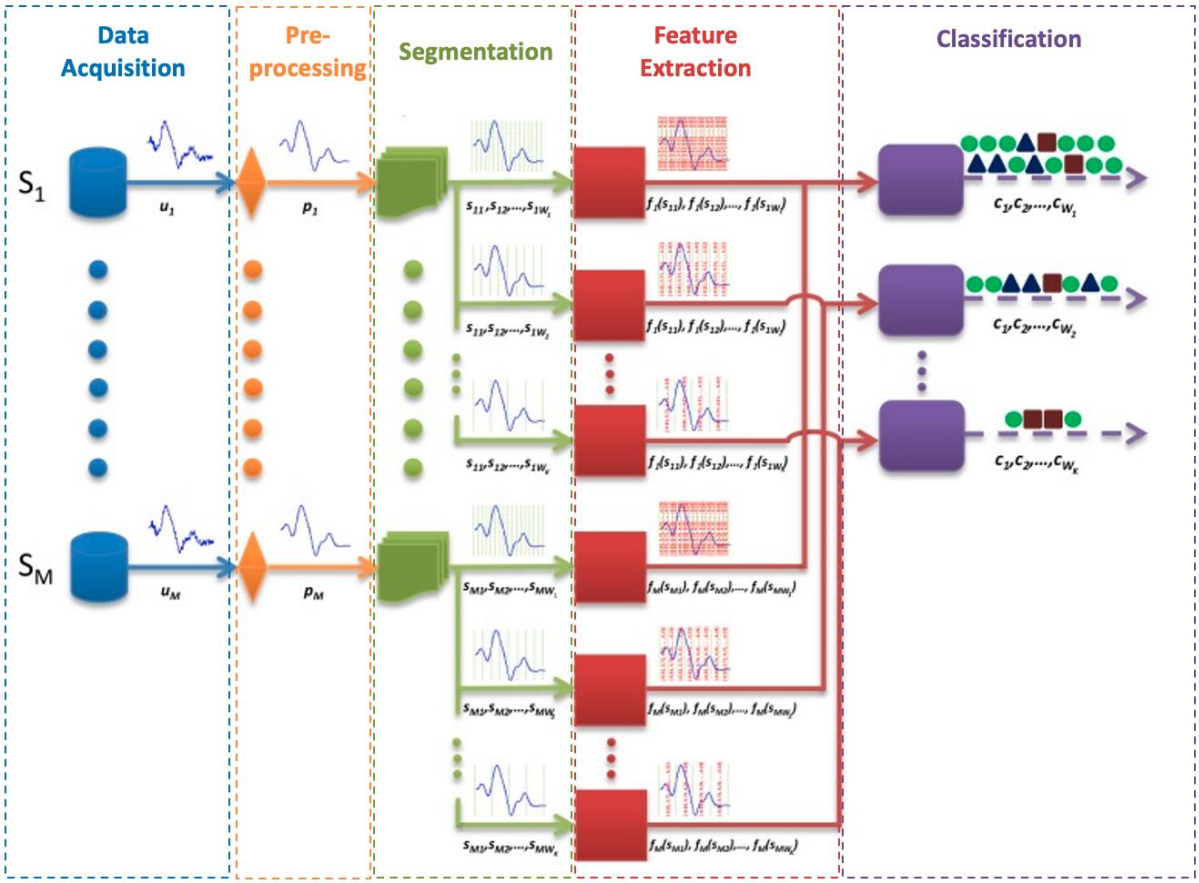


Figure 5.1.3-A

A typical activity classification process includes data acquisition, pre-processing, segmentation and feature extraction. The extracted features are then fed into the classification algorithms as inputs.

In the time domain, statistical and non-statistical analysis such as mean, median, min/max, standard deviation and area distribution will be applied on the three-dimensional signals and magnitudes in order to extract suitable features. The intensity of a movement, which is usually independent from the orientation of the dancer's limb and body, can be easily gleaned from the magnitude of x-, y- and z-values.

In the frequency domain, we can apply Fast Fourier Transform (FTT) to the time domain window to obtain the magnitude and frequency spectrum. FTT, works by sampling a signal over a period of time and divides it into its frequency components. This is followed by statistical analysis which include spectral energy, signal entropy, peak frequency, skewness, etc., to select suitable features for machine learning [4].

Kurtosis is a statistical measure that defines how heavily the tails of a distribution differ from the tails of a normal distribution. Kurtosis was used to identify whether the tails of the 100 samples in each sliding window contain extreme values, or outliers, that might skew the distribution. A high kurtosis value suggests the presence of significant outliers, resulting a heavier tail, which can be a characteristic of a particular move.

Skewness measures the degree of asymmetry in each of our sliding window which contain 100 samples. A high skewness value suggests the distribution of the 100 samples in each sampling window deviates significantly from normal distribution, which can also be considered as a characteristic of a particular move or motion.

In the final deployment, two categories of features are extracted from the raw signals captured by the accelerometer and gyroscope sensors, as shown in Table 5.1.3-A. The first category extracts the features for each of the x-, y- and z-axis from both sensors; the second category are features computed from a combination of data from each axis as well as data from the state machine flags. A total of 39 features are thus extracted for each sampling window, as shown in Table 5.1.3-A.

	Description	**No. of features**
Category 1 *(features extracted from each of the x-, y- and z-axis of accelerometer and gyroscope sensors)*	mean	6
	standard deviation	6
	variance	6
	kurtosis	6
	skewness	6
	peak-to-peak	6
Category 2 *(features computed from data in Category 1 and state machine flags)*	mean acceleration – the mean magnitude of the acceleration vector across the sampling window	1
	idle_sample_1 – the percentage of samples within the sampling window where the magnitude of the acceleration vector is above a certain threshold	1
	idle_sample_2 – the percentage of samples within the sampling window where z-axis of the accelerometer is outside a certain range of values	1

Table 5.1.3-A

A total of 39 features from two broad categories are extracted from each sampling window consisting of 100 samples for model training

Although the team explored other features such as entropy, energy band, correlation, etc., as well as other statistical characteristics in the frequency-domain, our assessment shows that these features do not meaningfully improve the classification results. In addition, adding more features would increase the clock cycles needed by FPGA which translates to higher power consumption which is an important metric of the assessment. Furthermore, Principal Component Analysis (PCA) on the 39 features revealed good clustering and separability of the data, as shown in the biplot and triplot in Figure 5.1.3-B and Figure 5.1.3-C, respectively. Hence, the 39 features in Table xx are chosen for the final training model.

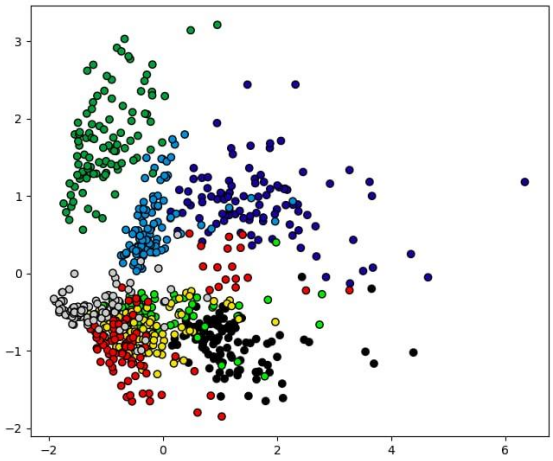


Figure 5.1.3-B
PCA biplot of the feature space reveals decent clustering of data

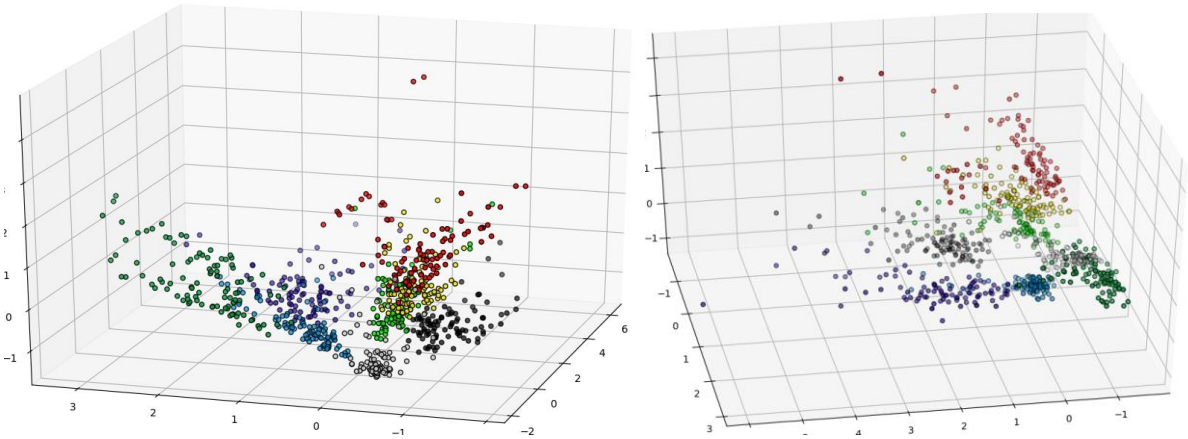


Figure 5.1.3-C
PCA triplot of the feature space shows decent separability of the data

5.1.4 Training Dataset

Our training dataset consists of 60 of each dance moves spread equally across group members and devices. We also captured the dance moves of a randomly selected group member to be separated from the training set and kept as a test set. There are 50 recorded dance data for each of the moves for training, where each recording contains a duration of approximately 7 to 10 seconds of dance data.

To increase the amount of data we have, for each recording we extracted a window of 100 samples, starting at 20 samples before the start of each dance move, as well as subsequent windows for a maximum of 300 windows per recording. This greatly increases the robustness of the model against differences in the exact moment where the start of a move is detected for different dancers. As a result, this also drastically increases the number of samples we have for training.

To further augment our data, for each window, we redistribute a randomized small amount of the acceleration in z-axis to x- and y-axis across the samples, such that the overall magnitude of the combined acceleration vector remains consistent. Doing so would also increase the robustness against variations in the way the sensor is positioned or oriented on each dancer's hand.

After the above-mentioned data augmentation processes, the total number of training and testing samples amount to 76782 and 7981 respectively, which is equivalent to 76782*5 seconds per sample, or 4.44340278 days of training data.

Model Training

5.1.6 Supervised Learning

This project would be using supervised learning models to classify and distinguish dance moves. Classical classifications such as Support Vector Machine and *k*-Nearest Neighbours would be employed to achieve said objective. In addition, this project also makes use of Feedforward Convolutional Neural Networks in identifying motions, fully leveraging the neural network acceleration power in FPGA.

The main goal of supervised learning is to learn a model from labelled training data that allows users to make predictions about new input data. The following section outlines the mechanisms of the different supervised learning models employed in this project.

5.1.7 Support Vector Machine (SVM)

SVMs are one of the best supervised learning algorithms. SVM looks for a hyperplane in an *n*-dimensional feature space that has the maximum margin between the support vectors for two or more classes, where *n* is the number of features. The different classes represent the individual dance moves that this project seeks to identify.

A maximum margin is good because it separates data points in the feature space that leads to lower generalization error [7]. An advantage of SVM is that it is robust to outliers and the optimal solution is stable. Even if our project collects more dance and motion data, as long as the support vectors remain unchanged, the SVM solutions remains intact. However, a drawback in this approach is that the support vectors would be susceptible to noise, hence, a proper de-noising technique have to be chosen.

Hard margin SVM is typically used for linearly separable dataset. However, in dataset that is nonlinear, soft margin SVMs with variable Slack constant (ξ) can be used. Another solution for nonlinear dataset is the use of kernel trick [8], as shown in Figure 5.1.7-A, to map the original feature space to a higher-dimensional feature space where a separating hyperplane can be obtained, followed by performing a linear SVM in said hyperdimensional space to classify a test instance.

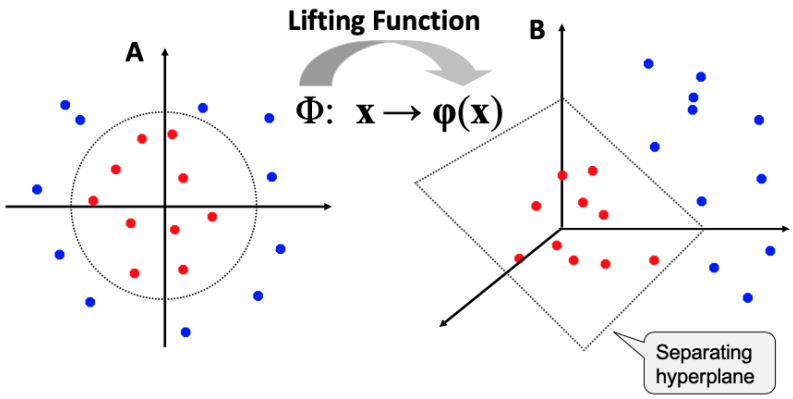


Figure 5.1.7-A:
Using kernel trick to lift the linearly inseparable feature space to higher dimensions so that a separating hyperplane can be found.

5.1.8 *k*-Nearest Neighbours (*k*-NN)

k-NN is known as an instance-based or memory-based supervised learning, which works by memorizing the labelled training dataset and use the memorized examples to classify new and unseen objects. The '*k*' in *k*-NN, which is usually odd number, refers to the number of nearest neighbours the classifier looks for in order to make its prediction. In scikit-learn, *k*-NN searches for the nearest neighbours using Euclidean distance (Minkowski metric with parameter = 2) between two points in a feature space. Once the neighbours are identified, a classification can be made by selecting the most common outcome or taking the average. However, it is also possible to assign weighting function or bias to k-NN, i.e., giving neighbours that are closest to the test data point a higher weighted score [9].

k-NN is chosen as one of the learning models to classify dance moves due to its ease of implementation and competitive accuracy among other training models. However, the challenge remains in calibrating the weighting function as well as finding the optimal *k* value as different value of *k*'s will yield different decision boundaries which may impact the classification performance, as shown in Figure 5.1.8-A.

A large *k* value will *typically* produce decision boundaries that are less fragmented and more robust to noise, as long as bias-variance trade-off is taken into account [9], however, it may be computationally expensive to perform recall.

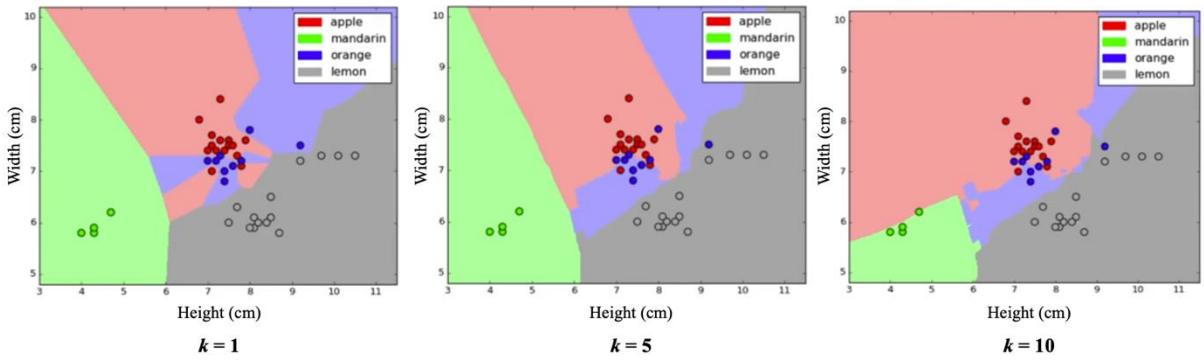


Figure 5.1.8-A
Different values of k's in k-NN yield different decision boundaries in a feature space, which affects classification accuracy.

5.1.9 Neural Networks

Neural network is part of a computational field known as Deep Learning, which in recent years have made great strides in motion recognition and thus becomes a model of choice for our project in classifying dance moves.

Classical classifiers such as SVM and Decision Tree often require extensive feature engineering, that is, developing domain-specific, sensor-specific or signal processing-specific features to analyze and process the captured sensor data. Such manual approach often results in the process being an expensive undertaking due to the need of domain expertise, as well as running the risk of over-engineering the features [10].

A key advantage of neural networks is that the features in the dataset need not be too "perfect", as neural network is capable of learning the relevant features from raw sensor data in order to make accurate predictions. In a data-rich setting, such automatic feature learning in neural network has high tolerance for imperfect features due to the large amount of data points that can be complemented. Furthermore, neural network is able to extract high-level representation in deep layer, which makes it ideal for time series classification and complex dance move recognition as required in this project [10].

The following section explores two popular neural networks techniques, namely Multilayer Perceptron (MLP) and Convolutional Neural Network (CNN) and discuss their strengths and suitability.

MLP was ultimately chosen as the model of our choice due to its ease of implementation on FPGA.

5.1.10 Multilayer Perceptron (MLP)

MLP, a variant of feedforward artificial neural network, is composed of at least three layers of nodes: an input layer, a hidden layer (note that there can be more than one hidden layer) and an output layer, as shown in Figure 5.1.10-A.

Inspired by human's nervous system, each node beyond the input layer acts as a neuron or perceptron that is triggered by a nonlinear activation function (e.g., a sigmoid function) once a threshold value is reached. MLP employs a supervised learning algorithm known as backpropagation for training and is capable of classifying data that is linearly inseparable.

Each hidden perceptron acts as feature extractors, and as the learning process proceeds, these hidden neurons gradually discover the salient features of the problem space [11].

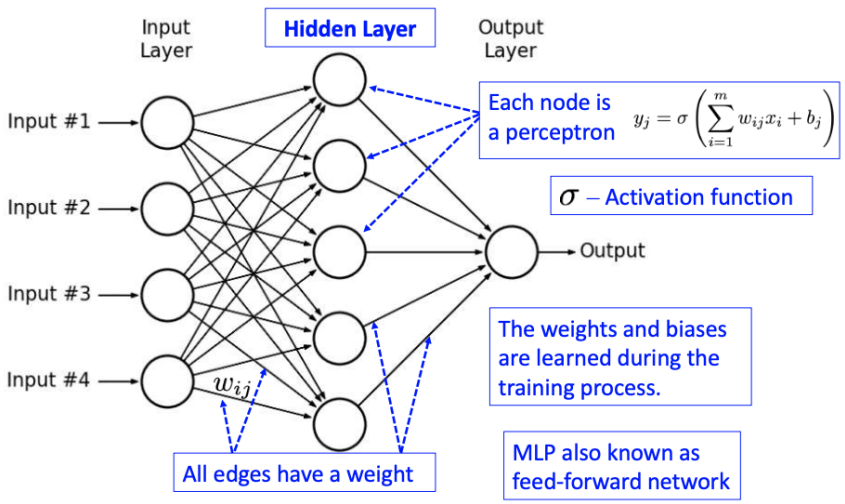


Figure 5.1.10-A
MLP structure as inspired by human's nervous system.

The general steps for MLP are as follows: apply all the data in the learning set, adjust the synaptic weights, reapply the data in the set. Repeat the process until some measure of convergence emerges. Figure 5.1.10-B depicts the MLP model structure for predicting motions in human activity recognition [12].

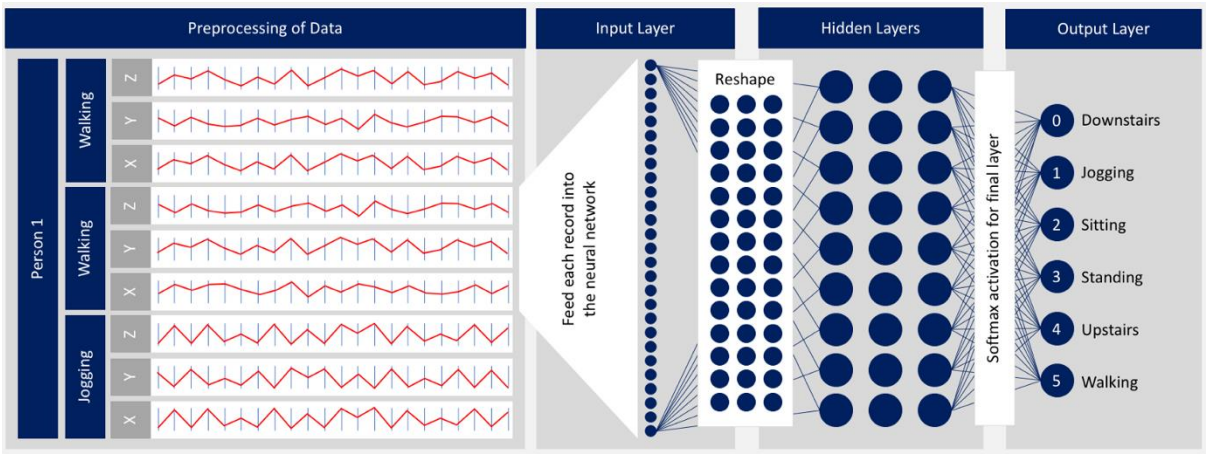


Figure 5.1.10-B
MLP model in predicting motions [12]

One major drawback of MLP is that the number of parameters can quickly become unmanageable because the layers are fully connected. All the nodes beyond the input layer in MLP are connected to one another resulting in a highly dense web, which creates redundancy and inefficiency. Hence, the convergence performance in MLP can be slow. Furthermore, the data must be reshaped and flatten before feeding as inputs to the MLP. MLP is also susceptible to local minima which would affect the training process.

5.1.11 Model Architecture Deployed in Final Evaluation

Due to the each of implementation on FPGA, MLP is chosen as the neural network model for various stages of assessment, including the final evaluation. The MLP architecture is shown in Figure 5.1.11-A, where it consists of four layers: input layer (39 nodes, corresponding to 39 features extracted), two hidden deep layers (consist of 16 nodes each) and finally, an output layer that has 8 nodes that corresponds to the 8 dance moves. It should be noted that the learning model is very small, with only 1048 parameters.

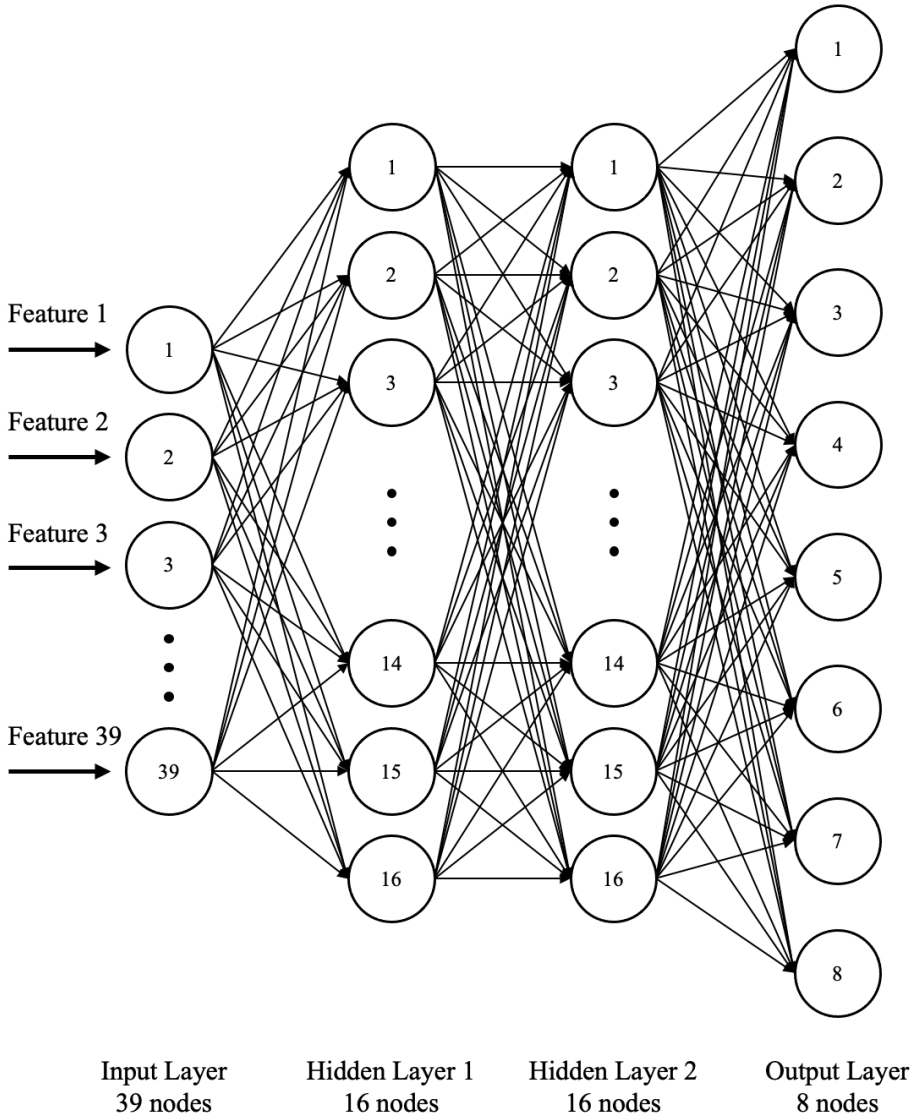


Figure 5.1.11-A
MLP architecture used in this project. It consists of the input layer (39 nodes), two hidden layers (16 nodes each) and an output later (8 nodes)

The model is trained with a learning rate of 0.0001 where the 5-fold cross validation yielded an average 99% validation accuracy on 200 epochs with Adam. Therefore, we are confident that this relatively small model architecture is able to generalise dance moves very well based on the selected feature vectors.

The final deployed model is trained without *k*-fold cross-validation to make use of more data for training. The final model produced a test accuracy of 98.4% on unseen test data.

The training histories for the model accuracy and model loss across all 200 epochs are shown in Figure 5.1.11-B and Figure 5.1.11-C, respectively

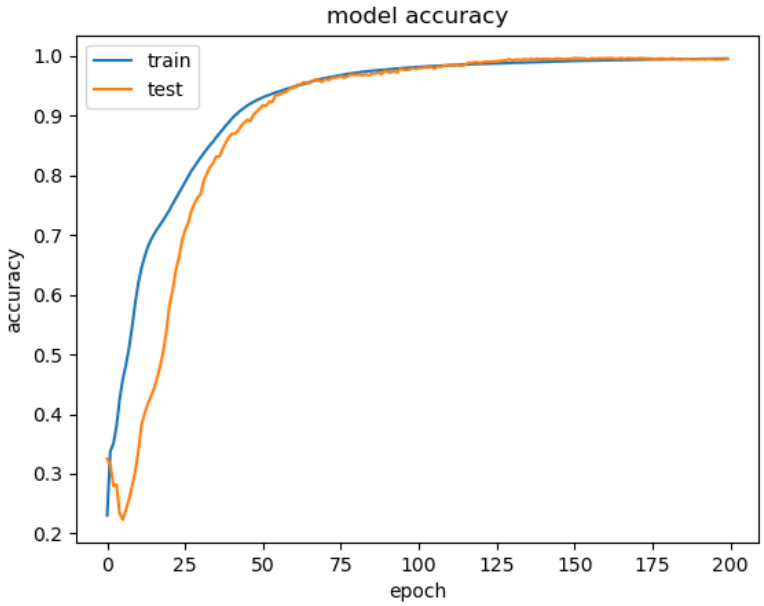


Figure 5.1.11-B
history of training accuracy across 200 epochs

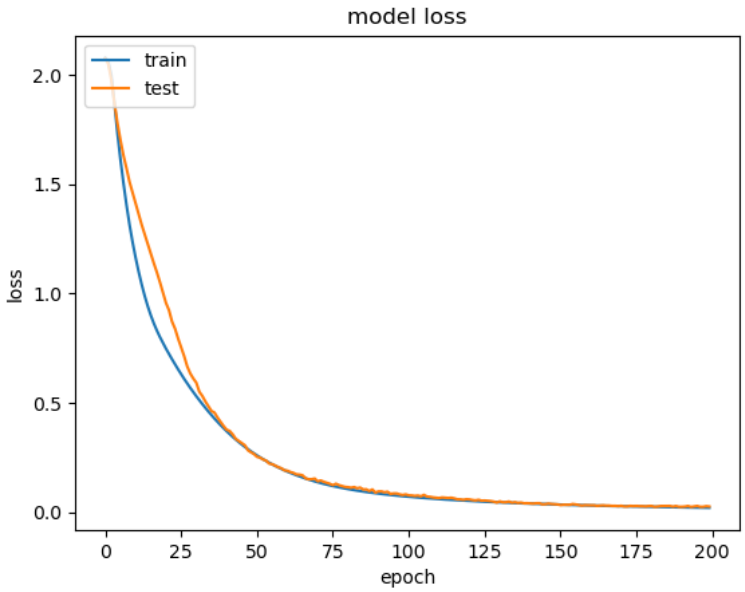


Figure 5.1.11-C
history of model loss across 200 epochs

Confusion Matrix and Classification Report of the above model is detailed in Section 5.1.16

5.1.12 Convolutional Neural Networks (CNN)

CNN is a variant of deep neural networks which if comprised of two main types of elements: convolutional layers and pooling layers, as shown in Figure 5.1.12-A. These two layers can be repeated at depth, providing multiple layers of abstraction the input signals [10]. Hence, CNN is capable of performing specialized functions such as feature extraction, subsampling, scaling, and shifting, including recognizing two-dimensional shapes such as the sensor data from accelerometer and gyroscope employed in this project

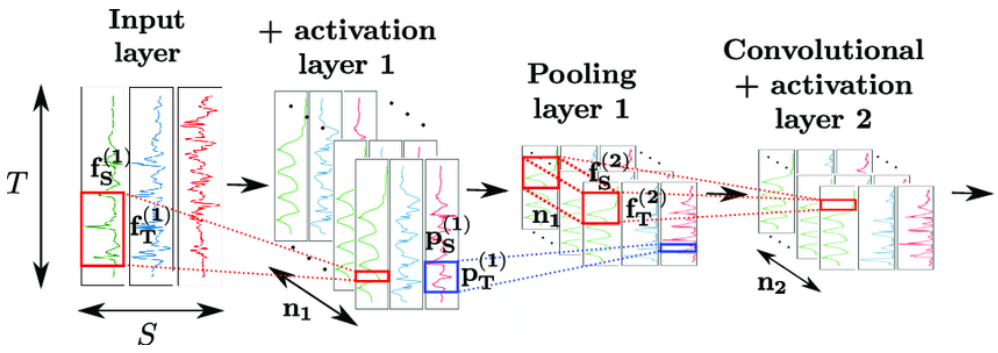


Figure 5.1.12-A

CNN architecture for sensor-based human activity recognition. T, S, and n_k denotes input sensor data's time length, number of sensor channels and convolutional kernels of the k^{th} layer, respectively [13]

CNN, when used on HAR, learns to map a given window of signal data of time length T to a motion where the learning model interprets across each window of data, followed by preparing an internal representation of the window [13].

CNN is more superior than MLP in multiple fronts. Firstly, the weights are lighter and shared, resulting in less wastage and ease of training as compared to MLP. Furthermore, the convolutional and pooling layers can penetrate deeper into the hidden layers, where the layers are connected in a lesser degree rather than fully connected.

However, due to the technical challenge in implementing CNN on the FPGA, MLP is chosen as the final evaluation model instead.

Model Training

5.1.14 Supervised Learning – Model Training

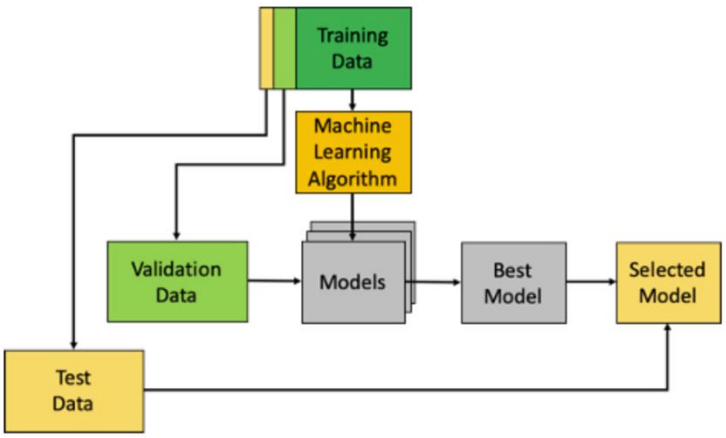


Figure 5.1.14-A:

Supervised learning paradigm, where the labelled data are partitioned to inform and assess the learning model's performance

Supervised Learning model training typically first divides the available labelled dataset into three sets [1], as shown in Figure 5.1.14-A:

- Training set: used for model parameter optimization;
- Validation set: used for hyperparameter tuning, gauge generalisation error and model selection;

- Test set: used only for final evaluation of the trained model and is performed after the training and validation processes are completed

In scikit-learn, this can be done by invoking the Train/Test split function which partitions the dataset in training and test sets, followed by calling the Fit Method on the training set to estimate the learning model. And finally applying the learning model by using the Predict Model to classify unseen data instances, or by using the Score Method to evaluate the trained learning model's performance on the test set.

The objective of dividing the original data into training and test sets is to use the test set as a way to estimate how well the model trained on the training data would generalize to new unseen data. The test set represented data that had not been seen during training but had the same general attributes as the original data set, i.e., the test set is drawn from the same underlying distribution as the training set [1].

5.1.15 *k*-fold Cross Validation

Cross-validation is a technique that extends beyond evaluating a single model. It conducts multiple Train/Test splits of the data, each of which is used to train and evaluate a separate model. This is useful because if the random state seed parameter in the Train/Test split function is varied, the learning model's accuracy score will be positively or adversely affected, depending on the specific samples that the training set uses. Hence, cross validation provides a more stable and reliable estimates on how the learning model is likely to perform on average by running multiple distinct Train/Test splits and averaging the results [14].

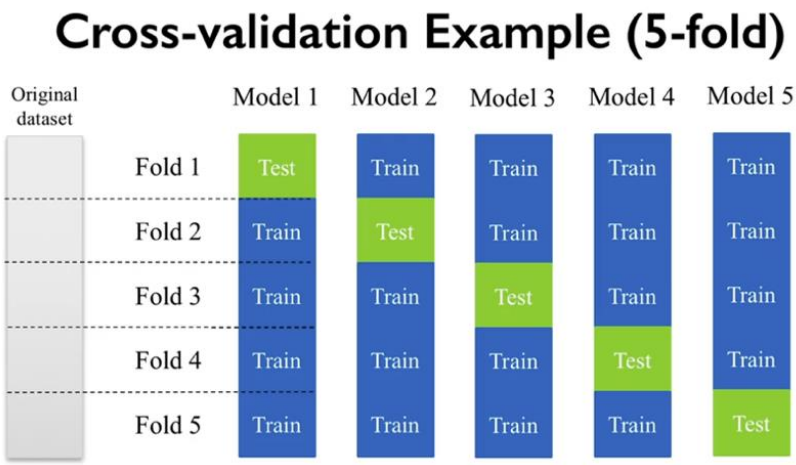


Figure 5.1.15-A
5-fold Cross Validation visualisation

The most common type of cross-validation is *k*-fold cross-validation where *k* is typically set to 3, 5 or 10. For example, to do five-fold ($k = 5$) cross-validation, the original dataset is partitioned into five segments (or "folds") of equal or close to equal size, followed by a series of five training models that is performed on each fold. For example, in Figure 5.1.14-A, Model 1 uses fold 2 to 5 as the training set and is being evaluated using fold 1 as the test set. A similar process is applied to Model 2 to 5. When all five processes are completed, their accuracy scores (five in total) corresponding to the fold are generated.

It's typical to then compute the mean of all the accuracy scores across the folds and report the mean cross-validation score as a measure of how accurate we can expect the model to be on

average. In scikit-learn, we can make use of the *cross_val_score* function to do cross-validation which reports the average performance over different experiments.

One advantage of computing the accuracy of a model on multiple splits instead of a single split, is that it gives us potentially useful information about how sensitive the model is to the nature of the specific training set. We can look at the distribution of these multiple scores across all the cross-validation folds to see how likely it is that by chance, the model will perform very badly or very well on any new data set, so we can get a sense of the worst-case or best-case performance estimate from these multiple scores. This extra information is computationally heavy, however, the increase in knowledge about how the model is likely to perform on unseen data is usually well worth the cost [14].

5.1.15 Data Leakage

Data leakage occurs when the test set (or validation set) leaks information to the model, giving rise to optimistic performance prediction and possibly invalidates the entire experiment. If pre-processing of data, e.g., normalization, is required, it should be done on the training set and not the entire dataset. For example, if normalization is performed on the test set, then information about the test set might be leaked into the training set and influences the model [15].

In the case of *k*-fold cross validation, each model must be discarded and restarted with a new test set after each experiment in order to prevent data leakage that might influence the learning model's structure and optimization of parameters.

We are also careful by segregating training and test set of the dance dataset from the beginning and prior to any feature extraction and data augmentation to prevent information of the test data being leaked to the training model.

Accuracy of a Classifier

5.1.16 Confusion Matrix

To choose the best classifier, we can use objective and quantifiable metrics to assess the effectiveness and performance of the learning model. This can be done via a technique known as Confusion Matrix – a table with four different combinations of predicted and actual values, namely true positive (TP), false positive (FP), false negative (FN) and true negative (TN). These four values are then used to derive useful performance measurement metrics such as sensitivity, specificity, precision, negative predictive value, accuracy and more importantly, the AUC-ROC Curve [16]. Figure 5.1.16-A shows a Confusion Matrix and the relationships between the different metrics.

Actual Class \ Predicted Class	Positive (P)	Negative (N)	
Positive (P)	True Positive (TP)	False Negative (FN)	**Sensitivity** (True Positive Rate) $\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}} = \frac{\mathrm{TP}}{\mathrm{TP+FN}}$
Negative (N)	False Positive (FP)	True Negative (TN)	**Specificity** (True Negative Rate) $\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN+FP}}$
	Precision (Positive Predictive Value) $\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP+FP}}$	**Negative Predictive Value** $\mathrm{NPV} = \frac{\mathrm{TN}}{\mathrm{TN+FN}}$	**Accuracy** $\mathrm{ACC} = \frac{\mathrm{TP+TN}}{\mathrm{P+N}} = \frac{\mathrm{TP+TN}}{\mathrm{TP+TN+FP+FN}}$

Figure 5.1.16-A

Confusion Matrix table and the relationship between different metrics. The row of the matrix represents the instances of an actual class, whole the column represents the instances in a predicted class.

A TP is an outcome where the learning model correctly predicts the positive class as positive. Similarly, a TN is an outcome where the learning model correctly predicts the negative class as negative.

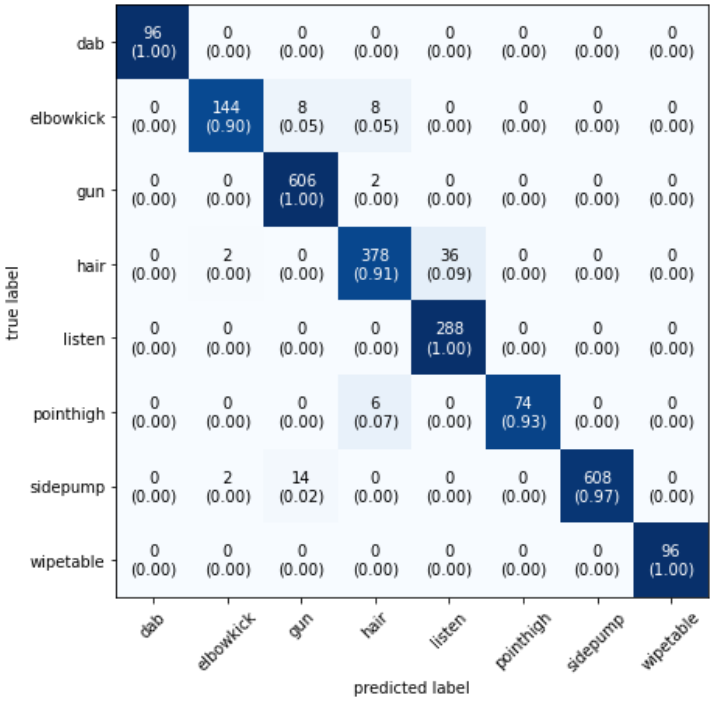
A FP is an outcome where the learning model incorrectly identifies a negative class as positive. Similarly, a FN is an outcome where the learning model incorrectly identifies a positive class as negative

Accuracy is the ratio of the number of correct predictions over the total number of predictions made. In other words, it measures the proportion of the total number of predictions that turn out to be correct.

Sensitivity, or known as recall, measures the positive examples that are correctly identified as positive by the learning model. Specificity, on the other hand, measures the negative examples that are correctly identified as negative by the learning model. The use of recall and specificity depends on the scenario and the consequences they could engender. For example, recall is more important when false negatives are catastrophic while specificity is more crucial when false positives are unacceptable [16].

Precision is an important metric when being right (positive prediction is correct) outweighs detecting all positives. F1-score (Eqn. 1) is the harmonic mean or weighted average of precision and recall, which is used when both precision and recall are important and a balance is required.

$$F_1 = \frac{2\mathrm{TP}}{2\mathrm{TP} + \mathrm{FP} + \mathrm{FN}} \quad \text{---------- (Eqn. 1)}$$



	precision	recall	f1-score	support
dab	1.00	1.00	1.00	96
elbowkick	0.97	0.90	0.94	160
gun	0.96	1.00	0.98	608
hair	0.96	0.91	0.93	416
listen	0.89	1.00	0.94	288
pointhigh	1.00	0.93	0.96	80
sidepump	1.00	0.97	0.99	624
wipetable	1.00	1.00	1.00	96
accuracy			0.97	2368
macro avg	0.97	0.96	0.97	2368
weighted avg	0.97	0.97	0.97	2368

Figure 5.1.16-B
Confusion Matrix and Classification Report of the training model

Figure 5.1.16-B shows the Confusion Matrix and its corresponding Classification Report of the evaluation model. The 8 dance moves achieved a relatively high *f1*-score which indicates good classifier robustness. However, elbow kick and hair have a relatively lower recall and precision score, which can be attributed to their similarity in motion that resulted in false classifications.

5.1.17 AUC-ROC Curve

AUC-ROC curve, as shown in Figure 5.1.17-A, is often employed to study or visualise the performance of the multi-class classification problem, or in this project, predicting the dance moves [17].

ROC (receiver operating characteristics) Curves is a graph showing the performance of a classification model at different probability thresholds. The curve, which plots TP rate against FP rate, summarises the trade-off between a learning model's recall and false positive rate using different threshold settings. It is often used when the observations are balanced between each class.

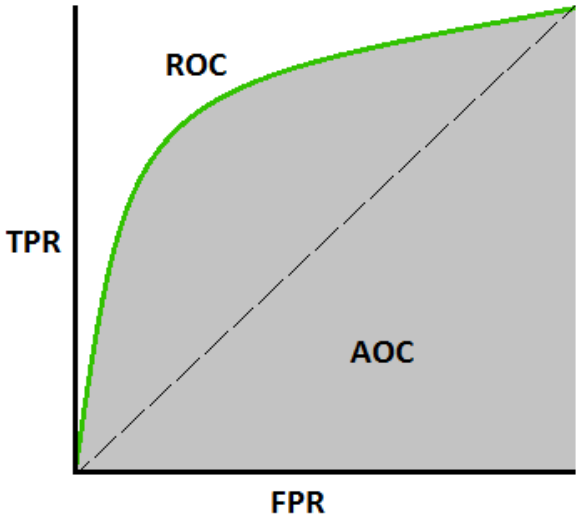


Figure 5.1.17-A
AUC-ROC Curve [17]

AUC (area under the curve) can be used as a summary of the trade-off or viewed as the degree of separability. It shows how effective a learning model is capable of predicting between classes.

Typically, an AUC-ROC value that is closest to 1 indicates a good learning model.

5.1.18 Generalization, Overfitting and Underfitting

Generalization refers to the learning model's ability to provide accurate predictions for new and unseen data. Learning models that are too complex for the amount of training data are said to be overfitting and are unlikely to generalize well to unseen data instances. The inadequate amount of training set constrains the model in seeing the global pattern thus limiting its generalization ability. On the contrary, models that are too simple that do not even do well on the training data, are said to be underfitting and are also unlikely to generalize well [18].

Figure 5.1.18-A shows the difference scenario when a learning model is underfitting, overfitting and well-fitting. The optimal strategy for supervised learning is to develop a model that has optimal fit such that it has a good generalization performance.

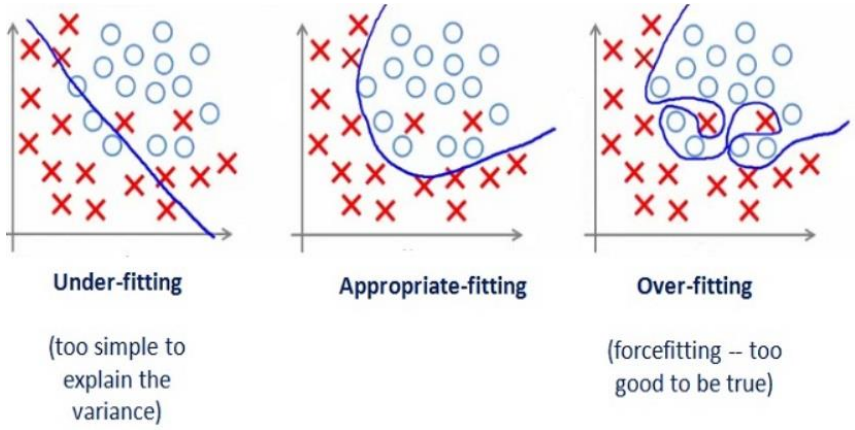


Figure 5.1.18-A
Examples of underfitting, overfitting and optimal fitting [18]

5.1.19 Packages and Libraries used for Model Training

The following packages and libraries will be explored to support this project's learning model algorithms. The list is not exhaustive.

- Anaconda (version 1.10.0)
- Keras (version 2.4.0)
- TensorFlow (version 2.4.1)
- Theono (version 1.0.5)
- Lasagne
- scikit-learn
- numpy, etc.

5.1.20 Predicting Dancers

Beyond classifying the dance moves, the team also explored if a dancer can be accurately identified based on the available training data. The model, detailed in the attached *Dancer Prediction – Final.ipynb,* achieved an average 98% accuracy after 5-fold cross validation. In other words, the system is fully capable of telling which dancer is performing the dance moves. This raises interesting privacy implications and is further discussed in Section 6.3.4 Ethical Impact.

Section 5.2 Software Dashboard - *Written by Liu Chaojie*

5.2.1 Dashboard Architecture.

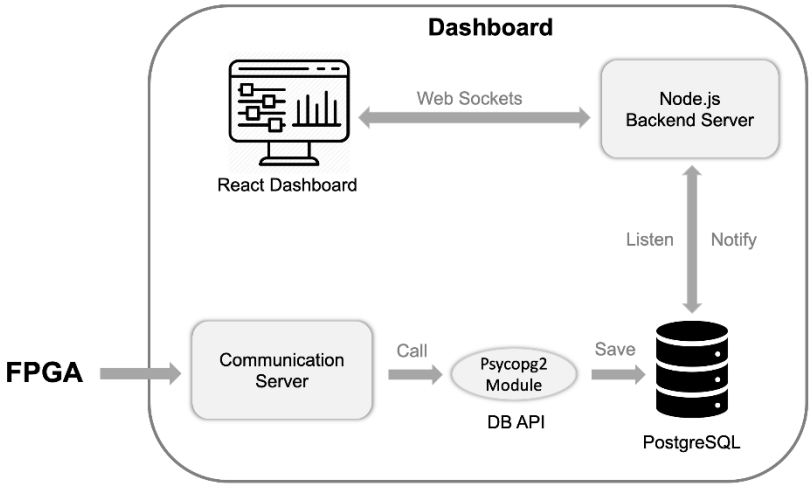


Figure 5.2.1-A
Architecture of Dashboard Subsystem

The Figure 5.2.1-A illustrates the Dashboard subsystem architecture and its interface with External Comms.

The communication server receives sensor data from FGPA and saves them to database via Psycopg2, a Python PostgreSQL adapter [1], which exposes database API to the communication server. Once the database detects new data, it will notify the Node backend server of data updates. This backend server works as a middleman pushing data from database to dashboard via socket. Then, the dashboard processes incoming data and renders the real-time display.

5.2.1.1 Tech Stack of the Dashboard

Figure 5.2.1.1-A
Tech stacks for dashboard

The database is PostgreSQL database, and the frontend is powered by React. The mid-tier server that connects database and frontend is powered by Node.

5.2.1.2 Separating Two Servers for Decoupling

Lower coupling is one of the characteristics of structured architecture design and good programming practices [2]. It requires that components with different responsibilities should be implemented separately. In our team, External Comms and Dashboard subsystems can be developed and optimized independently.

The dashboard laptop runs two separate servers to support data streaming. One is the communication server that communicates with FPGA and saves incoming data into database. Another one is the Node backend server that is a middleman for message transmission between database and dashboard. Two components with such different functionalities are separated for decoupling.

Moreover, the two servers are in different programming languages. The communication stack uses Python while the web stack uses JavaScript. They are separated also to avoid overhead on potential language conflicts.

5.2.2 Dashboard Design

The application supports three pages: Dashboard, Sensors and Records. Their descriptions are tabulated below:

Page	Description
Dashboard	Presents synchronization, prediction results and muscle fatigue in user-friendly UI.
Sensors	Displays sensor data streaming for all sensors in real-time.
Records	Lists completed dance sessions in recent days. Provides offline analytics for each session.

As in Figure 5.2.2-A, all pages can be easily accessed by their tabs in the navbar.

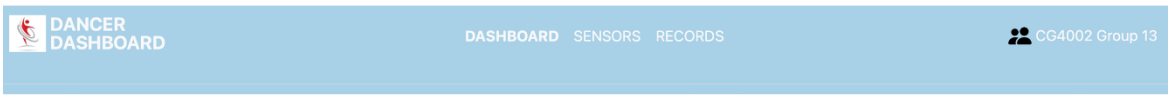


Figure 5.2.2-A
Application Navbar

5.2.2.1 The Dashboard Page

Figure 5.2.2.1-A
The Dashboard Page

The Figure 5.2.2.1-A gives an overview of the Dashboard Page. The page consists of synchronization, positions, muscle fatigue, speed ranking and dancer panels. Details of each part are explained below.

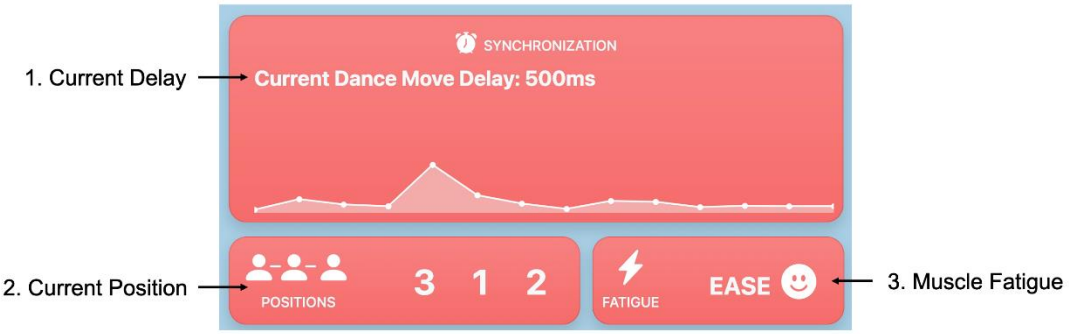


Figure 5.2.2.1-B
The Upper Left Part of Dashboard Page

Each marked component in Figure 5.2.2.1-B is tabulated here:

No.	Name	Description
1	Dancers Delay	Difference between the earliest dancer and the latest dancer who starts the current dance move.
2	Dancer Positions	Predictions of dancers' current positions.
3	Muscle Fatigue	Fatigue level of the dancer's arm muscle. It has three levels: ease, medium and hard.

Figure 5.2.2.1-C
Speed Ranking of Dashboard Page

Each marked component in Figure 5.2.2.1-C is tabulated here:

No.	Name	Description
1	Dancer ID	The ID used to identify three dancers.
2	Dancer Delay	The difference between the dancer's start time and the team's earliest start time of the dance move.

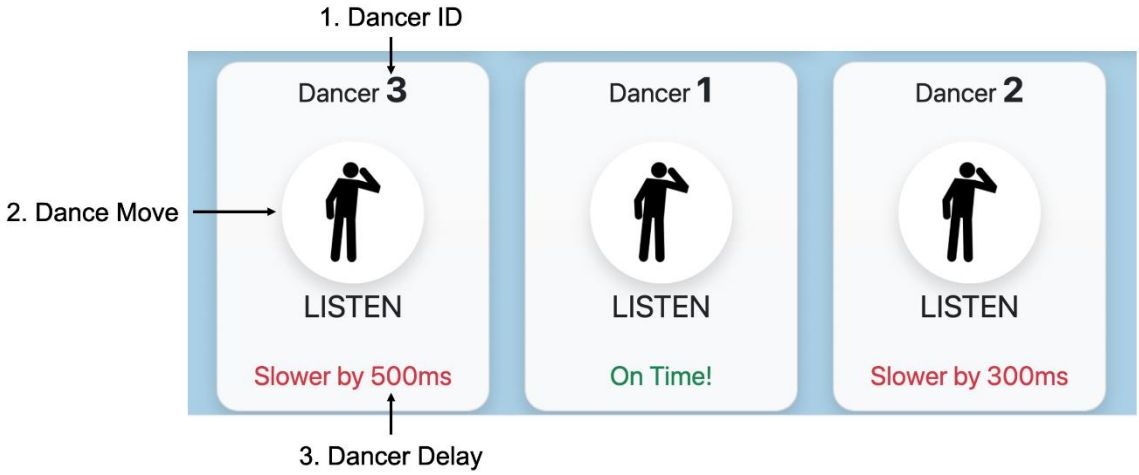


Figure 5.2.2.1-D
User Panels of Dashboard Page

Each marked component in Figure 5.2.2.1-C is tabulated here:

No.	Name	Description
1	Dancer ID	The ID used to identify three dancers.
2	Dancer Move	Prediction of the dancer's current dance move.
3	Dancer Delay	The difference between the dancer's start time and the team's earliest start time of the dance move.

5.2.2.2 The Sensors Page

Figure 5.2.2.2-A
An overview of the Sensors Page

Figure 5.2.2.2-B
Sensor Data Streaming

Each marked component in Figure 5.2.2.2-B is tabulated here:

No.	Name	Description
1	Dancer ID	The identifier of the dancer.
2	Dancer Move	Prediction of the dancer's current dance move.
3	Accelerometer Data	Real-time streaming of accelerometer data in the line chart.
4	Gyro Data	Real-time streaming of Gyro sensor data in the line chart.
5	EMG Data	Real-time streaming of EMG data in the line chart.
6	Muscle Fatigue	Fatigue level of the dancer's arm muscle. In sensors page it has three levels: low, medium and high.

5.2.3 Offline Analytics

5.2.3.1 Sessions in the week

The Records page shows sessions completed in recent days and a trendline recording the dancing frequencies in rec, as illustrated in the Figure 5.2.3.1-A.

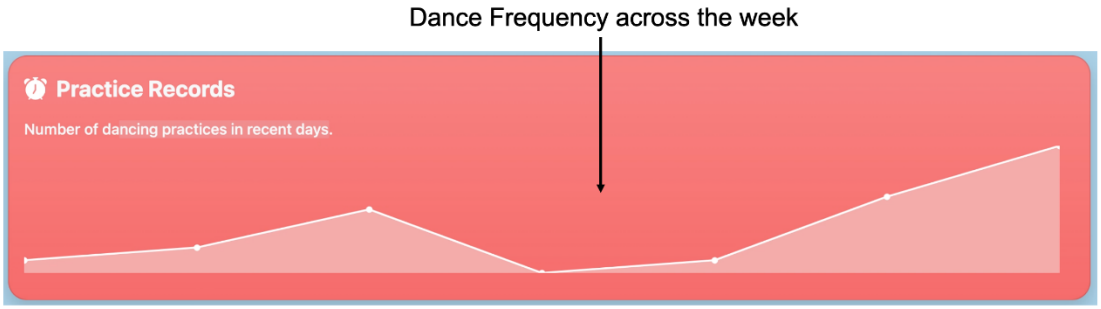


Figure 5.2.3.1-A
Dancing Frequency across the Week

User can view the detailed analytics of a session by clicking the corresponding session number in the Records page.

5.2.3.2 Summary of a session.

As illustrated by Figure 5.2.3.2-A, the summary consists of three labels.

Figure 5.2.3.2-A
The Upper Part of Summary Page

No.	Name	Description
1	Number of Dancers	The total number of participants in the session.
2	Highest Fatigue	The highest muscle fatigue reached during the session.
3	Number of Moves	The total number of moves during the session.

5.2.3.3 Checking accuracy with ground truth

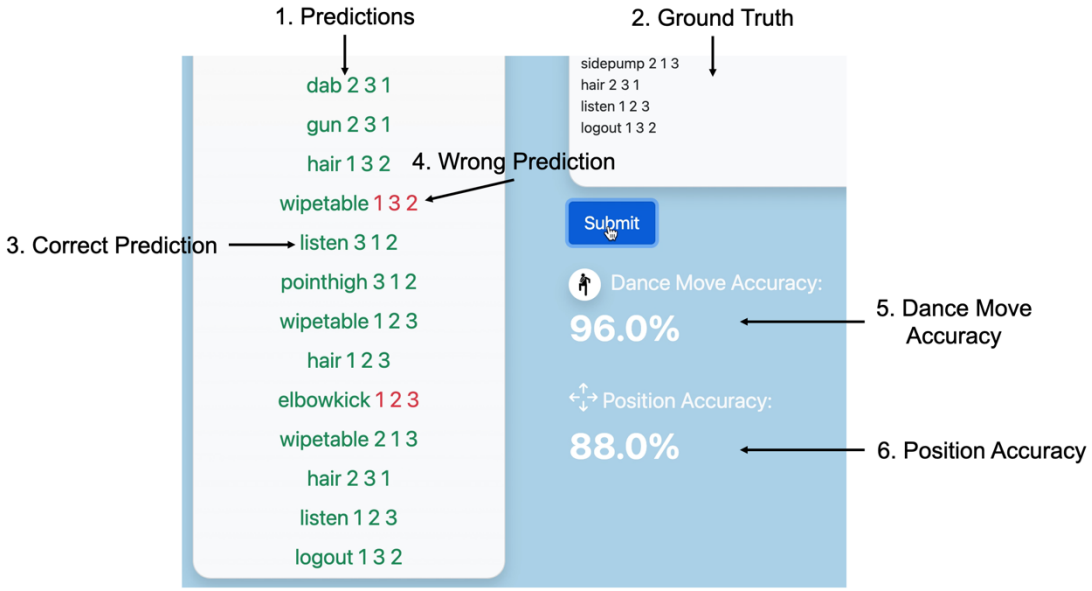


Figure 5.2.3.3-A
Checking Accuracy

User can input current session's ground truth into the textbox and click the button. The system will find out correctly classified and misclassified predictions in the left. As shown in Figure 5.2.3.3-A, both dance move accuracy and position accuracy will be calculated.

5.2.3.4 Analytics for each dancer

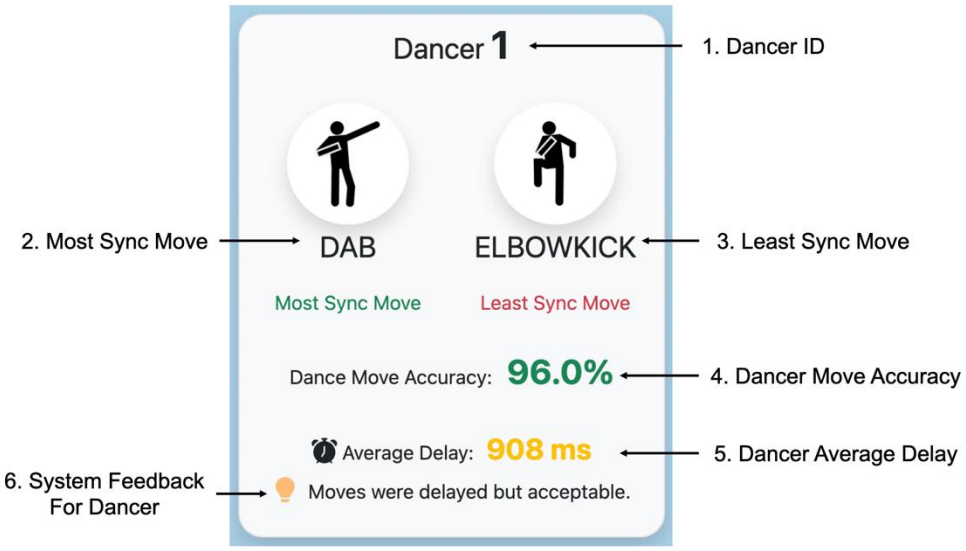


Figure 5.2.3.4-A
Dancer Analytics

As shown in Figure 5.2.3.4-A, dancers can see detailed analytics of their performance and the feedback/instructions provided by the system.

No.	Name	Description
1	Dancer ID	The move with the earliest time offset by the dancer.
2	Most Sync Move	The move with the fastest response of the dancer. This indicates that the dancer is familiar and practiced for this dance move.

3	Least Sync Move	The move with the longest response time of the dancer. This indicates that the dancer may be unfamiliar with this move and more practice is needed.
4	Dancer Move Accuracy	Number of correct dance / total number of dance move of the dancer * 100%.
5	Dancer Average Delay	The average move delay of dancer during the session. This helps dancers decide whether they should catch up or wait for their peers for better synchronization.
6	System Feedback	Instructions provided by system for the dancer.

5.2.3.5 Synchronization analytics of the session

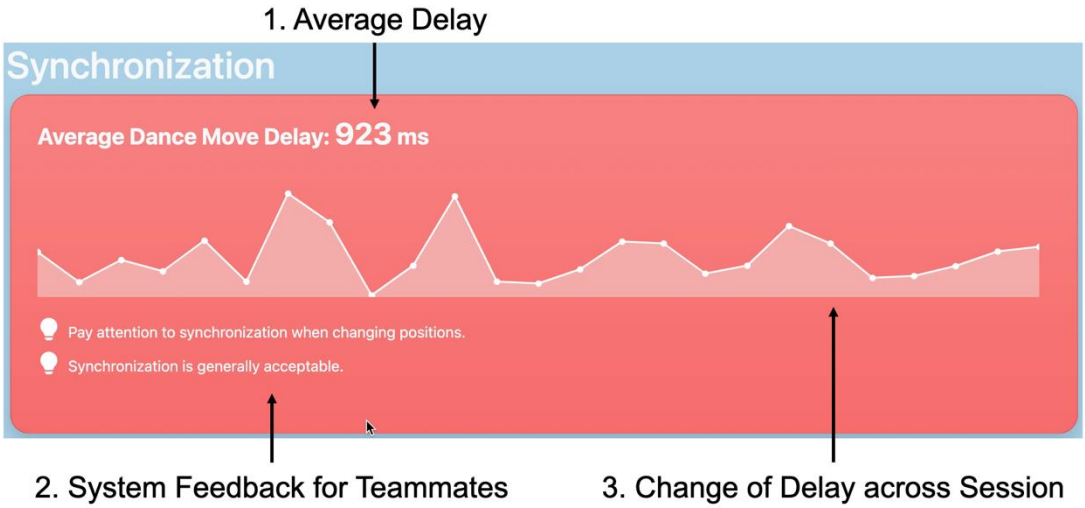


Figure 5.2.3.5-A
Synchronization Analytics

As shown in Figure 5.2.3.5-A, average dance synchronization is calculated for the team. Teammates can also see how their synchronization changed across the entire session. Based on that, system provides instructions for teammates on how to make improvement.

5.2.4 Storage of incoming sensor data

The incoming sensor data and prediction results are received by the communication server and then immediately saved to PostgreSQL database via Psycopg2.

5.2.4.1 Rationales of PostgreSQL over MongoDB

Both PostgreSQL and MongoDB satisfy the requirement. PostgreSQL is a well-built, robust relational database with older age. It requires a structured schema and follows ACID (Atomicity, Consistency, Isolation, Durability) compliance for storage minimization and transaction stability [4]. MongoDB is modern database which supports direct storage of JSON instead of requiring a schema [5]. It is well-designed for big data and can be highly distributed.

Our project does not utilize many advanced features of MongoDB. Specifically, we only store small amount of data without demands of distributed database. MongoDB's JSON-format storage brings no more convenience because the External Communication uses

Python. Moreover, both my teammate and I have prior knowledge in PostgreSQL. We finally decided to use PostgreSQL as our database.

5.2.4.2 Schema Implementation

The Beetle table stores Gyro and Accelerometer sensor data. Each row represents a set of data at one timestamp for one dancer.

```
create table Beetle
(
    uid        uid_t,
    local_time timestamp default CURRENT_TIMESTAMP,
    time       bigint,
    yaw        numeric not null,
    pitch      numeric not null,
    roll       numeric not null,
    x          numeric not null,
    y          numeric not null,
    z          numeric not null,
    activation activation_t not null,
    primary key (uid, local_time)
);
```

The EMG table stores EMG data processed by Hardware Sensor component, namely RMS (Root Mean Square), MAV (Mean Absolute Value) and ZCR (Zero Cross Rate). Each row represents a set of data at one timestamp.

```
create table EMG
(
    time       bigint,
    local_time timestamp default CURRENT_TIMESTAMP primary key,
    rms        numeric not null,
    mav        numeric not null,
    zcr        numeric not null
);
```

The DanceMove table stores dance move prediction results and synchronization delays. Each row represents a set of data at one stamp for all three dancers.

```
create table DanceMove
(
    local_time       timestamp default CURRENT_TIMESTAMP,
    start_time       bigint,
    start_time_one   integer,
    start_time_two   integer,
    start_time_three integer,
    prediction       dance_move_t not null,
    primary key (local_time)
);
```

The DancePosition table stores position prediction results. Each row represents a set of data at one timestamp for all three dancers.

```
create table DancePosition
(
    start_time  bigint,
    local_time  timestamp default CURRENT_TIMESTAMP primary key,
    left_slot   uid_t not null,
    middle_slot uid_t not null,
    right_slot  uid_t not null
);
```

The Session table stores start time and end time of a session. This is utilized for offline analytics per session in the Records page.

```
create table Session
(
    start_time bigint,
    local_time timestamp default CURRENT_TIMESTAMP primary key,
    end_time   bigint default null
);
```

5.2.5 Real-time streaming

Real-time streaming requires real-time data transmission at two parts. The first part is between the dashboard and backend server, which can be easily realized by Web Sockets. The second part is between the backend server and database.

The headache here is the second part. Most databases are designed to be retroactive: they can only be passively queried by external applications while it cannot notify external applications [6]. To tackle this problem, we use Listen and Notify functionalities provided by PostgreSQL.

5.2.5.1 Design of Listen / Notify Approach

PostgreSQL provides an asynchronous publishing-subscribing functionality for external applications, namely Listen and Notify [7]. A database function can publish notification to a channel that backend servers are listening to. In such way, the PostgreSQL can notify the server of incoming sensor data. The detailed process is illustrated in Figure 5.2.5.1-A.

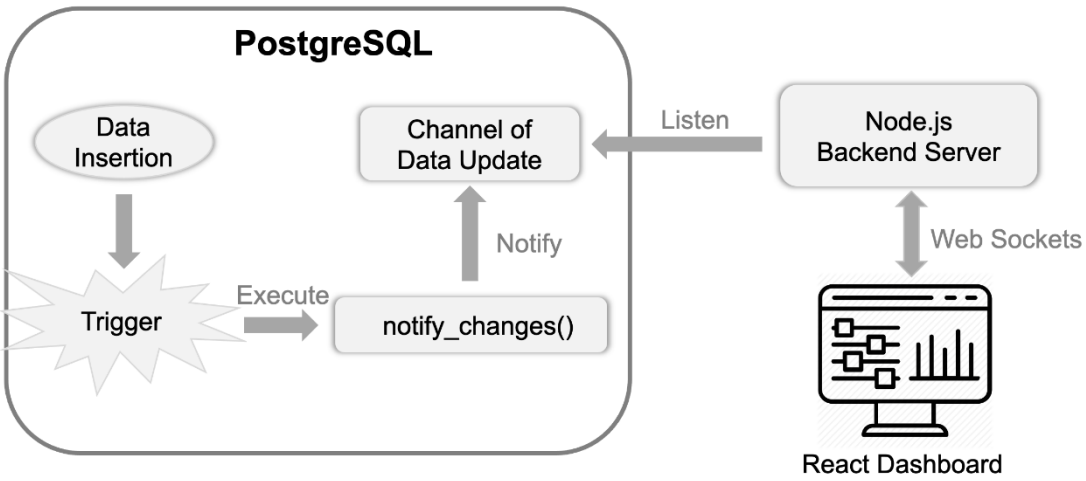


Figure 5.2.5.1-A
Real-time streaming by PostgreSQL Listen and Notify

When new sensor data are inserted into database tables, a trigger is released, which will execute the user defined *notify_changes( )* function. This function publishes notifications with payloads of newly inserted data. The server then hears notifications and transmit the message to the dashboard via web sockets. React will re-render the components according to the newly inserted data.

As stated above, this solution realizes a bijective relation between database update and frontend state update. This relation may crash the frontend when the sensor data arrives at an extremely high frequency, because React cannot handle such high frequency re-rendering.

Our solution is to mute the re-rendering but only update states. A 10 Hz clock is set to re-render the frontend once per 100ms to catch up updates of states. In fact, this at most 100ms delay is unobservable and our streaming is very smooth and reactive.

An example of the above process
Here is a situation for updating dancer positions.

First, the following query is sent to database by communication server to insert a new position.

```
insert into DancePosition(start_time, left_slot, middle_slot, right_slot) values(0, '1', '2', '3')
```

Then, the PostgreSQL notify the channel with following message.

```
{"table_name" : "DancePosition", "record" : {"start_time":0,"local_time":"2021-04-18T20:52:11.658803",
"left_slot":"1","middle_slot":"2","right_slot":"3"}}
```

Moreover, this notification is heard by the Node server who listens to the same channel. The server transmits the same message above immediately to the dashboard.

In the end, the dashboard parses the JSON message and updates its state.

5.2.5.2 Implementation of Listen / Notify Approach

Database Side
PostgreSQL trigger is used to set actions when each database table changes. The trigger function will *pg_notify* the *streaming_data* channel for each new row in JSON format.

```
create or replace function notify_dance_position_changes() returns trigger as
$$
declare
begin
    perform pg_notify(
            'streaming_data',
            json_build_object(
                    'table_name', 'DancePosition',
                    'record', row_to_json(new)
                )::text
        );
    return null;
end;
$$ language plpgsql;

drop trigger if exists dance_position_inserted on DancePosition cascade;
create trigger dance_position_inserted
    after insert on DancePosition
    for each row
execute function notify_dance_position_changes();
```

Server Side
The Node server listens to the same channel.

```
db.query(`listen ${notification_channel}`);
db.on('notification', processDBNotification);
```

Once notified, it will call the *processDBNotification* to send the data to the frontend.

```
const processDBNotification = async (data) => {
    let payload = data.payload;
    sendToClients(payload);
    console.log(payload);
};
```

The *sendToClients* function then sends the message via the web-socket as shown below.

```
const sendToClients = (data) => {
    wss.clients.forEach( callbackfn: client => {
        client.send(data);
    });
};
```

Frontend Side

React receives the JSON message from the server and parse it. The parsed information is then updated in states.

```
handlePositionData = (record) => {
    let {start_time, left_slot, middle_slot, right_slot} = record;
    this.setState( state: {
        positions: [left_slot, middle_slot, right_slot]
    });
}
```

5.2.5.3 Smooth 10Hz Refreshing Rate

As illustrated above, every newly inserted data in database will update the frontend states via the process described above. This process requires no polling at all. State updates are very efficient and with nearly no delay.

The only tiny delay is caused by the 10Hz fixed-frequency re-rendering, which is 100ms. In fact, during our experiment, this 10Hz refreshing rate is very responsive and smooth. This approach is also proved to be very stable because no lost data was ever detected during our demo.

5.2.5.4 Backup Plans

Backup 1. Frequent polling

The server can query the database at a fixed frequency to detect data changes. In such way the frontend refresh rate is bound with the backend query frequency. When the query frequency is relatively fast, the dashboard can pretend to be real-time. Its mechanism is illustrated in Figure 5.2.4-A:

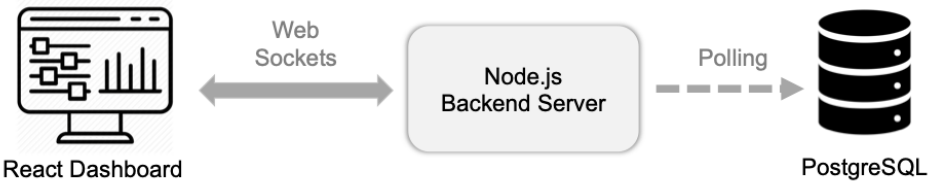


Figure 5.2.4-A
Real-time Streaming by Polling

Backup 2. Dash

This plan is using pure Python stack. The Dash library helps build real-time dashboard for Python data analytics extremely fast [8]. By using Dash, our dashboard subsystem can be simplified to serverless architecture as below:

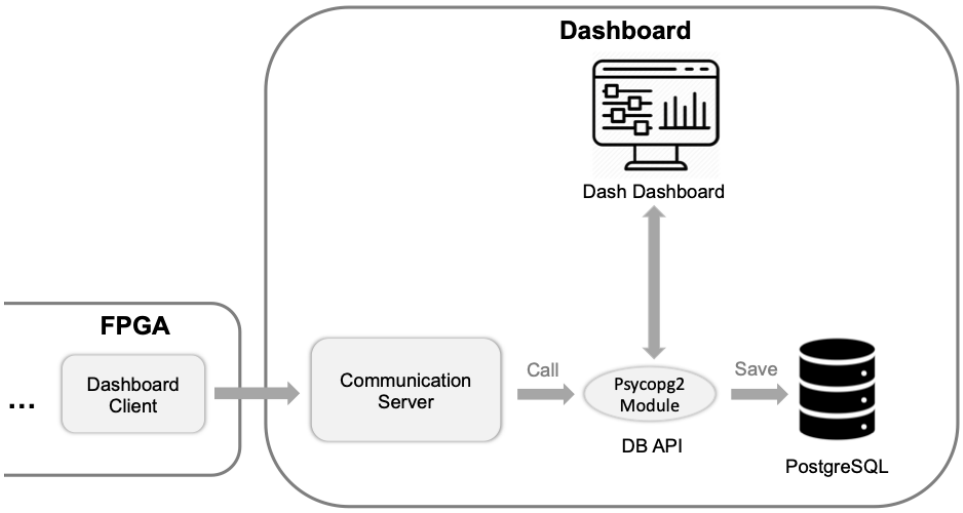


Figure 5.2.4-C
The Backup Architecture Using Dash

This backup plan is easy and saves time for development. However, the drawback is obvious. Dash is specifically designed for data visualization and hence it is far less customizable for UI development than JavaScript stacks. Adopting Dash stack may restrict us from building beautiful and customizable UI. Therefore, Dash is merely preserved as a compromise.

5.2.6 User Survey

Our user survey is in two forms.

The first form of questionnaires, consisting of both closed ended and open-ended questions. The open-ended questions can boost the response rate while the closed-ended questions unravel more profound insights [9].

The second form is face-to-face feedback from my teammates. These feedbacks value greatly and he my iterative improvement of the UI.

5.2.6.1 Questionnaire Design and Response

User Survey for CG4002 Dashboard Design

Close-ended Questions

1. By intuition, which color below do you think represents a late dance move?

   A. Warm Color B. Cold Color C. Faded Color (Grey)

   Your answer: A A B A A

2. Do you like the shadow effects around the boxes?

   A. Yes. B. No. C. It's too thick. D. It's too light.

   Your answer: C C C A C

3. Do you like to place positions of the three in one box or distribute them to dancer panels?

   A. In one box. B. Distributed

   Your answer: A A A A A

4. Do you prefer mere images or mere text or both to represent dance move prediction?

   A. Images B. Text C. Both

   Your answer: C A C A A

5. Do you prefer animations for IMU, or the current line chart is just good for you?

   A. Animation B. Line chart

   Your Answer: A A A B A

6. If your answer to Q5 is "animation", will you mind the lagging of the animation, given that network connection may be unstable?

   A. I don't mind. B. Lagging animations are worse than line charts.

   Your Answer: B A B B

7. Do you prefer IMU chart and Acceleration chart of the same person arranged as horizontal or vertical?

   A. Vertical. B. Horizontal.

   Your Answer: A B B B B B

Open-ended Questions

1. What was your first impression when you saw the dashboard?

   The UI should be further polished. It looks a bit flawed.

   Its functionality seems clear to me.

2. Which features on the dashboard are most informative to you?

   Chart and the prediction labels.

3. Is there any functionality you think you need but is missing on this dashboard? Please describe.

   Maybe a synchronization display of each player's motion.

   Auto-comparison between ground truth and prediction.

   A feedback channel.

5.2.6.2 Iterative Improvements from User Feedback

Change of Color Scheme
The original color scheme is black-grey-white. My teammate argued that it was not dynamic enough for our application. I changed the color scheme into light-blue and pink, which is now warmer and fits our theme better.

Separation of ML Predictions and Sensor
The old dashboard put everything including ML results and sensor data into one page. The UI looked very clustered. Finding specific information among them was difficult for my teammates.

Hence, I separated it into a main Dashboard page and a Sensor Page. They are now less clustered and more intuitive.

Change of Sensor Panel Color
The sensor panel was transparent, and it was the same color as the background blue. Teammates said that the blue panel made sensor line chart hard to notice.

Hence, I changed the sensor panel background to white.

Use Animation for Switching Panel for Positions
The old way to present dancer position was showing a sequence of numbers, such as '1 2 3'. However, my teammate told me that this was not intuitive and hoped that I could animate the positions of three user panels instead.

Therefore, I used CSS transform and CSS transition to build animation for switching user panel to represent their positions.

Section 6 Project Management Plan - *Written by Chun Lik*

6.1 Team Structure

This project adopts the chief-designer structure. The chief designer of this project, Sean, who is also a hardware expert, is elected due to his experiences in prior projects of similar nature. Chief designer designs the system architecture while heavily assisted by domain specialists, e.g., hardware, communication and software experts, as shown in Figure 6.1-A. This structure allows individual group members to focus solely on the areas in which they have sound and expertise knowledge.

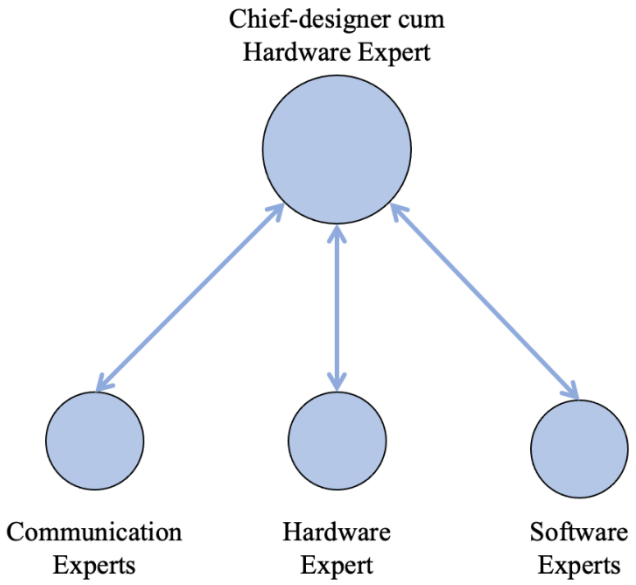


Figure 6.1-A
Chief-designer team structure and closely supported by domain experts

6.2 Progress Tracking

This project will use revision control for the purpose of managing software, coding and documentation. Revision control is suitable for large scale projects such as CG4002 Capstone Project that requires close collaboration between team members. Furthermore, such approach affords the project team the ability to experiment and implement new ideas and recover from earlier versions. This project will be using GitHub extensively as it provides the above-mentioned functionalities. In addition, GitHub's issue tracker provides a convenient platform for the team to discuss and diagnose coding issues to ensure compatibility,

A weekly milestone table is also set up to ensure project schedule and objectives are within scope.

Scope / Week	HW Sensors	HW FPGA	Comms Internal	Comms External	SW Machine Learning	SW Dashboard	Joint Work
3	Prepare hardware (solder pins etc) Research and order extra hardware	Research types of neural networks and available frameworks	Research on BLE communications and real-time scheduling	Research on TCP Sockets communication and Encryption protocol.	Research on ML classical classification techniques suitable for Human Activity Recognition (HAR)	Research on dashboard design and study the dashboard template online.	Decide on team structure, scope assignment and collaboration platforms.

4	Prepare hardware (solder pins etc) Research and order extra hardware	Learn how to use HLS and use it to create a matrix multiplier	Establish a BLE connection between a Beetle and laptop	Create client script for evaluation server.	Research on Deep Learning and Neural Networks variants for HAR. Identify suitable dataset for HAR. Develop workable SVM, *k*-NN and Decision Tree learning models based on WISDM dataset	Brainstorm features and analytics for the dashboard. Research on real-time data streaming from database.	Initial Design Report submission
5	Build wearable and test detection algorithms	Start with the construction of neural network	Successfully conduct a 3-way handshake between a Beetle and laptop	Establish comms between Ultra96 and laptop.	Implement Convolutional Neural Networks based on WISDM dataset	Implement data streaming on a sample label. Improve the UI design with teammate feedbacks.	Review feedbacks on Initial Design report
6	Refine detection algorithms	Continue building neural network	Establish connections between 3 Beetles and laptop using the communicatio n protocol	Establish comms between Ultra96 and dashboard server.	Apply model evaluation and select the best learning models.	Compose UI in React. Make the database.	Progress checkpoint. 1-on-1 walkthrough process on implementation
R	Prepare for Integration	Ensure neural network works with random values	Test out sending data between 3 Beetles and laptop concurrently for one minute	Integrate with the rest of the team.	Refine and tweak both classical and neural networks models to correctly distinguish dance moves. Perform feature engineering if necessary	Experiment with the functionalities on dashboard using sample input.	
7	Integrate with the rest of the team.	Start integration, implementatio n of model	Integrate with the rest of the team	Ensure data is received by Ultra96 and results are passed to evaluation server.	Integrate ML with the rest of the team	Integrate the dashboard with the rest of the team.	Individual subcomponent test
8	Maintain hardware and conduct necessary design revisions	Prepare for first checkpoint, ensure comms able to input data and receive output	Test out with real sensor data	Ensure data is received by Ultra96 and results are passed to dashboard server.	Continue working and troubleshoot integration issue	Refine database schema and data streaming method to fit other subsystems.	
9	Maintain hardware and conduct necessary design revisions	Explore methods for optimization of timing and power (based on first system checkpoint results)	Work on improving latency and reliability of the BLE connection	Integrate with the rest of the team.	Evaluate final integration results and prepare for first system checkpoint	Evaluate integration result for system checkpoint.	First system checkpoint
10		Work with ML to tune neural network		Integrate with the rest of the team.	Continual refinement to ML and improve accuracy if necessary	Optimize the performance of dashboard and backend server.	
11	Maintain hardware	Further optimizations		Optimize codes.	Review performance of second evaluation test and tweak ML	Test against the dashboard with various input.	Second evaluation test.

12	Maintain hardware	Polish system, finalize any last changes	Tie up loose ends and get the scripts ready for the final evaluation	Polish and finalize changes.	Polish, finalize and stabilize ML algorithms in preparation for Final Evaluation	Do the quality assurance against unexpected user actions.	
13	Last round of testing, work on final report						Final evaluation; Final Design Report submission

Table 6.2-A

Weekly milestone and expected progress timeline

Items	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
[ALL] Discuss system architecture, research											
[ALL] Work on initial design report											
[Sensor] Build wearable											
[FPGA] Learn about HLS, execute simple matrix multiplier on board											
[C-Int] Establish BLE connection between Beetles and laptop											
[C-Ext] Write script to connect with evaluation server											
[ML] Research on deep learning and neural networks											
[DB] Research on UI design.											
[Sensor] Processing of movements											
[FPGA] Program CNN on board using random values											
[C-Int] Setup handshaking and protocol											
[C-Ext] Comms between Ultra96 and laptop											
[ML] Comparison with another model											
[DB] Conduct user survey											
[ALL] Subcomponent Test											
[ALL] Start integration											
[Sensor] Finetune feature extraction											
[FPGA] Implement neural network											
[C-Int] Ensure data is passed from Beetles and laptop											
[C-Ext] Ensure data is received by Ultra96 and results are passed to evaluation server											
[ML] Train neural network											
[DB] Work to receive and display real time data											
[ALL] First system checkpoint											

[ML] Further training of model for other dance moves											
[C-Int] Work on improving reliability											
[FPGA] Optimize for power and speed											
[ALL] Second evaluation test											
[ALL] System improvements and testing											
[Sensor] Maintenance of wearable											
[DB] Refinement of features and design											
[ALL] Finalize Design Report											
[ALL] Final Evaluation											

6.3 Societal and Ethical Impact - *Written by Xin Yan*

With additional sensors and further training of new movements, our dance detector can be easily adapted to other forms of activity detectors. Generally, any form of activity that requires movement classification can utilize our dance detector to estimate the accuracy of the movements. On top of its versatility, it is cheap and easy to maintain. Hence, we see the dance detector having a lot of potential to be adapted to other usages, bringing benefits to society. Of the vast number of possible implementations of the activity detector, we have narrowed it down into 3 main categories, namely sports, music, and disability aid.

6.3.1 Societal Impact: Sports

Like how our dance detector is used for dancing and training of dancers, the same can be done for sports. When it comes to learning new moves for a sport, new athletes can train the coordination of their movements with the device. For sports like basketball or handball which consist of multiple different moves such as crossover, dribbling etc, the device can be used to track the accuracy of the athletes' moves. It can also be further adapted to detect if the athlete violated any rules in performing the new move, such as a potential "travel" during the execution of the new move. We can see the dance detector being easily adapted to be a movement detector for such sports.

Seasoned athletes can also use the device to perfect their motion (for example, canoeing or dragon boating). Such sports require athletes to have perfect form execution with good rhythm. For this implementation, our detector can have additional features such as a timer countdown to indicate the start of a new "pull" for the athlete to train their rhythm. An angle sensor and proximity sensor can also be added to ensure that the form is accurate. Since our original detector already possess the sync delay feature, the dragon boaters can "paddle" together with one device each to estimate each of their lag time.

Apart from sport activities, the device can be easily enhanced to classify gym movements as well. Not only can it help fitness enthusiasts track the accuracy of each of their repetition in the gym, it can also help ensure people who partake in virtual fitness classes have the correct form at the comfort of their own homes. Having the right form is pivotal as it prevents potential injuries that can occur from such strenuous activities. The detector can also implement an injury prevention alert that warns the users if their form is wrong, or they are overexerting themselves (from the EMG data).

Lastly, for athletes who have sustained injuries and require rehabilitation, the dance detector can also be adapted to accommodate them as well. This can help physiologists check in on their patients' rehabilitation records to ensure that they are recovering well.

6.3.2 Societal Impact: Music

Our dance detector can also be enhanced to accommodate musical instruments learners. For instruments like drums, a lot of coordination is required to produce the sounds desired. The dance detector can be adapted to help a learner perfect his piece by ensuring his movements and rhythm is right. It can also help to ensure that the drummers' strength for each of the beat is correct.

While it can be deemed as a redundancy since the accuracy can be determined through listening to the piece, it can help alleviate the instructors' lives especially when it is a fast-paced piece. It can also benefit those who are self-taught, or those who are tone deaf. Furthermore, it can also help a hearing-impaired individual learn how to drum or play other musical instruments since the dashboard allows the user to see his or her own performance.

6.3.3 Societal Impact: Disability Aid

The dance detector can be adapted as a disability aid. It can train with sign languages and allow society to communicate with individuals who have speech or audio impairments. This is very beneficial to the society as it encourages more inclusivity and eliminates the barriers of communication.

For instance, the detector can be adapted as an action to words system. This can help to translate the speech or audio impaired individual's sign languages into words for one to understand. Then words exchanged can then be generated displayed on the dashboard in real time.

6.3.4 Ethical Impact

When it comes to such devices which require a large amount of data collection to train the system, privacy is the main concern. Malicious intent of hacking the data can lead to a serious breech in data privacy, where hackers can leak your data to other 3rd parties who may use the data for other purposes. Leaking of such sensitive data can have a gravely negative influence someone's life.

In addition, since these devices are installed at their own homes, without proper installation and proper security measures, users of these devices may risk cyber-attacks. This can include hackers modifying the training data or distorting the whole system etc.

These can be prevented by ensuring the data encryption is strong enough and that the network is not susceptible to attacks. Some precautions against cybersecurity attacks include setting up a firewall, strong passwords, control access, regular system updates etc. This can help to reduce the potential attacks and keep malicious activities at bay.

Section 5.1.20 revealed that a dancer can be accurately identified if the model has access to their training data. However, this is achievable only if the training data has been labelled manually – in this case, the name of the dancers. Hence, users participating in the project should acknowledge and provide consent on how their data might be used.

References

Hardware Sensors

[1] DFROBOT, “SKU DFR0339,” *DFRobot*, 2020. [Online]. Available: https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339. [Accessed: 29-Jan-2021].

[2] DFROBOT, “Bluno Schematic,” *DFRobot*, 2020. [Online]. Available: https://github.com/Arduinolibrary/DFRobot_Bluno_Bettle/blob/master/DFR0339-Bluno%20beetle%20V1.0.pdf?raw=true. [Accessed: 29-Jan-2021].

[3] Alldatasheet.com, “ATMEGA328P Datasheet(PDF) - ATMEL Corporation,” *ALLDATASHEET.COM - Datasheet search site for Electronic Components and Semiconductors and other semiconductors.* [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/241077/ATMEL/ATMEGA328P.html. [Accessed: 07-Feb-2021].

[4] Texas Instruments, “CC2540,” *CC2540 data sheet, product information and support | TI.com*. [Online]. Available: https://www.ti.com/product/CC2540. [Accessed: 07-Feb-2021].

[5] Alldatasheet.com, “HT7850 Datasheet(PDF) - Holtek Semiconductor Inc,” *ALLDATASHEET.COM - Datasheet search site for Electronic Components and Semiconductors and other semiconductors.* [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/205882/HOLTEK/HT7850.html. [Accessed: 07-Feb-2021].

[6] Alldatasheet.com, “XC6206P332MR Datasheet(PDF) - Torex Semiconductor,” *ALLDATASHEET.COM - Datasheet search site for Electronic Components and Semiconductors and other semiconductors.* [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/243854/TOREX/XC6206P332MR.html. [Accessed: 07-Feb-2021].

[7] “MPU-6000 and MPU-6050 Product Specification Revision 3.” [Online]. Available: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf. [Accessed: 07-Feb-2021].

[8] “GY-521 MPU6050 3-Axis Acceleration Gyroscope 6DOF Module,” *Powered by HAOYU Electronics*. [Online]. Available: https://www.hotmcu.com/gy521-mpu6050-3axis-acceleration-gyroscope-6dof-module-p-83.html. [Accessed: 07-Feb-2021].

[9] Arduino, “arduino/ArduinoCore-avr,” *GitHub*. [Online]. Available: https://github.com/arduino/ArduinoCore-avr/tree/master/libraries/Wire. [Accessed: 07-Feb-2021].

[10] Jrowberg, “jrowberg/i2cdevlib,” *GitHub*. [Online]. Available: https://github.com/jrowberg/i2cdevlib. [Accessed: 07-Feb-2021].

[11] Alldatasheet.com, “APDS-9960 Datasheet(PDF) - AVAGO TECHNOLOGIES LIMITED,” *ALLDATASHEET.COM - Datasheet search site for Electronic Components and Semiconductors and other semiconductors.* [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/918047/AVAGO/APDS-9960.html. [Accessed: 07-Feb-2021].

[12] “GY-9960-3.3 APDS-9960 RGB Infrared Gesture Sensor Module,” *Powered by HAOYU Electronics*. [Online]. Available: https://www.hotmcu.com/gy996033-apds9960-rgb-infrared-gesture-sensor-module-p-329.html. [Accessed: 07-Feb-2021].

[13] Sparkfun, “sparkfun/APDS-9960_RGB_and_Gesture_Sensor,” *GitHub*. [Online]. Available: https://github.com/sparkfun/APDS-9960_RGB_and_Gesture_Sensor. [Accessed: 07-Feb-2021].

[14] H. Zhang, “140: Basic Concepts of Linear Regulator and Switching Mode Power Supplies,” *AN*. [Online]. Available: https://www.analog.com/en/app-notes/an-140.html. [Accessed: 07-Feb-2021].

[15] Alldatasheet.com, “CR2032 Datasheet(PDF) - Energizer,” *ALLDATASHEET.COM - Datasheet search site for Electronic Components and Semiconductors and other semiconductors.* [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/160971/ENERGIZER/CR2032.html. [Accessed: 07-Feb-2021].

[16] “TZT 0.9V To 5V DC DC Step Up Power Module Voltage Boost Converter Board,” *aliexpress.com*. [Online]. Available: https://www.aliexpress.com/item/32807311456.html?spm=a2g0o.productlist.0.0.45426f73Rw452Y&algo_pvid=09b75639-b8c7-4cf5-9314-e7185bf5d3df&algo_expid=09b75639-b8c7-4cf5-9314-e7185bf5d3df-1&btsid=0bb0624716127047927373895e8913&ws_ab_test=searchweb0_0%2Csearchweb201602_%2Csearchweb201603_. [Accessed: 07-Feb-2021].

[17] “MyoWare Muscle Sensor Kit - Digi-Key.” [Online]. Available: https://media.digikey.com/pdf/Data%20Sheets/Sparkfun%20PDFs/MyoWare_Muscle_Sensor_Kit_Guide.pdf. [Accessed: 07-Feb-2021].

[18] “SURFACE ELECTROMYOGRAPHY DETECTION AND RECORDING.” [Online]. Available: https://www.delsys.com/downloads/TUTORIAL/semg-detection-and-recording.pdf. [Accessed: 07-Feb-2021].

[19] MartinBloedorn, “MartinBloedorn/libFilter,” *GitHub*. [Online]. Available: https://github.com/MartinBloedorn/libFilter. [Accessed: 07-Feb-2021].

[20] B. N. Cahyadi, W. Khairunizam, I. Zunaidi, L. H. Ling, A. B. Shahriman, M. R. Zuradzman, W. A. Mustafa, and N. Z. Noriman, “IOPscience,” *IOP Conference Series: Materials Science and Engineering*, 01-Jun-2019. [Online]. Available: https://iopscience.iop.org/article/10.1088/1757-899X/557/1/012004. [Accessed: 07-Feb-2021].

[21] Kosme, “kosme/arduinoFFT,” *GitHub*. [Online]. Available: https://github.com/kosme/arduinoFFT. [Accessed: 07-Feb-2021].

[22] Alldatasheet.com, “MT3608 Datasheet(PDF) ,” *ALLDATASHEET.COM - Datasheet search site for Electronic Components and Semiconductors and other semiconductors.* [Online]. Available: https://pdf1.alldatasheet.com/datasheet-pdf/view/1131968/ETC1/MT3608.html. [Accessed: 14-APR-2021].

[23] Farnell.com. “*Varta Rechargeable Nickel-Metal-Hydride Button Cells*”. [online] Available at: http://www.farnell.com/datasheets/1725942.pdf [Accessed 18 April 2021].

[24] Biz.maxell.com. "*ML2032*". [online] Available at: https://biz.maxell.com/en/rechargeable_batteries/ML2032_DataSheet_16e.pdf [Accessed 18 April 2021].

[25] Texas Instruments. "*TPS6120x Low Input Voltage Synchronous Boost Converter With 1.3-A Switches*". [online] Available at: https://www.ti.com/lit/ds/symlink/tps61200.pdf [Accessed: 18-APR-2021].

FPGA

[1] Xilinx Research Labs, "What is FINN?," *About FINN*, 2020. [Online]. Available: https://xilinx.github.io/finn/about. [Accessed: 25-Jan-2021].

[2] Xilinx Research Labs, "End-to-End Flow," *End-to-End Flow - FINN documentation*, 2020. [Online]. Available: https://finn.readthedocs.io/en/latest/end_to_end_flow.html. [Accessed: 25-Jan-2021].

[3] J. Johnson, "Create a custom PYNQ overlay for PYNQ-Z1," *FPGA Developer*, 15-Mar-2018. [Online]. Available: https://www.fpgadeveloper.com/2018/03/create-a-custom-pynq-overlay-for-pynq-z1.html/. [Accessed: 07-Mar-2021].

[4] Xilinx, "Overlay Tutorial," *Overlay Tutorial - Python productivity for Zynq (Pynq)*, 2018. [Online]. Available: https://pynq.readthedocs.io/en/v2.5.1/overlay_design_methodology/overlay_tutorial.html. [Accessed: 09-Mar-2021].

[5] P. Sharma, "CNN Tutorial: Tutorial On Convolutional Neural Networks," *Analytics Vidhya*, 28-Dec-2018. [Online]. Available: https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/. [Accessed: 16-Mar-2021].

[6] A. Bastin, "How to Configure the Number of Layers and Nodes in a Neural Network," *Data Science Central*, 17-Feb-2019. [Online]. Available: https://www.datasciencecentral.com/profiles/blogs/how-to-configure-the-number-of-layers-and-nodes-in-a-neural. [Accessed: 18-Apr-2021].

[7] *Ultra96 Hardware User's Guide*, Revision 1, Avnet Inc., 2018.

[8] Xilinx Research Labs, "Introduction," *FINN-HLS*, 2020. [Online]. Available: https://finn-hlslib.readthedocs.io/en/latest/index.html. [Accessed: 27-Jan-2021].

[9] SlayStudy, "Implementation of softmax activation function in C/C++," *SlayStudy*, 2020. [Online]. Available: https://slaystudy.com/implementation-of-softmax-activation-function-in-c-c/. [Accessed: 04-Apr-2021].

[10] Amiq-Consulting, "CNN-using-HLS," *GitHub*, 31-Mar-2020. [Online]. Available: https://github.com/amiq-consulting/CNN-using-HLS/tree/master/modules. [Accessed: 18-Mar-2021].

[11] R. Panicker, *High Level Synthesis Flow*, 06-Jun-2020. [Online]. Available: https://wiki.nus.edu.sg/display/ee4218/High+Level+Synthesis+Flow. [Accessed: 22-Jan-2021].

[12] Xilinx Research Labs, "Network Preparation," *Network Preparation - FINN documentation*, 2020. [Online]. Available: https://finn.readthedocs.io/en/latest/nw_prep.html. [Accessed: 25-Jan-2021].

[13] Xilinx, "HLS Pragmas," *SDAccel Development Environment Help for 2019.1*, 19-Jun-2019. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html. [Accessed: 02-Feb-2021].

[14] J. Faraone, M. Kumm, M. Hardieck, P. Zipf, X. Liu, D. Boland, and P. H. Leong, "AddNet: Deep Neural Networks Using FPGA-Optimized Multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 115–128, 2020.

[15] Avnet, "PMBus on the Ultra96," *Ultra96-PYNQ Github repository*, 30-Sep-2020. [Online]. Available: https://github.com/Avnet/Ultra96-PYNQ/blob/master/Ultra96/notebooks/common/ultra96_pmbus.ipynb. [Accessed: 20-Mar-2021].

Internal Comms

[1] ARM Ltd and ARM Germany GmbH, *Advantages of using an RTOS*. [Online]. Available: https://www.keil.com/rl-arm/rtx_rtosadv.asp#:~:text=RTOS%20Concept&text=An%20advanced%20RTOS%2C%20such%20as,a%20number%20of%20tasks%20simultaneously. [Accessed: 28-Jan-2021].

[2] "Communication Protocols," *Communication Protocols - an overview | ScienceDirect Topics*. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/communication-protocols. [Accessed: 30-Jan-2021].

[3] B. Ray, *Bluetooth Vs. Bluetooth Low Energy (BLE): What's The Difference?* [Online]. Available: https://www.link-labs.com/blog/bluetooth-vs-bluetooth-low-energy. [Accessed: 29-Jan-2021].

[4] "BLEmicro_V1.1_SKU_TEL0084," *DFRobot*. [Online]. Available: https://wiki.dfrobot.com/BLEmicro_V1.1_SKU_TEL0084. [Accessed: 29-Jan-2021].

[5] "Three-Way Handshake," *Three-Way Handshake - an overview | ScienceDirect Topics*. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/three-way-handshake. [Accessed: 30-Jan-2021].

[6] J. Scarpati, "What is Network Time Protocol (NTP)? - Definition from WhatIs.com," *SearchNetworking*, 23-Feb-2017. [Online]. Available: https://searchnetworking.techtarget.com/definition/Network-Time-Protocol. [Accessed: 31-Mar-2021].

External Comms

[1] “AES,” *AES - PyCryptodome 3.9.9 documentation*. [Online]. Available: https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html. [Accessed: 07-Feb-2021].

[2] “DBMS Concurrency Control: Timestamp & Lock-Based Protocols,” *Guru99*. [Online]. Available: https://www.guru99.com/dbms-concurrency-control.html#5. [Accessed: 07-Feb-2021].

[3] J. Matson, “Choosing the correct Time Synchronization Protocol and incorporating the 1756-TIME module into your Application,” *Rockwell Automation*. [Online]. Available: https://literature.rockwellautomation.com/idc/groups/literature/documents/wp/enet-wp030_-en-e.pdf.

[4] N. Jennings, “Socket Programming in Python (Guide),” *Real Python*, 12-Dec-2020. [Online]. Available: https://realpython.com/python-sockets/#socket-api-overview. [Accessed: 07-Feb-2021].

[5] “Python PostgreSQL Tutorial Using Psycopg2 [Complete Guide].” [Online]. Available: https://pynative.com/python-postgresql-tutorial/. [Accessed: 07-Feb-2021].

[6] “Quorum queues,” *RabbitMQ*. [Online]. Available: https://www.rabbitmq.com/. [Accessed: 07-Feb-2021].

[7] “Classic modes of operation for symmetric block ciphers,” Classic modes of operation for symmetric block ciphers - PyCryptodome 3.9.9 documentation. [Online]. Available: https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html. [Accessed: 16-Apr-2021].

[8] “SSH Port Forwarding Example,” *SSH port forwarding - Example, command, server config*. [Online]. Available: https://www.ssh.com/ssh/tunneling/example#what-is-ssh-port-forwarding,-aka-ssh-tunneling. [Accessed: 07-Feb-2021].

[9] “sshtunnel,” *PyPI*. [Online]. Available: https://pypi.org/project/sshtunnel/. [Accessed: 07-Feb-2021].

Machine Learning

[1] Pant, A. (2019, January 23). Workflow of a machine learning project. Retrieved February 3, 2021, from https://towardsdatascience.com/workflow-of-a-machine-learning-project-ec1dba419b94

[2] Khan, A., Hammerla, N., Mellor, S., & Plötz, T. (2016, January 14). Optimising sampling rates for accelerometer-based human activity recognition. Retrieved February 03, 2021, from https://www.sciencedirect.com/science/article/abs/pii/S0167865516000040?via%3Dihub

[3] Banos, O., Galvez, J., Damas, M., Pomares, H., & Rojas, I. (2014, April 9). Window size impact in human activity recognition. Retrieved February 03, 2021, from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4029702/

[4] Brownlee, J. (2020, August 15). Discover feature engineering, how to engineer features and how to get good at it. Retrieved February 04, 2021, from https://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/

[5] Dataset. (n.d.). Retrieved February 02, 2021, from https://www.cis.fordham.edu/wisdm/dataset.php

[6] Malyi, V. (2017, July 18). Run or walk. Retrieved February 04, 2021, from https://www.kaggle.com/vmalyi/run-or-walk

[7] Gandhi, R. (2018, July 05). Support vector machine - introduction to machine learning algorithms. Retrieved February 04, 2021, from https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

[8] Bhattacharyya, S. (2020, October 21). Understanding support Vector Machine: Part 2: Kernel TRICK; Mercer's theorem. Retrieved February 03, 2021, from https://towardsdatascience.com/understanding-support-vector-machine-part-2-kernel-trick-mercers-theorem-e1e6848c6c4d

[9] Schott, M. (2020, February 27). K-Nearest neighbors (KNN) algorithm for machine learning. Retrieved February 01, 2021, from https://medium.com/capital-one-tech/k-nearest-neighbors-knn-algorithm-for-machine-learning-e883219c8f26

[10] Brownlee, J. (2019, August 05). Deep learning models for human activity recognition. Retrieved February 01, 2021, from https://machinelearningmastery.com/deep-learning-models-for-human-activity-recognition/

[11] Uniqtech. (2019, June 13). Multilayer perceptron (MLP) VS convolutional neural network in deep learning. Retrieved February 01, 2021, from https://medium.com/data-science-bootcamp/multilayer-perceptron-mlp-vs-convolutional-neural-network-in-deep-learning-c890f487a8f1

[12] Nils. (2018, December 15). Human activity Recognition (har) tutorial with Keras and CORE ML (part 1). Retrieved February 02, 2021, from https://towardsdatascience.com/human-activity-recognition-har-tutorial-with-keras-and-core-ml-part-1-8c05e365dfa0

[13] Frédéric Li (2018, February) Comparison of Feature Learning Methods for Human Activity Recognition Using Wearable Sensors. Retrieved February 02, 2021, from https://www.researchgate.net/publication/323423384_Comparison_of_Feature_Learning_Methods_for_Human_Activity_Recognition_Using_Wearable_Sensors

[14] Krishni. (2018, December 21). K-Fold cross validation. Retrieved February 02, 2021, from https://medium.com/datadriveninvestor/k-fold-cross-validation-6b8518070833

[15] Brownlee, J. (2020, August 14). Data leakage in machine learning. Retrieved February 02, 2021, from https://machinelearningmastery.com/data-leakage-machine-learning/

[16] Narkhede, S. (2021, January 14). Understanding confusion matrix. Retrieved February 05, 2021, from https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

[17] Narkhede, S. (2021, January 14). Understanding AUC - ROC CURVE. Retrieved February 03, 2021, from https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

[18] Bhande, A. (2018, March 18). What is underfitting and overfitting in machine learning and how to deal with it. Retrieved February 02, 2021, from https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76

Software Dashboard

[1] “psycopg2,” *PyPI*, 07-Sep-2020. [Online]. Available: https://pypi.org/project/psycopg2/. [Accessed: 06-Feb-2021].
[2] "Structured design," Stevens, Wayne P.; Myers, Glenford J.; Constantine, Larry LeRoy (June 1974).. IBM Systems Journal. **13** (2): 115–139. doi:10.1147/sj.132.0115

[3] Liu SH;Lin CB;Chen Y;Chen W;Huang TS;Hsu CY; “An EMG Patch for the Real-Time Monitoring of Muscle-Fatigue Conditions During Exercise,” *Sensors (Basel, Switzerland)*. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/31337107/. [Accessed: 02-Feb-2021].
[4] “Transactions,” *PostgreSQL Documentation*, 07-Feb-2013. [Online]. Available: https://www.postgresql.org/docs/8.3/tutorial-transactions.html. [Accessed: 01-Feb-2021].
[5] “MongoDB vs PostgreSQL,” *MongoDB*. [Online]. Available: https://www.mongodb.com/compare/mongodb-postgresql. [Accessed: 01-Feb-2021].
[6] “Real-Time Data Fetch,” *Real-Time Data Fetch | Cube.js Docs*. [Online]. Available: https://cube.dev/docs/real-time-data-fetch. [Accessed: 02-Feb-2021].
[7] K. Lelonek, “How to use LISTEN and NOTIFY PostgreSQL commands in Elixir?,” *Medium*, 03-Oct-2018. [Online]. Available: https://blog.lelonek.me/listen-and-notify-postgresql-commands-in-elixir-187c49597851. [Accessed: 05-Feb-2021].
[8] Plotly, “Introducing Dash,” *Medium*, 16-Nov-2017. [Online]. Available: https://medium.com/plotly/introducing-dash-5ecf7191b503. [Accessed: 06-Feb-2021].
[9] R. Mishra, “This is all you need to know to conduct a UX Survey,” *Medium*, 21-Jun-2018. [Online]. Available: https://uxplanet.org/this-is-all-you-need-to-know-to-conduct-a-ux-survey-50400af45920. [Accessed: 04-Feb-2021].