CG4002 Computer Engineering Capstone Project
2022/2023 Semester 1

"Laser Tag++"
Final Design Report

Group **17**	Name	Student #	Specific Contribution
Member #1	Cui Minjing	A0211171Y	Hw Sensors
Member #2	Anderson Leong Ke Sheng	A0223111E	Hw AI
Member #3	I Muthukumar	A0201661U	Comms Internal
Member #4	Tan Hui En	A0221841N	Comms External
Member #5	Ho Ming Jun	A0199272R	Sw Visualizer

Section 1 System Functionalities (Everyone)

This project is about designing a wearable system that enables an AR laser tag game.

Section 1.1 Feature List

- Single Player Mode - One Player game
- Versus Play Mode - Two Player game
- Laser tag gun that can shoot infrared rays
- A target suit that mounts infrared sensors to detect shots
- A wearable device on the wrist that captures motion information
- Mobile phone mounted on the gun using Augmented Reality to display real-time information of the game
- Gun features: Shoot opponent
- Game gestures: shield, grenade, reload, logout
- System is wireless
- System can communicate the output to the evaluation server in a secure manner
- System can detect game actions and reflect the actions on the game
- System has a visualiser displaying real-time game actions and players status
    - User's HP, Ammo, Grenade count, Shield HP, Shield Timer
    - Opponent's Shield Activated, HP, Ammo, Grenade Count
    - Visual display when out of Ammo, Shield Charges, Grenades
    - Sound effects for different actions, getting shot or out of ammo
- A processor where the game engine runs and stores game states.

Section 2 Overall System Architecture (Everyone)

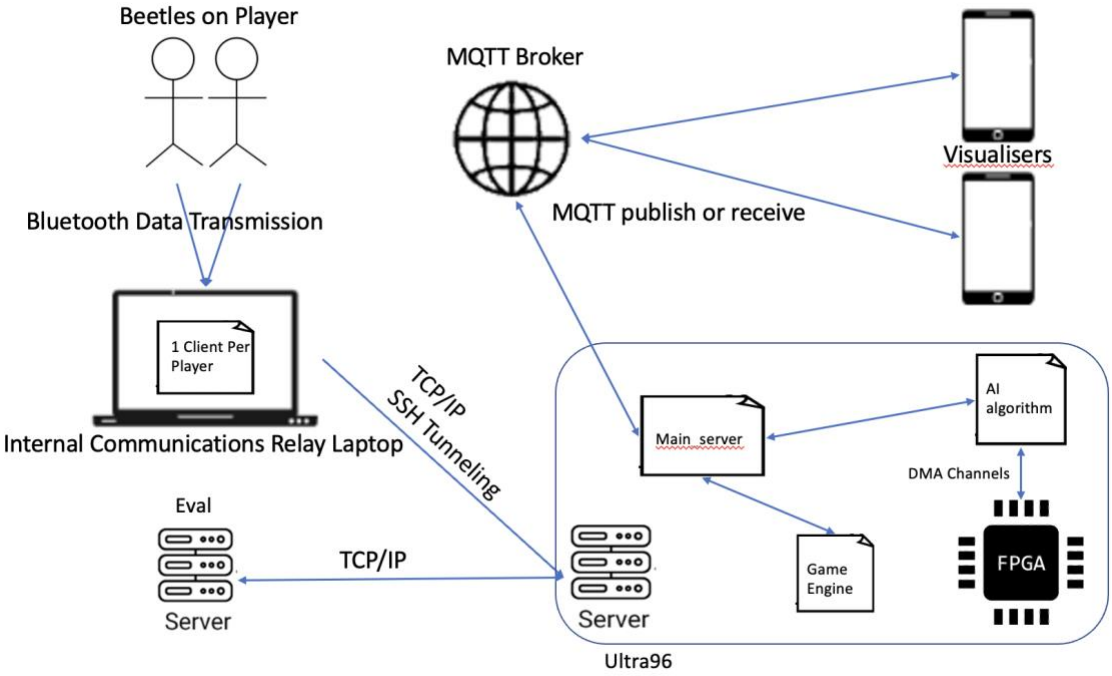


Figure 2. High Level Architecture

Section 2.1 Beetle Implementation

Each player equips three Beetles, on the vest, gun and glove. The Beetle on the vest implements the Infra-red Detector which acts as the shooting target. The Beetle on the gun implements the Infra-red Transmitter and button which controls the shooting action. The Beetle on the glove having the IMU sensor can detect the orientation and movement of the players' hand.

Section 2.1.1 Communication Protocol

The internal communication will consist of serial communication between the Beetles and laptop via Bluetooth.

For the external communication, there are two different protocols used between the devices as listed in Table 2.1.1 below.

Laptop and Ultra96	TCP/IP with SSH Tunneling
Ultra96 and evaluation server	TCP/IP
Ultra96 and visualizer	MQTT

Table 2.1.1 External Communications Protocols

Section 2.1.2 Hardware Components

The hardware components used are as follows:

i. 6 Bluno Beetles DFR 0339
ii. Ultra96
iii. 2 HX-121 IR Transmitter and Receiver Pairs
iv. 6 Lilypad board
v. MPU6050 GY-521 three axis accelerator
vi. 6 AAA batteries
vii. 2 buttons
viii. breadboard and wires

Section 2.1.3 Software Interfaces

The beetle sends information from the sensors to the laptop using serial communication via Bluetooth. The information is then transmitted by establishing a TCP/IP connection with SSH Tunneling between the laptop and the Ultra96, and between the Ultra96 and the evaluation server. The Ultra96 will then relay the information to an algorithm that utilizes a neural network run on the FPGA to determine gestures. Detected actions are run through the game engine and the latest game state is then sent to the software visualiser for display via MQTT.

Section 2.2 System Final Form

1) Vest: A vest mounted with an IR Sensor
2) Gun: A gun with a stand to place handphones mounted with a button to shoot
3) Glove: To detect user hand movements to initiate the appropriate action.
4) All Equipments are powered by Lilypads and equipped with a beetle each

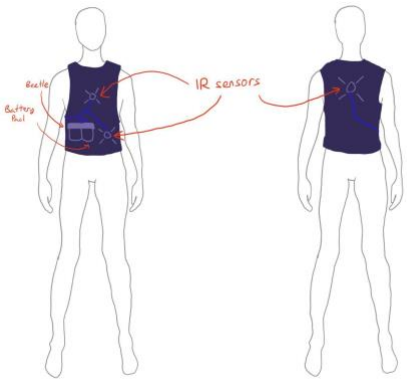


Figure 2.2.1 Design diagram of vest with Beetle and IR Sensors	Figure 2.2.2 Actual picture of vest with Beetle and IR Sensors

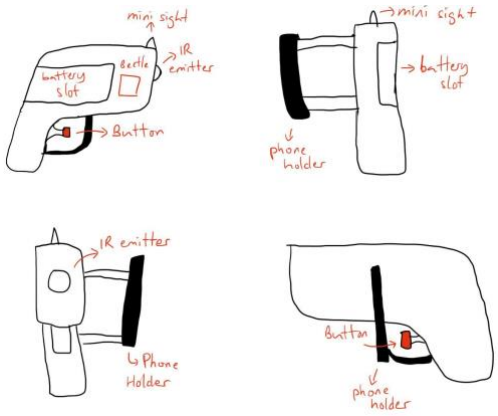



Figure 2.2.3 Design diagram of gun with Beetle and IR Sensors	Figure 2.2.4 Actual image of gun with Beetle and IR Sensors
	
Figure 2.2.5 Design diagram of glove with Beetle and IMU	Figure 2.2.6 Actual image of glove with Beetle and IMU

Section 2.3 Main Algorithm

Loop Until Logout Action Detected:

1. Beetles read sensor data and send data to the relay laptop via bluetooth.
2. Relay laptop sends the data to the Ultra96 via TCP/IP with SSH Tunneling.
3. Main server on Ultra96 relays necessary information to the AI Algorithm.
4. AI Algorithm utilizes an FPGA accelerator to run data on trained weights.
5. Main server receives results from the AI Algorithm.
6. Main server runs actions through the game logic in the GameEngine on Ultra96.
7. Main server reads the latest GameEngine state and publishes to MQTT Broker.
8. Visualisers receive published state from MQTT Broker subscription for display.
9. Visualiser app publishes whether grenades hit or missed to the MQTT Broker.

When Logout is detected, Visualiser displays a game over message. More detailed high level sequence diagrams can be found in Appendix 2.3.

Section 3 Hardware Sensors (Min Jing)

In this section, an overview of all hardware components will be displayed first in section 3.1. Then, the whole system is divided into three circuits and the detailed schematic, pin table, and power supply are shown in Section 3.2. In Section 3.3, the libraries for each sensor data will be discussed.

Section 3.1 Hardware components

Section 3.1.1 Bluno Beetles DFR 0339

Bluno Beetle is an arduino uno board with Bluetooth 4.0. It is small, light and sturdy and thus chosen for this project. It can collect data from sensors and transmit the sensor data to a computer via Bluetooth. In this system, 6 Beetles will be used in total, 3 per player. Each beetle will be connected to the sensors on the gun, the vest and the glove separately. The power supply of each Beetle is 5V.

Figure 3.1.1 Bluno beetle DFR 0339

Section 3.1.2 HX-M121 Infra-Red Transmitter and Receiver

IR transmitters and receivers are used to remotely send signals. 2 pairs of IR sensors will be used as the gun and the target on each player. The gun equips the transmitter while the vest equips the corresponding receiver. This infrared receiving distance is said to be 10 meters and requires supply voltage, 5V.

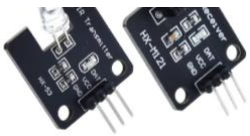


Figure 3.1.2 HX-M121 Infra-Red transmitter and receiver

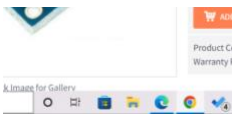


Figure 3.1.3 MPU6050 GY-521 triple axis accelerometer

Section 3.1.3 MPU6050 GY-521 Triple Axis Accelerometer

Triple axis accelerometer is an electronic device that measures the acceleration and position of the object in 3 axes as it has 3 accelerometers and 3 gyroscopes built into it. The 3-axis data collected from the IMU sensor can help with gesture detection later. 2 IMU sensors will be used, with each equipped on the player glove to collect the hand motion data of the player. The supply voltage ranges from 2.375V to 3.46V.

Section 3.1.4 Lilypad

Lilypad board is a small and inconspicuous voltage regulator. This board can pop up a AAA battery voltage to 5V. In this system, 6 LilyPads are used, each with a AAA battery as Beetle power supply.

Figure 3.1.4 Lilypad

Figure 3.1.5 Buzzer

Section 3.1.5 Buzzer

In this system, 2 buzzers are used. Each of them are connected to the gun circuit and to inform the users that the gun has shot through audio output.

Section 3.2 Circuit schematics and pin tables

Section 3.2.1 Gun Circuit

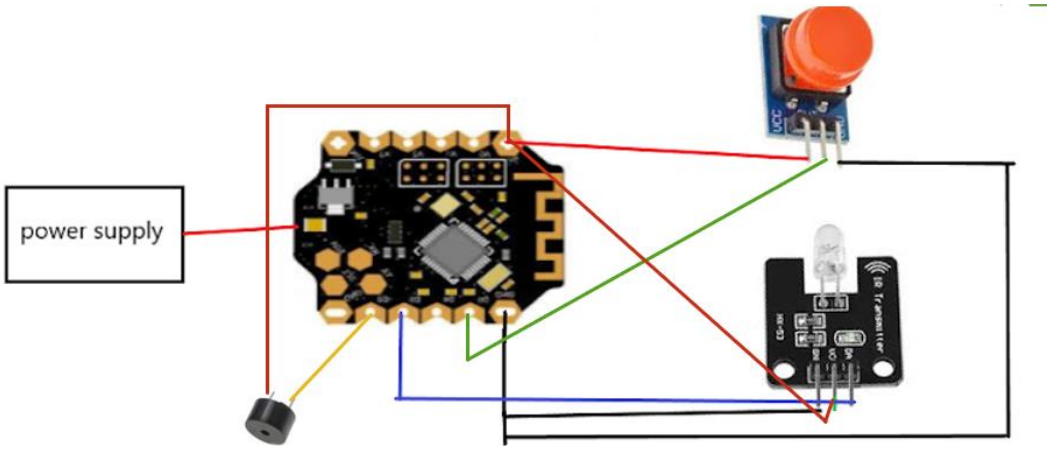


Figure 3.2.1 The gun circuit schematic

Beetle	Button	IR Transmitter	Buzzer
Vcc	Vcc	Vcc	Vin
GND	GND	GND	
D05	OUT		
D03		OUT	
D02			OUT

Table 3.2.1 The gun circuit pin table

Section 3.2.2 The vest circuit

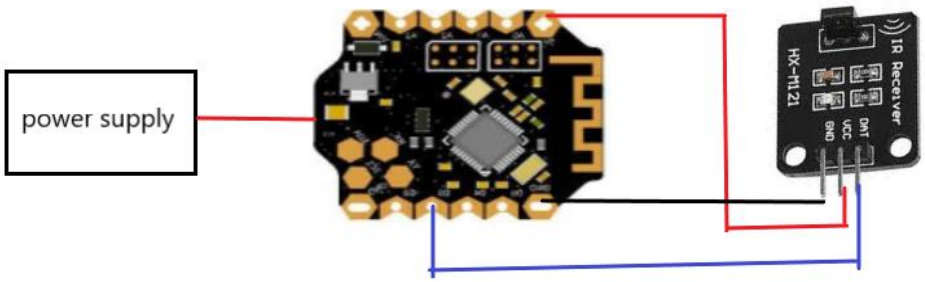


Figure 3.2.2 The vest circuit schematic

beetle	IR receiver
Vcc	Vcc
GND	GND
D03	OUT

Table 3.2.2 The vest circuit pin table

Section 3.2.3 Glove Circuit

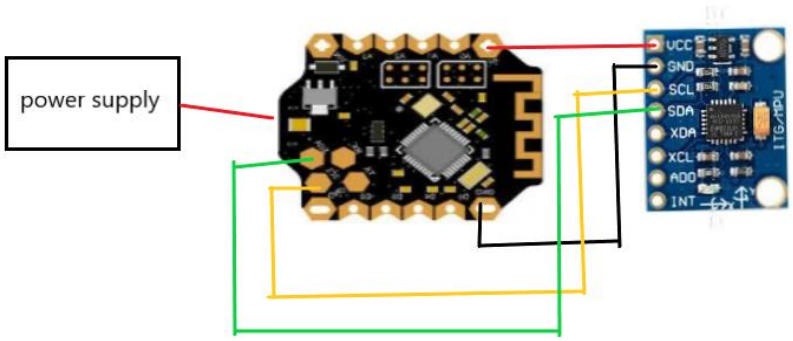


Figure 3.2.3 The glove circuit schematic

beetle	triple axis accelerometer
Vcc	Vcc
GND	GND
SCL	SCL
SDA	SDA

Table 3.2.3 The glove circuit pin table

Section 3.2.4 Power Supply Circuit

The following image shows the power supply circuit for each beetle. Each Lilypad will mount one 1.5V AAA battery. The Lilypad maintains a constant supply voltage of 5V and directly connects to each beetle as the power supply.

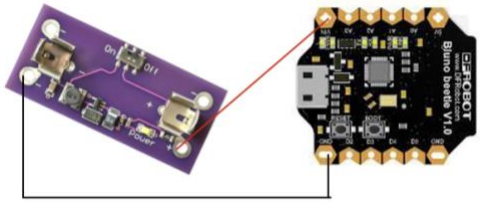


Figure 3.2.4 The power supply circuit schematic

Lilypad	**Beetle**
positive	**Vin**
negative	**GND**

Table 3.2.4 The power supply circuit pin table

Section 3.3 Algorithms

Section 3.3.1 Algorithm for IMU (Glove)

The "MPU6050.h" arduino library is used for IMU code which can help collect and calibrate IMU data. To set up, ***mpu.initialize()*** is called to initialize and calibrate the data. Calling ***mpu.getMotion6()*** gets the gyro and acceleration data from the IMU sensor. The line **`*mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz)*`** directly stores the collected data to the variables *ax,ay,az,gx,gy,gz*, where ***a*** stands for accelerometer, ***g*** gyroscope, and ***xyz*** the 3 axis. While experimenting, the data was found to range from -35000 to 35000 which can cause Bluetooth data transmission difficulties. Hence, a map function is utilized to scale the data to 0-255. For example: **`*ax = map(ax, -35000, 35000, 0, 255)*`**. The following image shows the data from the serial port. We can see the data collected from IMU fluctuate during as it moves and stabilize when it stays put.

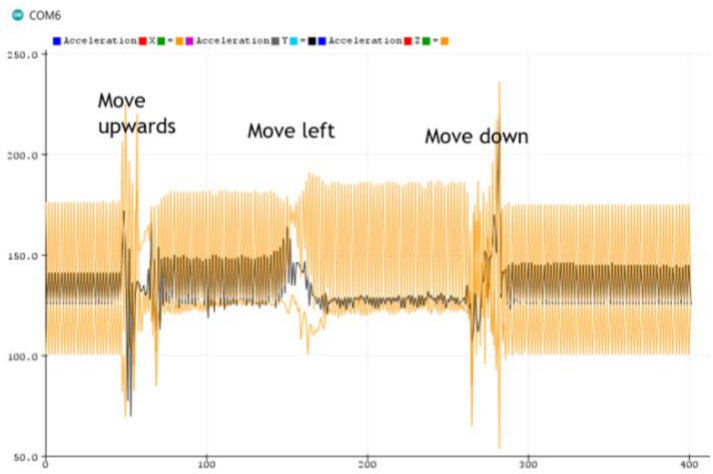


Figure 3.3.1 IMU data shows in serial port

Section 3.3.2 Algorithm for Vest circuit

The arduino library “”IRremote.hpp” is used to program the IR receiver to receive data from IR transmitters. ***IrReceiver.begin(PIN_NUMBER)*** is used to start the IR receiver. In this system, PIN_NUMBER is **3**. ***IrReceiver.decode()*** function is called to detect the receiving of data. If data is received, the function returns true. Then the data received is recorded in ***IrReceiver.decodedIRData.decodedRawData***. To avoid friendly fire and interference from other devices, an ***expectHex*** value is set for each pair of vest and gun. Only when the *IrReceiver.decodedIRData.decodedRawData* is the same as *expectHex* data, would we detect a valid shot. Then the function ***IrReceiver.resume()*** is resumes receiver for next receive.

Section 3.3.3 Algorithm for Gun circuit

The arduino library “”IRremote.hpp” is used to program the IR transmitter to send data to the IR sensors in the vest. ***IrSender.begin(PIN_NUMBER)*** is used to start the IR transmitter. In this system, PIN_NUMBER is **2**. To send data, ***IrSender.sendNEC(sAddress & 0xFF, sCommand, sRepeats)*** function is used in the gun circuit. In this system, taking player 1 as example, the address is *0xFF91* and data sent is *0x91* with no repeat. Hence actual function in code is ***IrSender.sendNEC(0xFF91, 0x91, 0);***.

Section 3.4 Power supply design

Following table 3.4.1 shows work voltage and current for each component in the system according to the datasheet of each. We found that the power supply for the system should be 5V. Then we have 2 choices for the power supply design: use 4 AAA batteries to create a 6V power supply or use only 1 AAA battery and a voltage regulator to pop it up to 5V. The first choice has more lasting time but because of the weight, it is not that wearable. Then table 3.4.2 shows the total power and lasting time of each circuit when using the voltage regulator. We found that all lasting time is enough for the whole game. Also, during the actual test, all circuits can run for more than 2 hours. Hence, eventually we chose the second choice, Lilypad and 1 AAA battery as the power supply.

	Voltage (V)	Current(mA)	Power(mW)
beetle	5	14	70
MPU-6050	2.375-3.46	3.8	13(at most)
IR transmitter	5	20	100
IR receiver	5	5	25
Buzzer	5	6	30

Table 3.4.1 The power of each component

	Power(mV)	Lasting time(h)
Laser gun	200	6.6
Vest	95	13.89
Glove	83	15.9

Table 3.4.2 The total power and lasting time for each circuit when user voltage regulator

Section 4: Hardware AI (Anderson)

To facilitate the gesture recognition features for action detection, we will be using Artificial Intelligence, or Machine Learning. This recognition system needs to be generalisable, accurate, and fast to match the real time requirement of a laser tag game. Generalisability can be achieved by having a greater variety of data, accuracy depends on the model itself, and an FPGA accelerator is utilized to help in achieving the speed factor.

Section 4.1 Data Collection and Preprocessing

As we train the model, python Pandas is utilized for data manipulation both during gameplay and training of the model along with useful libraries such as Sklearn, Scipy and Numpy for preprocessing the data.

Section 4.1.1 Data Collection

A python Keyboard library was utilized during data collection to label the IMU data based on keypress in real time as our trainers repeated the gestures to be classified such that the data is labeled as correctly as possible and subsequently stored in a .txt file. This text file stored data in the order *ax*, *ay*, *az*, *gx*, *gy*, *gz*, *Activity_code* to be read using pandas *read_csv()* function. During data collection, the IMU samples at a rate of around **40 Hz**, and by the end of the data collection for actual training of the model, a total of 591991 data points were collected, with the distributions as follows:

```
idle       349338
grenade     65999
exit        61689
shield      58449
reload      56516
```

The actions were conducted with obvious pauses at the end of the actions to give the labeling person time to release the keypress such that the downward motions of certain actions like shield or reload will not be classified as part of the actions.

Section 4.1.2 Data Preprocessing

Faulty data such as data collected during a fragmentation or abnormally large numbers that exceed the expected values of 0 - 255 are filtered out. Furthermore, we segment the data by implementing a sliding window of size 20, equating to roughly half a second, and stride 5 steps, equating to about one-eighth of a second as it slides, and label each window of data based on the mode of the activity of each window. This method of preprocessing allows us to also be able to label transitions for idle states to the gestures much more accurately when more than half the action is the gesture. Feature extraction was not conducted to train the data due to the nature of a Convolutional Neural Network (CNN) that does the feature extraction on its own which most of the time and brings better results than the standard manual extraction and Multi-Layer Perceptron (MLP). [25]

Section 4.1.3 Data Visualisation

Data collected were visualized to visibly see if there are distinct features in the data that a machine would be able to differentiate. Refer to Appendix 4.1.3 for the visualisations.

Section 4.2 Building The Model

Section 4.2.1 Model Architecture

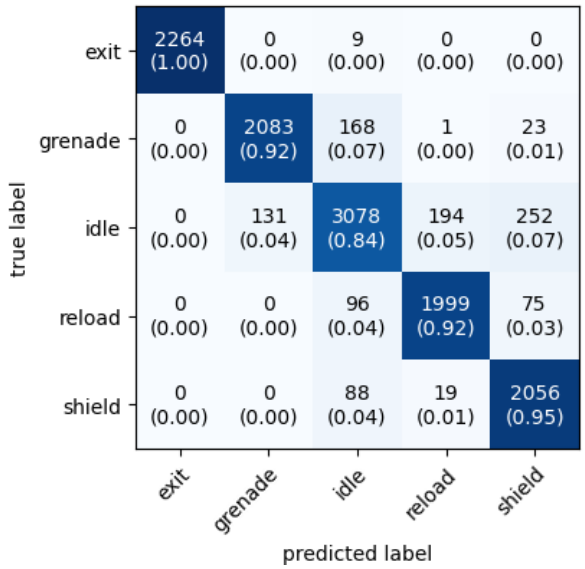
A Convolutional Neural Network (CNN) model is deployed for gesture classification in the final product [8] and it was built and trained using the TensorFlow library. It consists of an Input Layer taking in 120 data points, a 1-dimensional Convolutional layer of kernel size 3 with 20 filters, another 1 dimensional Convolutional layer of kernel size 4 with 25 filters, a Dense neural layer with 24 neurons, and a layer with 14 neurons, with an output dense layer with 5 layers. Every layer in the model is ReLU activated except for the final output layer, which is SoftMax activated. Model Summary in Appendix 4.2.1.

Section 4.2.2 Model Tuning

This final architecture was arrived at after experimenting with different numbers of layers, filters, and neurons. This final architecture was arrived at due to its small network size yet high accuracy. A dropout of 0.4 was used due to the number of our training data being small.

Section 4.2.4 Training and Validation

Our training dataset was shuffled, balanced, and split into 2 parts where 80% of it would be used to train our model, while the remaining 20% would be used for validation during the training of the model. We trained for 2,000 epochs to arrive at a validation accuracy of **91.58%**. Code base for training models and extracting weights and biases in Appendix 4.2.4



Confusion Matrix of Our CNN model

Section 4.3 Model Evaluation

Heavy emphasis was placed on the exit having no false positives to reduce the chances of accidentally logging out as much as possible. Idle predictions involve many random actions as well and thus the lower number of correct predictions.

Section 4.3.1 Testing

A separate dataset is collected during data collection to make it easier to simulate actual model deployment as the data is being parsed into the model to verify manually whether the predictions of gestures are correct. Testing dataset will be used to simulate deployment to create the algorithm later described in Section 4.5.

Section 4.4 Ultra96 Synthesis and Simulation Setup

The final model will then need to be implemented on the FPGA attached to the Ultra96. The weights and biases from the model are extracted and input into the FPGA accelerator. These weights and biases are then converted into C++ array format and input into a header file as parameters to be read and used in a C++ implementation of a CNN model. Vivado HLS has a library that helps export C++ implementations as hardware IPs to be stitched together in Vivado to generate a bitstream to be uploaded into the FPGA.

Section 4.4.1 How Model Predicts

Implementing the model in C++ requires the convolutional layers and dense neural network's weights and biases to be used correctly. When data is passed through to the Convolutional layer, take the first convolution as an example where there are 20 data points each in 6 channels, there will be 20 sets of 6 different kernels of size 3 and 20 biases. Refer to Appendix 4.4.1.1 for illustration of our architecture.

For the first convolutional layer, a kernel run slides through the dataset with dedicated kernels for each channel and sums them up into a single point, where every single result would have the filter bias added to it and subsequently run a ReLU function on the final sum in each point. Which results in 18 data points with 20 channels after running through 20 filters. This is followed by another convolutional layer with the same concept of running the kernel through the data point with 20 channels instead of 6 from the first convolutional layer. Refer to Appendix 4.4.1.2 for an illustration for easier understanding.

For the fully connected dense layers, each point in the data is multiplied by each weight to each hidden neuron, where the neuron sums up the product of the weights and the data points before adding the bias of the neuron followed by our activation function, ReLU. This is actually a dot product of the weights and the input layers followed by addition of the biases before going through the activation function. Assuming 24 neurons in the hidden layer, 24 * 375 matrix undergoes a dot product with the input layer of matrix 375 * 1 dimension, resulting in a result of matrix 24 * 1 dimension to undergo the next dense layer. A scaled down illustration of the matrices functions on 2 Neurons in the hidden layer and 3 data points is as follows where **a** and **b** are the results. Refer to Appendix 4.4.1.3 for the math formula.

Section 4.4.2 Implementing Calculations in C++

The next challenge was to implement the two above mentioned types of networks into C++ implementation.

```
data_t val;
conv0: for(int i=0; i<CONV_0; i++){
    for (int j=0; j<(20-KERNEL_0+1);j++){
        c0.convolute(in_buffer_0,w_conv_0,i,j,val);
        conv_out_0[j*CONV_0+i]=ReLu<data_t>(val+conv_b_0[i]);
```

For the convolutional layer, for every filter (20 for the first conv layer), a Conv class is created, and calls a function taking in the filter index *i*, the input *in_buffer_0*, weights of the layer *w_conv_0*, and the step *j*, and outputs to *val*, which is then summed with the bias of the filter and run through a ReLu function. The final output is input into the *conv_out_0* layer output buffer at the *i* index of the *j* layer. for us to conduct our next layer's calculation by just accessing it since out 120 data points will also be flattened to a single array.

```
l0:for(int i=0; i<LAYER_0; i++){
    m1.dot_prod(conv_out_1,w_layer_0,i,val);
    in_buffer_1[i]=ReLu<data_t>(val + bias_0[i]);
```

Dense layers dot product is implemented similarly by having a Matrix class created to call the necessary functions. The output from the previous layer is used as input, in this case *conv_out_1*, the weight *w_layer_0*, the *i* index to extract the correct layer weights for that iteration, and output to *val* which is then summed with the bias, and run through a ReLu function at the end and input into a buffer to be used in the next layer. The final layer does not undergo ReLu activation but is sent to the DMA output channels to undergo SoftMax using a python function as exponential functions are much more complex in C++.

Section 4.4.3 Optimizing the Network Implementation

A few optimisation options can be used during implementation of the convolutional and dense layers in C++, namely array_partitions, unroll, and pipelining. As seen in our code base snippets in Appendix 4.4.3.1 to 4.4.3.3, we called #pragma HLS <optimization option>. These optimizations allow for better resource utilization, leading to better throughput, and thus better speedup as more data can be computed at the same time. [21]

Without optimization, our resource utilization was very low, around 10% of the resource available. Comparing the case where we implemented pipelining, unrolling, and partitions, we notice there is an increasing resource utilization trend. Refer to Appendix 4.4.3 for report output tables.

Referring to Appendix 4.4.3, from noOptimization, to addPipelines, to addUnroll, to addPartitioning, and fullyOptimized versions, we notice that there is an increasing trend of resource utilization. The difference between addPartition and fully Optimization is the way the arrays are partitioned, which trades off the Flip Flops for more usage of the LookUp Table. The final implementation would be the fully optimized model where LUT is more utilized as it is theoretically faster than FFs in accessing multiple data.

Section 4.4.3.1 Pipelining

Pipelining was used to copy only the weights and data that are necessary to be used for calculations. This speeds up the read and write since processes do not need to wait for the previous iteration to complete before starting. We can use pipelining at this stage as the weights and data points are independent of each other, thus pipelining would not lead to inaccuracies. We see in the table above where latency decreases with pipelining. Refer Appendix 4.4.3.1 for pipeline code use. [22]

Section 4.4.3.2 Unrolling

Unrolling was used to calculate multiplications and summing the multiplications to the same output address. With unroll, we can achieve the same effect as calling the same functions wrapped in the loop multiple times in parallel that all add to the same address which allows us to achieve better lower latency. Refer Appendix 4.4.3.2 for unrolling code use. [23]

Section 4.4.3.3 Array Partitioning

Array partitioning breaks large arrays, such as input buffers and weights, into smaller chunks to allow for concurrent reading and writing as more read and write ports would be available. Array partitions have a dim parameter, which when not specified, completely partitions, and just as the addPartitions report, utilizes more FFs while having the correct input dim uses more LUTs. For our convolutional weight arrays, we partition only the 3rd dimension only the 2nd dimension for dot product as we only these respective areas are required to be copied over in the pipelining stage. Refer Appendix 4.4.3.3 for array partitioning code use. [24]

Section 4.5 Model Deployment

Section 4.5.1 Start of Motion Detection

The data transfer from our hardware to external comms to pass to the AI never stops. This necessitates some form of detection to know when a motion begins. Referring to Appendix 4.5.1.1, we observe that the range or standard deviation of the accelerometer seems to be the best form of action detection as the spikes in these values beyond a certain threshold often corresponds to a start of motion (Our test data is also labeled on the fly during collection, allowing us to determine the correspondence). Standard deviation of the accelerometer was chosen as threshold as it was observed to produce more obvious spikes during motion start. Feature extraction found in the following public repo was referenced to be used in my implementation.

Section 4.5.2 Algorithm to Filter out Noisy Predictions

Section 4.5.2.1 Sliding Window

Even with the threshold for start of motion set, as our model takes in 20 data points, which is half a second of input to provide an output, a timing that is usually a fraction of the timing of an actual gesture. Thus, we append to a data frame with the 6 new data points until there are 20 sets in the data frame, check threshold before inputting into the FPGA, remove the 5 data sets at the back and loop the process, making us a sliding window of stride 5. There are also cases of misclassification of the data. Refer to Appendix 4.5.2.1 for noisy predictions and the actual labels that should have been output.

Section 4.5.2.2 Percentage Certainty Threshold

As our FPGA model returns 5 floating points, which we run through a python SoftMax function, to obtain the percentage certainty of each action. We can use this percentage certainty as a second check by only allowing a reading above a percentage *P* to be classified as a class, otherwise, it would be detected as not a gesture, *idle*.

Section 4.5.2.3 Margin of Error Acceptable

Since we implement a sliding window, and a gesture can range from 1 and a half second to 3 or more seconds, the ideal model should predict the same motion a few times in a row. Thus, I keep a list of consecutive prediction hits, and to account for actions not meeting acceleration threshold implemented or low classification percentages that are borderline, I allow for this list to take in a variable number *E* of idle predictions by having a countdown on the margin of error.

Section 4.5.2.4 Required Consecutive Detection

If the size of the list is more than a certain length *L*, we can be more certain that it is an action. We then extract the mode (most frequent prediction) in this temporary list and output it as the result of the prediction algorithm. With these logic checks in place, we successfully clean our data off noise and output the predictions correctly as in Appendix 4.5.2.2. Furthermore, the variables *L*, *P*, *std_a Threshold*, and *E* can all be easily tweaked to adjust the sensitivity of the deployment of the model, making it possible to recalibrate new models easily. The codebase to achieve this printing to prove authenticity can be found in Appendix 4.5.3 and algorithms found in Appendix 4.5.4.

Section 4.6 Progressions Made Since Initial Design

Section 4.6.1 Removed Points

Section 4.5 Evaluation of AI Design with Given Dataset in initial design was removed as there were many overlaps with other sections. Evaluations were done during training (confusion matrix) and FPGA co-simulation was done during HLS stage to ensure the weights and biases were applied correctly and that the C++ implementations give correct outputs. Resource utilization was also already shown in section 4.4, and usage of PMBus to power consumption [2] has been demonstrated in individual progress check but not applied in any of my optimizations for its lack of significance to the AI during the individual performance check. Also, ultimately 2 models were implemented as opposed to 3, as the complexity of the LSTM RNN [1] to implement in C++ resulted in failure. Comparison with the remaining MLP and CNN made CNN the obvious choice. [25] CNN being a more advanced form of deep learning also better suits gesture detection as the computer algorithm may be able to extract features that manual extractions may otherwise be unable to. DMA usage and stitching of exported IPs were demonstrated in the individual progress check and not this report.

Section 4.6.2 Iteration from Online Dataset to Actual Datasets

Section 4.6.2.1 Online Dataset and HLS MLP Implementation

From the online dataset [18], data was split based on user into different text files, and there were accel and gyro folders for the respective data. The way to read the data from text files had to change as I collected all 6 data along with the label into a single row to use instead. Appendix 4.6.2.1 shows the difference in folder structure that allows me to easily exclude data that are suspected to be faulty. I moved on to make a neural network (MLP) with 77% validation accuracy for the individual progress check with confusion matrix in Appendix 4.6.2.1.a. As I intended to use CNNs, having only just figured out dot product function in C++ implementation and not the Convolutional process in C++ until later, I was unable to achieve a desirable model accuracy to present by the time without feature extraction.

Section 4.6.2.2 HLS CNN Implementation and Real Data

After getting the C++ CNN implementation working, I was given datasets by the internal comms which I used to get the first desirable result of the as shown in Appendix 4.6.2.2. I then suspected faultiness in the data given to me, along with the small size for datasets collected for the too good to be true results. Inspired by the online data, I requested the internal comms for the ability to label on the fly and we collaborated to make the most accurate dataset collector possible. During data collection, all 5 people involved did gestures without guidance to simulate unseen data, which worked well to achieve 10/14 actions for 1 player evaluation, 36/40 correct actions submitted to eval_server on 2 player evaluations, and 39/40 correct actions submitted to eval_server on unseen player evaluations. Accuracy improved in 2 player from 1 player mode as the model was trained with less data for the 1 player mode there was inaccurate or mislabeled data due to having less experience with gathering data. Data was recollected for 2 Player mode to achieve greater precision in labeling and to collect every team member's data, leading to the AI's success as data labels were cleaner, more precise, and had an increased variety. The final confusion matrix is shown in the previous section, Section 4.2.4. Not having a 100% confusion matrix is acceptable as there are other checks done by the algorithm and 100% might actually be an overfitted model and thus above 90% accuracy would already be enough. This additional algorithm was not proposed before due to the misconception that the trained FPGA model alone would suffice.

Section 5 Internal Communications (Muthu)

The main aim of internal communications is to pass the data from the sensors to the ultra96. The data included the number of bullets shot by the player(captured by the button trigger in the gun), the number of times the player was hit(captured by the sensors in the vest) and the data from the IMU sensors which were used for prediction of the user movements. This section will elaborate on the various protocols adopted to ensure that the data can be passed on effectively.

Section 5.1 Manage Tasks on Beetles and the Processes on Laptop

The initial plan to tackle the multiple tasks on the Beetles, was using the millis() function in arduino, rather than the built in delay function, to minimize delay. Using the delay function seems to hold the processor while that task is being executed. This would be prevented if the millis function is used [3].

Section 5.1.1 Arduino

There were primarily three tasks that were occurring concurrently in all the beetles. One of the tasks was to collect the relevant data from the sensors. The second task was to transmit it over to the relay node. The third task was to check for serial communication from the laptop to check for the handshake packet, which is essential so that the laptop can establish a connection with the laptop.

Section 5.1.2 Relay Node

The initial plan was to use threading within the relay node to tackle the different beetles and the ultra96. Hence, in order to prepare for the 1 player game, there were 5 threads which were implemented in the relay node: 3 threads for the three beetles which were connected through Bluetooth to the relay node, 1 thread to communicate with the ultra96 and 1 thread to communicate with the visualiser using the ultra96. To manage these threads, I used the threadpool executor library in python. This library allowed efficient handling of the threads without the need to worry about race conditions or dealing with the global lock in python.

However, after the 1 player testing, the threads were converted to processes since the threadpool executor did not function very well. There was significant latency when the beetles tried to communicate with the relay node.

For the 2 player game, we decided to use two relay nodes, with one relay node for player 1 and one relay node for player 2. Converting from threads to processes was a small change which decreased any possible latency that might have arisen from using threads. In the end, we used 4 processes. 3 processes for the different beetles and 1 process for the ultra96. MQTT was implemented using the paho-mqtt library. This library had a start function which started the communication thread. This thread was called within the process which started the ultra96.

Section 5.2 Setup and Configuration of BLE interfaces

For the communication in between the Bluetooth devices, there will be peripheral devices and central devices. Peripheral devices collect the relevant data that will be transferred to the central device, where the data can be processed [19]. In this scenario, the laptop will be the central

device and the beetles will be the peripheral devices. The setup of the BLE interfaces involved the various AT commands, using Arduino, to set the beetles to the required configuration. The AT commands can be seen in the appendix 5.2.1.

Section 5.2.1 Relay Node

The following code shows how the beetles were initialised in the relay node. First, the beetles were connected by utilising the MAC addresses that were obtained using the AT commands. Subsequently, I obtained the 'dfb0' service and the first characteristic within the service as it was this service which received the messages that were transferred during Bluetooth communication. Each beetle was also assigned a Delegate object, which was necessary since this delegate object is the only way to check what data has been received through Bluetooth communications. The initialisation function can be seen in appendix 5.2.2.

After all these setup, peripheral.waitForNotifications(timeout) function is called continuously within the process to check for data transmission. This function returns false if no data has been received after the timeout. Else, it will return true and will call the handleNotification function within the delegate object.

Section 5.3 Protocol

The BLE allows the baud rate to be set from 9600 to 115200 [5]. Initially, the baud rate was set to 9600. However, it was later changed to 115200. The packet structure in appendix 5.3.1 the packet format which was used to send data from the beetle to the relay node used in IMU sensors. This packet houses 12 bytes of data. These 12 bytes correspond to the 6 different readings which were captured by the IMU. Each of the readings were allocated 2 bytes. The packet structure in appendix 5.3.2 shows the packet format which was used to send data from the beetle to the relay node used in the Gun or the vest. The major difference is that there is now a sequence number, which is allocated 2 bytes. Furthermore, the data is also split into 6 different values, similar to the IMU. However, only the first value is set to 1, which indicates that the sensor has obtained a reading.

The following diagram shows an example of the data sent from the Gun/vest. Note that the data, which is bolded, is essential.

170	2	4	3	**1**	0	0	0	0	0	Pad	Check

The following diagram shows an example of the data sent from the IMU. It has 6 essential data points.

170	2	4	3	**127**	**130**	**65**	**80**	**100**	**170**	Pad	Check

The header of the packet is 170. Followed by that is the type of device, type of packet, followed by the data itself. The checksum is appended at the end of the packet. For the acknowledgement packet, it consists of the header and the acknowledgement number itself. Subsequently, its padded with 0's to fill up the 18 bytes worth of data.

For the handshake algorithm, the relay node will first send the SYN PACKET to the beetle. Subsequently the beetle will send the ACK PACKET back to the beetle. The beetle will then send the SYNACK packet to complete the handshake protocol.

The following tables summarizes the types of ACK,SYN and SYNACK packets that were implemented:

	Initial	Final
SYN PACKET	'H00000000000000000000'	[170,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
ACK	'A00000000000000000000'	[170,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,170 ^ 2]
SYNACK	'S00000000000000000000'	[171,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

Initially, the packets contained just letters appended with 19 bytes worth of 0's. However, after much consideration, they were changed to arrays with the first index being the header values, followed by being padded with 19 0's.

The initial plan was to use Cyclic Redundancy Check to check for any errors while transmitting the data over the network.If any error is observed, the packet will be forced to be resent after a while.This can either be implemented from scratch or using relevant libraries [16], which will be decided later on. However, this was found to be an incorrect approach to do checksum. Hence, I decided to do the XOR operation for all the bytes in the data to retrieve the checksum. This is easily implemented on both the beetles and the relay node.

Section 5.3.1 Receiving Data on the Relay node

After receiving data, I calculated the checksum of the packet to ensure that it was accurate. If it's accurate, the data is passed to a process Data function, where the data is retrieved using the python struct unpack feature. Considering that there are 3 types of data (from the IMU, Gun and the Vest), the unpack method would unpack the 3rd to 14th byte of data.

```
def processData(self, data):
    recv = unpack('HHHHHH', data[3:15])
    packetNo = recv[0]
    print("proces Data")
    print(self.s (parameter) self: Self@MyDelegate
```

This method would then unpack the data to obtain the 6 data points, which will be sent to the client. The IMU sensor would send 6 relevant data points. However, the Gun/Vest would only send a value of 1 to the client.

Section 5.3.2 Communication with the Visuliser

To ensure that the user is updated whenever the beetles disconnect or reconnect with the relay node, it was essential that there must be a certain form of communication set up between the internal comms and the software visualiser component. I used the paho MQTT library to subscribe and connect to the visualiser. Whenever, there are any form of disconnections, the messages in the following format will be sent:

Status	Message
Gun disconnected	{"imu": " ","gun" : "no","vest":" "}
Gun reconnected	{"imu": " ","gun" : " ","vest":" "}
IMU and gun disconnected	{"imu": "no","gun" : "no","vest":" "}

As seen from the table above, the format of the message sent to the MQTT player is fixed. The message is a dictionary typed object. When a device is disconnected, the value of the device is set to be “no”, which is then sent to the visualiser. When the device is reconnected, a new message is sent, with now the value of the device set to be an empty string.

To aid this functionality, certain global variables were implemented. The first variable, a message queue, mqttQueue, which receives a message from individual beetles whenever they disconnect. The message is a set with the following format: (“device” :”status”). The status is either “DC”, which stands for disconnected, or “RC”, which stands for reconnected. Hence, the beetles constantly update the queues regarding the connection status of the devices.

This mqttQueue is then tracked in the client process. If any item is placed in the queue, this item will be retrieved since we are always checking if the size of the queue is above 0, indicating that the queue is not empty. Then, the message is retrieved and sent to the publish function of the mqtt class. This publishes function reads the status and then updates a global list which tracks the disconnected devices. Subsequently, the disconnected list is iterated through to generate the appropriate message to be sent to the visualiser. A list tracking the disconnected devices is important since more than one device might not be connected. Hence, the message sent to the visualiser must update all the devices which are disconnected.

Section 5.3.3 Stop and Wait

The stop and wait algorithm [27] is as such:

The beetle sends a packet to the relay node, say Packet 0. The relay node computes the checksum and if it has received the appropriate packet, it will send ACK 1. This confirms that the relay node has received packet 0 and is waiting for packet 1. Subsequently, the beetle will send Packet 1 and the relay node will send ACK 2.
If the packet from the beetle to relay node gets lost, or the packet at the relay node ends up corrupted, or the ACK packet from the laptop is lost, the beetle resends the previous packet after a timeout. The stop and wait protocol are only used for the guns and the vest. It is not used for the IMU devices since it slows down the amount of data sent by the IMU drastically.

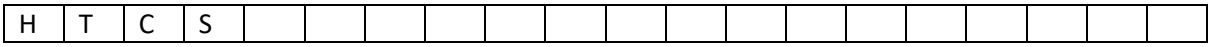
Section 5.4 Handling Reliability Issues

Despite all the measures taken, there will still be many problems such as the loss of connection. This section aims to address these problems and provide a solution for them.

Section 5.4.1 Packet Fragmentation:

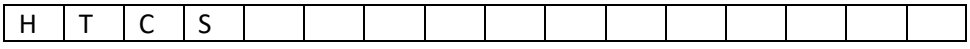
If the data is larger than the maximum transfer size, then it will have to be split into smaller packets. Then, for each packet, there will be an offset and packet id [4] added so that it can be reassembled once it has been transferred over. The following diagram represents how packet fragmentation is implemented at the time of two-player testing.

Perfect Scenario: Header == 170 && length == 20. Check for checksum and append the result.

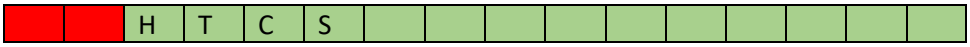


Fragmentation:

1)Header == 170.Append to buffer.

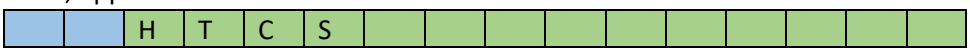


2)Header !=170:



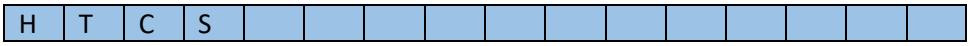
If the buffer is empty, remove the first few bytes then append the rest of the data to the buffer.

Else, append the data to the buffer.



Data received from the beetle:

Existing Buffer:



Buffer appended with the required data.

A packet is not fragmented if it meets the 3 of the following criteria: Correct header, Correct checksum and a perfect length of 20 bytes. Else, check if the header of the data received is 170. If it is, append the data to the buffer. Else, remove the first few bytes till the appropriate header is obtained. At every iteration, check the fragmented buffer if its length is greater than 20. Then, proceed to pop from the buffer and append it to a new buffer, until the header of the buffer is 170 again (Start of the next packet). For the retrieved data, do the relevant checksum to verify it. The image in appendix 5.4.1 shows the following algorithm in action.

Section 5.4.2 Connection Loss

Initially, if there was a connection loss between the laptop and the Beetles, then the communication in between the devices will be stopped. In that case, the scanner class from the bluepy library will be used [17]. It will automatically scan for devices that are relaying information . The data obtained can be used to set a connection.

However, this approach was changed since it was inefficient.In order to deal with connection loss, the individual MAC addresses of the devices are stored and once the device disconnects, the relay node will constantly search for the device. If the MAC address of the device can be found, then the relay node will connect back with the beetle. In order to ensure that the relay node searches for the device, the btled disconnect error will be called when the device is disconnected. Then, the relay node will catch this exception and try to reconnect back.

Section 5.5 Issues that had to be resolved

Section 5.5.1: Temporary stoppage of the IMU

During prolonged testing, we found out that the IMU stops sending data after some time (usually 15-20 minutes). There were many possible reasons for this, but it was primarily due to the IMU sending too much data with no pause in between two subsequent packets. A potential way to resolve this would be to reduce the frequency of the IMU, which would result in retraining of the data. A potential work around was to alert the user when such a situation occurred by sending a disconnect message. This would prompt the user to reset the beetle connected to the IMU, which resolved the issue. Since this issue was found quite late, around week 12, we did not retrain the model since it would involve additional work and we did not want to affect a model which was already working well.

Section 6 External Communications (Hui En)

This section explains the implementation of the external communication between Ultra96, laptop, visualizer, and the evaluation server. External communication involves integrating and enabling data transfer between the various components of the project over the internet. The game logic component is also integrated within the external communication code.

Section 6.1 Communication between Ultra96 and Laptop

The laptop will be the relay node for sending data from the sensors to Ultra96 to be processed. The laptop will need to implement SSH tunneling to connect to Ultra96, before sending data over to Ultra96 via TCP. The data received by the Ultra96 is shown in the table below.

IMU	`{"P": 1, "D": "IMU", "V": [127,127,127,127,127,127]}`
GUN	`{"P": 1, "D": "GUN", "V": 1}`
VEST	`{"P": 1, "D": "VEST", "V": 1}`

Table 6.1: data format received from laptop

The member *'P'* denotes the player number. The member *'D'* denotes which device the data belongs to. The member *'V'* denotes the value for the corresponding device; a list of 6 data points for IMU, value of **1** if the gun is shot or if the vest is hit. Upon receiving the data from laptop, the code will store each data to their respective buffers.

```
IMU_buffer = queue.Queue()
GUN_buffer = queue.Queue()
vest_buffer = queue.Queue()
```

Section 6.2 Communication between Ultra96 and Evaluation Server

The evaluation server is used to evaluate and guide the gameplay during the evaluation of the system. The JSON we send to evaluation server and receive from evaluation server is the same. The table illustrates the data transfer between evaluation server and evaluation client that is running on Ultra96.

Evaluation client (Ultra96) to evaluation server Format: **Len_crypt(JSON)** Contains the action done by each player, and the corresponding player state as a result of the action done	Evaluation server to evaluation client (Ultra96) Format: **Len_JSON** Contains the correct player state in the event that the player state we send over is incorrect, thus preventing error carry forward
{	

```
            "p1": {
                  "hp": 10,
                  "action": "grenade",
                  "bullets": 1,
                  "grenades": 1,
                  "shield_time": 0,
                  "shield_health": 0,
                  "num_deaths": 1,
                  "num_shield": 0
            },
            "p2": {
                  "hp": 100,
                  "action": "shield",
                  "bullets": 2,
                  "grenades": 2,
                  "shield_time": 10,
                  "shield_health": 30,
                  "num_deaths": 5,
                  "num_shield": 2
            }
      }
```

Table 6.2: data format between evaluation client and server

Ultra96 connects and sends data over to the evaluation server via TCP. The data sent over is encrypted for security. Upon receiving the correct player state from the evaluation server, we make sure to replace our player state with the correct one.

Section 6.3 Communication between Ultra96 and Visualizer

After the player has performed an action, and the action is being processed, we will send the updated player state with the actions performed and the corresponding player state. The table below shows the data format transfer between Ultra96 and visualizer.

Sending to visualizer	Receive from visualizer
`The same format as the JSON in the table above`	`yes1` **or** `yes2`

Table 6.3: data format between Ultra96 and visualizer

When a grenade action is done by the player, the visualizer will send over data indicate if the grenade hit the opponent. *yes1* means that player 1's grenade hit player 2. *yes2* means that player 2's grenade hit player 1. This allows us to correctly update the player state to reflect the grenade damage.

We communicate with the visualizer using MQTT protocol, and we use the public broker *test.mosquitto.org*. To publish to visualizer, we subscribe to topic "visualizer17", and to receive from visualizer, we subscribe to topic "grenade17". An important note is that we changed the MQTT QoS from 0 to 1, as we realise messages were sometimes lost. QoS 1 guarantees that a message is delivered at least once.

Section 6.4 Concurrency

We require concurrency due to the need to integrate and allow communication between the various components of the system. The structure of the code is different for both 1-player and 2-player mode. The table below will illustrate the general differences in terms of concurrency.

Category	1-player	2-player
Concurrency	Thread	Process
number of threads/processes started	1. Game Engine 2. Eval client 3. AI detector 4. Laptop Server 5. MQTT publish 6. MQTT receive	1. Eval client (process) 2. Player 1 AI (process) 3. Player 2 AI (process) 4. Player 1 Laptop Server (process) 5. Player 2 Laptop Server (process) 6. MQTT publish (thread) 7. MQTT receive (thread)

Table 6.4: the changes from 1-player to 2-player

We will explain some of the changes we made from 1-player mode to 2-player mode.

Section 6.4.1 Switching from Thread to Process

While threading is enough for a 1-player game, we came to realise that it is very challenging to use threading for 2-player game.

1. Since we have an additional player, the system is now receiving twice as many data compared to 1-player mode. We suspect that this was causing significant reduction in the speed of the program. We had an instance where we waited for close to 10 seconds for the program to detect the action performed by the player.
2. We also encountered some deadlock issues that we attempted to fix with a timeout, but the problem resurfaced. The deadlock issue was also inconsistent,, with some runs encountering a deadlock but others were able to complete the full run.
3. There were also issues with the Python Global Interpreter Lock (GIL) that we could not understand.

The various reasons mentioned caused us to switch to using multiprocessing which was able to fix all of the issues that we encountered while using threading.

Section 6.4.2 Game Engine Not a Thread

Before switching to multiprocessing, we attempted to speed up the program by reducing the number of threads running. Hence we analysed the code and found that Game Engine thread was redundant and was able to remove the thread, and found a new way of implementing Game Engine. This change however, did not speed up the program significantly.

Section 6.4.3 Two Laptop Server

We initially thought of having only one laptop server to receive data from the relay laptop. However, by connecting all 6 devices to one laptop client made the code very slow for the internal communication. Hence, we decided to create 2 laptop clients, one for each player, resulting in the need for 2 laptop servers to receive data.

Section 6.5 Walkthrough of Code Logic

The diagram shows the overall flow of the program implemented. We will give a brief outline of how the code works, and highlight some decisions we made. You may refer to the code base given in Appendix 6.5 for the links to see the whole algorithm in detail.

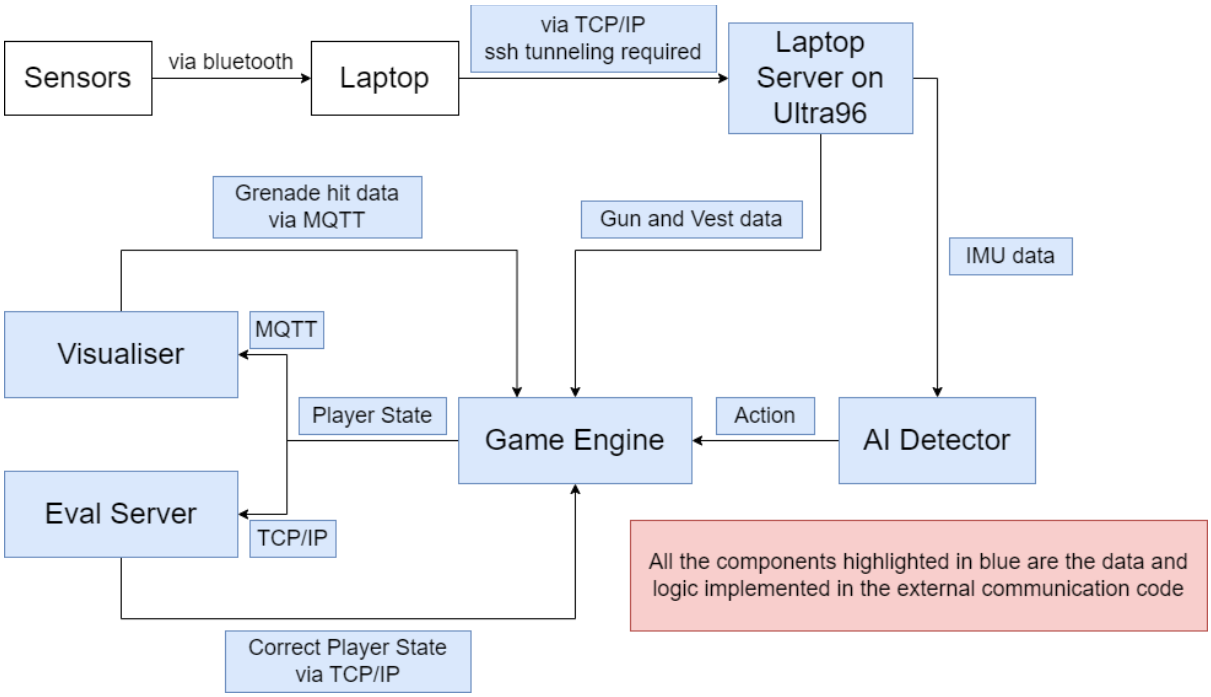


Figure 6.5: Flow of the system

1. We receive a continuous stream of data from Laptop, we store the data into their respective buffers, refer to section 6.1. We made a switch from using python list to using queues as buffers, because queues are thread safe and process safe, and popping elements from queue (O(1)) is faster than list (O(n)).
2. IMU data needs to be processed by AI_Detector, hence we pass them into the AI_Detector to predict the action. The actions predicted can be any of the following; ["logout", "grenade", "idle", "reload", "shield"]. We discard any “idle” actions, otherwise, we store the predicted action in another buffer, ACTION_buffer.
3. We then begin checking these 4 buffers: ACTION_buffer, GUN_buffer, and vest_buffer.
    a. Any actions in ACTION_buffer will be popped out and pass through game engine.
    b. Any elements in GUN_buffer, means that the gun has been shot, this will also be passed into the game engine.
    c. In addition, when checking GUN_buffer, we will also check for elements in vest_buffer to see if the bullet hits the opponent. This will be passed into the game engine.
4. In the game engine, we will update the player state accordingly. When the action is grenade, we will check the vis_recv_buffer which will contain the data we receive from visualizer indicating whether grenade hit or not. We implemented a delay as we realized that MQTT can be slightly slow in sending data over. This ensures we wait for a short time to check if grenade is hit, instead of accidentally missing the data because the program is too fast.
5. We will then send the updated player state to MQTT before sending it to Evaluation Server. There are some nuances in the code to explain.

    a. For 1-player game, we are able to send the player state to Evaluation Server and Visualizer at the same time.
    b. For 2-player game, we have to wait for both players to carry out their respective actions, update the player state before sending it to Evaluation Server. However, we do not send the player state to Visualizer together with Evaluation Server.
    c. Note that immediately after a player does an action, we will send the updated player state to Visualizer. This will ensure that when a player performs an action, the action is immediately reflected on the Visualizer.
    d. A previous version of the code sends the player state to visualizer and evaluation server at the same time. This means that if player 1 performs an action, and player 2 performs an action 5 seconds later, the display on player 1's action will suffer a delay of 5 seconds.
    e. This was a poor implementation as there is a chance that players cannot tell if their action is not displayed because of a failed prediction or due to a delay in the opponent's action.
6. Upon sending over to Evaluation Server, we will wait to receive the correct state from Evaluation Server, and replace our player state with the correct one.
7. That completes one set of actions.

Section 7 Software Visualizer (Ming Jun)

The software visualiser is an AR application that runs on a mobile phone. Its main purpose is to display the relevant game information as an overlay on top of the image displayed from the phone's camera. This provides an enhanced, engaging and more immersive experience for the players as the visualiser will provide them with real-time effects and feedback when they are playing the game. The design choices and ideation will be explained in this section.

Section 7.1 Design

Section 7.1.1 User survey and feedback

Throughout the design of my visualiser app, I did a few user surveys and got some feedback which influenced how my final design turned out to be. My user survey includes asking for feedback on AR effects, UI design, and an open ended feedback section for additional feedback. Listed below are the key feedback points which I took into consideration finalising my design.

- First person shield should be more translucent to allow for viewing clarity
- First person and opponent shield colours should be different
- Reduce the size of the grenade explosion to minimise obstruction to the view
- Show blood effects when taking or dealing damage
- Add sound effects for shooting, grenade and shield
- Show warning messages that appear on the screen when something is wrong
- Add voice-overs for warning messages to ensure that the user is alerted
- Show current and previous action to allow the user to keep track of his/her actions

Taking the points above into consideration, I decided to implement my design which is explained in the sections below.

Section 7.1.1 Information Displayed

The information displayed in the software visualiser is listed below.

- The system has a visualiser that displays real-time information of the game actions and players' status
    - User's and Opponent's HP
    - User's and Opponent's Bullet Count, Grenade Count, Shield Durability
    - User's and Opponent's Shield Countdown Timer
    - Text to show when opponent is detected by the camera
    - AR effects for different actions:
        - Blood effect when dealing and taking damage
        - Throwing a grenade
        - Grenade explosion
        - First person shield active
        - Opponent shield active
    - Sound effects for different actions:
        - Shooting
        - Grenade exploding
        - Using a shield

        - Shield breaking
        - Reloading
    - Alert messages displayed with voiceover for:
        - Out of bullets
        - Out of grenades
        - Out of shields
        - Shield already active
        - Reload unsuccessful
    - Warning messages displayed with voiceover for hardware disconnections
    - Gameover screen with winner displayed

Section 7.1.2 Overlay

The software visualiser design comprises the game overlay and the gameover screen. Figure 7.1.2.1 and 7.1.2.2 in Appendix 7.1.2 shows the game overlay and gameover screen respectively.

The game overlay is inspired and modified from the popular first person shooter game Valorant. With that in mind, the core design choice I chose to focus on is minimalism and clarity. The UI is designed in a way that provides the user with all the information required, whilst minimising the obstruction of the camera's view on the screen.

The scoreboard at the top provides a clear view of the current game status, with the colours green and red differentiating players 1 and 2. The current player's hp and shield values and bars are at the bottom left of the screen while the bullet, grenade and shield counts are at the bottom right. Similarly, the opponent's information is shown at the top right of the screen. The player's current and previous action is shown at the bottom of the screen. The rest of the screen space is used for the camera's field of view.

Section 7.1.3 Design Constraints

There are some constraints that need to be considered when developing the app. Table 7.1.3 below shows some of the constraints considered.

Constraints	Details
Phone Storage	The AR app has to be able to fit on the phone including all the assets and effects it has to display.
Battery Usage	The AR app should not drain the phone's battery too fast.
Performance	The AR app should perform all the tasks smoothly with as little latency as possible.
User Experience	The UI should be intuitive and easy for the user to understand and use.
Camera Field of View	The live display on the AR app is limited to the camera's field of view and hence has to be taken into consideration when designing the overlay.

Table 7.1.3 Design Constraints

Section 7.2 Software Architecture

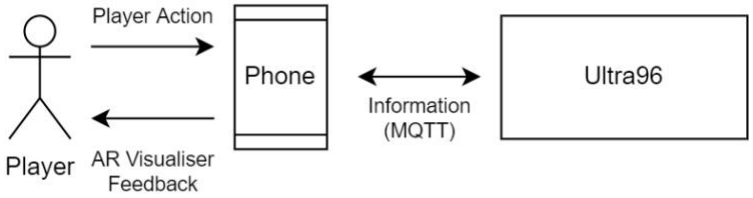


Figure 7.2 Architecture of AR Visualiser

The player is detected and tracked by the AR in the phone using image target tracking (see section 7.3.1). The phone receives action information of both players from the Ultra96 via MQTT and displays the corresponding effects or status on the AR visualiser for the players to see. A more detailed explanation of how the effects are displayed is explained in section 7.2.2 and 7.3. The AR software will primarily be developed for Android platforms, and two Android phones will be used for the game.

Section 7.2.1 Frameworks and Libraries

Unity is the main development framework for the AR visualiser. Unity is open-source, making it easier to obtain information and resources. It also has extensive libraries that are very useful for AR development. Coupled with Unity's simplicity of use, it is easier to navigate and utilise its interfaces and thus allows for faster app development.

Vuforia is the library used to develop the AR effects. Its features are easy to use and work well with unity. Github is used for tracking and version control. It also helped to simplify the process for player two integration which is explained in Section 7.2.3. Refer to Appendix 7.2.1 for the link to the Github repository of my code.

Section 7.2.2 Data inputs and networking

Information exchange is done between the software visualiser and the Ultra96 in the form of messages. MQTT will primarily be used to transmit information between the Ultra96 and the phones. MQTT is lightweight, efficient, reliable and secure [20], making it a good choice for our design.

JSON is the main format of the messages chosen to transmit between the devices. JSON is arguably the standard for exchanging data between web, mobile and back-end services. It is a relatively concise, flexible data format that can work well with many programming languages, making it a very good choice for message transmission. Table 6.2 in Section 6.2 shows the JSON message format used to send player information. Once the message is received on the phone, the software visualiser reads the information from the JSON and displays the relevant data and effects on the overlay. In Table 6.2, Player 1 performs the grenade action and hence a grenade launching effect will be shown on the screen. On the other hand, Player 2 will see a grenade exploding effect on the screen instead. Simultaneously, Player 2 performs a shield action and both phones will show a shielding effect on Player 2.

Figure 7.2.2.1 shows JSON message format of the player's hardware disconnection status. In Figure 7.2.2.1, player 1's IMU has disconnected and hence the visualiser is notified using the JSON message. The visualiser will then display the corresponding disconnection warning. Figure 7.2.2.2 shows what the disconnection warning looks like. Refer to Appendix 7.2.2 for Figures 7.2.2.1 and 7.2.2.2.

Similarly, when a failed shoot, reload, grenade or shield action is received, a message will be shown to alert the player of the failed action. Figure 7.2.2.3 in Appendix 7.2.2 shows an example of the failed message.

When the opponent is detected on the screen and a grenade is thrown at them, a grenade hit will be sent to the ultra96 from the visualiser. Table 6.3 in Section 6.3 above shows the format of the message sent. The message sent is either "yes1" or "yes2" in plain text form. The number in the message indicates which player threw the grenade.

Section 7.2.3 Player Two Point of View

The initial design was based on player one's point of view. In order to create the point of view for player two, I decided to modify the same code that I already developed. I used Github to aid me in this process. I created a new branch from my current player one's point of view code. Thereafter, I swapped all the variables corresponding to player one and two respectively. By doing so, player two's point of view is now complete, and I generated an entirely new and separate app solely for player two. Hence, I have two separate and distinct apps for both player one and player two.

Section 7.3 Augmented Reality

Section 7.3.1 AR Anchors and Effects

The use of AR anchors is to ensure that the AR effects appear in the correct positions on the screen. The software used in our app to anchor these effects is Vuforia. Vuforia allows for the use of an image to serve as the image target. When the camera detects the image target, the AR effects can be generated relative to the position of the image target on the camera screen. Figure 7.4.1 in Appendix 7.4.1 is used as the AR anchor. Our app uses this image target to generate shield, grenade and bullet hit effects for both player 1 and player 2. See Appendix 7.4.2 for the full images of the AR effects.

Section 7.4 Challenges Faced

Section 7.4.1 Two Player Integration

During the initial two player integration for the software visualiser, the app only had basic information of the player and that of the opponents' as seen in Figure 7.1.2.1 in the appendix. When we were doing the two-player test run, players continued to perform the same actions despite it being unsuccessful or invalid. Similarly, when any of the hardware components disconnected, there was no way for the player to know unless the internal communications person verbally informs them. This was not going to work for the final evaluation as verbal communication was not allowed.

In order to resolve this issue, we decided to get the internal communications to send a disconnection message directly to the visualiser whenever there is a disconnection. This will then trigger the warning message as described in Section 7.2.2 above. Similarly, the external communications will send a message indicating that an action has failed, and trigger the message as described in Section 7.2.2 above. The messages are played out loud using a voiceover so that the players can not only see but hear the message as well. This ensures that the player is aware of the warning message and can take the necessary steps to resolve the issue.

Section 8: Social and Ethical Impact (Everyone)

The use of augmented reality wearables comes with many social and ethical challenges. In this section, we will elaborate on some of the major concerns regarding these wearables and what can be done to mitigate these issues.

Section 8.1 Social Impacts

Companies like Meta have been pouring more and more resources into VR. Our AR laser tag game is similar where the virtual world is layered onto the actual physical world. With companies like Apple also rumored to be producing and releasing their VR headset. VR may become more and more prevalent. However, VR involves complete immersion into the virtual world. Our AR laser tag may be part of the motion towards blending the VR world and the physical world. Although this project is currently just a game at this stage, many of its features can be adapted and applied to future possible use. When AR is prevalent, certain sensors may be equipped for daily life. Pedestrians can make gestures a short moment before reaching traffic lights, and traffic lights will be notified earlier to allow for pedestrian crossing, reducing wait times. This is very possible as today, impact sensors on iPhones are already capable of sensing accidents [26]. We can make SOS systems as well by allowing sensors to recognise and learn stress signals unique to the users, such that during emergency events such as during a crime attack, people within the vicinity can be alerted to help while the police are contacted automatically.

Section 8.2 Ethical Impact

The most common concern that people have with such wearables is the safety of their personal data. By using these wearables, the data collected from the users can be highly intrusive. In the event of a data breach or misuse by the company collecting these data, the consequences can be very serious and would endanger many users. Hence, it is of utmost importance for us to design the system with users in mind first. The users should have as much control over what data can or cannot be used by the companies.

In order to protect users, the system could be designed in such a way whereby the data is collected and processed locally only. After the processing is done, the data is discarded and hence it will not be transmitted or used anywhere. To ensure that the data cannot be accessed by hackers, robust security measures need to be put in place to protect the data.

In a VR realm, the environment of the user is usually stationary in a single room, however, in AR prevalent worlds, much of the user's activity can be tracked as long as it is being used. Malicious parties may be able to acquire personal records of people's daily lives and plan out more elaborate crimes such as knowing when the user's belongings are most vulnerable to be robbed or stolen. Confidential information may accidentally be shared through AR cameras and the liability of these data leaks would be blurred as well. Thus, we should approach AR with caution as we innovate while figuring ways to prevent unintended detriments.

Section 9 Project Management Plan (Everyone)

Week	Tasks
21/8 - 27/8 Week 3	● External ○ Begin basic implementation of protocols ● Internal ○ Being basic implementation of protocols ○ Download and install ubuntu ○ Try to connect to the BLE module using handphone ● Software Visualiser: ○ Being basic implementation of the app ○ Come up with a potential design for the app, including layout and other features ● Hardware AI & Hardware Sensors: ○ Obtain relevant components and familiarize with the sensors ○ Begin soldering the beetle and the relevant sensors ○ Start designing relevant ML models
28/8 - 3/9 Week 4	● All members to continue the aforementioned tasks for the previous weeks ● ML Models accuracy to be above 70% ● Begin design of the prototype for the shield, guns and gloves
4/9 - 10/9 Week 5	● First component test: ○ All members to meet up before evaluation to check on the status
11/9 - 17/9 Week 6	● Individual subcomponent test: ○ All members to meet up before evaluation to check on the status ○ After test, members to rectify any problems ○ One Model to be implemented on FPGA ○ Members to also communicate regarding combining all the relevant components
18/9 - 24/9 (Recess Week)	● Members should try to combine their components ○ Internal and External comms should try to combine their code to set up the protocol ○ Hardware sensors and Internal comms personnel should test if data from sensors can be picked up by the laptop ○ Hardware AI to try to test out the ML models using data obtained from the sensors ○ External Comms personnel to communicate with software visualiser personnel to check if data ● Members meet up to finalize design of the guns, shields and

	gloves
25/9 - 1/10 Week 7	● Mount the relevant sensors on the guns,shields and gloves ● Continue integration of subcomponents ● Prepare for 1-player game ● Collection of data for machine learning model ● Integrate Machine Learning model with the necessary algorithms such as start of motion detection and noise filtering
2/10 - 8/10 Week 8	● Finalize the relevant implementations for the 1-player game
9/10 - 15/10 Week 9	● Evaluation of 1-player game ○ Members to perform the evaluation and work on the relevant feedback from it ○ AI algorithms should be complete, collet more data for next iterations if needed ● Begin work on 2-player game
16/10 - 22/10 Week 10	● Finalize the relevant implementations for the 2-player game
23/10 - 29/10 Week 11	● Evaluate the 2-player game by team members
30/10 - 5/11 Week 12	● Two player game evaluation ○ Team to work on the feedback received after evaluation ● Prepare for the advanced game system that will be tested next week
6/11 - 10/11 Week 13	● Final Evaluation ○ Ensure that all the subsystems are working ● Prepare for final report ○ Detail all the findings and highlight main differences from initial report

References

[1] A. Murad and J.-Y. Pyun, "Deep recurrent neural networks for human activity recognition," *MDPI,* 06-Nov-2017. [Online]. Available: https://www.mdpi.com/1424-8220/17/11/2556. [Accessed: 21-Aug-2022].

[2] Avnet, "Ultra96-PYNQ/ultra96_pmbus.ipynb at master · Avnet/ultra96-PYNQ," *GitHub*. [Online]. Available: https://github.com/Avnet/Ultra96-PYNQ/blob/master/Ultra96/notebooks/common/ultra96_pmbus.ipynb. [Accessed: 21-Aug-2022].

[3] B. Earl, "Multi-tasking the Arduino - part 1," *Adafruit Learning System*. [Online]. Available: https://learn.adafruit.com/multi-tasking-the-arduino-part-1/overview. [Accessed: 21-Aug-2022].

[4] B. Moyer, "Fragmentation: When a message is too big," Inside the IoT, 16-Nov-2019. [Online]. Available: https://www.insidetheiot.com/fragmentation-when-a-message-is-too-big/. [Accessed: 21-Aug-2022].

[5] "Bluno_sku_dfr0267," DFRobot. [Online]. Available: https://wiki.dfrobot.com/Bluno_SKU_DFR0267. [Accessed: 21-Aug-2022].

[6] "Fundamental concepts | arcore | google developers," *Google*. [Online]. Available: https://developers.google.com/ar/develop/fundamentals. [Accessed: 21-Aug-2022].

[7] "How to use python threading lock to prevent race conditions," *Python Tutorial - Master Python Programming For Beginners from Scratch*, 14-Jul-2022. [Online]. Available: https://www.pythontutorial.net/python-concurrency/python-threading-lock/#:~:text=A%20race%20condition%20occurs%20when,value%20of%20the%20shared%20variable. [Accessed: 21-Aug-2022].

[8] J. Brownlee, "Deep learning models for human activity recognition," *Machine Learning Mastery,* 05-Aug-2019. [Online]. Available: https://machinelearningmastery.com/deep-learning-models-for-human-activity-recognition/. [Accessed: 21-Aug-2022].

[9] M. Stenina, "Markerless ar: Use cases and how-to - wikitude blog," *Wikitude,* 21-Jul-2022. [Online]. Available: https://www.wikitude.com/blog-wikitude-markerless-augmented-reality/. [Accessed: 21-Aug-2022].

[10] M. Woolley, "Bluetooth low energy -it starts with advertising," *Bluetooth® Technology Website,* 29-Mar-2022. [Online]. Available: https://www.bluetooth.com/blog/bluetooth-low-energy-it-starts-with-advertising/. [Accessed: 21-Aug-2022].

[11] Real Python, "An Intro to threading in Python," *Real Python,* 22-May-2022. [Online]. Available: https://realpython.com/intro-to-python-threading/#starting-a-thread. [Accessed: 21-Aug-2022].

[12] "Sensors overview : android developers," *Android Developers*. [Online]. Available: https://developer.android.com/guide/topics/sensors/sensors_overview. [Accessed: 21-Aug-2022].

[13] V. Nunavath, S. Johansen, T. S. Johannessen, L. Jiao, B. H. Hansen, S. Berntsen, and M. Goodwin, "Deep learning for classifying physical activities from accelerometer data," *Sensors (Basel, Switzerland)*, 18-Aug-2021. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8402311/. [Accessed: 21-Aug-2022].

[14] "Bluepy - a bluetooth le interface for python¶," *bluepy - a Bluetooth LE interface for Python - bluepy 0.9.11 documentation*. [Online]. Available: https://ianharvey.github.io/bluepy-doc/index.html. [Accessed: 21-Aug-2022].

[15] "Python - multithreaded programming," *Tutorials Point*. [Online]. Available: https://www.tutorialspoint.com/python/python_multithreading.htm. [Accessed: 21-Aug-2022].

[16] "CRC," *PyPI*. [Online]. Available: https://pypi.org/project/crc/. [Accessed: 21-Aug-2022].

[17] I. Harvey, "The scanner class¶," *The Scanner class - bluepy 0.9.11 documentation*. [Online]. Available: https://ianharvey.github.io/bluepy-doc/scanner.html. [Accessed: 21-Aug-2022].

[18] "WISDM smartphone and smartwatch activity and biometrics dataset data set", *UCI Machine Learning Repository*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/WISDM+Smartphone+and+Smartwatch+Activity+and+Biometrics+Dataset+. [Accessed: 21-Aug-2022].

[19] "Core Bluetooth Programming Guide," *Core Bluetooth Overview*, 18-Sep-2013. [Online]. Available: https://developer.apple.com/library/archive/documentation/NetworkingInternetWeb/Conceptual/CoreBluetooth_concepts/CoreBluetoothOverview/CoreBluetoothOverview.html. [Accessed: 21-Aug-2022].

[20] MQTT - The Standard for IoT Messaging
"The standard for IOT messaging," MQTT. [Online]. Available: https://mqtt.org/. [Accessed: 21-Aug-2022].

[21] "How better FPGA Resource Utilization can save you time and money," *InAccel*. [Online]. Available: https://inaccel.com/how-better-fpga-resource-utilization-can-save-you-time-and-money/. [Accessed: 13-Nov-2022].

[22] "pragma HLS pipeline," *Documentation Portal*, 19-Oct-2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-pipeline. [Accessed: 13-Nov-2022].

[23] "pragma HLS unroll," *Documentation Portal*, 19-Oct-2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-unroll. [Accessed: 13-Nov-2022].

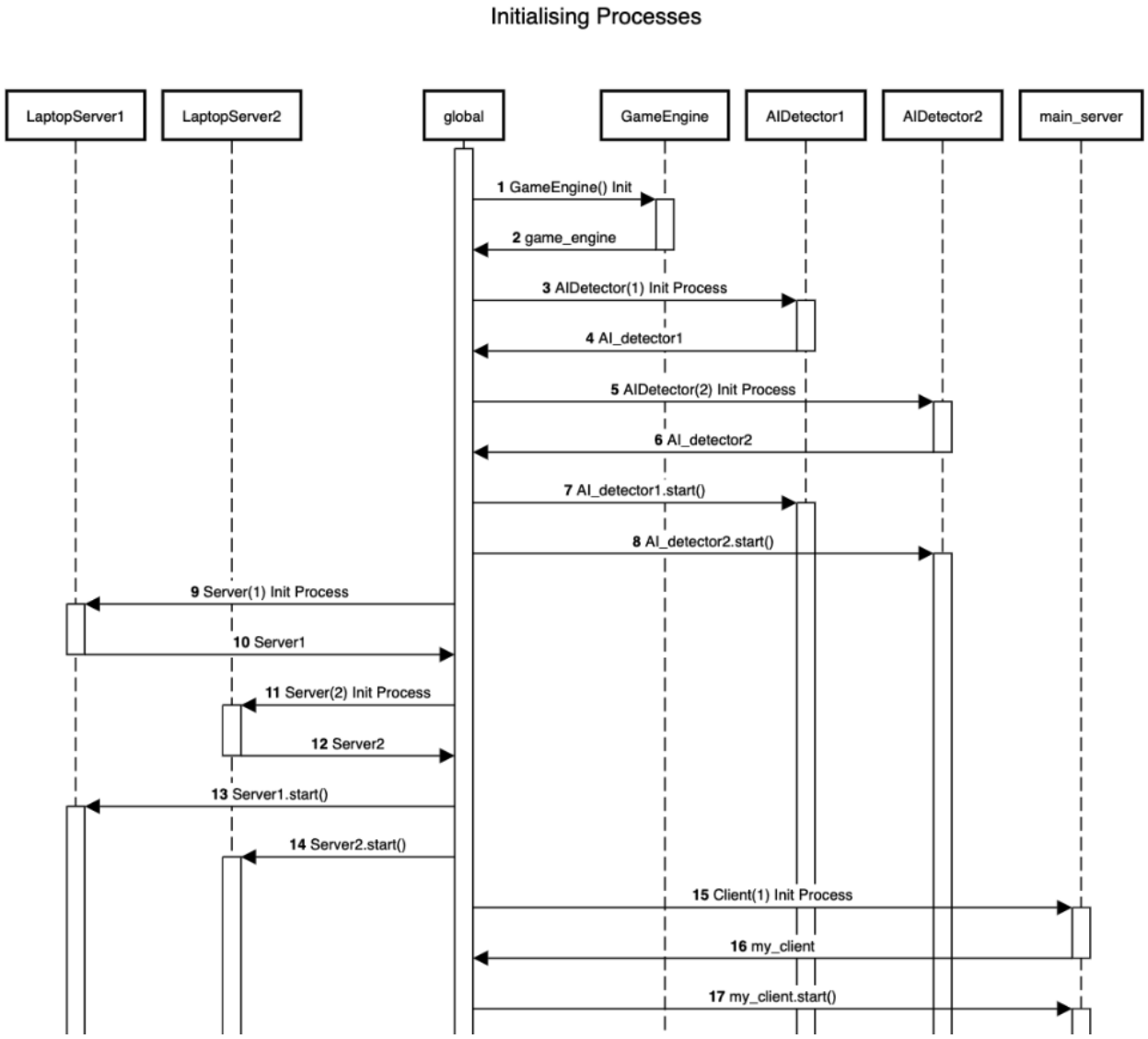
[24] "pragma HLS array partition," *Documentation Portal*, 19-Oct-2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-array_partition. [Accessed: 13-Nov-2022].

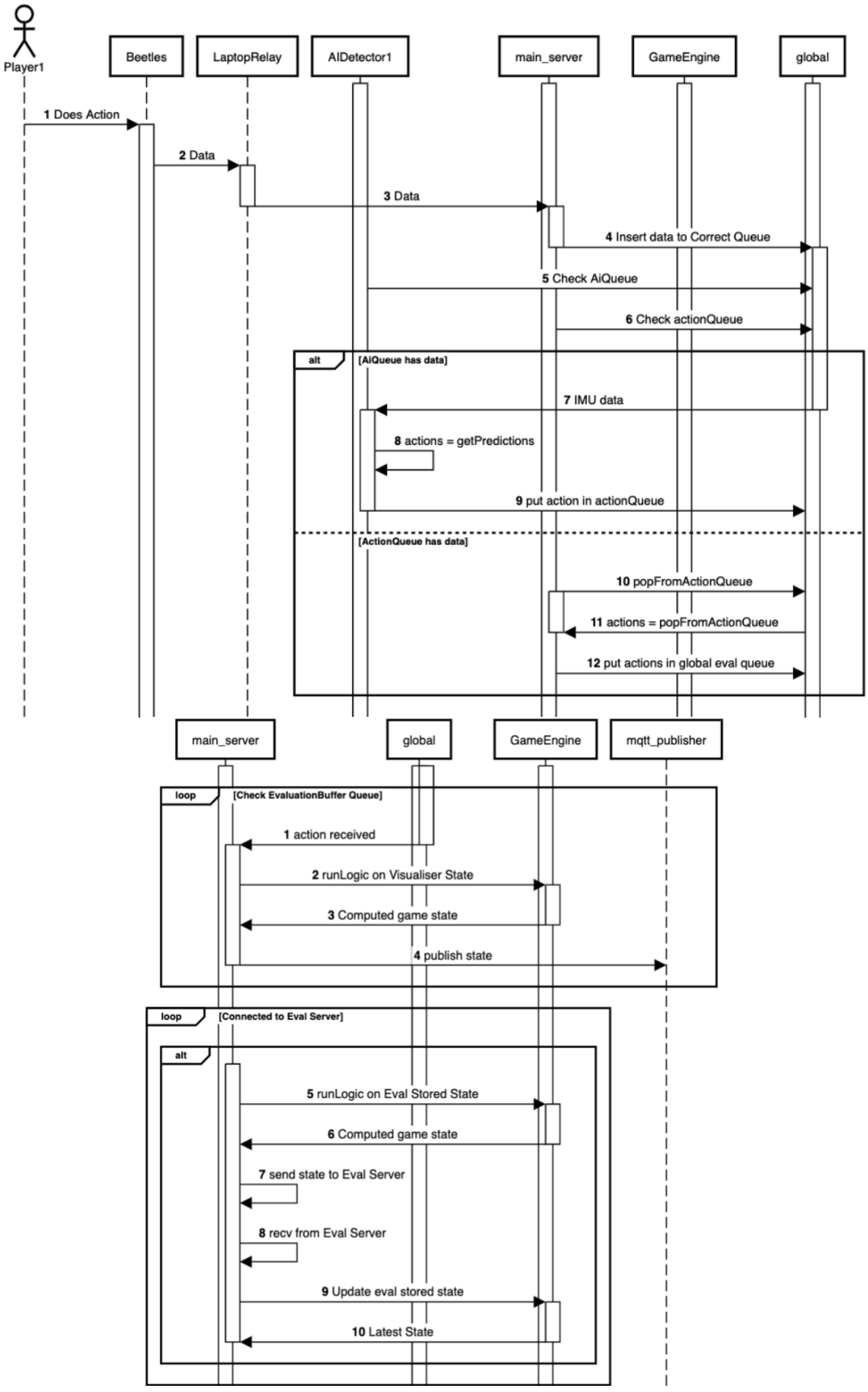
[25] S. Sen, "Multilayer Perceptron model VS CNN," *Medium*, 04-Aug-2020. [Online]. Available: https://medium.com/the-owl/multilayer-perceptron-model-vs-cnn-5be5cf87897a. [Accessed: 13-Nov-2022].

[26] “Manage crash detection on iPhone 14 models,” *Apple Support*. [Online]. Available: https://support.apple.com/en-gb/guide/iphone/iph948a628e9/ios#:~:text=Turn%20Crash%20Detection%20on%20or%20off&text=You%20can%20turn%20off%20alerts,they%20will%20still%20be%20notified. [Accessed: 13-Nov-2022].
[27] “Stop and wait arq,” *GeeksforGeeks*, 14-Jun-2022. [Online]. Available: https://www.geeksforgeeks.org/stop-and-wait-arq/. [Accessed: 13-Nov-2022].

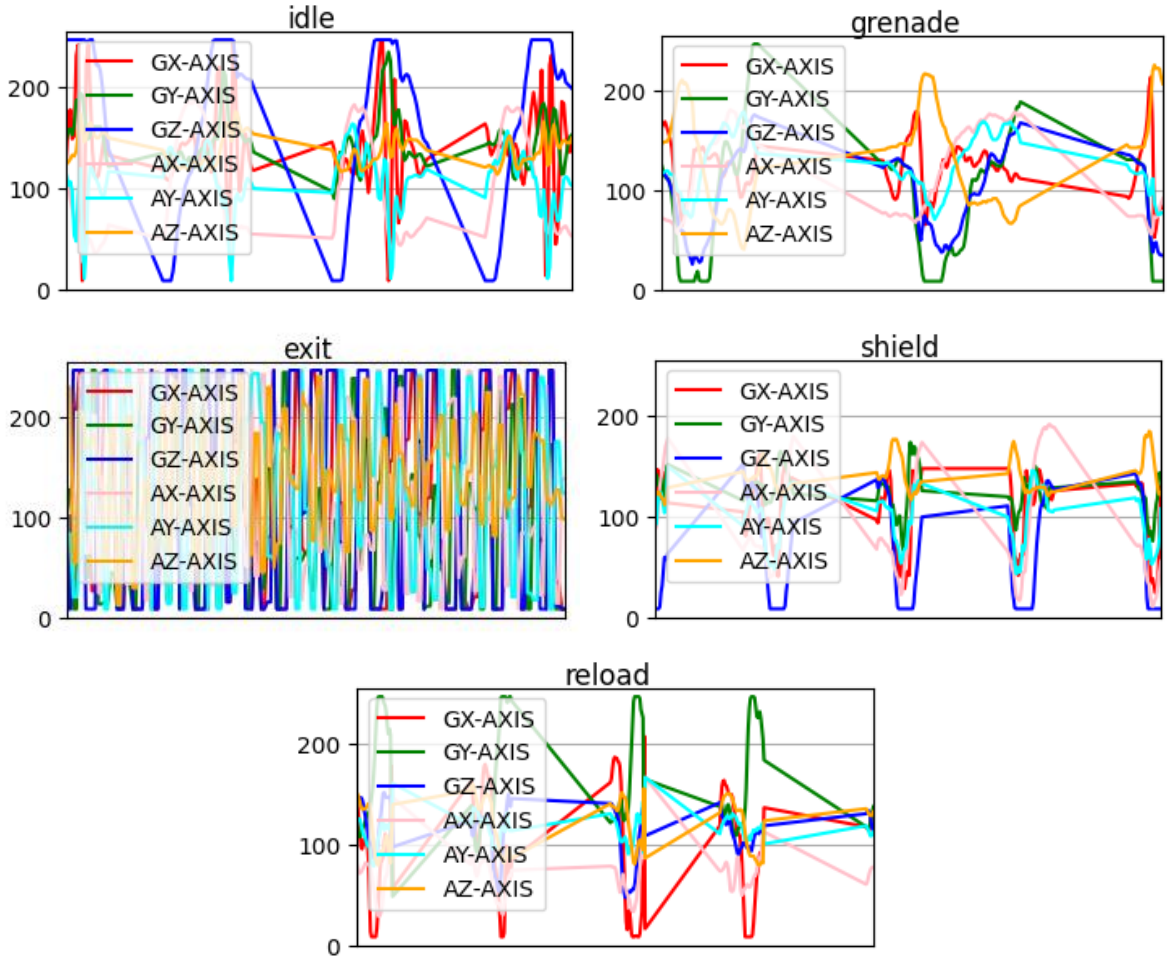
Appendix

Appendix 2.3 Sequence Diagrams




Player1
Beetles
LaptopRelay
AIDetector1
main_server
GameEngine
global
1 Does Action
2 Data
3 Data
4 Insert data to Correct Queue
5 Check AiQueue
6 Check actionQueue
alt
[AiQueue has data]
7 IMU data
8 actions = getPredictions
9 put action in actionQueue
[ActionQueue has data]
10 popFromActionQueue
11 actions = popFromActionQueue
12 put actions in global eval queue
main_server
global
GameEngine
mqtt_publisher
loop
[Check EvaluationBuffer Queue]
1 action received
2 runLogic on Visualiser State
3 Computed game state
4 publish state
loop
[Connected to Eval Server]
alt
5 runLogic on Eval Stored State
6 Computed game state
7 send state to Eval Server
8 recv from Eval Server
9 Update eval stored state
10 Latest State

Appendix 4.1.3 Data Visualisation



Appendix 4.2.1 Model Summary

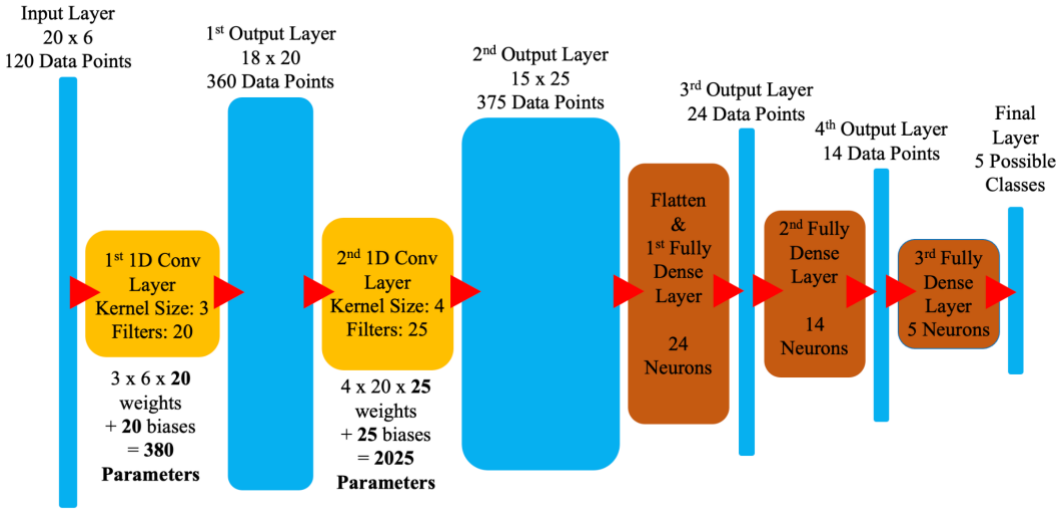
Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 18, 20)	380
conv1d_1 (Conv1D)	(None, 15, 25)	2025
flatten (Flatten)	(None, 375)	0
dense (Dense)	(None, 24)	9024
dense_1 (Dense)	(None, 14)	350
dropout_3 (Dropout)	(None, 14)	0
dense_2 (Dense)	(None, 5)	75

Total params: 11,854
Trainable params: 11,854
Non-trainable params: 0

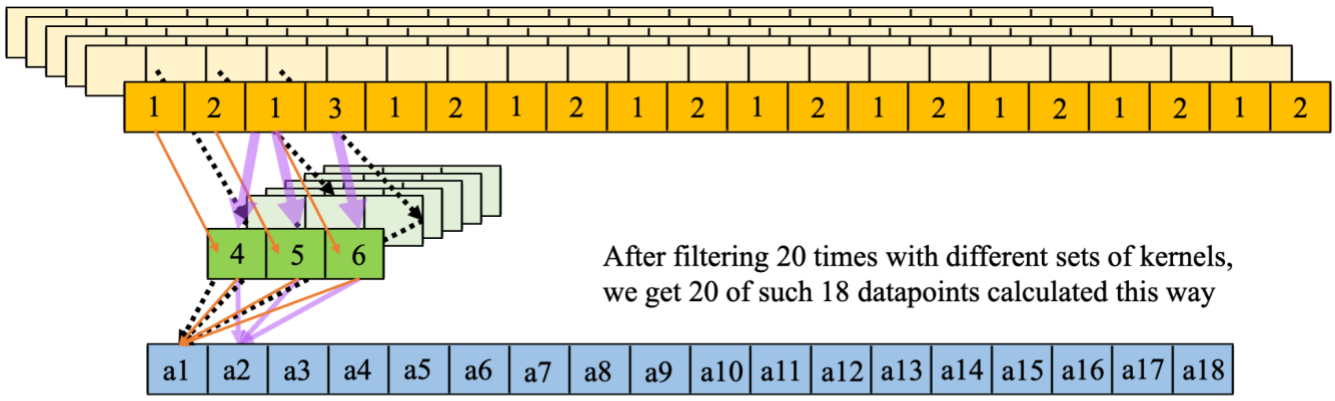
Appendix 4.2.4 Trainer Code base

https://github.com/uosjapuelks/capstoneml/blob/ultra_96v/notebook/conv_trainer.ipynb

Appendix 4.4.1.1 CNN Architecture



Appendix 4.4.1.2 Convolutional Network Calculation Illustration



a1 = ReLU(conv_bias[0] + 1*4 + 2*5 + 1*6 + (2nd channel*kernel_channel2[0]+…)+…)

a2 = ReLU(conv_bias[0] + 2*4 + 1*5 + 3*6 + (2nd channel*kernel_channel2[0]+…)+…)

Appendix 4.4.1.3 Mathematical Formula for Dense Layers

Dot Product

$$\begin{bmatrix} w1 & w4 \\ w2 & w5 \\ w3 & w6 \end{bmatrix} \cdot [d1 \quad d2 \quad d3] = [a' \quad b']$$

Add biases + ReLU function

$$ReLU([a' \quad b'] + [b1 \quad b2]) = [\boldsymbol{a} \quad \boldsymbol{b}]$$

Appendix 4.4.3 Hardware AI

Timing

Clock		addPartitioning	addUnroll	fullyOptimized	noOptimization	usePipelines
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns	10.00 ns	10.00 ns
	Estimated	10.922 ns	10.922 ns	10.922 ns	8.944 ns	8.944 ns

Latency

		addPartitioning	addUnroll	fullyOptimized	noOptimization	usePipelines
Latency (cycles)	min	333881	333401	333881	643258	533722
	max	333881	333401	333881	643258	533722
Latency (absolute)	min	3.647 ms	3.641 ms	3.647 ms	6.433 ms	5.337 ms
	max	3.647 ms	3.641 ms	3.647 ms	6.433 ms	5.337 ms
Interval (cycles)	min	333881	333401	333881	643258	533722
	max	333881	333401	333881	643258	533722

	addPartitioning	addUnroll	fullyOptimized	noOptimization	usePipelines
BRAM_18K	34	55	76	55	55
DSP48E	29	28	28	29	29
FF	102339	10337	62640	4268	4259
LUT	28115	16519	59890	4987	5236
URAM	0	0	0	0	0

Appendix 4.4.3.1 Pipelining

```
// Load Kernel weights
for (int i=0; i<in_filts;i++) {
    for (int j=0; j<kernelSize; j++){
    #pragma HLS pipeline
        conv_kernels[j][i][k]=kernel[j][i][k];
```

Appendix 4.4.3.2 Unrolling

Convolutional Multiplication Loop

```
for (int i=0; i<in_filts;i++) {
    for (int j=0; j<kernelSize; j++){
        #pragma HLS unroll
        int idx = j*in_filts+i;
        conv_p += in_int[idx]*conv_kernels[j][i][k];
```

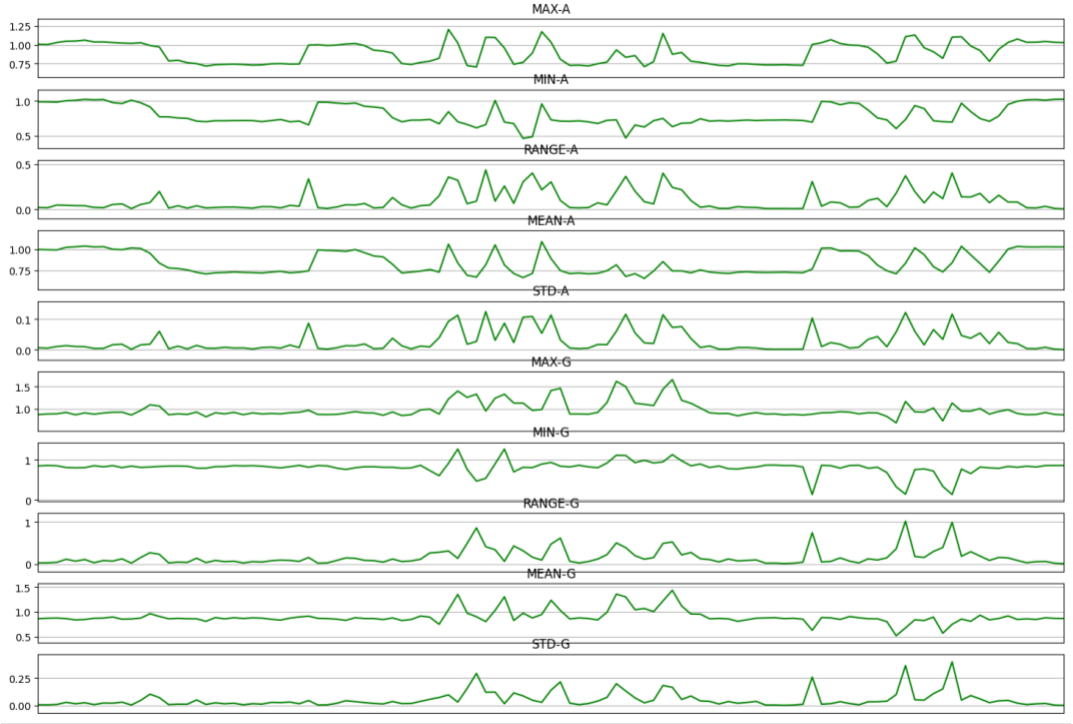
Dot Product Multiplication Loop

```
for(int i=0; i<n; i++) {
#pragma HLS unroll
    dotp += a_int[i] * b_int[i][j];
```

Appendix 4.4.3.3 Array Partitions

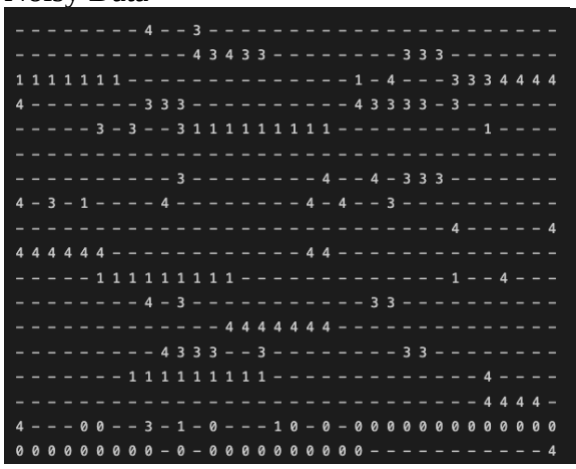
```
#pragma HLS array_partition variable=input complete
#pragma HLS array_partition variable=kernel dim=3 complete
```

Appendix 4.5.1.1 Features Extracted

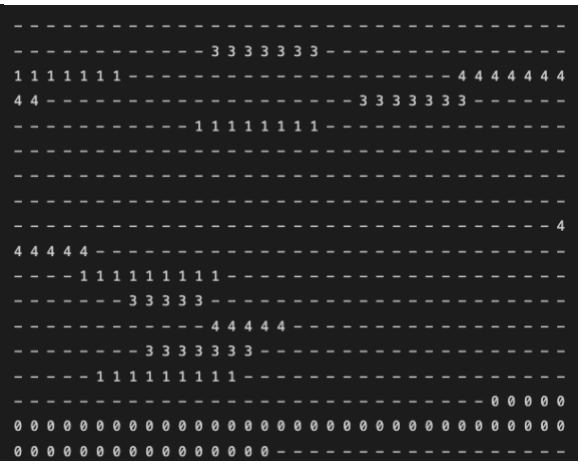


Appendix 4.5.2.1 Noisy Predictions vs Actual Data

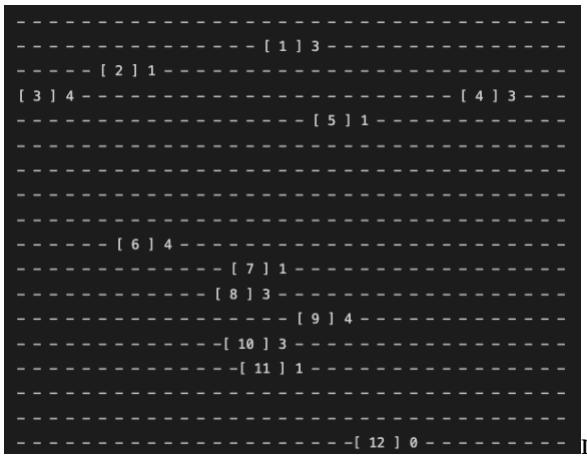
Noisy Data



Actual Data



Appendix 4.5.2.2 Algorithm and Model Combined Output



number behind the [] brackets are the output

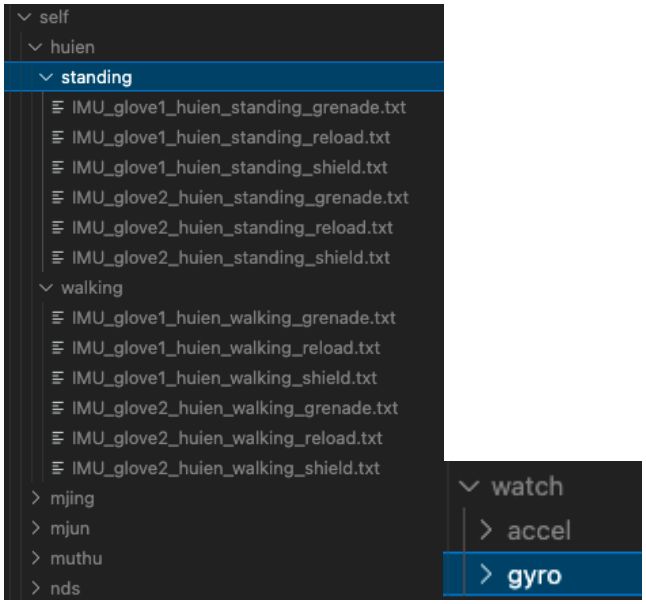
Appendix 4.5.3 Codebase for Printing as Above

DISCLAIMER: Code in codebase may have different outputs as these were saved outputs
https://github.com/uosjapuelks/capstoneml/blob/ultra_96v/notebook/eval_conv.ipynb

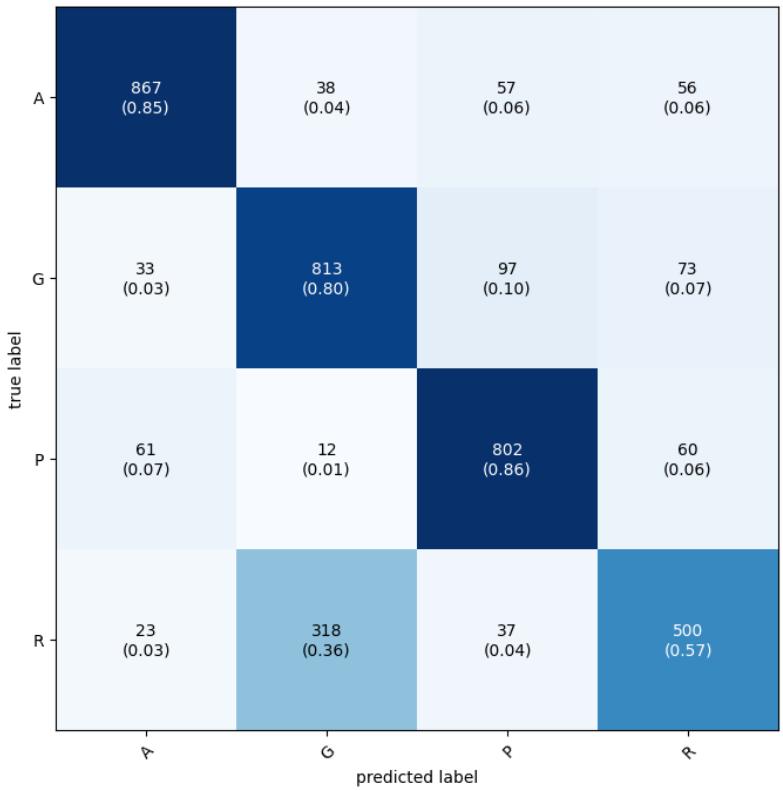
Appendix 4.5.4 Codebase for Algorithm

Main AI Algorithm:
https://github.com/uosjapuelks/capstoneml/blob/ultra_96v/scripts/start_detector.py
AI FPGA usage:
https://github.com/uosjapuelks/capstoneml/blob/ultra_96v/scripts/ai.py

Appendix 4.6.2.1 Self Collected Data vs Online Data

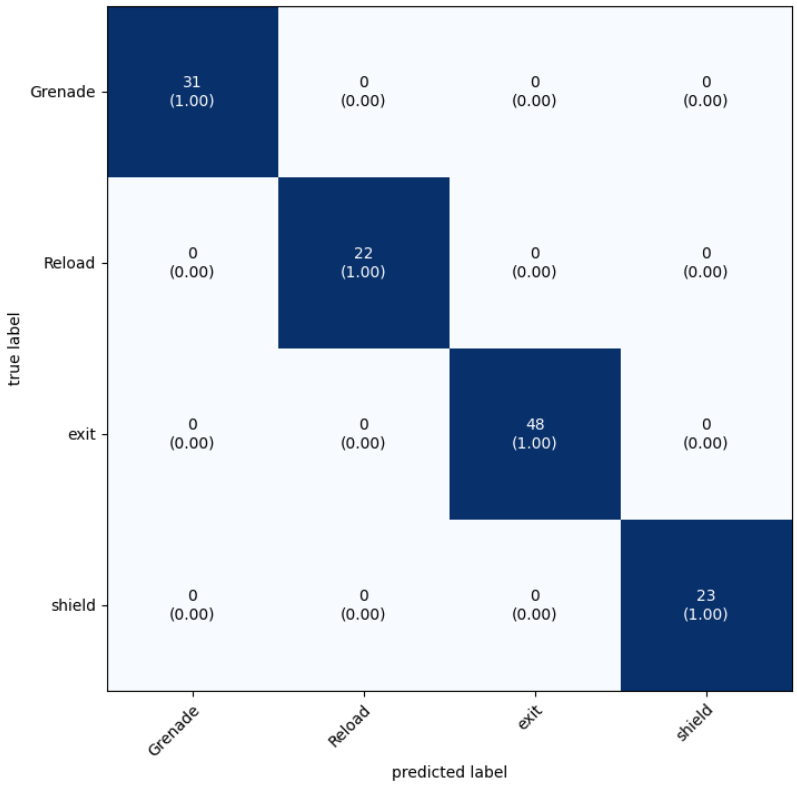


Appendix 4.6.2.1.a Confusion Matrix on Online Dataset with MLP



Appendix 4.6.2.2 Training of First batch of self collected data

Confusion matrix of model trained on **first batch of self collected data**



Appendix 5.2.1 AT Commands used

Command	Purpose
AT+MAC=?	To obtain the MAC addresses of the beetles
AT+BLUNODEBUG	This was set to ON when debugging to observe the beetles. However, it was turned off after debugging.
AT+SETTING	Set to DEFPERIPHERAL
AT+UART	Set to 115200 to ensure that the Baud rate is 115200
AT+EXIT	To exit the AT mode.

Appendix 5.2.2 Initialisation function

```
    def initialise(self):

        addr = self.ADDRESS
        print("start")
        connect = 0

        initialise = 0
        while initialise == 0:
            while (connect == 0):
                print("\r{} :Trying to Connect".format(self.name), end="")
                try:
                    currentBeetle = btle.Peripheral(addr)
                    self.peripheral = currentBeetle
                    print("Device Connected")
                    connect = 1
                except Exception as e:
                    pass

            # acquiring the services and characteristics of the beetle
            initialise = 0
            err = 1
            while (connect == 1 and initialise == 0):
                try:
                    self.service = currentBeetle.getServiceByUUID('dfb0')
                    self.characteristic = self.service.getCharacteristics()[0]
                    currentBeetle.withDelegate(MyDelegate(self))
                    initialise = 1
                    print("initialised")

                except btle.BTLEDisconnectError as c:
                    print("Disconnected before intialisations")
                    connect = 0
                except Exception as e:
                    print(e)
                    initialise = 0
```

Appendix 5.3.1 Packet structure of IMU data

```
typedef struct Packet {
  uint8_t header;
  uint8_t device;
  uint8_t type;
  uint16_t rawX;
  uint16_t rawY;
  uint16_t rawZ;
  uint16_t accX;
  uint16_t accY;
  uint16_t accZ;
  uint32_t pad;
  uint8_t checksum;

} packet;
```

Appendix 5.3.2 Packet structure of data from Gun/Vest

```
typedef struct Packet {
  uint8_t header;
  uint8_t device;
  uint8_t type;
  uint16_t seq;
  uint16_t rawX;
  uint16_t rawY;
  uint16_t rawZ;
  uint16_t accX;
  uint16_t accY;
  uint16_t accZ;
  uint8_t pad1;
  uint8_t pad2;
  uint8_t checksum;

} packet;
```

Appendix 5.4.1

```
--------------------------------------------------------------------------------
b'\x00\x80\x00~\x00\x7f\x00\x00\x00\x00\x00g\xaa\x04\x04t\x00\x8f\x00\xb7'
Extended Buffer:[170, 4, 4, 116, 0, 143, 0, 183, 0, 128, 0, 126, 0, 127, 0, 0, 0, 0, 0, 103, 170, 4, 4, 116, 0, 143, 0, 183]
Fragmented Data[170, 4, 4, 116, 0, 143, 0, 183, 0, 128, 0, 126, 0, 127, 0, 0, 0, 0, 0, 103]
Retrieved Data:(116, 143, 183, 128, 126, 127)
Clearing buffers
Remaining buffer:[170, 4, 4, 116, 0, 143, 0, 183]
--------------------------------------------------------------------------------
b'\x00\xb7\x00\x80\x00~\x00\x7f\x00\x00\x00\x00\x00`\xaa\x04\x04s\x00\x8f'
Extended Buffer:[170, 4, 4, 116, 0, 143, 0, 183, 0, 183, 0, 128, 0, 126, 0, 127, 0, 0, 0, 0, 0, 96, 170, 4, 4, 115, 0, 143]
Fragmented Data[170, 4, 4, 116, 0, 143, 0, 183, 0, 183, 0, 128, 0, 126, 0, 127, 0, 0, 0, 0, 0, 96]
Clearing buffers
Remaining buffer:[170, 4, 4, 115, 0, 143]
--------------------------------------------------------------------------------
b'\x00\xb7\x00\x80\x00~\x00\x7f\x00\x00\x00\x00\x00`\xaa\x04\x04t\x00\x8f'
Extended Buffer:[170, 4, 4, 115, 0, 143, 0, 183, 0, 128, 0, 126, 0, 127, 0, 0, 0, 0, 0, 96, 170, 4, 4, 116, 0, 143]
Fragmented Data[170, 4, 4, 115, 0, 143, 0, 183, 0, 128, 0, 126, 0, 127, 0, 0, 0, 0, 0, 96]
Retrieved Data:(115, 143, 183, 128, 126, 127)
Clearing buffers
Remaining buffer:[170, 4, 4, 116, 0, 143]
--------------------------------------------------------------------------------
```

Appendix 6.5 Github Repository for External Communications Code

One Player Implementation Code Base:
https://github.com/huien77/CG4002_ext_comm/tree/correct_oneplayer
Two Player Implementation Code Base:
https://github.com/huien77/CG4002_ext_comm/blob/main/README.md

Appendix 7.1.2 Overlay

Figure 7.1.2.1 Game overlay

Figure 7.1.2.2 Gameover screen

Appendix 7.2.1 Github Repository for Software Visualiser Code

Code for Player 1 Point of view: homingjun/CG4002-Laser-Tag at Integration-P1-POV (github.com)
Code for Player 2 Point of view: homingjun/CG4002-Laser-Tag at Integration-P2-POV (github.com)

Appendix 7.2.2 Data Inputs and Networking

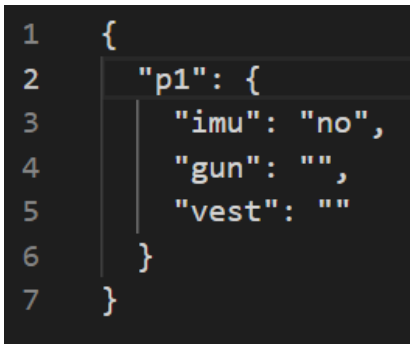


Figure 7.2.2.1 Player's hardware disconnection status

Figure 7.2.2.2 IMU disconnection warning

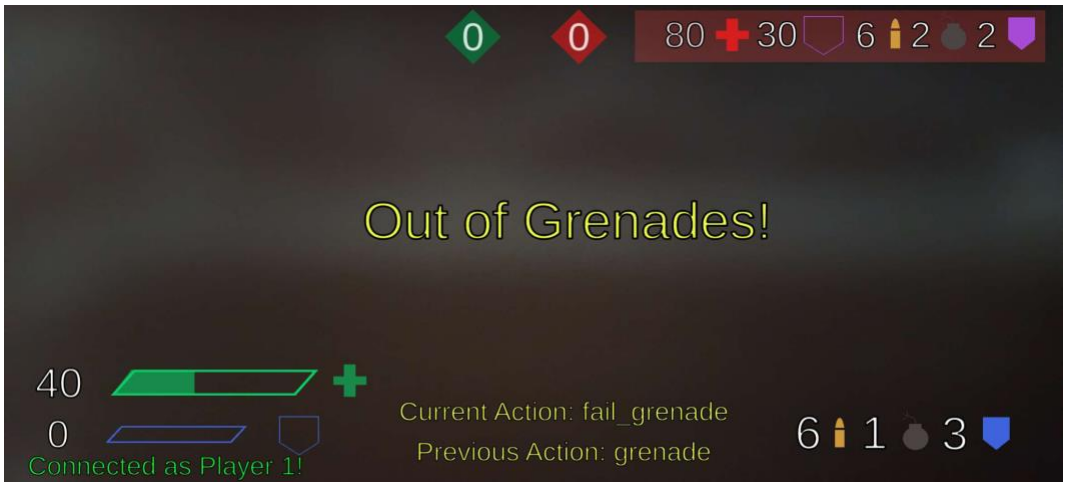


Figure 7.2.2.3 Failed grenade action message

Appendix 7.3 Image Target and AR Effects

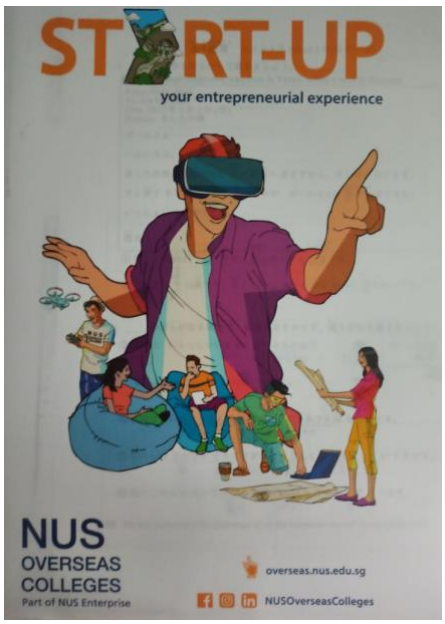


Figure 7.3.1.1 Image Target


Figure 7.3.1.2 First Person Shield Activated

Figure 7.3.1.3 Opponent Shield Activated

	
Figure 7.3.1.4 Grenade Throw	Figure 7.3.1.5 Grenade Explosion
	
Figure 7.3.1.6 Blood effect when dealing damage	Figure 7.3.1.7 Blood effect from taking damage