Technical Report
Graduation Assignment

LED Input Detection for Gadget Board

Aemics BV

Saxion University of Applied Sciences

Created by Luciano Regis Orlandini (460952)

This page is intentionally left blank.

Information

Institutions:

Aemics B.V.

Address:

Zutphenstraat, 81

7575 EJ – Oldenzaal

The Netherlands

Saxion University of Applied Sciences

Address:

M.H. Troomplan, 28

7513AB – Enschede

The Netherlands

Supervisors:

Engr. Tom Oude Nijhuis – R&D Manager

t.oudenijhuis@aemics.nl

Engr. Erik Karstens – Lecturer LED

w.f.karstens@saxion.nl

Student:

Mr. Luciano Regis Orlandini

4th Year Applied Computer Science Student

l.orlandini@aemics.nl

460952@student.saxion.nl

Address:

Eschenstr. 10

48599 – Gronau (Westf.)

Germany

Document:

Technical Report – LED Input Detection for Gadget Board

Version 1.0

This page is intentionally left blank.

Foreword

This technical report details the entire work performed by the student Luciano Regis Orlandini during the graduation assignment 'LED Input Detection for Gadget Board', provided by the company Aemics BV. The assignment took place between February and July 2022, under supervision of Engr. Tom Oude Nijhuis, R&D Manager at Aemics and Engr. Erik Karstens, lecturer LED at Saxion University of Applied Sciences.

The report is intended for the engineers at Aemics to work with these results in the future and for the examiners at Saxion University of Applied Sciences during the graduation assessment.

I, Luciano Orlandini, would like to thank Aemics BV and Saxion University of Applied Sciences for the opportunity and all experiences gathered during this assignment.

Luciano Orlandini

Oldenzaal, June 2022

Summary

An electronic Gadget Board with a 12 x 12 matrix of LEDs was previously designed by the company Aemics to be used as a display for applications running on a paired mobile phone. The purpose of this assignment was to develop a proof-of-concept using the light detection capability of the LEDs as a reliable mean to detect when and where a user would touch the board's matrix, analogue to a touch-screen. Once the concept was proven the applications running on the mobile phone could be updated to accept this new input source and additional games could be created and played via this function.

Work began with a detailed analysis of the existing hardware and the supplied software. This also included tests to unravel all functionality already implemented on the Gadget Board. The analysis showed that it already performs an ADC of the light detected by every LED and transmits this information via serial. Hence a previous report from an Aemics engineer was studied on light detection with diodes and additional research was undertaken on such technology, as well as on possibly suitable algorithms to interpret these values. After the analysis and research phases, it was found that machine learning classifiers could be suitable for a program to achieve the objective of this assignment. Development of the proof-of-concept has then started with Scrum being used as a project management methodology and the project development was planned in three-week-long sprints.

A software has been designed comprising all steps required to run algorithms such as data gathering and pre-processing, model training & validating and testing directly on the Gadget Board. The model created detected touch with different precision parameters, first in an area comprising 16 LEDs in a 4 x 4 quadrant, then 4 LEDs in a 2 x 2 square and finally a single LED. It reached accuracies above 95% when detecting the first 2 precision settings and above 80% on the last. After a successful implementation the software was ported to work on a mobile application and additional games were developed.

This proof-of-concept has shown that artificial neural network models are capable of making associations between the ADC values transmitted by the Gadget Board, to accurately and quickly predict when and where the LED matrix is being touched, increasing its functionality and opening the door for further development. Its performance was influenced by environmental light which needed to be the same as when the data was gathered to train the model. This neural network was the third attempt, as two simpler implementations were also tried and tested but could not provide the same levels of accuracy.

Future recommendations include increasing accuracy level for the single LED precision mode by spending additional time on data gathering and in different lighting conditions to provide versatility to the Gadget Board. Further research on microcontrollers and embedded machine learning, as well as hardware additions such as a buzzer and OLED screen, could potentially enable the entire application to run exclusively on the Gadget Board, removing the need of a paired mobile phone.

Content

List of Figures

List of Tables

References & Abbreviations

For clarity and simplification, the people, institutions and hardware / software parts within this project will be referenced throughout the document as per Table 1 and the abbreviations used as per Table 2. The text styles shown in Table 3 will be used to highlight specific references mid-text:

Table 1 - References

Student	Luciano Regis Orlandini
Company Supervisor	Tom Oude Nijhuis
Learning Supervisor	Erik Karstens
Company	Aemics BV

Table 2 - Abbreviations

ADC	Analogue-to-Digital Conversion
GB	Gadget Board
I/O	Input / Output
ISR	Interrupt Service Routine
LED	Light Emitting Diode
MCU	Microcontroller Unit
ML	Machine Learning
OTG	On-The-Go
PCB	Printed Circuit Board
POC	Proof-Of-Concept
USB	Universal Serial Bus

Table 3 - Text Styles

`"file_name.typ"`	File name
`function_name()`	Programming function
`myValue=3`	Variable name
`"Text String"`	Content of string-type variable
`'0'`	Content of char / int / float-type variable
Step 1	Direct reference to a POC Development Framework

1 Introduction

1.1 Project Background

1.1.1 Company

Aemics BV is located in Oldenzaal and has around 25 employees. The main business model consists of clients approaching the company with an idea or issue which then are translated into requirements. To assess feasibility of the project, a proof-of-concept is made followed by a prototype. No matter the state of the product, Aemics will help realise the idea.

Besides the development of prototypes, the company manufactures on site within the production department. The advantages of having these facilities are to help the realisation of prototypes and the reduction of production lead times. Products can also be packed and distributed directly from Aemics' main building. Additionally, the company develops their own products, such as a PYggi board [1].

1.1.2 Assignment

In a previous graduation period, the company has developed an electronic Gadget Board (GB) as per Figure 1:

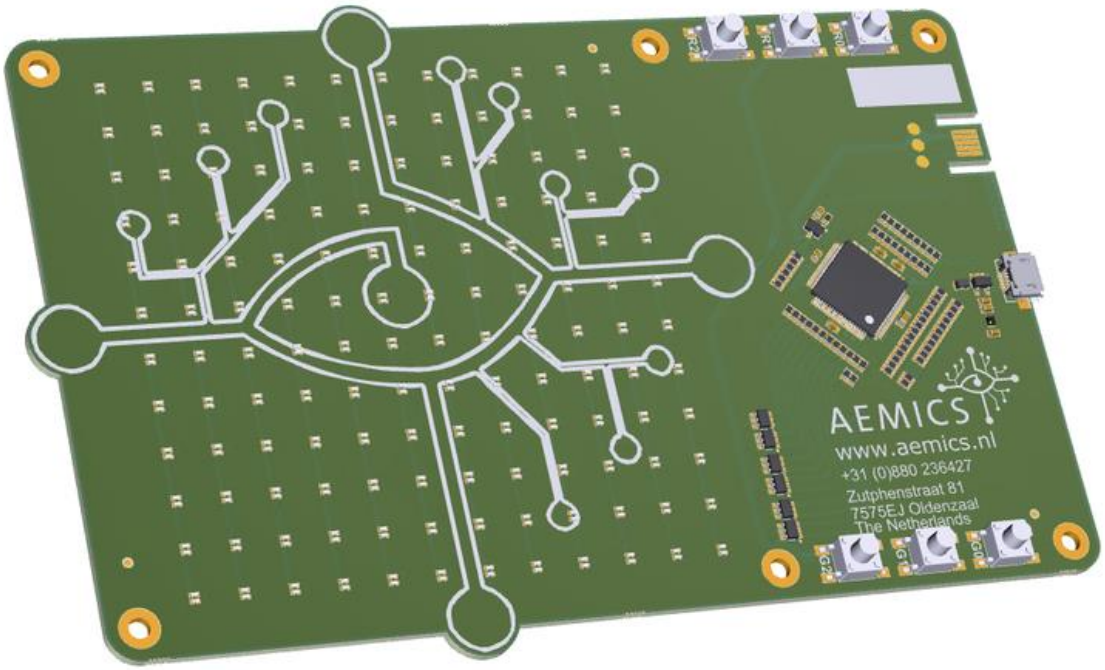

Figure 1 - Gadget Board

The gadget runs a simple Tic-Tac-Toe implementation, but is suitable for other games such as Checkers, Go, or a sketch board. The game's logic runs in a mobile app hosted on a smartphone connected to the board, while the gadget's logic runs on its Microcontroller Unit's (MCU) embedded software as a 'stupid' LED and button controller.

The GB's only output is via the 12 x 12 LED matrix shown where the Aemics Logo is printed, forming an image analogous to a square 144-pixel screen. The only user input source is via push-buttons. A previous internal study at the company [2] has shown that it is possible to use the LEDs as an extra input source, as they sense light changes in the environment. The hardware is prepared to support this.

1.1.3 Objective

The main objective of the assignment is to develop an algorithm that makes use of the sensing capabilities of the Gadget Board's LEDs to configure them as an input source, similar to a touch-screen. The algorithm needs to run fast enough so the input detection does not have a considerable input-lag. The above aims at improving user experience and increasing board functionality, as a specific part of the LED matrix becomes selectable via touch.

Having LEDs instead of pressure sensors is a low-cost alternative and removes the need for a display. It has to be taken into consideration that not only a single LED detects light changes when it is being touched, but the surrounding ones may or may not be affected by the user's other fingers or hand.

Once this functionality is implemented, the algorithm needs to be ported to the embedded software or mobile app. Additional games can also be developed to demonstrate different usage scenarios.

1.1.4 Boundaries

The student was given access to a Gadget Board with its embedded software and used his own laptop and a smartphone for research and development. A desktop computer was also provided when working at Aemics. Only the LED matrix present on the board was used during development as its hardware constraints had to be considered.

When additional hardware was needed, the student discussed possibilities of implementing any changes to the GB with the company supervisor.

1.1.5 Benefits

This system will be used both as a proof-of-concept and to demonstrate some of the capabilities of Aemics, as it is a product designed and produced in-house. It can be used as advertisement for the company during open days. Robustness and stability were paramount to any programs developed, as recipients are not able to modify the source code nor reprogram the board. A successful implementation may speed up future development of new products, if they share any features with this system.

1.2 Project Management

1.2.1 Methodology

The company works in 3-week sprints, with a company-wide stand-up meeting on Mondays. Additionally, a meeting between all students and their supervisors took place on Thursdays, where more details on all assignments were discussed. The student adapted the above and used a Scrum framework [3] to plan each sprint by creating user stories and estimating the time required for them. The sprint duration aforementioned helped plan developmental milestones.

1.2.2 Control

A project backlog was created and a sprint review and retrospective with the company supervisor happened at the beginning and end of each one respectively. Decisions on adjustments were made to ensure the project remained on track.

The built-in project management system at Aemics was used to keep track of sprints and user stories. The system automatically filled in burndown charts and provided visibility of the project's progress to all stakeholders.

1.3 Project Approach

By taking the objectives and equipment supplied into account, an analysis & research period was required prior to any development. This enabled the student to gain a better understanding of the task at hand. The work realised during the semester was divided in phases:

1.3.1 Phase I: Original System Analysis

A complete study on the original Gadget Board and software provided was undertaken, to serve as a starting point for further development. Datasheets as well as colleagues at Aemics were consulted when analysing hardware components, as not all of their possible capabilities were visible by only inspecting the board with its initial software. The code provided was read and tested and online resources were consulted to clarify functionality of some libraries and functions being called by the program.

1.3.2 Phase II: Initial Research

During this phase, the student became familiar with unknown technologies discovered during ***Phase I***. Studies on new techniques, such as Machine Learning (ML) and other algorithms were conducted to provide clarity on which could be implemented to achieve the assignment's objectives.

Additional research was required as development progressed, as further technologies or libraries were uncovered and/or as new features desired came to light.

1.3.3 Phase III: Definition of System Requirements

Once there was sufficient understanding of the equipment provided by the company and on relevant existing technologies, the project's objective was scrutinised and system requirements were defined. These would to need to be achieved via a proof-of-concept and tested to ensure the developed program would meet the company's expectations.

1.3.4 Phase IV: Proof-Of-Concept (POC)

A POC application was designed without hardware limitations (i.e., on a desktop computer) to prove the assignment was feasible. During development, some of the initial design choices were modified as more knowledge on the existing technologies was gathered.

1.3.5 Phase V: Porting

With a complete and functional POC, the software was ported to run on a smartphone. Development used the original mobile application as a starting point, but a complete redesign of it ensued. New games were also created to demonstrate extra functionality of the algorithm developed.

This page is intentionally left blank.

2 Phase I: Original System Analysis

2.1 System Architecture

The original system consists of two subsystems: the Gadget Board with its embedded software and a smartphone with a mobile app. A wired USB connection provides direct communication between the two. The system architecture in Figure 2 demonstrates the interactions between the subsystems with a user:

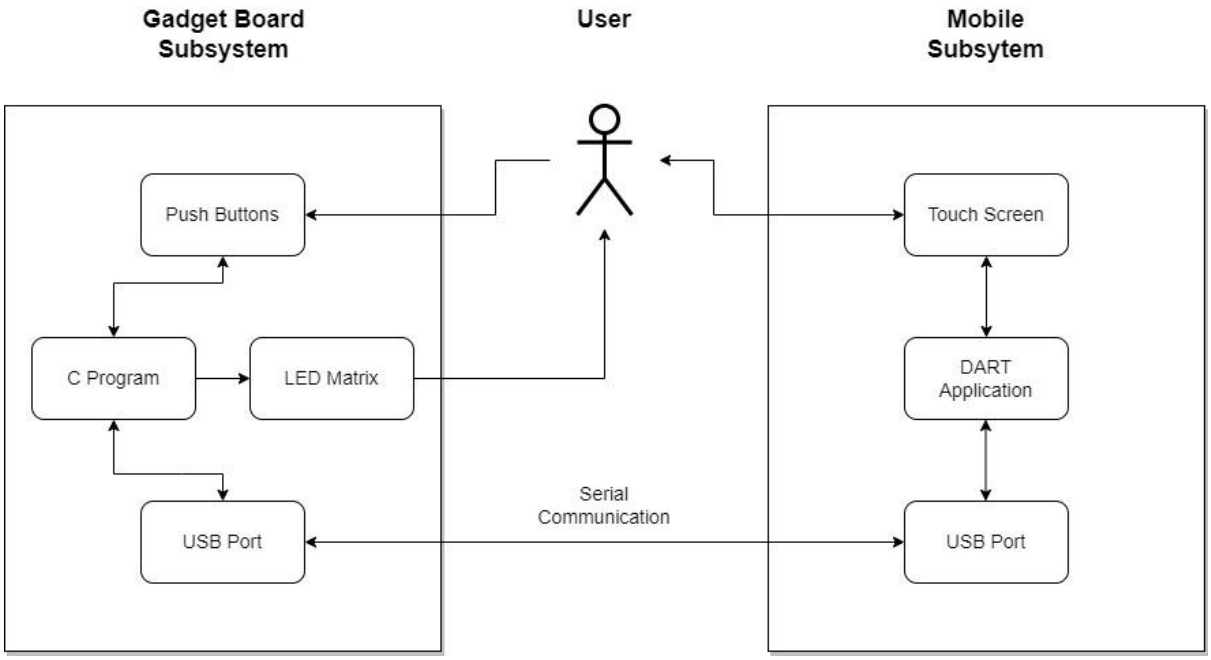

Figure 2 - System Architecture

2.2 Hardware

The Gadget Board's physical properties relevant to this project are:

- A 200 x 100 mm Printed Circuit Board (PCB).
- 144 mono-colour red LEDs and 144 mono-colour green LEDs.
- 6 push-buttons on the right-hand side of the board, three at the top, three at the bottom.
- A PIC18F87J94-I/PT 8-bit microcontroller.
- A micro-USB connector for serial communication.
- A connector for programming the microcontroller.

The LEDs are assembled in pairs – one for each colour – set out as a 12 x 12 matrix with a 5 mm clearance in all directions between each pair. This matrix is used as a display. The embedded software uses timers which manipulate the lights to create 5 intensity settings, these being: switched off, or lit up with 4 different levels of perceivable brightness.

The LEDs are connected in rows of 12; the red and green are wired individually. This means 12 MCU pins, one for each row of red and 12 for each row of green. Each column is connected to 12 other pins, but with all pairs connected together instead. This forms a closed system where instead of being wired to ground, each row is connected to an I/O pin of a column.

The simplified hardware schematic in Figure 3 showcases the circuit aforementioned:

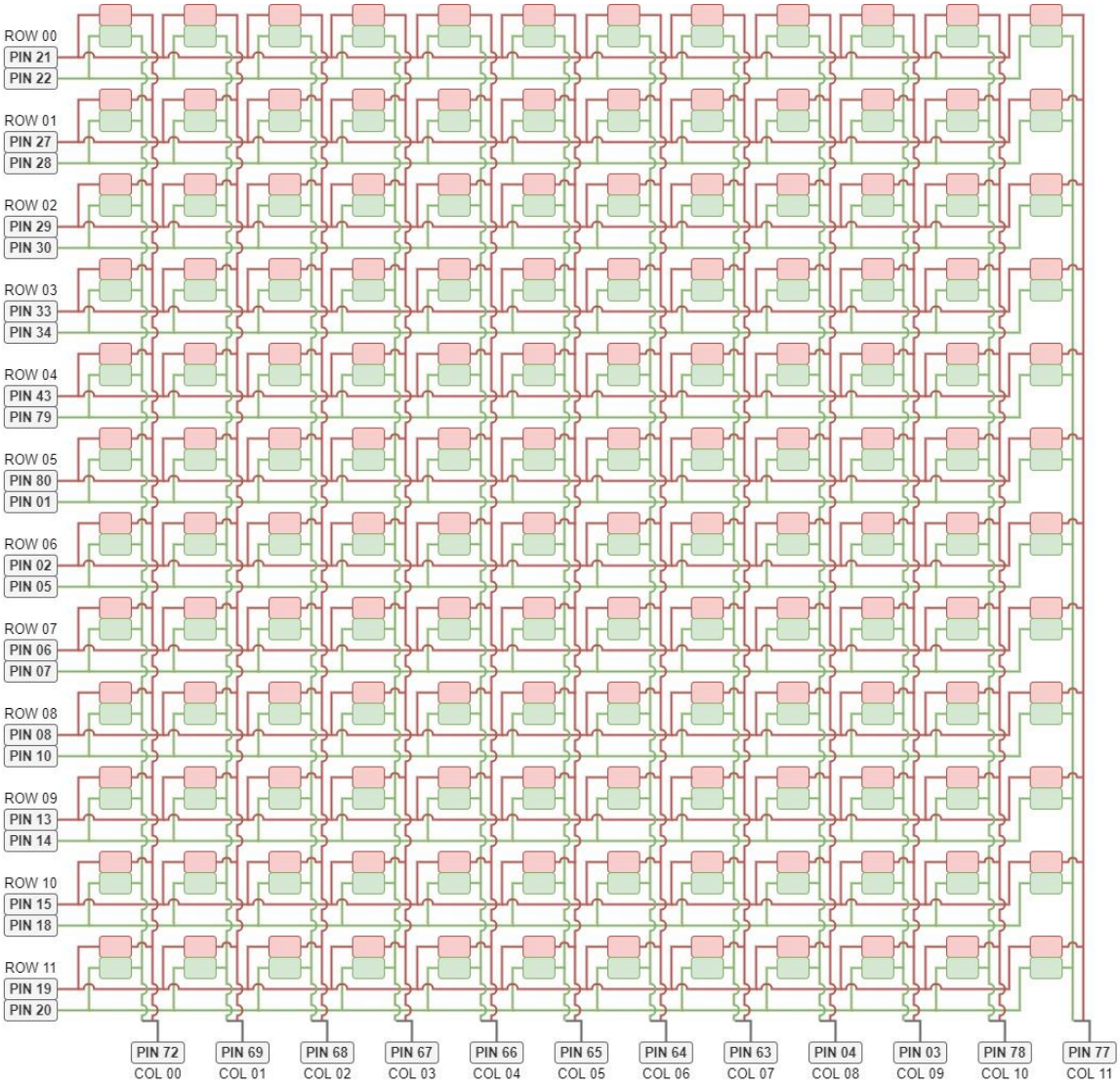

Figure 3 - Hardware Connections

2.3 Software

2.3.1 Gadget Board

The microcontroller is programmed to control the LED's status – on or off – and to communicate via serial with the mobile phone's subsystem. This consists of receiving instructions and reporting readings on the light detection of LED's or state of the push-buttons – pressed or not-pressed.

The embedded software triggers an Analogue-to-Digital Conversion (ADC) of the light detected by all 144 red LEDs and transmits this data via serial, at a frequency of 3.47 Hz. The message includes the header "LM=", followed by a 12 x 12 matrix with the conversion values; each index represents one LED.

When a push-button is pressed, a message with the header "BM=" is transmitted, followed by an array of size 6, representing the state of each of the buttons. Examples of both serial messages are seen on the following page:

```
LM=[[550 555 551 563 562 560 552 558 559 556 557 564]
    [558 561 559 564 551 556 555 550 560 562 554 553]
    . . .
    [552 564 562 557 560 558 559 556 550 561 554 551]
    [551 559 560 554 557 564 555 552 562 558 550 553]]
(12, 12), dtype=”String”

BM=[[0 0 0 1 0 0]]
(1, 6), dtype=”String”
```

2.3.2 Mobile App

The mobile app is a Flutter App in Dart language, which contains the logic for Tic Tac Toe. It reads user input – either via the phone’s touch-screen or the board’s push buttons – and sends serial messages to the GB, with instructions to turn specific LEDs on or off, according to the state of the game. A full analysis of the embedded software and mobile app is available in Appendix I.

2.4 Layers

The different interactions between the subsystems is further detailed via the layer diagram in Figure 4:

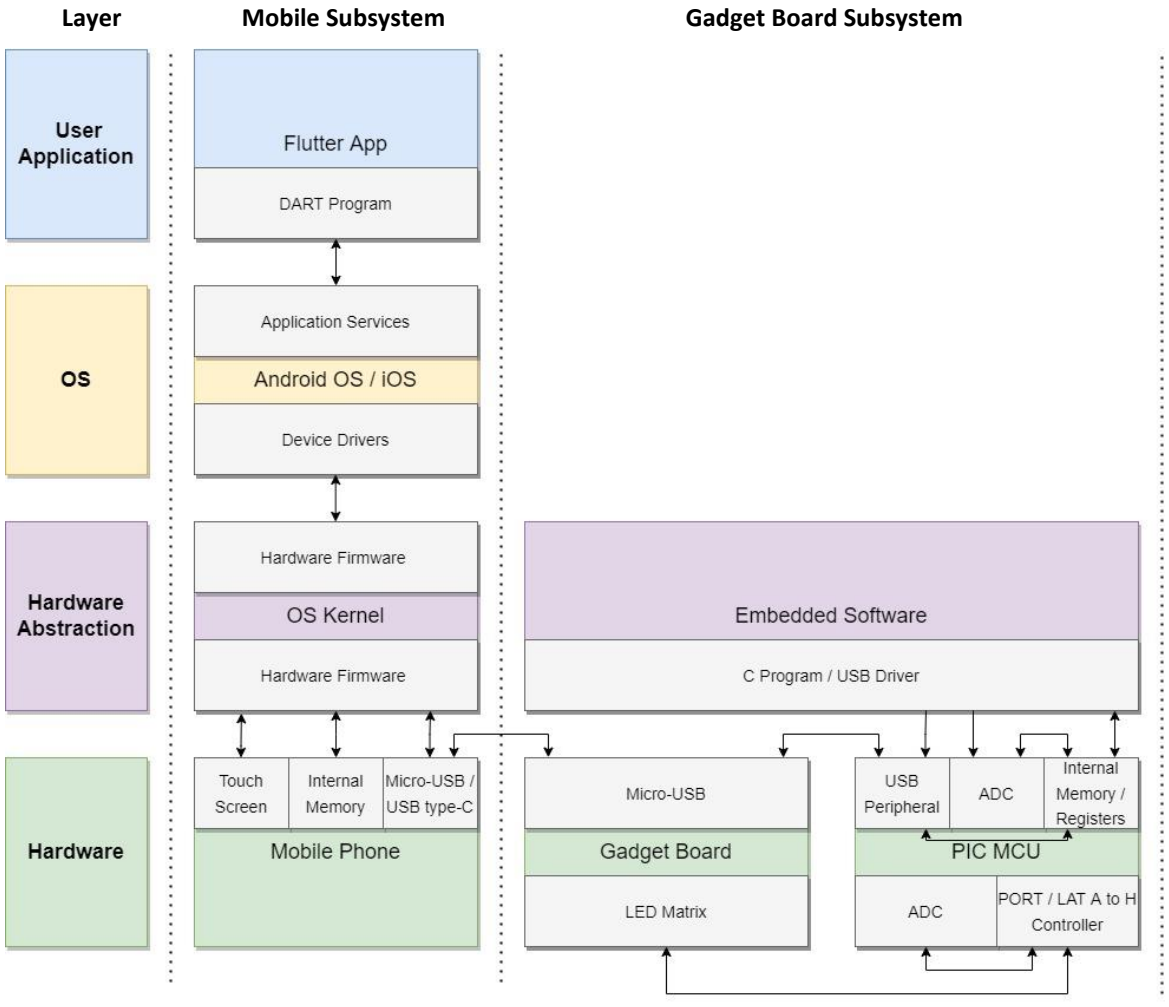

Figure 4 - Layer Diagram

2.5 Initial Testing

2.5.1 Overview

During the ADC process, the green LED within a pair is switched on while the red one is polled by the MCU. Touching the pair caused more green light to reflect into the red LED, which increased its ADC value, while any shadow cast on others decreased theirs instead. Pointing a flashlight to the GB increased all values according to the intensity of the light beam reaching each part of the matrix.

Despite all LEDs having the same specifications, they did not always produce the exact same values. This could be due to different performances between electrical components, such as capacitors, varied wire resistance or slightly different placement angles on the board.

2.5.2 Tests

Test programs were written in Python as the first attempt to detect touch using the ADC values of the LEDs. The script interpreted the serial messages with header "LM=" coming from the GB and instructed the embedded software to switch on a specific LED as per the following conditions:

1. Taking the matrix index with the highest value.
2. Taking the matrix index where the value is higher than a predefined threshold.
3. Calculating the average between three matrices and taking the index with the highest value, above the predefined threshold.

2.5.3 Results

Condition 1 proved not workable, as even when the matrix is not being touched, there is always a 'highest value', meaning random LEDs were being switched on at every reading.

Condition 2 worked occasionally. Sometimes after the user removed their finger from the GB, several LEDs reported a high reading for reasons beyond the scope of this assignment, triggering misreads. As the user approached a certain LED, others reached the threshold beforehand, causing misfires. Any threshold set would also only work in rooms with similar lighting conditions and the program would require constant adjustments. Darker rooms produced lower ADC values across the board, while sunny rooms produced the opposite, causing the threshold to either never being reached or being constantly reached by all LEDs

Condition 3 provided a better response than Condition 2, although the same problems remained.

2.5.4 Conclusion

The initial testing showed that a smarter algorithm was required, as all ADC values needed to be analysed as a whole and put into perspective, as the user could approach the board from different directions, causing various patterns of collateral shadowing.

External light greatly influenced readings, so any program needed to constantly adapt itself, as this product is intended as a mobile gadget, susceptible to environmental changes.

3 Phase II: Initial Research

This chapter outlines details on the information gathered during the initial research phase, from how LEDs detect light and the ADC process, to machine learning basics & other algorithms which could be suitable for this project. To provide direction, a main research question and several sub-questions have been defined.

3.1 Research Question

Is it feasible to use the output of the light detection capability of the LEDs within the Gadget Board, to detect when and where the LED matrix is being touched?

3.1.1 Sub-Questions

1. How do LEDs detect light and how can this be measured and any output values interpreted?
2. What is an Analogue-to-Digital Conversion and how can any output values be interpreted?
3. What types of algorithms are suitable for fast predictions based on input data?
4. How can such algorithms be deployed?

3.2 Technical Research

Below follows a summary of the findings for each of the sub-questions defined above. Full detail is available in Appendix II. The main question is answered throughout the report, in context with the application developed.

3.2.1 Sub-Questions 1 & 2

Research on the capabilities of LEDs detecting light showed that these diodes generate an analogue signal directly proportional to the intensity of light shone upon them. If amplified, this signal can be measured by a microcontroller and translated to computer language via an Analogue-to-Digital Conversion.

3.2.2 Sub-Questions 3 & 4

The research for suitable algorithms for the problem at hand indicated that machine learning is the best solution. Details were found on use-case scenarios of several existing methods – such as supervised, semi-supervised, unsupervised or reinforcement learning – and model types – such as classifying, regressing and clustering, among others.

Supervised classifiers stood out since no existing data was available, hence it could be gathered to fit the purpose of the assignment. These models require labelled data, where several inputs correspond to a specific output. They will learn the association on how certain parts of the input affect the output and are able to make predictions when given new, unlabelled datapoints.

When looking at ways to gather and store such data, a few different formats were found – such as Comma-Separated Values (CSV) [4] and binarized with Pickle – the early being the most popular. Techniques for data processing have been studied – such as normalizing and scaling – as well as ways to clean up datapoints in preparation for fitting them to a ML model. These techniques were used according to the needs of the Model in question.

Two different training processes have been reviewed, with Batch Learning being ideal, where the model will be trained once using all available data and then deployed to another application.

This page is intentionally left blank.

4 Phase III: Definition of System Requirements

4.1 Analysis

The objective was analysed in line with the findings during ***Phases I*** and ***II*** of the project, to set out the priorities of the POC to be developed and to define what would be considered a successful implementation for each of the following areas:

4.1.1 Accuracy

The accuracy level when reading user input on the LED matrix affects usage of the Gadget Board. Four definitions of it have been agreed upon between the student and company supervisor, as seen on Table 4:

Table 4 - Accuracy Definition

LEVEL	DEFINITION	SCENARIO
0	Inaccurate	Board does not detect any specific touch.
1	Semi-Accurate	Board detects touch on a 4 x 4 LED quadrant
2	Accurate	Board detects touch on a 2 x 2 LED square
3	Very Accurate	Board detects touch on a single LED

The company would consider this area successful once the POC could read input with an accuracy level of 1 as a minimum, as that would already provide a mean for simple tasks to be executed on the Gadget Board.

4.1.2 Data

At the beginning there was no data available on the light changes sensed by the LEDs, therefore a gathering process was required, as data is fundamental to machine learning projects. Its format and quantity depended on the chosen ML model, but datapoints needed to be labelled and follow the structure of the information being transmitted by the Gadget Board. Any data processing should occur on the POC and not on the board's embedded software due to its hardware limitations. The company would consider this area successful once there was a method of gathering and storing relevant data efficiently, with some level of automation.

4.1.3 Hardware

The LEDs used for development had to be the ones present on the Gadget Board to reflect a real usage scenario, where ambient light and temperature varied depending on location and to guarantee that the placement and distance between each LED were consistent.

The company would consider this area successful if the POC could run with the GB connected to a normal computer, despite this allowing for a significant higher processing power than a mobile phone. This would serve as minimal proof that it is feasible to use an LED matrix as an input source.

A decision was made between the student and company supervisor on when to port the program to the mobile phone subsystem, according to the POC's results.

4.1.4 Software

After a successful implementation of a mobile port of the POC, a decision was made on which games could be developed to run on the system. Any new game was seen as an extra.

4.2 Moscow

Clear, measurable and prioritised requirements were needed so verification via unit and integration tests would be possible. The following MoSCoW [5] specifications were used to create and identify their priorities:

- MUST: Must have this requirement to meet the assignment's needs.
- SHOULD: Should have this requirement, but project success does not rely on it.
- COULD: Could have this requirement if it does not affect anything else on the project.
- WON'T: Would like to have this requirement later, but delivery won't be this time around.

Table 5 details all requirements. The POC…

Table 5 - Moscow

ID	MUST
M-01	…detect user input on the LED matrix with accuracy level 1.
M-02	…detect user input on the LED matrix within a period of 1000 ms*.
M-03	…gather and store labelled data to train and validate any ML model.
M-04	…provide metrics on any ML model to aid with analysis and improvements.
M-05	…use the LED matrix on the existing Gadget Board as a source of data.
M-06	…have clear, commented code and include instructions for future development.
ID	**SHOULD**
S-01	…detect user input on the LED matrix with accuracy level 2 or higher.
S-02	…detect user input on the LED matrix within a period of 500 ms.
S-03	…be ported to the mobile phone and integrated with the existing game of Tic Tac Toe.
S-04	…have additional games and functionality on top of the existing one.
S-05	…save live usage data locally to periodically retrain ML model.
ID	**COULD**
C-01	…save combined live usage data in a cloud database to periodically retrain ML model.
C-02	…be lightweight to run on low to mid-range mobile phones
ID	**WON'T**
W-01	…have Over-The-Air support for future updates.
W-02	...allow the user to develop its own tasks or games via the mobile phone app.
W-03	…have an LCD display to show game score.

**This time period was established as it would represent low input-lag, making the game playable.*

5 Phase IV: POC – Development Process

This chapter explains the development process followed throughout the POC.

5.1 Iterations

Development was planned with the possibility of multiple iterations being realised, where each would consist of a complete development cycle using a specific ML model and test suite. An iteration would be complete once the POC would cover all different accuracy levels and could be tested directly on the Gadget Board with its performance metrics saved for future analysis.

This approach would allow for comparison between different models, where decisions could be made, as for example, between using one with mediocre accuracy yet lightweight or one with high-accuracy but demanding computing resources. The most appropriate would eventually be ported to the mobile phone subsystem.

5.2 Stages

Every iteration comprised three stages to cover accuracy levels 1, 2 and 3. For every stage, the possible touch positions within the LED matrix were defined by the POC's logic. These were unique and did not overlap. For simplification, a touch position is referred to as an ***index***. All stages shared ***Index 0***, which represents the LED matrix not being touched. Specification on the remaining indexes are in Figure 5:

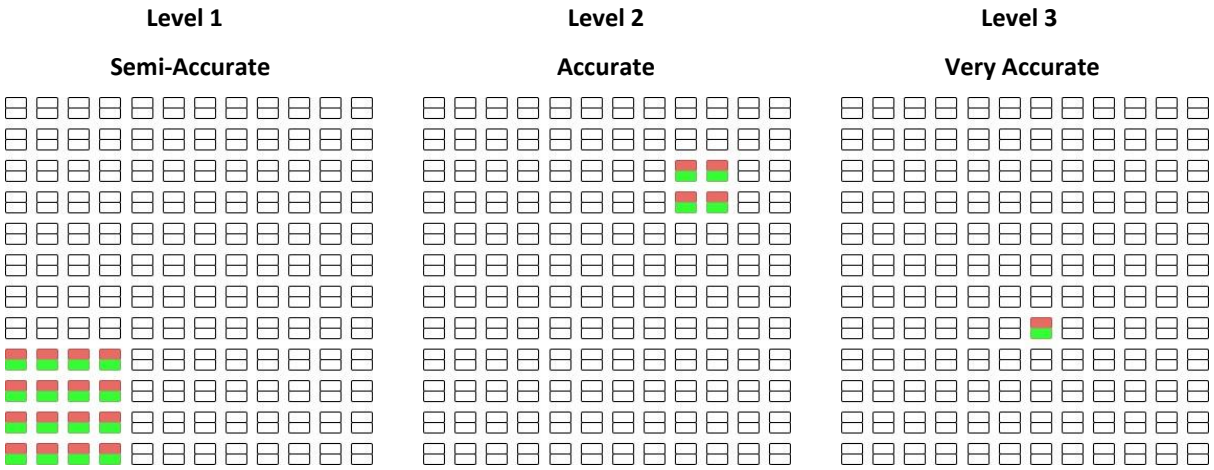

Figure 5 - Accuracy Modes

5.2.1 Stage 1: Semi-Accurate

On a 12 x 12 LED matrix, there can be 9 unique 4 x 4 quadrants. In Figure 5, from top to bottom, left to right, they count from 1 to 9. The quadrant highlighted represents ***index 7***.

5.2.2 Stage 2: Accurate

On a 12 x 12 LED matrix, there can be 36 unique 2 x 2 squares. In Figure 5, from top to bottom, left to right, they count from 1 to 36. The square highlighted represents ***index 11***.

5.2.3 Stage 3: Very Accurate

On a 12 x 12 LED matrix, there can be 144 unique LEDs. In Figure 5, from top to bottom, left to right, they count from 1 to 144. The LED highlighted represents ***index 91***.

5.3 Steps

Each stage contained four steps, each with parts of the POC which together would achieve the project requirements. The steps were:

- ***Step I***: Data Gathering
- ***Step II***: Model Training & Validating
- ***Step III***: Testing on Gadget Board
- ***Step IV***: Stage Review.

The program developed for each step required three variations, one for every stage. Once development within a step was completed, work moved on to the next. During ***Step IV***, if results were satisfactory, work would progress to the next stage, otherwise it would return to ***Step I***.

5.4 Complete Cycle

The development process described was required as it was not possible to gauge if results of a single step were optimal until the program got tested directly on the Gadget Board. Each iteration comprising three stages of four steps had its own feature branch on GitLab for version control. Figure 6 represents the complete process:

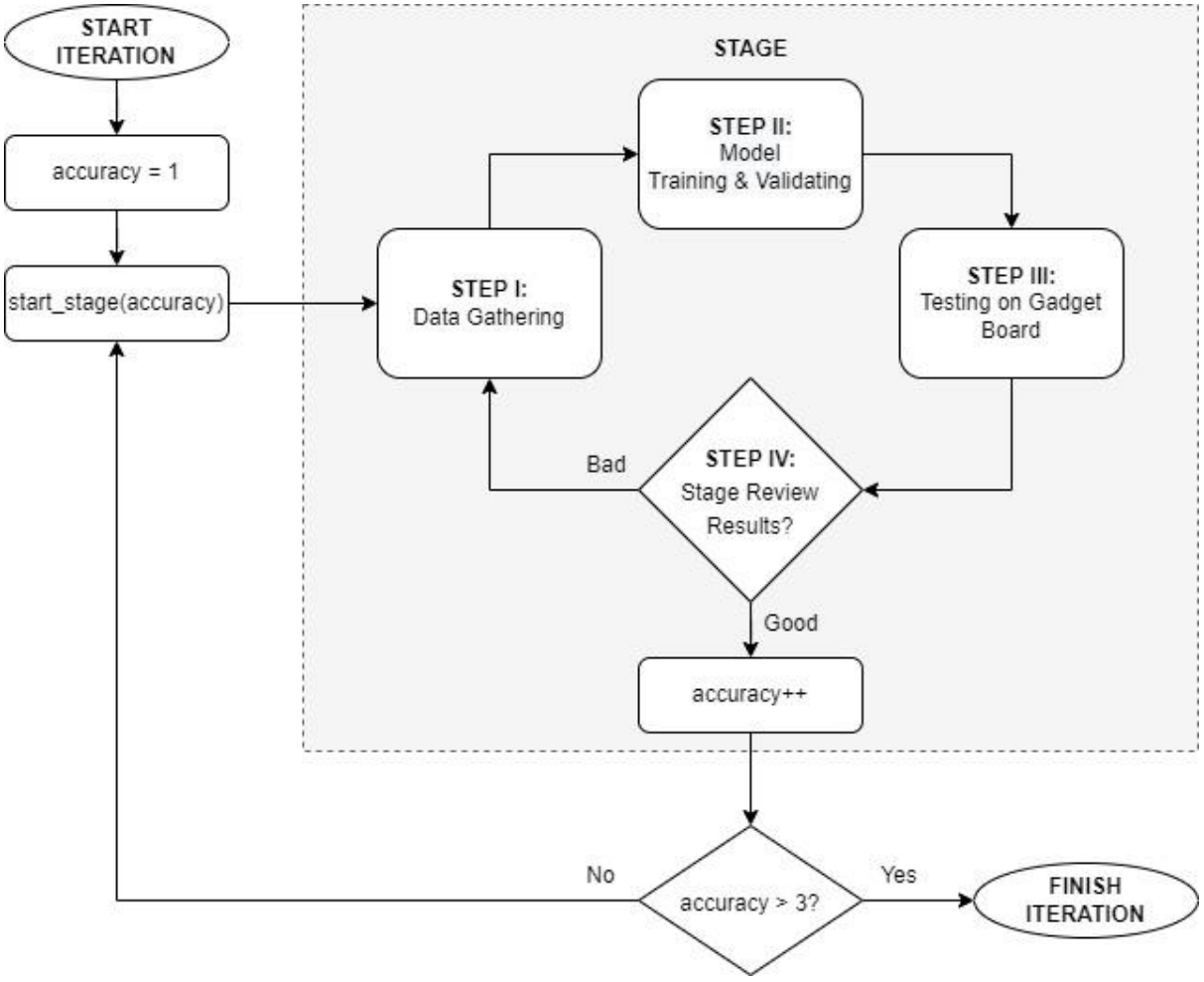

Figure 6 – Development Process

6 Phase IV: POC – Functional Design

This chapter details the functional design of the POC, explaining what was planned for ***Steps I*** to ***IV***, to complete every stage of a development iteration. This design was inspired by the defined system requirements. The motivation for the initial choices made references to the findings of ***Phases I*** & ***II***.

6.1 Step 1: Data Gathering

The program for ***Step I*** was planned as a game, to make the data gathering process appealing, keeping interest for longer than if it was a simple repetitive task. The game Simon Says [6] was the inspiration. Its objective was to efficiently build a database of classified data, where all datapoints would contain a label indicating which ***index*** they represented. This data format was to comply with requirements of the classifier models researched. Each accuracy level needed its own database, as the range of labels was different between them.

6.1.1 Overview

The game logic tells the user where to touch the LED matrix and stores the live ADC readings coming from the Gadget Board, which are momentarily influenced by such touch. For ***Stage I***, the user is prompted to touch one of the 9 quadrants, for ***Stage II***, one of the 36 squares and for S***tage III***, one of the 144 LEDs. During a single gameplay, multiple inputs are requested, one at a time, to gather data as quickly as possible.

Since the board's LEDs are not capable of physically detecting touch, the logic must notify the user when to do so and provide a control method so only good data is stored.

6.1.2 Progression:

The program runs in rounds, starting at 1 and ending after round 9. The total number of rounds is arbitrary, a limit is imposed to incentivise users to play through the three variations covering all accuracy levels.

In every round, the logic temporarily switches on the LEDs for randomly generated ***indexes*** and then notifies the user to touch the board there. At this point, the live ADC values coming via serial are stored in a database including the current ***index*** shown to the user to provide a label to the datapoint. The number of indexes shown within a round are the same as its number, so a logical progress is made. Once a round is complete, the game moves on to the next. The pseudocode below represents round 1 of the game played during ***Stage 3***:

```
level = 1
ForEach range(level)
    round[].append(random_number([1, 144]))
EndForEach

ForEach round as index
    led_matrix[index] = 1
    sleep(1)
    led_matrix[index] = 0
EndForEach

notify_user()
user_touch[12][12] = read_adc_values()
stop_notification()
save(accuracy, (user_touch + index))
level++
```

6.2 Step 2: Model Training & Validating

The program for ***Step II*** was planned as a standalone script. Its objectives were to pre-process the data gathered during ***Step I***, train & validate a ML model, save its metrics for future analysis and export it for use in a different script. This model had to correctly categorise an unclassified array of 144 values within the range of labels for the relevant accuracy level.

6.2.1 Overview

The logic loads the relevant database and verifies that there are datapoints covering all possible outputs for the current accuracy level. Depending on the ML model used, the datapoints may need to be pre-processed according to the techniques researched. The data is then fitted to a model, with some of it set aside to be used during validation.

Finally, the validation results are displayed as graphs so the information is meaningful. These metrics serve the purpose of identifying where possible problems are, as the model might perform better for some outputs than others. They are saved and made accessible for future reference when comparing with results of unit tests and of ***Step III***. This analysis happens during ***Step IV***.

6.2.2 Flowchart

The chart in Figure 7 shows the progress flow of the script described above:

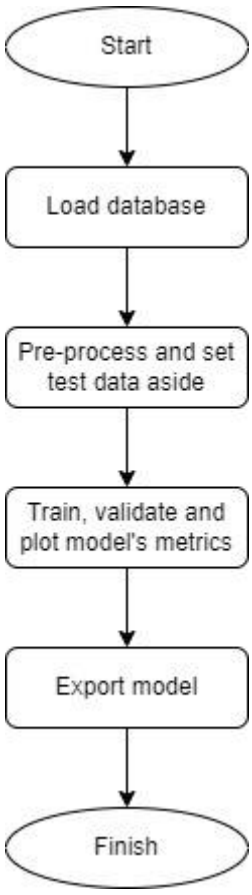

Figure 7 - Model Training & Validating

6.3 Step 3: Testing on Gadget Board

The objective of this step was to cross-check that the accuracy reported by the ML model exported on ***Step II*** was the same when running directly on the Gadget Board. Additionally, if the accuracy was high enough, this program would serve as an initial proof that using the LED's sensing capabilities for input detection was feasible, thus answering the Main Research Question. Validation of this step would occur after extensive testing.

This logic was planned as the same game of ***Step I***, but instead of saving the ADC readings arriving from the Gadget Board, it passes them to the model created during S***tep II*** and uses the prediction returned to check if the user input was the same as the ***index*** generated for the current game round.

6.3.1 Overview

Only a few variables and in-game functions are modified so the game runs as the description above. The overall logic to generate and display rounds and notify the user to touch the board remains unaffected.

6.3.2 Progression

The logic analyses the model's prediction of the user input. While it detects ***index 0***, nothing happens. When it detects another ***index***, it compares it to the level generated. Feedback is provided to tell the user what was the prediction and if their answer is correct or not. When the prediction matches the ***index*** displayed by the logic, the game progresses to the next level or is reset in case there is a mismatch. The pseudocode below represents round 1 of the game played during ***Stage 1***:

```
level = 1
ForEach range(level)
    round[].append(random_number([1, 9]))
EndForEach

ForEach round as index
    led_matrix[index] = 1
    sleep(1)
    led_matrix[index] = 0
EndForEach

notify_user()
While prediction == 0
    user_touch = read_adc_values()
    prediction = model.predict(user_touch)
Endwhile
stop_notification()

If prediction == index
    correct()
    level++
Else
    incorrect()
    level = 1
Endif
```

6.4 Step 4: Stage Review

The objective of this step was to assess if the current stage's performance was satisfactory, by reviewing metrics of ***Step II*** and any notes taken during ***Steps I*** and ***III***. These results would also compare with the ones from different stages, to check whether the ML model being used was suitable for all accuracy levels. The following may be found during review:

Overfitting

if the performance observed in ***Step III***, which uses live data does not match the accuracy indicated in ***Step II***, it may tell that the model is overfitted, as it will perform well with known data but not with new, unseen data-points.

Low-Accuracy

Any model should offer an accuracy of at least 90% so gameplay is not cumbersome, especially if the game's logic relies on precise input. The algorithm should prefer having a higher number of false negatives than false positives, meaning it is better to not detect any touch than to detect an incorrect position as that could break the game.

Performance

If the user is required to touch the board for a long period before a correct prediction is made, it could mean that either the model requires optimisation or the current data gathered in ***Step I*** is not sufficient to teach a clear distinction between outputs. A satisfactory performance level is one where the user is able to play the game without problems analogue to high input latency.

6.4.1 Actions

If any of the above is experienced, then the current model is not performing to a satisfactory level and work should return to ***Step I*** where changes can be made. When the result is indeed satisfactory, work shall progress to the next stage.

7 Phase IV: POC – Set Up

Once the POC's functionality was designed, explaining what it would do, work progressed to its technical phase where how the program would run was established. Prior to that, a few scripts were written to facilitate communication with the Gadget Board, the original system received some updates and a framework has been defined.

7.1 Helper Modules

Importable helper modules have been written in Python containing functionality regularly needed by the POC. These modules contain an "`__init__.py`" script specifying which methods could be imported.

7.1.1 Board Handler Module

This module contains functions to read incoming serial messages, switch on LEDs and display specific patterns on the LED matrix, such as a bootscreen, numbers, blinking patterns and etc. Details can be found in the source code.

Led Matrix Class

A key functionality of this module is to define the class `LedMatrix`, which allows the initialisation of 12 x 12 x 1 objects used to hold information on the state of all LEDs. The 3rd dimension is an object of class `Pixel`, which contains the intensity level for the red and green LEDs. The Python snippets below show the class definitions:

```
class LedMatrix:
    def __init__(self, width=WIDTH, height=HEIGHT):
        self.width = width
        self.height = height
        self.matrix = [
            [Pixel(0,0) for _ in range(width)] for _ in range(height)
        ]

class Pixel:
    def __init__(self, red, green):
        self.red = red
        self.green = green

    def to_byte(self):
        return ((self.green<<4) & 0xF0) | (self.red & 0x0F)
```

Read Serial Buffer

The Python code snippet on the following page shows `read_serial_buffer()`, developed to make a single reading of an entire matrix of ADC values. Planning ahead, these readings would get flattened, being converted from a 12 x 12 matrix into a size 144 array. This would allow for better readability of datapoints, once this information was saved into a database:

```
def read_serial_buffer():
    with serial.Serial('/dev/ttyACM0', 460800) as serial_connection:
        led_values = serial_connection.readline().decode('utf-8')
        serial_connection.close()
        if led_values.startswith('LM='):
            led_values = np.array(json.loads(led_values[3:]))
            return led_values.flatten()
        else:
            return None
```

The return value was converted as such:

```
led_values = [550 555 551 563 562 560 ... 555 552 562 558 550 553]
(1, 144), dtype=”numpy.array”
```

Set LED Matrix

Any serial message received by the Gadget Board needs a header so it can be interpreted correctly. The header `“Set\0”` indicates the message contains a `LedMatrix` class object, where its indexes contain a `Pixel` class object with an intensity level between `‘0’` and `‘4’` for the red and green LEDs. The Python code snippet below shows `set_led_matrix(led_matrix)`, developed to enable the POC to easily light up specific LEDs:

```
def set_led_matrix(led_matrix):
    with serial.Serial('/dev/ttyACM0', 460800) as serial_connection:
        serial_connection.write(b'Set\r')
        for col in range(led_matrix.width):
            bytes_array = []
            for row in range(led_matrix.height):
                bytes_array.append(led_matrix.matrix[col][row].to_byte())
            serial_connection.write(bytes_array)
        serial_connection.close()
```

7.1.2 Data Handler Module

Each datapoint saved by the POC had to comprise 145 columns (144 features + 1 label) and be cleaned up to only contain integers separated by a comma to comport with CSV format. Any spaces or non-numerical characters had to be removed. Three databases have been initialised, one for each accuracy level. The Python code snippet below shows `write_csv_data(database, features)`, developed to achieve the above:

```
def write_csv_data(database, features):
    with open(database, mode='a+', newline='') as file:
        writer = csv.writer(file, delimiter=',')
        writer.writerow(features)
    file.close()
```

This module also contained other functions to manipulate data, where specific datapoints could be deleted or to load the entire dataset into memory.

7.2 System Updates

A few updates to the original system have been implemented to add functionality needed by the POC and to improve performance of the Gadget Board. The embedded software has remained similar to the original, with the exception of a few changes in the program's main and in the ISR.

Performance Improvements

A performance improvement was done by including a manual call at the end of the main loop for the USB driver to check for new serial messages and update the USB buffer. This call was previously made in the ISR. With this change, the driver checks only occur once per cycle and not at every single interrupt. This has made the ISR's overall duration regular, as the USB messages vary in length and by consequence in reading time.

Another improvement was to the ADC process. The sampling time was minimally increased, which provided more consistent readings and a Boolean state now allows for the entire process to be toggled on or off, as required by developers. Toggling the ADC process off is useful when displaying animations on the Gadget Bord as it removes the distracting blinking of green LEDs which occur during the ADC polling, it makes the duration of the ISR shorter and stops serial broadcasting, reducing load on the MCU's processor.

New Functionality

A Piezo Buzzer was added to the GB, allowing for audio output which would serve to notify the user of certain events. The MCU's code was updated with a function that produced the square wave required to make the buzzer ring. The program's Interrupt Service Routine called this function when *TMR2IF=1*, meaning at every 400 µs if *PR2=200*. This effectively generated a 2.5 KHz square wave.

This functionality could be toggled between on and off, by changing the data direction of the pin connected to the buzzer between input and output. The default direction was input, meaning no sound is emitted. The C code snippet in the following page shows the update to the MCU's embedded software and the Python one shows `buzz()`, developed to toggle the buzzer's status:

```
if (strcmp((const char*)tmp_msg, "Set\0") == 0) {
    read_from_usb = 1;
} else if (strcmp((const char*)tmp_msg, "Buz\0") == 0) {
    toggle_buz = !toggle_buz;
    pin_io(&_BUZZER, toggle_buz);
}
```

```
def buzz():
    with serial.Serial('/dev/ttyACM0', 460800) as serial_connection:
        serial_connection.write(b'Buz\r')
        serial_connection.close()
```

The embedded software now checks for additional headers when reading messages stored in the USB buffer. If the serial message contains the header…

- "Set\0", it is followed by a 12 x 12 x 1 matrix indicating which LEDs should be on or off. This information is stored in a *led_matrix* array of the embedded software, which is later accessed to update all LED's state.
- "Adc\0", it indicates a request to toggle the status of the ADC process between on and off.
- "Buz\0", it indicates a request to toggle the data direction of the buzzer pin between input and output, effectively turning the buzzer on or off.

7.3 Framework:

7.3.1 Simon Says

The scripts for ***Steps I*** and ***III*** were planned to be written in Python as per the helper modules and to be contained within their own module. As per the functional design, both steps make use of the same Simon Says game logic, with only a few parameters and functions being different to cater for the different functionality between them. To choose between steps, a command line argument needs to be added when calling the module via terminal. The bash commands below shows how to call the game in 'Data Gathering' mode as per ***Step I*** or in 'Testing on Gadget Board' mode as per ***Step III*** respectively:

```
python3 simon_says

python3 simon_says 1
```

The module's "`__main__.py`" asks the user to choose the accuracy level and parses the command line argument, storing the information in variables used throughout the game. It then calls the scripts required for the gameplay logic with these variables passed as a parameter.

7.3.2 No Touch Data

A separate script was required to create datapoints where the user is not touching the LED matrix at all, or is just hovering their hand above it at different distances – between 1 and 5 centimetres away. These datapoints were needed to teach the ML algorithm the state where only ambient light or user proximity is being detected. This script calls `read_serial_buffer()` 100 times per execution and appends ***index 0*** to return values before saving it contemporarily to all databases.

7.3.3 Jupyter Notebooks

Jupyter Notebook [7] was used for the scripts of ***Step II***, allowing the code to be written modularly, offering an easy solution to swap out libraries or functions being employed, without the need of a complete rewrite. It also allows for the execution of code in snippets and displays its output and plots directly underneath their respective calls. When only certain parts of the code need changing and to be reran, other time intensive tasks such as machine learning model training do not need to be repeated.

7.3.4 Test Suite

A test suite has been created to provide an automatic check against specific system requirements. Unit tests on the POC's code were designed prior to development, to aid in planning the program and to ensure all required functionality would be included in a modular and verifiable way. This was realised using Python's `unittest` [8] library. Details will follow in the next chapter.

8 Phase IV: POC – Technical Design

This chapter contains information on the technical design of each part of the POC including its test suite.

8.1 Overview

Scripts covering the complete development cycle were designed modularly, enabling features to be swapped out so the program would work on ***Stages 1***, ***2*** and ***3***. The POC as a whole comprises the following subsystems:

Gadget Board Subsystem

- Gadget Board hardware.
- Embedded software for PIC18F87J94 MCU.

Python Subsystem

- Test Suite
- Simon Says game module.
- Jupyter Notebook scripts for the three accuracy levels.
- Board Handler & Data Handler importable Python modules.

Figure 8 demonstrates the system's architecture.

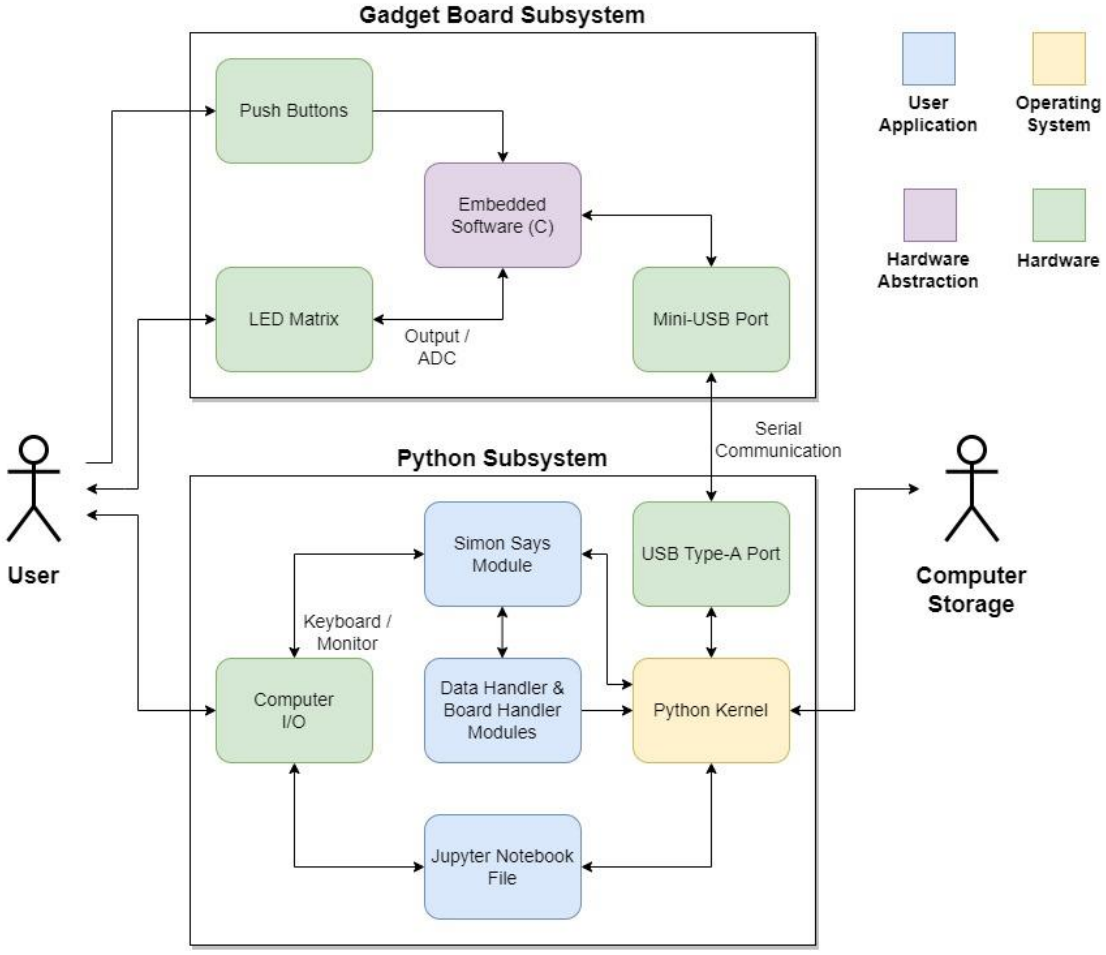

Figure 8 - POC Architecture

8.2 Test Suite

Each individual test was planned to call individual functions from the POC and assert that the returned values would be as expected. Any function called would need to be implemented in the program's logic and behave the same regardless of the current accuracy level and game mode.

8.2.1 Summary

Four tests have been created to certify that the POC achieves certain functionality or specific system requirements. A summary can be viewed on Table 6:

Table 6 - Unit Tests

TEST ID	TEST NAME	SYSTEM REQUIREMENT
T-01	Game Round Generation	--
T-02	Labelled Data Gathering	M-03
T-03	Model Accuracy	M-01 / S-01
T-04	Model Speed	M-02 / S-02

8.2.2 Tests

T-01: Game Round Generation Test

This test verifies that the gameplay's rounds are always within the correct range (i.e., it does not ask the user to touch ***index 10*** during ***Stage 1***, where the highest is ***index 9***). In this test, 250 different rounds are created via the function `generate_round(level, accuracy)` and its outcome is asserted to be within the range of the accuracy level passed as a parameter. The code snippet below show this unit test:

```
def test_level_generation(self):
    for _ in range (250):
        for accuracy in range (1, 4):
            round = poc.generate_round(1, accuracy)
            self.assertTrue(round >= accuracy.min and round <= accuracy.max)
```

T-02: Labelled Data Gathering Test

This test verifies that the POC is storing data in the correct format: 144 features + 1 label. The Gadget Board needs to be connected to the computer. The function `dg_get_input(level, accuracy)` is called 50 times and the length of the return value is asserted to be equal to 145. The following code snippet below shows this unit test:

```
def labelled_data_gathering(self):
    for accuracy in range (1, 4):
        for _ in range (0, 50):
            datapoint = dg_get_input(0, accuracy)
            self.assertTrue(len(datapoint == 145)
```

T-03: Model Accuracy Test

This test verifies that the ML model trained during ***Step II*** performs with an accuracy similar to the one reported by its metrics. It iterates through all possible outputs within an accuracy level and simulates real ADC readings by retrieving a random datapoint from relevant database, where the label is the same as the current ***index*** being analysed. This datapoint is passed to the model via `predict(accuracy, datapoint)`. Three executions for every accuracy level take place. Results are stored and an average is taken at the end. The test is considered successful if the average for all models is greater or equal to 90%.

Although this test provides an automated validation, performance of the model in a real-life test scenario may still be different. This might indicate the available data is not varied enough to cover the many possible lighting conditions a user may experience when using the Gadget Board. The following code snippet shows this unit test:

```
def ml_model_accuracy(self):
    for accuracy in range (1, 4):
        average_accuracy = 0.0
        count = 0
        for _ in range (0, 3):
            for index in range (0, i.max):
                db = database[accuracy].loc[data['144'] == index]
                db.pop(filtered_db.columns[-1])
                datapoint = db.values[random.randint(0, len(db)-1)]
                prediction = poc.predict(accuracy, datapoint)
                if prediction == index:
                    count += 1
            average_accuracy += count / 50 * 100
        average_accuracy /= 3
        self.assertTrue(average_accuracy >= 90.0)
```

T-04: Model Speed Test

This test evaluates the speed in which the ML model for all accuracy levels performs a prediction. It does not test its accuracy, as it does not matter whether the prediction is correct or not.

The test retrieves a random datapoint from the relevant database and defines a timer start point as *`start=time.time()`*. The datapoint is then passed to the model via `predict(accuracy, datapoint)` and an end time is defined with *`end=time.time()`*. Finally, *`end`* is subtracted by *`start`* and the execution time becomes known. Fifty executions take place for each accuracy level, with an average time taken at the end. The test is considered successful if the average for all models is lesser or equal to 1000 ms. The code snippet below show this unit test:

```
def ml_model_speed(self):
    for accuracy in range (1, 4):
        average_speed = 0.0
        for _ in range (0, 50):
            datapoint = db[accuracy].values[random.randint(0, len(db)-1)]
            start = time.time()
            poc.predict(accuracy, datapoint)
            end = time.time()
            average_speed += end - start
        average_speed /= 50
        self.assertTrue(average_speed <= 1)
```

8.3 Step I: Data Gathering

8.3.1 Simon Says Logic

The logic for this step has *accuracy* passed as a parameter and a command line argument to distinguish execution with the one from ***Step III***. It runs as per the flowchart in Figure 9 and subsequent explanations:

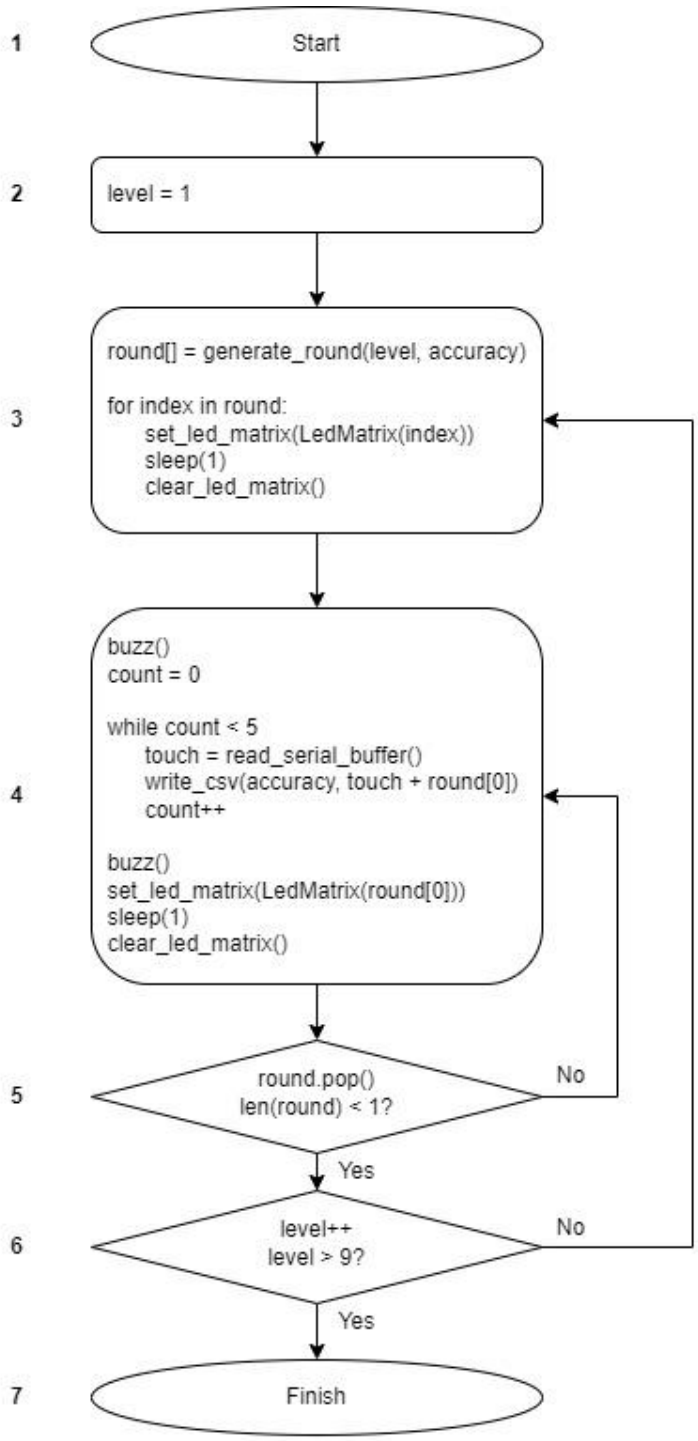

Figure 9 - Simon Says Step I

Blocks 1 and 2 show the start of the program and initialisation of `Level=1`, to indicate the first round.

Block 3 contains the function to generate rounds which is called by unit test ***T-01***. The `Level` parameter indicates how many ***indexes*** it will generate and `accuracy` determines the range of the outputs generated. It also displays all outputs of a round to instruct the user in which order they should touch the LED matrix of the Gadget Board.

Block 4 shows that five unique datapoints are saved at every ***index*** being touched by the user, with the first index of `round` appended to it. After 9 rounds. the user would have been prompted to touch 45 positions on the board, making a total of 225 datapoints being gathered in a full gameplay. Once the buzzer stops ringing, the correct ***index*** in play is shown to the user. If it is different to where the LED matrix was being touched, the user must close the game and manually delete the last five datapoints in the database, as they contain bad data.

Block 5 removes the first index of `round` and checks if any are left. If not, the game moves on, otherwise it returns to block 4. Block 6 increments the level and checks if that was the last one. If not, the game returns to block 3 with an updated `Level` parameter, otherwise the gameplay is over.

8.4 Step II: Model Training & Validating

8.4.1 Model

For each development iteration, a different ML model was used to train the algorithm. The choice was narrowed down to Supervised Learning models as there was classified data available. The first model planned was Multinomial Naive Bayes [9] which is normally used to identify words, but offers a short training time and performs well. The choice of a simple model for the first iteration was to get quick results, so the following steps in a stage could also be completed, in order to provide a starting point.

As results using this model were insufficient, the following iteration used a multinomial Logistic Regression [10] which is used to calculate the probability of an output based on the input. The last iteration used a Sequential neural network from Sklearn via the TensorFlow library, which allowed for a more complex model with deep layers performing more associations between the inputs than the previous models.

8.4.2 Data Pre-Processing

The database for the accuracy level of the current stage is loaded into a Pandas Dataframe [11]. These provide a thin wrap around NumPy arrays, allowing for functions and properties of both libraries to be used during pre-processing. As Dataframes, these are equivalent to 2D arrays.

The data gets split into features *X* and labels *y*, both have as many rows as in the database. *X* has 144 columns, which contain the ADC readings for each LED and *y* only one column, with the label for the datapoint. Below follows an example of a few datapoints for ***Stage 1***, where columns 0 to 143 are ADC readings for each LED – usually a value between 500 and 600 – and 144 is the ***index*** they represent:

```
   0   1   2   3   4   5   6   7   8 ... 136 137 138 139 140 141 142 143 144

529 538 529 528 526 531 531 533 530 ... 562 565 571 567 564 563 556 561   3
532 536 529 530 528 529 537 540 534 ... 563 560 573 567 567 565 556 565   7
532 541 528 531 528 532 532 535 528 ... 565 563 573 551 550 548 543 549   4
534 536 525 580 567 529 537 534 545 ... 563 580 512 501 503 505 526 530   5
519 530 520 525 535 544 539 554 555 ... 569 563 568 550 550 587 534 576   8
550 523 544 528 526 531 531 575 580 ... 562 565 571 567 564 563 556 561   2
(6, 145), dtype=”Numpy.array”
```

The data for ***Stages 2*** and ***3*** looks similar, except the label in column 144 has a range between 0 and 36, and 0 and 144 respectively.

The Sklearn library [12] offers different types of scalers that can be applied to X. At first, the model was trained without scaling or normalising the data because any changes applied during training would also be required on live data coming from the Gadget Board during gameplay.

X and y are split into training and testing samples using Sklearn's `train_test_split()` function. The percentage of data to remain for training started at 80% and got adjusted to get better results.

8.4.3 Histogram

To visualise the distribution of outputs across the database, the histogram of y is plotted as per Figure 10.

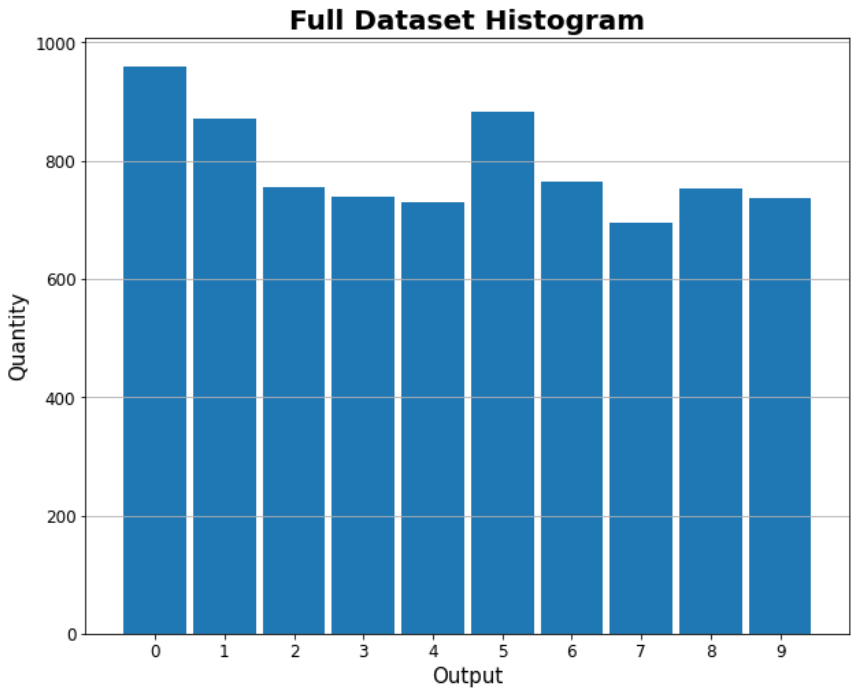

Figure 10 - Histogram

This is an example for ***Stage 1***. The graph shows there are on average between 700 and 800 datapoints for each output from '1' to '9' and short of 1000 for output '0'. This step is important to check if any of the outputs is over or under represented in the dataset.

8.4.4 Training & Validating

After pre-processing, the model is fitted and trained. The samples that were separated for testing are automatically used to measure its performance, that is seen by calling the model's *history* property. With these metrics, graphs are plotted, including a visualisation of the confusion matrix [13], providing detail on where the model is best performing, by highlighting the amount of true and false positives and true and false negatives for every expected output. Practically, the confusion matrix provides visibility on which particular part of the Gadget Board is reading user touch more and less accurately.

8.4.5 Saving & Exporting

The final task is to export the model. Sklearn's models provide a simple function to achieve that, called `sklearn.model.save("name")`. After calling this function, a new folder is created. This folder and the model inside are accessible by separate scripts via the function `sklearn.model.load_model("name")`. From then onwards, the pre-trained model will be stored into a variable and can be used to predict new values.

8.5 Step III: Testing on Gadget Board

8.5.1 Simon Says Logic

The logic for this step has *accuracy* passed as a parameter and a command line argument to distinguish execution with the one from ***Step I***. It runs as per the flowchart in Figure 11 and subsequent explanations:

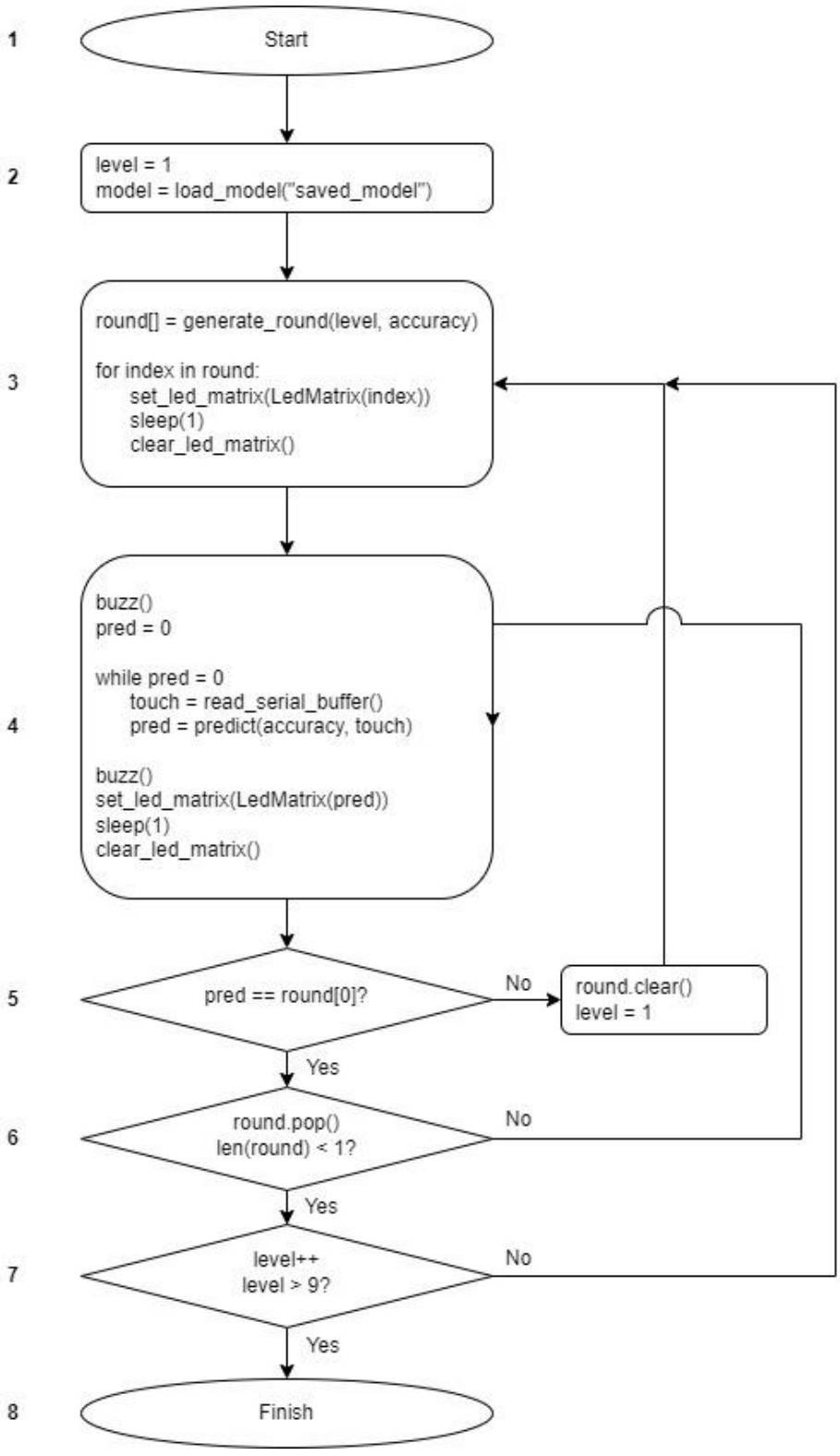

Figure 11 - Simon Says Step III

Blocks 1 and 2 show the start of the program and initialisation of *level=1*, to indicate the first round. Additionally, the ML model created in ***Step II*** is loaded into a variable for later use. Block 3 works exactly as per the implementation of ***Step I***.

Block 4 shows *pred* being initialised to '0', which means the LED matrix is not being touched. While the buzzer is ringing, the logic calls `read_serial_buffer()` and passes its return value to the model loaded in Block 2. While the prediction remains '0', the logic will repeat this process, until another prediction is found. At that point the buzzer stops ringing and the detection is shown to the user via the call `set_led_matrix()`.

Block 5 compares the predicted value to the current round's ***index***. If it is a mismatch, the logic clears *round* and sets *level=1*, effectively resetting the game. Block 6 runs when there is a match. It removes the first index of *round* and checks if any are left. If not, the game moves on, otherwise it returns to block 4.

Block 7 increments the level and checks if that was the last one. If not, the game returns to block 3 with an updated *level* parameter, otherwise the gameplay is over.

8.6 Summary

This chapter described the entire technical functioning of the POC, including the unit tests realised to confirm it is certifying relevant system requirements.

As the POC was planned for multiple iterations, analysis of the results was constantly needed so the correct adjustments could have been made. The following chapter shows the results for each of the three iterations realised.

9 Phase IV – POC: Results

The different ***iterations*** within this chapter highlight the progress made due to the development framework used. At the end of each subchapter, the lessons learnt of each cycle are detailed and changes to the program are explained and motivated.

9.1 1st Iteration: Naïve Bayes

The ML model targeted was Sklearn's Naïve Bayes, due to its very low training time and ability to run with raw values, without the need for data scaling or normalisation. This choice was to enable simple development of all steps to gather preliminary results quickly.

The following overview is a summary of the results and process flow during this iteration. Details on the progress of every development step is included afterwards.

9.1.1 Overview

The results reported by the model using test data separated during `train_test_split()` were compared to the ones from unit tests of all stages. State of these tests was verified to check which system requirements were certified. Finally, a conclusion on whether this model allowed for a smooth user experience when playing the game during ***Step III*** was reached. The above can be seen on the Table 7:

Table 7 - Naive Bayes Results

STAGE	TEST DATA ACCURACY (%)	UNIT TEST ACCURACY (%)	CERTIFIED REQUIREMENTS	GAME PLAYABLE DURING STEP III?
1	77	75	T-01 / T-02 / T-03 / T-04	No
2	3	0	T-01 / T-02 / T-03 / T-04	No
3	--	--	T-01 / T-02 / T-03 / T-04	--

These were the best results achieved after a few cycles, as on the first few runs of Steps ***II***, scores were lower and improvements were identified during ***Step IV***. Due to the poor performance of ***Stage 2***, this iteration was interrupted as it was determined that a new machine learning model was required to handle complex scenarios with multiple possible outputs.

9.1.2 Step I

Several datapoints have been gathered for ***Stages 1*** and ***2*** – around 6000 for each in total. The amount per label deviated up to 20%, as rounds were generated randomly. The total quantity gave an average of 600 datapoints per output in ***Stage 1*** and 162 in ***Stage 2***. It was already noticeable that as the number of possible outputs increased, an exponentially greater amount of data was needed.

This data was gathered exclusively in a single office at the company, with controlled environmental lighting conditions. Due to the low amount of possible ***indexes*** on ***Stage 1***, hand placement on the LED matrix during gathering was varied as seen in Figure 12:

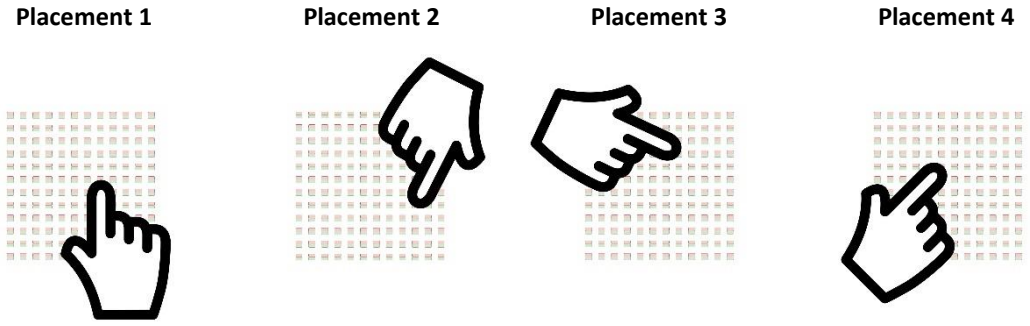

Figure 12 - Hand Placement

For ***Stage 2***, the variations were limited to placements 1 and 2 in Figure 12, due to the 4x greater amount of outputs, which allowed for less gathering time for each.

9.1.3 Step II

After separating the Pandas Dataframe into features and labels, the data was split into training and testing samples, with a ratio 80 / 20 respectively. After the first cycle, this value was changed to 90 / 10, to allow more data for training. This has produced a slightly better result, showing that having more data could improve the model's performance further.

The data was then fitted into a `sklearn.naive_bayes.MultinomialNB()` model, trained and validated. The performance of this model could be scrutinised with aid of the confusion matrix plotted from its history, which is interpreted as such: the index seen on the y-axis is the one being tested and the values within each column along the x-axis represent how many times the respective index has been predicted instead. There were plenty of wrong predictions as seen in Figure 13.

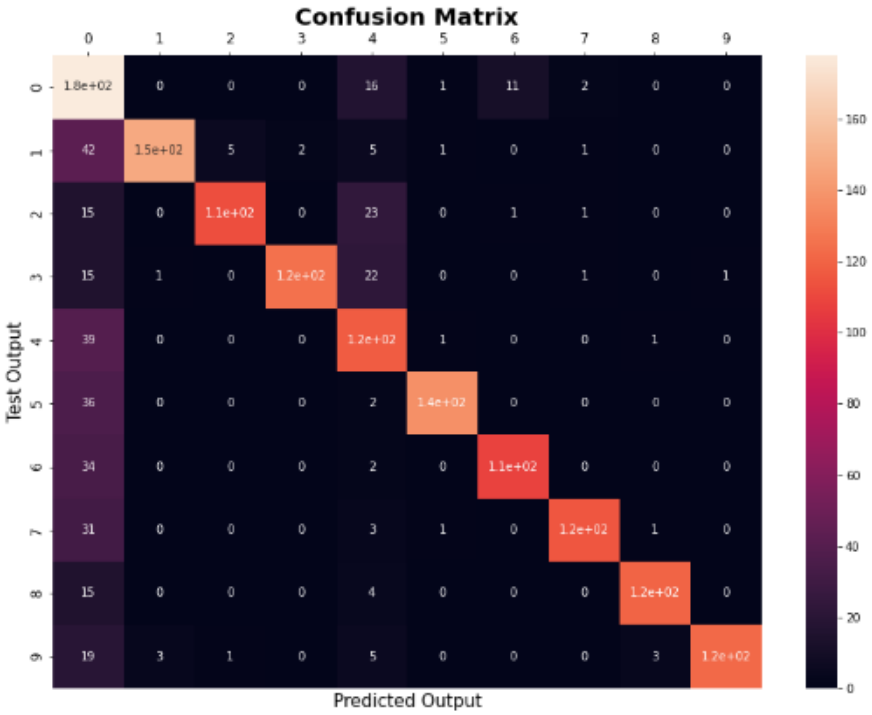

Figure 13 - Confusion Matrix

9.1.4 Step III

The model exported from ***Step II*** has been loaded into the Simon Says game and played directly on the Gadget Board during ***Stage 1*** only. No attempts were made to play the game in ***Stage 2*** as the reported accuracy was close to nil. Not much testing was possible as accuracy at the reported level has made the game unplayable. At most, the first two rounds would work as expected.

9.1.5 Step IV

The first run of ***Stage 1*** happened with 1000 datapoints – 100 per output. This has shown very low accuracy results, around the 45% mark. After researching on probable causes, the first adjustment was to add more data-points to provide richer information to the model, so it could better distinguish between outputs. The confusion matrix has shown that some outputs performed better than others, indicating that a more targeted data gathering approach was necessary. As more data was added, accuracy rose until it stagnated at the 77% mark. Other variations of the Naïve Bayes model were also tested to no avail, these being the `GaussianNB()` and `ComplementNB()`. Considering all steps above, the following was observed.

1. The amount of datapoints is insufficient on some or all ***indexes***, especially when looking at the performance of ***Stage 2***.
2. Data distribution must be evened out, so all outputs show the same performance.
3. Either bad data has been gathered during ***Step I*** or it required further pre-processing.
4. Naïve Bayes is unsuitable due to the type of data being used, as it is normally used to classify emails as spam.
5. Tests during ***Step III*** show that a higher accuracy is needed as the game was unplayable.

9.2 2nd Iteration: Logistical Regression

The ML model targeted was Sklearn's Logistic Regression, which is normally used with numerical inputs.

9.2.1 Overview

As per the previous iteration, Table 8 shows accuracy results for the test data and unit tests, as well as which system requirements are being certified and if it was concluded that this model allows a game to be playable:

Table 8 - Logistic Regression Results

STAGE	TEST DATA ACCURACY (%)	UNIT TEST ACCURACY (%)	CERTIFIED REQUIREMENTS	GAME PLAYABLE DURING STEP III?
1	93	91	T-01 / T-02 / T-03 / T-04	Yes
2	99	99	T-01 / T-02 / T-03 / T-04	Partially
3	98	95	T-01 / T-02 / T-03 / T-04	No

Curiously, despite the high accuracy reported by ***Step II***, the game was only fully playable in ***Step III*** for ***Stage 1***.

9.2.2 Step I

To resolve issues encountered in points 1 and 2 of ***Step IV*** from the previous iteration, a new modality has been added for ***Step I***, called Manual. This script was not a game, it simply asked the user which ***index*** needed to be trained and gathered 50 datapoints for it, with the user being expected to keep their finger on that position throughout.

More directed datapoints have been gathered for all levels using this new mode, with a good distribution between all different outputs. This time, data was gathered in different rooms with varying lighting conditions. ***Stage 1*** had over 8000 datapoints, ***Stage 2*** around 10000 and ***Stage 3*** over 30000. This process was very time intensive and quality of datapoints degraded over the final stage as it was only realised by one person.

9.2.3 Step II

Fitting the Logistic Regression model was unsuccessful at the start, as that particular section within Jupyter Notebook was running indefinitely until the application eventually crashed. This problem was fixed by adding a step to the data pre-processing: scaling.

The entire dataset was fitted to a Min Max scaler from Sklearn, which scaled all values between 0 and 1, with type `float32`. Afterwards, the model training time dropped to single-digit minutes. Its performance was still very low, in the 30 to 40% mark. Analysing the datapoints showed that several had columns with deviant values, below 300 or above 800, while the vast majority stood between 450 and 650. A small code was added to remove these values from the dataset, as per the snippet below:

```
for i in range (0, 144):
    indexes = data[(data[str(i)] < 450) | (data[(str(i))] > 650)].index
    data.drop(indexes, inplace=True)
data.reset_index(drop=True, inplace=True)
```

Any datapoint with at least one outlier got removed from the Dataframe. This has caused a drop of around 15% of the total amount, but accuracy results soared.

9.2.4 Step III

Stage 1 was fully playable using this new model. Several gameplays were completed without a single incorrect prediction. During ***Stage 2***, the model worked most times, but errors were somewhat frequent, with a real estimated accuracy at around 80%, while during ***Stage 3***, real accuracy was estimated at about 20% or lower. While approaching a particular position, the model would often 'misfire' and predict one of the other outputs along the way. This has made testing time consuming, also for the fact that it took several gameplays for all possible outputs to come in play, based on the nature of the game.

As the datapoints were scaled during training, the same was required with new readings coming directly from the Gadget Board. The entire database was loaded once at and a scaler object was initialised with this data. This was required so scaling of new datapoints would have the same weights as they had during ***Step II***.

9.2.5 Step IV

The model's real-life performance showed signs of overfitting, as it was performing well with known data, but poorly with new and unseen data. The lessons learnt during this iteration were the following:

1. ***Stages 2*** and ***3*** still require more quantity and variation of datapoints. The new Manual mode is prone to gathering the same datapoint repeatedly, while the original provided a more spontaneous touch, which is replicated in a real life gameplay. At times, the algorithm would make different detections when touching the same spot from different angles.
2. A new game mode is required with a quicker pace, to gather spontaneous data and test all possible outputs in a timely manner.

3. More data for ***index 0*** is required or a threshold for when the model makes a prediction is needed.
4. The Logistic Regression is not powerful enough for handling too many categories.

9.3 3rd Iteration: Neural Network

This iteration used a more complex model, an artificial neural network. The exact choice was the Sequential model from Keras. When predicting, this model provides an array with as many indexes as the possible output, showing the probability of the inputs being associated with each of the outputs. The sum of all percentages was always 100. The highest probability indicated the prediction, but also showed how sure the model was of such answer. For example, a 51% probability of the prediction being ***index 5*** would make it the highest, but likely not the expected result.

9.3.1 Overview

As per the previous iteration, Table 9 shows accuracy results for the test data and unit tests, as well as which system requirements are being certified and if it was concluded that this model allows a game to be playable:

Table 9 - Neural Network Results

STAGE	TEST DATA ACCURACY (%)	UNIT TEST ACCURACY (%)	CERTIFIED REQUIREMENTS	GAME PLAYABLE DURING STEP III?
1	94	93	T-01 / T-02 / T-03 / T-04	Yes
2	99	100	T-01 / T-02 / T-03 / T-04	Yes
3	97	95	T-01 / T-02 / T-03 / T-04	Partially

The results show that this model could better handle the multiple categories of ***Stages 2*** and ***3*** as the game was now playable in all accuracy levels, albeit with a few reservations for the final stage.

9.3.2 Step I

To overcome issues 1 and 2 of ***Step IV*** from the previous iteration, a different mode was created, named Sequential. This is a game, with an idea similar to the original Simon Says, but it shows one random output which the user must repeat immediately. The gameplay has a single round, where all possible outputs are randomly loaded into a list which is iterated through the logic. It gathers fewer datapoints at a time than the main gameplay, but these are distributed evenly through all possible outputs. This mode forces the user to touch the board spontaneously due to its faster pace, gathering more real-life data. After playing Sequential a few times, ***Stage 2*** ended with 14000 datapoints. Due to time constrains, around 90% of the data for ***Stage 3*** was gathered using the Manual mode. The process stopped when 46000 entries in total were gathered.

9.3.3 Step II

The same scaling and outlier removing tasks were kept for this iteration, but a different process was applied when splitting the dataset into training and testing samples. As a way to use all available datapoints and still separate some for validating, a Stratified K-Fold technique was used, which made four copies of the data and split each in different sets of training and validating samples.

The neural network was fitted with input and dense layers [14] with as many neurons as there were outputs for each accuracy level. The final layer had a `softmax` activation which generated the array of probabilities for each output. Different amounts of `epochs` were tried during training, with the final amount set at 25, as in each, the model would adjust the weights of how much each datapoint input affected the output, increasing accuracy. Such result can be seen in Figure 14:

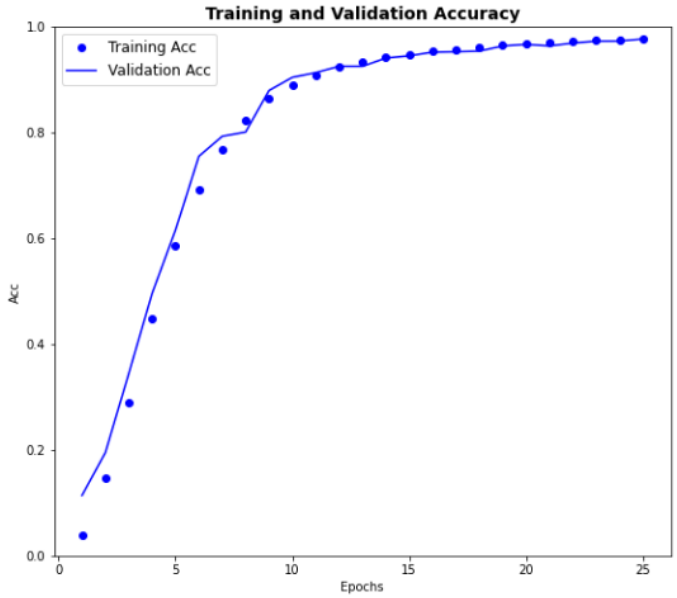

Figure 14 - Training and Validation

During this iteration, some research was made on how to run this model on a mobile device, which revealed that having it converted to TensorFLow Lite [15] format would enable this feature. To achieve that, the original model had to be initially saved in TensorFlow format. This was done by using this library from the beginning, which provides a wrapping around Sklearn's. The model was then saved and converted to `.tflite` format at the end of the Jupyter Notebook as per the code snippet below:

```
neural_net.save('nn_led')
converter = tf.lite.TFLiteConverter.from_saved_model('nn_led')
tflite_model = converter.convert()
with open('nn_led.tflite', 'wb') as f:
    f.write(tflite_model)
```

9.3.4 Step III

During this iteration, both ***Stages 1*** and ***2*** were fully playable. ***Stage 3*** has also seen improvements, but incorrect predictions still occurred, with a real estimated accuracy of about 80%. The Sequential game mode was used throughout testing, as it enabled a quick verification of all outputs.

The logic had a threshold implemented, so it would only accept the model's prediction if it was above 80% probability, otherwise it would accept it as if no touch occurred. It was noticeable that during the final stage, the hand placement had to be always the same as used during training, while for previous stages this problem was overcome by varying hand placement during data gathering. The same was seen based on the environmental lighting, as the models have only worked well in the rooms were the data was gathered.

9.3.5 Step IV

After this iteration, a decision was made between the student and company supervisor to port the POC to a mobile phone, despite the fact that several improvements could still be made. These improvements are discussed in the Conclusion & Recommendations chapter.

10 Phase V: Porting

10.1 Overview

The mobile port of the POC application needed the TensorFlow ML model created during ***Step II*** in a format compatible with the Flutter App in Dart language. This has been achieved via the following:

5. Python's TensorFlow library has a function which converts any model previously saved with `model.save()` to `.tflite` format (TensorFlow Lite).
6. Flutter has the dependency `tflite_flutter` [16], which enables the use of `.tflite` models within mobile devices.

The App has been completely redesigned to enable easy adding / removing of features and games, to comply with the industry standard Dart style-guide and to include performance improvements.

10.2 User Experience

Three games have been developed for the Mobile Subsystem to demonstrate all different accuracy levels:

- Tic Tac Toe for Semi-Accurate.
- Whack-A-Mole for Accurate.
- Drawing Board for Very-Accurate.

Details on their gameplay and logic are included in Appendix III. The new user interface provides a smooth experience, where all games are visible at a glance. The App's icon is an image of the Gadget Board. The program's main displays a splash screen for 3 seconds with the company's logo for branding and then the contents of the start screen, with a list of available games as clickable icons and playing instructions. Tapping an icon will navigate to the relevant game's page. Screenshots of the above description can be seen in Figure 15:

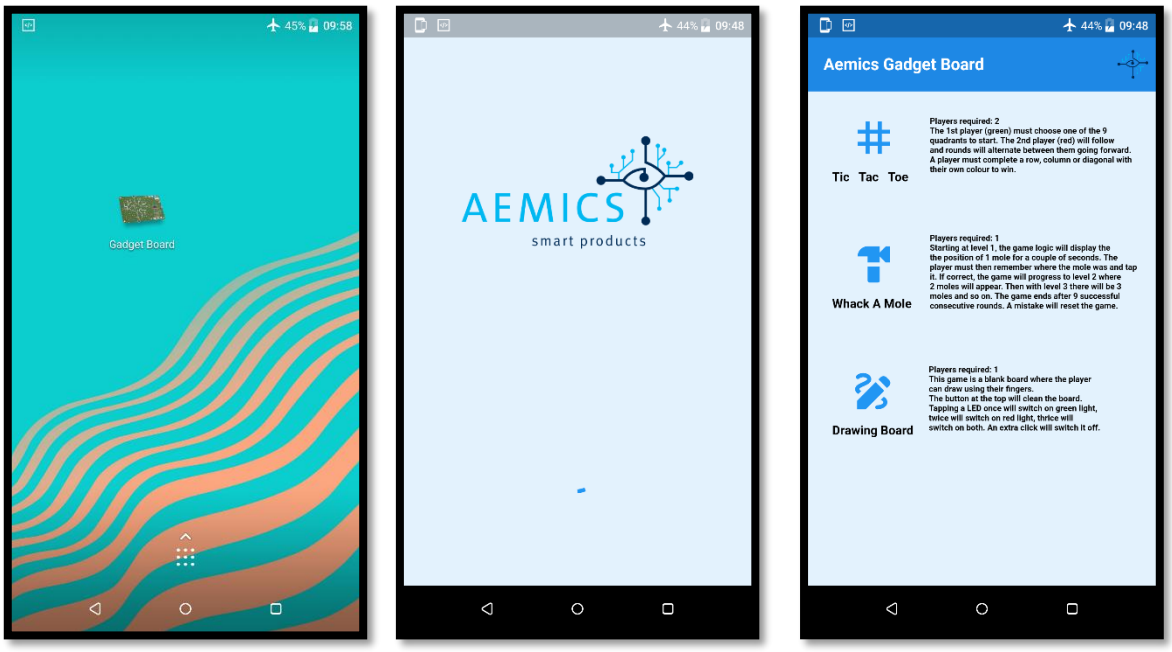

Figure 15 – App Icon / Splash Screen / Start Screen

Each game's page has a unique design as per Figure 16:

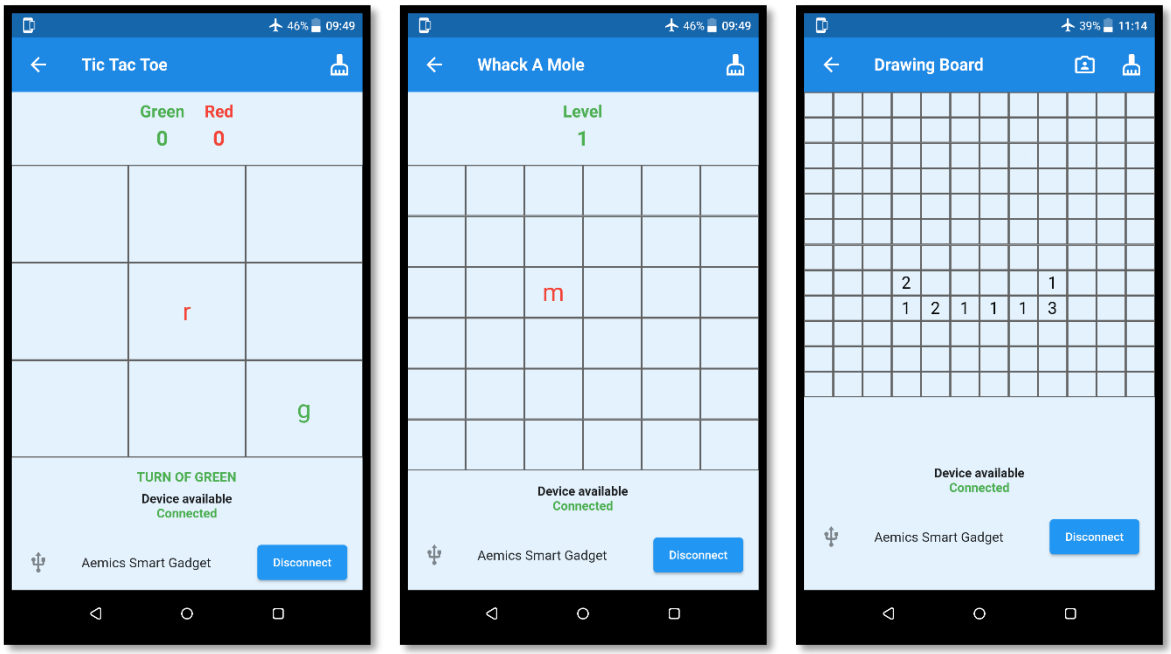

Figure 16 – Tic Tac Toe / Whack-A-Mole / Drawing Board

10.3 Utilities

10.3.1 Helper Functions

The existing text formatting function and pop-up generator have been grouped together into one file.

10.3.2 Board Handler

As per the Board Handler module of the POC, this Dart file includes functions used to communicate via serial with the Gadget Board. It declares a class to hold a `ledMatrix` object, which is a 12 x 12 x 1 List where each index represents an LED on the GB and contains the light intensity value between `'0'` and `'4'`. It also defines functions to light up the LED matrix in predefined patterns.

Another function is `updateLeds()`, to transmit via serial the command `"Set\0"` followed by a `ledMatrix` object, both converted to bytes. This function is called with a frequency of 4 Hz, providing an animation refresh rate of 4 FPS to the LED matrix. The code snippet below shows this function:

```
void updateLeds(UsbPort port, LedMatrix ledMatrix) {
  String command = 'Set';
  port.write(Uint8List.fromList(makePicMsg(command)));
  for (var row = 0; row < 12; row++) {
    for (var col = 0; col < 12; col++) {
      port.write(Uint8List.fromList([ledMatrix.getMatrix(row, col).toByte()]));
    }
    sleep(const Duration(milliseconds: 1));
  }
}
```

10.3.3 Neural Network

This file holds functions required to use the .tflite model:

Model

This function is called by each game logic as soon as they are started by the user, with the accuracy level passed as a parameter. It loads into a global variable the relevant .tflite model, which is stored in a folder in the App's root.

Data Scaling

As in the POC application of ***Step III***, live ADC values coming from the Gadget Board need the same scaling realised during training of the ML model. Dart language does not have a library with the MinMax scaler used, therefore all 144 ADC values are processed using the equivalent mathematical formula. This scaling needs the same parameters used when training the model back in ***Step II***. To achieve that without loading the entire dataset, an extra task was added to the POC to read and export the scaler weights into a text file, which is then loaded by the Flutter App and used during scaling. The code snippet below shows how ADC readings are scaled and saved into a new List:

```
var scaledMatrix = List<double>.filled(144, 0.0);

for (var i = 0; i < 144; i++) {
  scaledMatrix[i] = (double.parse(adcValues[i]) - scalerMin[i]) /
      (scalerMax[i] - scalerMin[i]);
}
```

Predictor

Once the live data has been scaled, the model's function predict() is called. Its return value is a List with as many indexes as the model's possible outputs, showing the probability for each prediction. The index of the highest value is then collected which represents the ***index*** detected. A minimum percentage can be defined at this point, so the function will only return a valid result if the probability is greater or equal than this threshold. The code snippet below shows this function:

```
int predict(List<double> scaledMatrix, int range) {
  var prediction = 0;
  var maxValue = 0.0;
  var result = List.filled(1, List<double>.filled(range, 0.0));

  neuralNet.run(scaledMatrix, result);

  for (var i = 0; i < result[0].length; i++) {
    if (result[0][i] > maxValue) {
      maxValue = result[0][i];
      if (maxValue > 0.50) {
        prediction = i;
      }
    }
  }
  return prediction;
```

10.4 Performance

The `.tflite` model's performance has been measured using an Android Benchmark App [17] from a Linux's terminal and a mobile phone connected to the computer. This tool measures and calculates statistics for the following performance metrics:

- Initialization time
- Overall memory usage

Although the tool is started from the computer's terminal, the benchmark itself runs on the mobile phone.

10.5 Results

The results can be seen in Figures 17, 18 and 19, for the models of ***Stages 1***, ***2*** and ***3*** respectively:

```
06-17 16:44:01.053  5687  5687 I tflite  : The input model file size (MB): 0.080856
06-17 16:44:01.053  5687  5687 I tflite  : Initialized session in 18.158ms.
06-17 16:44:01.054  5687  5687 I tflite  : Running benchmark for at least 1 iterations and at least 0.5 seconds but terminate if exceeding 150 seconds.
06-17 16:44:01.554  5687  5687 I tflite  : count=44968 first=539 curr=9 min=9 max=740 avg=9.64121 std=14
06-17 16:44:01.554  5687  5687 I tflite  : Running benchmark for at least 50 iterations and at least 1 seconds but terminate if exceeding 150 seconds.
06-17 16:44:02.554  5687  5687 I tflite  : count=76761 first=16 curr=11 min=9 max=3492 avg=11.2869 std=21
06-17 16:44:02.554  5687  5687 I tflite  : Inference timings in us: Init: 18158, First inference: 539, Warmup (avg): 9.64121, Inference (avg): 11.2869
06-17 16:44:02.554  5687  5687 I tflite  : Note: as the benchmark tool itself affects memory footprint, the following is only APPROXIMATE to the actual
memory footprint of the model at runtime. Take the information at your discretion.
06-17 16:44:02.554  5687  5687 I tflite  : Memory footprint delta from the start of the tool (MB): init=0.84375 overall=0.882812
```

Figure 17 - Semi-Accurate Model

```
06-17 16:46:33.613  5724  5724 I tflite  : The input model file size (MB): 0.094788
06-17 16:46:33.613  5724  5724 I tflite  : Initialized session in 3.121ms.
06-17 16:46:33.613  5724  5724 I tflite  : Running benchmark for at least 1 iterations and at least 0.5 seconds but terminate if exceeding 150 seconds.
06-17 16:46:34.113  5724  5724 I tflite  : count=39798 first=41 curr=11 min=10 max=738 avg=11.1099 std=13
06-17 16:46:34.113  5724  5724 I tflite  : Running benchmark for at least 50 iterations and at least 1 seconds but terminate if exceeding 150 seconds.
06-17 16:46:35.113  5724  5724 I tflite  : count=66526 first=20 curr=14 min=10 max=686 avg=13.3273 std=10
06-17 16:46:35.114  5724  5724 I tflite  : Inference timings in us: Init: 3121, First inference: 41, Warmup (avg): 11.1099, Inference (avg): 13.3273
06-17 16:46:35.114  5724  5724 I tflite  : Note: as the benchmark tool itself affects memory footprint, the following is only APPROXIMATE to the actual
memory footprint of the model at runtime. Take the information at your discretion.
06-17 16:46:35.114  5724  5724 I tflite  : Memory footprint delta from the start of the tool (MB): init=0.875 overall=0.914062
```

Figure 18 - Accurate Model

```
06-17 16:46:49.931  5757  5757 I tflite  : The input model file size (MB): 0.085176
06-17 16:46:49.931  5757  5757 I tflite  : Initialized session in 3.503ms.
06-17 16:46:49.931  5757  5757 I tflite  : Running benchmark for at least 1 iterations and at least 0.5 seconds but terminate if exceeding 150 seconds.
06-17 16:46:50.431  5757  5757 I tflite  : count=40507 first=39 curr=11 min=10 max=743 avg=10.9017 std=11
06-17 16:46:50.431  5757  5757 I tflite  : Running benchmark for at least 50 iterations and at least 1 seconds but terminate if exceeding 150 seconds.
06-17 16:46:51.431  5757  5757 I tflite  : count=68017 first=19 curr=14 min=10 max=1687 avg=13.0474 std=15
06-17 16:46:51.431  5757  5757 I tflite  : Inference timings in us: Init: 3503, First inference: 39, Warmup (avg): 10.9017, Inference (avg): 13.0474
06-17 16:46:51.431  5757  5757 I tflite  : Note: as the benchmark tool itself affects memory footprint, the following is only APPROXIMATE to the actual
memory footprint of the model at runtime. Take the information at your discretion.
06-17 16:46:51.431  5757  5757 I tflite  : Memory footprint delta from the start of the tool (MB): init=0.851562 overall=0.90625
```

Figure 19 - Very Accurate Model

All models have a file size smaller than a MB, can run multiple iterations in the µs realm and use less than 1 MB of RAM. Since all models use numerical 16-bit data only, modern day mobile processors can handle them with ease.

11 Conclusion & Recommendations

This assignment included elements of a research based as well as a design project. Not only there was a Research Question to answer, but the company also had certain design expectations which were defined during the System Requirements phase. The following subchapters will cover the conclusions and recommendations for both aspects aforementioned.

11.1 Conclusion

11.1.1 Main Research Question

The proof-of-concept aimed at answering the research question ***'Is it feasible to use the output of the light detection capability of the LEDs within the Gadget Board, to detect when and where the LED matrix is being touched?'***:

The results shown in chapter 9 proved that it is indeed possible to analyse the output values of the detections made by the LEDs and associate them with a particular position within the Gadget Board's matrix. This processing occurred in the sub-second realm, making this input mode responsive in addition to reliable. This POC has increased functionality of the board, which is proven by the contents of chapter 10.

11.1.2 System Requirements

Cross-checking results with all individual System Requirements assured the company that the POC developed met all expectations set out at the start of the semester.

All requirements in the ***MUST*** section have been certified, with ***M-01***, ***M-02*** & ***M-03*** being done so via unit tests. This ensures the same standard of certification will remain in future software revisions. Due to the nature of the remaining requirements, certification was only possible by manually analysing the different parts delivered.

Most requirements in the ***SHOULD*** section were also certified. ***S-01*** and ***S-02*** are proved by the same unit tests mentioned above. ***S-03*** and ***S-04***'s proof of certification are explained in chapter 10. The Performance subchapter within chapter 10 also certified one requirement in the ***COULD*** section: ***C-02***. The lightweight nature of the TensorFlow Lite models ensures that any mobile phone with an OS capable of installing the app will have hardware resources to spare when running it.

S-05, C-01 nor all ***WON'T*** requirements were achieved, but project success did not depend on these, which form part of future recommendations.

11.2 Recommendations

The student has a few recommendations of changes to the system as well as for future software revisions:

11.2.1 Software

Data

Extra time can be spent gathering data for all different accuracy levels, especially number 3. Basing on performance of the same model for the previous accuracy levels, an amount between 700 and 1000 datapoints for each output is required, adding up to a requirement of 145000 entries to the database, which currently has fewer than 50000. Additionally, data must be richer. Hand placement must cover all different angles and gathering should take place in environments with varied lighting conditions or data normalisation is required. This will ensure the Gadget Board will make correct predictions regardless of where the user is and from which side they approach it.

Mobile App

The current mobile app is still fairly simple. Since the Gadget Board is to be used as a way to advertise the company's capabilities, a more refined user interface is recommended, with animated transitions and a page dedicated to company information.

The LED matrix is capable of handling more complex gameplays than the ones available, so development of extra games are advised. All current games are still static, meaning the user can wait for an indefinite amount prior to making a selection. Moving animations with time sensitive inputs would showcase the responsiveness of the algorithm.

Cloud Support

The mobile phone is already connected to the internet, but this application is not. Connectivity is recommended for a variety of purposes:

- Allowing for online multiplayer game modes
- Allowing for over-the-air software updates
- Continuing gathering data and storing it in the Cloud, so the databases will grow continuously larger, allowing for model improvements.

11.2.2 Hardware

Extra Output

The buzzer added to the Gadget Board during development could also be used to enrichen user experience by providing sounds to the gameplay as well as other notifications. The current microcontroller also supports an OLED screen, which could be added to the board to keep score of games. This is currently only possible in the mobile app interface, but requires the user to look away from the board.

LED Spacing

At times it felt as the LED pairs were too close together, making it difficult to select a single one with the index finger. A test of increasing clearance between pairs is recommended, as it could help with overall precision of the algorithm. A possible drawback is that the quality of the image displayed by the LED matrix may decrease.

Independence

Online research has shown that TensorFlow Lite models are also compatible with 32-bit microcontrollers, as opposed to the current 8-bit one. There are cheap options of controllers which also include WiFi and Blue-tooth. An investigation is recommended on the possibility of swapping the MCU and attempting to keep all logic for games and machine learning on the Gadget Board. This would remove the need for a paired mobile phone. This solution would also require that a power source is fitted to the board, as it currently receives its energy from the phone via an USB OTG cable.

Appendix I – Detailed Software Analysis

Gadget Board Embedded Software

Overview

The original embedded software is programmed on a PIC18F87J94's internal memory. It uses 15 of the 128 KB available. An additional amount of 4 KB of SRAM can be accessed for the software operation. It is written in C language and compiled with the MPLAB's XC8 C compiler.

Start Up

On power up, the program sets the PIC MCU's configuration bits. Details on these are commented on the source code. The main processor frequency is set to 64 MHz and the timer's to 4 MHz. The slower timer provides a more meaningful control of it, where changes are noticeable to the human eye. It then initialises variables and components, such as system tasks, USB, timer, ADC, interrupts and serial communication. A key variable is `led_matrix`, a 12 x 12 x 1 matrix. From left to right, the array dimensions represent the LED's column, row and intensity value, set between `'0'` and `'4'`. The 3rd dimension is a single byte split in 2 nibbles, for the green and red LEDs respectively.

Main Program Loop & Interrupt Service Routine (ISR)

The program has two cycles, its main loop and the ISR, which gets called regularly.

Main Loop

The main loop runs indefinitely within a ***while*** loop. It first checks if there is data ready to transmit via serial USB. This data is a full 12 x 12 matrix of ADC values or an array of 6 values representing the state of each push-button, `'1'` for pressed, `'0'` for not-pressed. These are preceded by the headers `"LM="` and `"BM="` respectively.

The function then reads the USB buffer and checks for new data from the mobile subsystem. If this data contains the header `"Set\0"`, it is followed by a 12 x 12 x 1 matrix indicating which LEDs should be switched on or off. The main loop stores this information in `led_matrix`.

ISR

The ISR is triggered by any system interrupt. In this program most interrupts are from the USB device driver and a timer `TMR2`. The function first checks if any new messages have arrived via serial communication and updates the USB memory buffers as required. It then runs an ADC and LED control function called `timer_400us()`.

This function checks if the timer's period register flag `TMR2IF` is set and only runs when true, skipping the rest of the code when false. This ensures this part of the ISR will only and always execute after the interval defined on the timer's period register. When the flag is set, the function performs two main tasks:

- Runs the ADC process (details to follow).
- Reads the `led_matrix` variable and switches on or off all LEDs accordingly.

The flowchart in Figure 16 showcases a high-level overview of the program as described above.

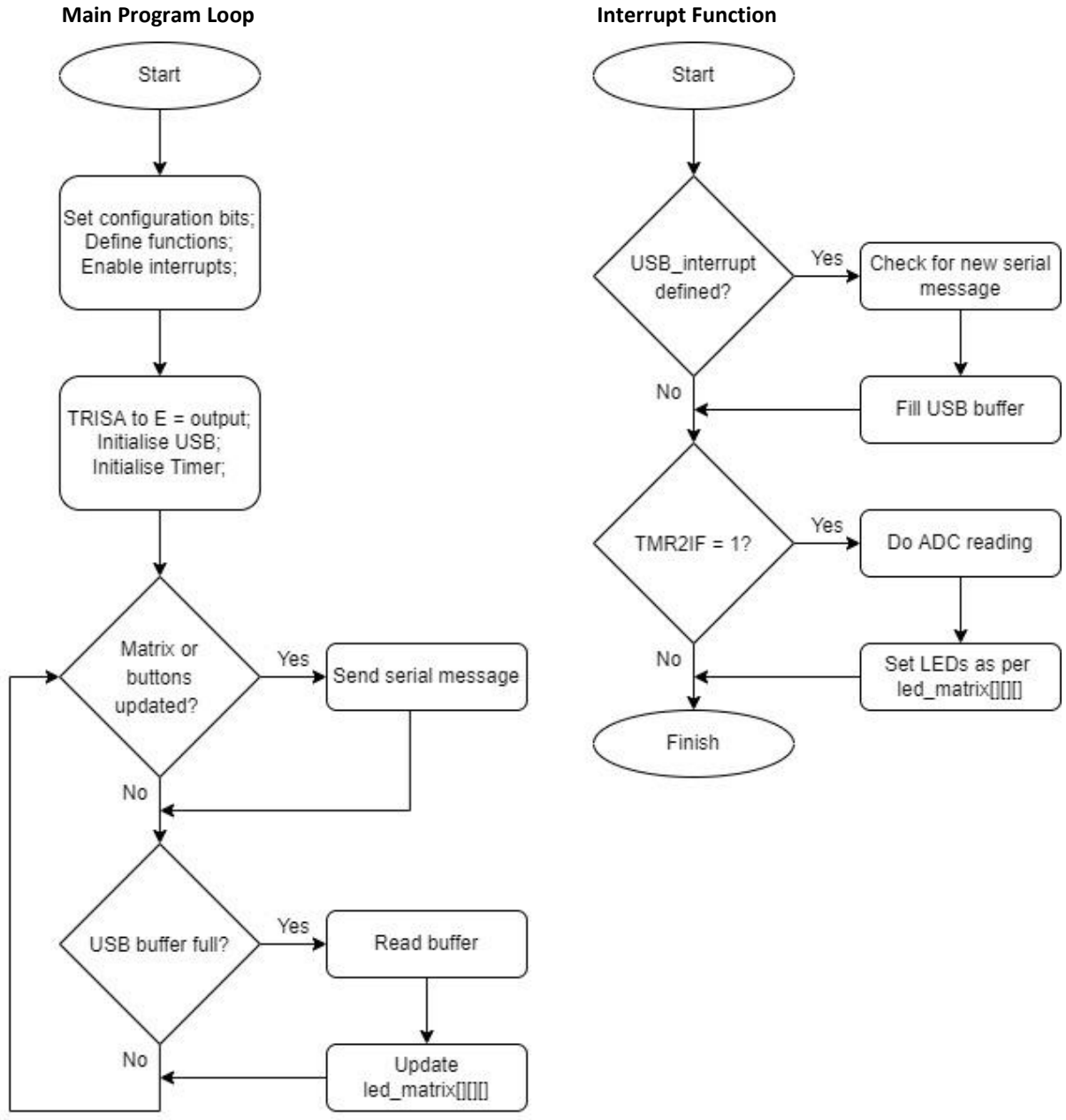

Figure 20 - Embedded Software High-Level Flowchart

Timer

The timer performs a vital function of controlling the interval in which `timer_400us()` runs, within the ISR.

TMR2 is initialised with a pre-scaler of 16 – the clock speed of 4 MHz is divided by 16 – and post-scaler of 1:2 – interrupt will only trigger half the times it would have normally done. The timer's period register *PR2* is given an 8-bit unsigned value between `'0'` and `'255'`. *TMR2*'s count resets once it matches this. Due to the post-scaler setting, every second time the timer cycles the amount set on *PR2*, the *TMR2IF* flag is set and the interrupt function is called.

The interrupt interval can be calculated with the formula below. For it, *PR2=200*, which is what this program is currently set to.

```
Interval (s) = {[1 / (Osc_Speed / Pre_Scaler)] * PR2} / Post_Scaler
Interval (s) = {[1 / (4000000 / 16)] * 200} / 2
Interval (s) = {0.0008} / 2
Interval (s) = 0.0004 => 400 µs
```

To put into perspective, 400 µs means this interval occurs 2500 times per second.

A wire was soldered to a debug pin in the microcontroller. For testing purposes, this pin was set to `'1'` at the beginning of `timer_400us()` after the check if *TMR2IF* is true and set to `'0'` at the end of the function. With help of an oscilloscope, where the vertical rulers are set 400 µs apart, it is possible to verify that the function runs at a set interval of 400 µs. Changing *PR2=250* brings the interval up to 500 µS. This is seen in Figure 17:

PR2 = 200; **PR2 = 250;**

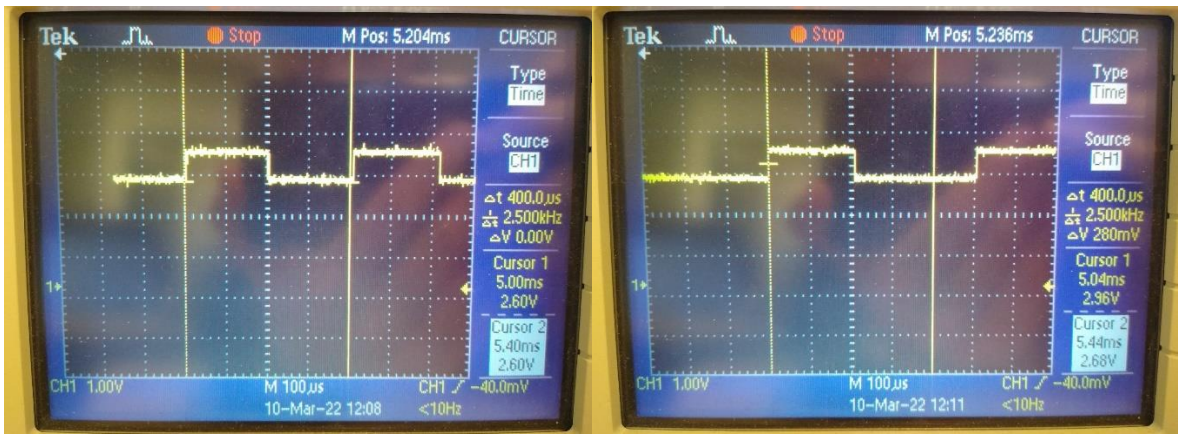

Figure 21 - Interval Measurements

The figure above also shows that `timer_400us()` itself takes 200 µs to run. This is seen by the duration in which the debug pin is set to logic `'1'`. The remainder time before the next interval is when the program's main function is running. Changing *PR2*'s value to below 175 made the interrupt cycle too short to run both `timer_400us()` and the program's main, effectively breaking the program. Maxing out *PR2=255* allows more time for both functions to run, but it also makes the whole program run slower.

ADC

The ISR performs ADC readings for every single red LED, one at a time, and saves the value into a 12 x 12 array. Each of the 144 possible indexes of this array represent a single LED pair. Two of these arrays are declared, so one can be transmitted while the other is being filled with new values. The MCU's ADC has a 12-bit resolution, so the voltages sampled can have a value between 0 and 4095 decimal – *0=0 V* and *4095=3.3 V*, which is the MCU's reference voltage.

The program lights up the green LED of a pair, then switches the data direction of the red to input. The red LED does not only detects green light, but also any environmental light with a shorter wave length than red. At this point the ADC sampling starts. If the LED is being covered by the user, the green light reflects into the red LED and discharge is slower, providing a more stable reading for that particular LED, which reflects into a higher value once sampling is finished. The process above causes a visible effect on the board, where every green light blinks a few times in a noticeable waterfall-like sequence which can be distracting.

It takes 4 interrupt cycles to sample a single LED, as explained on the flowchart in Figure 18.

- **Cycle 1:** Disables entire column previous to the target LED and the entire subsequent row. This is to avoid nearby light to 'spill' into the target LED.
- **Cycle 2:** Turns on green LED, change the red's direction to input and starts the ADC sampling.
- **Cycle 3:** Manually stops ADC sampling, which triggers conversion of the value which is saved in the ADC's buffer.
- **Cycle 4:** Waits until conversion is complete, read the ADC buffer and adds value to an array.

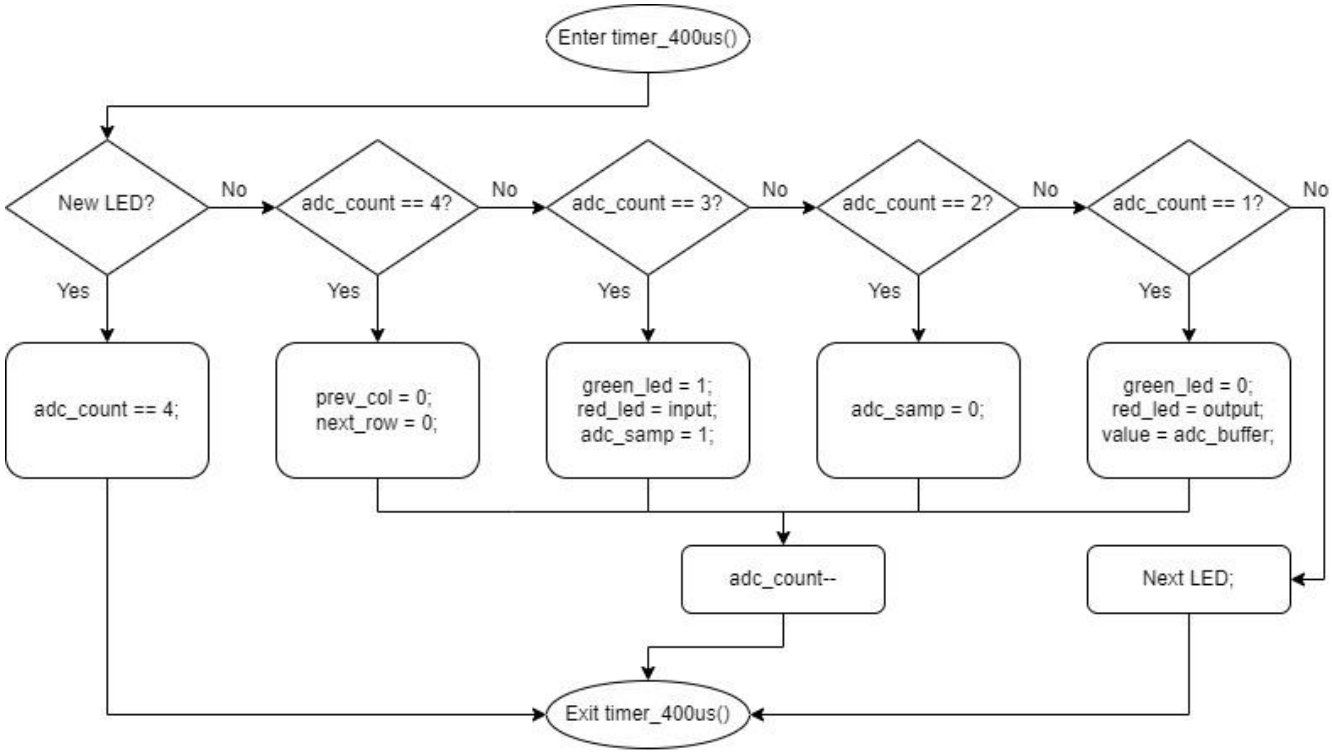

Figure 22 - ADC Flowchart

Once the ADC is complete for all LEDs, a Boolean is set to notify the main program loop. Considering the 4 cycles required plus the extra one to switch to the next LED, 720 cycles are required to fill the 12 x 12 array with ADC values for the entire matrix.

With *PR2=200*, 400 µs pass between the start of each cycle, making a complete matrix readout last 288 ms. This means 3.47 complete readings per second.

Mobile App

The Flutter App is divided into a few Dart language files: "`main.dart`", used to initialise the App and call "`startscreen.dart`", which contains logic for 2 pages, one with instructions on how to play and one with the entire Tic Tac Toe game logic. "`constants.dart`" & "`alert_dialog.dart`" are two small files containing a function to easily format text and to display an alert box. Both are used by `startscreen.dart`.

Interface

On open, the App displays a simple start page with navigation buttons and instructions on how to use it. Upon selecting the Tic Tac Toe game, a few elements appear, such as the grid, score and whose turn it is.

The screenshots in Figure 19 show the 2 pages currently available for the app.

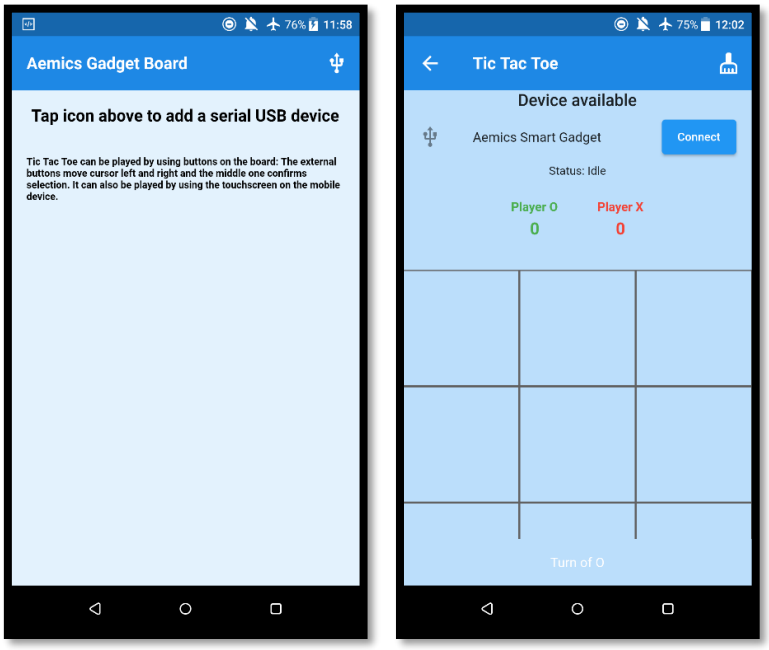

Figure 23 - Mobile App

App Logic

On start-up, the mobile app checks if any serial devices are plugged-in via an On-The-Go (OTG) wire. If true, the user is able to establish a connection to it by pressing a button on the device's touch screen. When the app connects to the GB, the LED matrix becomes a display and the game logic starts.

The application declares four 12 x 12 x 1 Lists, where each index represents one pair of red and green LEDs and can be given values between `'0'` and `'4'` – representing the 5 intensity levels of the LEDs. These Lists are regularly sent over via serial to the GB, one after the other, at a pre-defined interval of 250ms. The messages are preceded by the header `"Set\0"` as required by the board's embedded software. The predefined interval is created so the game's animation have a refresh rate of 4 frames-per-second. As the game progresses, these four arrays will have their values changed to display the required image.

The logic also sets up an event to listen for a specific USB serial message. This message is an array coming from the GB, containing information on which push button is currently active. Depending on which button was pushed, the game logic will adapt accordingly.

The flowchart in Figure 20 showcases the mobile subsystem's logic.

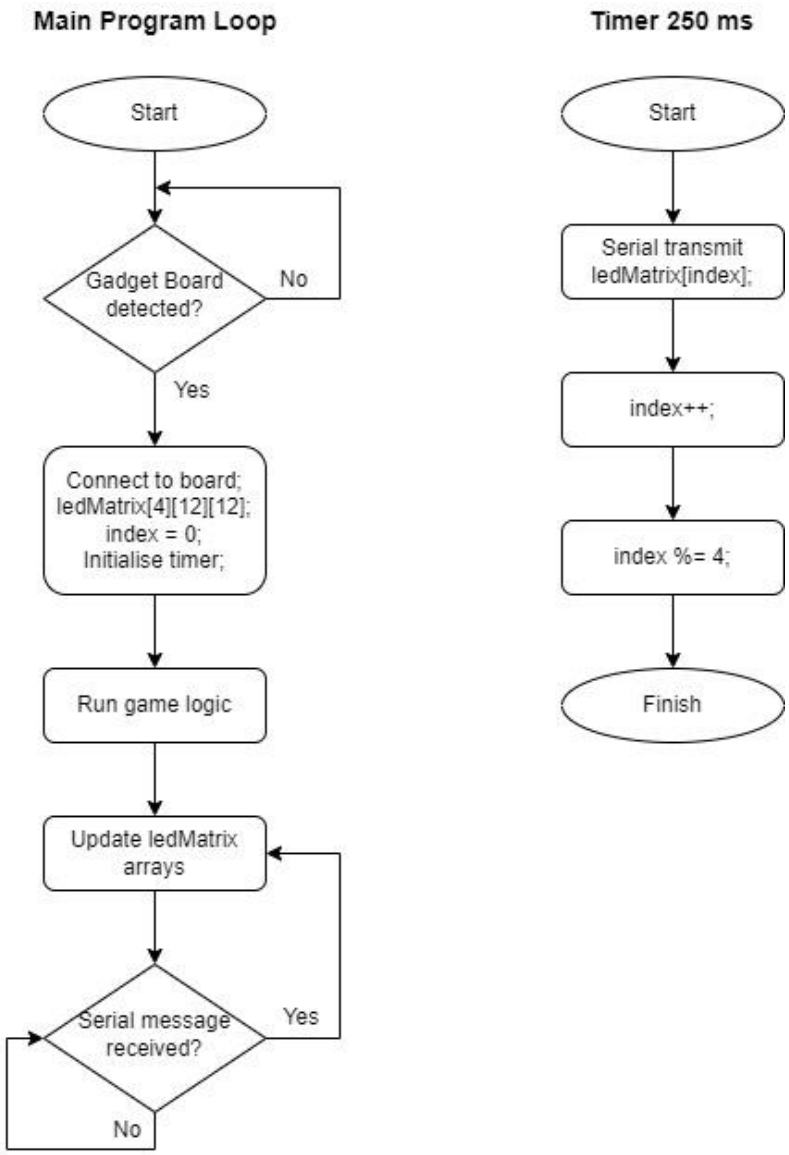

Figure 24 - Mobile Subsystem Flowchart

Return to Phase I: Original System Analysis

Appendix II – Technical & Machine Learning Research

LED Light Detection

Background

When exposed to light, photodiodes produce a current directly proportional to the photon intensity. This current flows in the opposite direction to a normal operation in a diode. As more photons hit the photodiode, the current increases causing a voltage across it [18].

LEDs are sensitive to wavelengths equal to or shorter than the one it predominantly emits. For example, red and yellow LEDs detect light emitted by a green one, but the opposite is not true.

Microcontroller

Measuring these currents and voltages is difficult without amplification. However, most modern microcontrollers have bidirectional I/O ports with configurable internal pull-ups or tri-state (high-impedance) inputs. Using the latter, the circuit can make a very accurate and precise measurement of the photocurrent [19].

In detector mode, the LED charges to +5 V very quickly (100 to 200 ns). This charge is sustained by the diode's inherent capacitance, typically 10 to 15 pF. At this stage the microcontroller should be switched to the high-impedance mode (input). Under reverse-bias conditions, a simple model for the LED is a capacitor in parallel with a current source, which represents the current induced by light intensity detected.

Due to physical properties outside the scope of this research, if more light falls on the LED, it will discharge quickly. In darkness, it will discharge slowly [20].

A MCU circuit can be configured to read the voltage discharged by the LED as an analogue signal and convert it to digital in order to make use of the value in an application.

Analogue to Digital Conversion

ADC [21] allows MCUs such as Arduinos, Raspberry Pi and other such components to communicate with the real world. Analogue signals are continuously changing values which come from various sources and sensors that can measure sound, light, temperature or movement and many digital systems interact with their environment by measuring the analogue signals from such transducers.

While analogue signals can provide an infinite number of different voltage values, digital circuits work with binary signals which have only two discrete states: logic HIGH or LOW. It is necessary to have an electronic circuit which can convert between the two different domains of continuously changing analogue signals and discrete digital signals. This is where Analogue-to-Digital Converters come in.

Such converters take a snapshot of an analogue voltage at one instant in time and produce a digital output code which represents this measurement. The number of bits used to store this data depends on the resolution of the converter.

A converter takes an unknown continuous analogue signal and converts it into a value considering the reference voltage range. For example, a 4-bit ADC with a reference voltage of 5 V, will have a resolution of one part in 15, where `0000 = 0 V` and `1111 = 5 V`. An 8-bit ADC will have a resolution of one part in 255, where

`00000000 = 0 V`, `01111111 = 2.5 V` and `11111111 = 5 V`.

Machine Learning Methods

Supervised Learning

This method [22] learns an association between input data samples and corresponding outputs after performing multiple training data instances. These algorithms require classified (labelled) data, containing multiple input parameters and one or more resulting output value. Each datapoint contains:

- X: Input parameters or features
- Y: Output

The algorithm will study the data in *X* and look at the output already given for *Y* to establish which conditions within *X* are required to produce that particular output. It is called supervised because the whole process of learning can be thought of as it is being supervised by a teacher. Examples of supervised machine learning algorithms include:

- Decision tree
- Random Forest
- KNN
- Logistic Regression

These can be divided into the following broad classes: Classification or Regression.

Classification

Predicts categorial output labels or responses for the given input data. For example, certain parameters or characteristics point that the subject is a dog or a cat.

Regression

Predicts output labels or responses which are continuous numerical values, for the given input data. For example, different key performance indicators will predict a specific stock value for a company.

Unsupervised Learning

This method is useful when the input data is not classified (labelled) and the program needs to extract useful patterns from it. Examples of unsupervised machine learning algorithms include:

- K-means clustering
- K-nearest neighbours

These can be divided into the following broad classes: Clustering, Association, Dimensionality Reduction and Anomaly Detection.

Clustering

Finds similar as well as relational patterns among data samples and clusters them in groups. A real-world example of clustering would be to categorise all customers of a store by their purchasing behaviour.

Association

Analyses large datasets to find patterns which further represent the interesting relationships between various datapoints. It is also termed as Association Rule Mining or Market basket analysis, which is mainly used to analyse customer shopping patterns.

Dimensionality Reduction

Reduces the number of variables for each data sample by selecting sets of principal or representative features. The reason behind this is the problem of feature space complexity which arises when data sets have too many features.

Anomaly Detection

Find out the occurrences of rare events or observations that generally do not occur. This is able to differentiate between anomalous or normal datapoints.

Semi-Supervised Learning

This method generally uses small supervised learning components (i.e., small amount of classified data) and large unsupervised learning components (i.e., plenty of unclassified data). One approach is to build the supervised model based on small amounts of classified data and then build the unsupervised model by applying the same to the large amounts of unclassified data to get more classified samples. Then, train the model on them and repeat the process. Another approach is to first use the unsupervised methods to cluster similar data samples, classify these groups and then use a combination of this information to train the model.

Reinforcement Learning

This method is used when the developer needs an agent to train over a period of time, so that it can interact with a specific environment. The agent will follow a set of observation strategies and will take actions in regards to its current state. The following lists the main steps of reinforcement learning methods:

- 1: Prepare an agent with some initial set of strategies.
- 2: Observe the environment and its current state.
- 3: Select the optimal policy in regards to the current state of the environment and perform important action.
- 4: The agent can get corresponding reward or penalty as per accordance with the action taken by it in previous step.
- 5: Update the strategies if required.
- 6: Repeat steps 2-5 until the agent has learnt and adopted the optimal policies.

Data Gathering for Machine Learning

Format

Data is the most important requirement to start a ML project. The most common format used is CSV, which is a simple file format used to store tabular data (number and text) such as a spreadsheet in plain text. In Python, CSV files can be uploaded in different ways, but the following aspects need to be considered:

File Header and Delimiter

The header contains the information for each field or column and the comma character is the standard delimiter to separate values. The same delimiter must be used for header and data because it specifies how data fields should be interpreted. It is important to consider the role of delimiters while uploading the CSV file into ML projects because alternatives can be used, such as a tab or white space.

Quotes

Double quotation mark is the default quote character. It is important to consider the role of quotes while uploading the CSV file into ML projects because different quotes can be used.

Data Pre-Processing

After selecting the raw data for training, it may need pre-processing, to convert the selected data into a format the ML algorithms will understand. The following techniques are possible:

Scaling

Data rescaling makes sure that attributes are at the same scale. Generally, values are changed to fall within the range 0 and 1. Certain ML algorithms like gradient descent and k-nearest neighbours require scaled data. This can be achieved with the help of the MinMaxScaler class of scikit-learn Python library.

Normalization

This is used to change values of each row of data to a length of 1. It is mainly useful in a sparse dataset where there are lots of zeros. This can be achieved with the help of the Normalizer class of scikit-learn Python library.

Binarization:

This technique makes the data binary. A threshold can be chosen, so that any values above it will be converted to 1 and below it to 0. This technique is useful when there are probabilities in the dataset that need to be converted into crisp values. This can be achieved with the help of the Binarizer class of scikit-learn Python library.

Standardization

This is used to transform data attributes with a Gaussian distribution. It differs the mean and standard deviation to a distribution with a mean of 0 and a deviation of 1. This technique is useful in ML algorithms like linear regression and logistic regression, that assumes a Gaussian distribution in input dataset and produces better results with rescaled data. This can be achieved with the help of the StandardScaler class of scikit-learn Python library.

Learning Processes

ML Models can be fitted and trained following two different processes:

Batch Learning

The model needs to be trained in a single batch, using the entire available training data:

- 1: Train the model using all data
- 2: Stop training once there is a satisfactory result/performance
- 3: Deploy this trained model into production. Here, it will predict the output for new data samples.

Online Learning

The training data is supplied in multiple incremental batches, called mini-batches:

- 1: Train the model by providing a mini-batch of training data to the algorithm
- 2: Provide the mini-batches in multiple increments to the algorithm
- 3: Provide new data samples
- 4: The model will keep learning over a period of time based on the new data samples

Frameworks

Python is a powerful yet simple language to work with, due to the many libraries and features available which could be useful for this application. Some of these features / libraries are:

- Serial communication
- Random number generation
- Array handling with NumPy
- Time delay
- CSV, Pickle, Pandas integration for data storage

Additionally, several popular Python libraries make development of ML algorithms simple and efficient. All are open-source and contain a vast array of documentation and examples online:

- Sklearn
- Keras
- TensorFlow

Return to Chapter 3 – Phase II: Initial Research

This page is intentionally left blank.

Appendix III – In-App Games

On every game's main page, the user must click 'Connect' to pair the Gadget Board with the mobile phone and start the main logic.

Tic Tac Toe

The existing logic has been updated to make use of the Semi-Accurate accuracy level, as it also divides the LED matrix into 9 blocks. Each ***index*** represents one of Tic Tac Toe's possible playing slots.

User Experience

This game requires two players. Player one should tap the LED matrix on the desired ***index*** and it will light up in green. Player two should then tap another one and it will light up in red instead. Nothing happens when tapping a position already lit up.

If one player completes either a row, column or diagonal, a pop-up will show, stating the winner. If all positions have been tapped and there are no winners, the pop-up will state the game ended in a draw. The score for each player is updated at the end of every match.

Logic

The logic triggers the ADC polling and starts predicting every single full matrix array coming from the Gadget Board. If the prediction remains `'0'` nothing happens. If it is not `'0'`, the quadrant relevant to the number predicted (between `'1'` and `'9'`) lights up with green or red, depending on which player's turn it is. Additionally, a List with size 9 registers either `'g'` for green or `'r'` for red at the same index as per the prediction. The logic knows whose turn it is with the help of a Boolean. After a tap is detected, this Boolean changes state.

From the minimum number of turns required to provide a winner onwards, after every tap the board checks the List with size 9 and verify if there is a winner. If true, `alertDialogue()` is called with the current player being passed as a parameter so they are announced as a winner. The score variable of the winner increments by 1.

If there are still no winners after turn number 9, `alertDialogue()` is called with `"Draw"` passed as a parameter.

Whack-A-Mole

The Whack-A-Mole logic makes use of the Accurate accuracy level. This game divides the LED matrix in 36 playable squares.

User Experience

Starting at level 1, one of the square's red LEDs switches on for 2 seconds to represent the mole and then it switches off. The user must tap the board on the same spot to hit it. A pop-up then appears stating if the user hit or missed the mole. If it is a hit, the game moves on to level 2 where two moles appear. If the user remembers correctly where both moles were, the game moves on to the next level, always adding one extra mole.

The game ends after level 9, if all moles were hit. If the player makes a mistake at any point, the pop-up at the end of the round informs how many moles were missed and the game resets.

Logic

The logic defines an empty growable List, a variable `level=1` is initialised and `generateRound()` is called, which generates as many random unique numbers between `'1'` and `'36'`, as the value held by `level` and adds them to the empty List. These numbers will also be registered on a List with size 36, at the relevant index with the character `'m'` to represent a mole. The logic iterates through the first List and light up in red the relevant square on the LED matrix for 2 seconds, before clearing it.

The logic then triggers the ADC polling and starts predicting every single full matrix array coming from the Gadget Board. If the prediction remains `'0'` nothing happens. If it is not `'0'`, the square relevant to the number predicted (between `'1'` and `'36'`) lights up in green and a `hits` variable is incremented by 1. Additionally, the List with size 36 registers either `'*'`, if the index predicted already contains a `'m'` or `'-'` if it was empty. If the value of `hits` is the same as `level`, `checkWinner()` is called.

This function iterates through the List with size 36 and counts how many times it finds a `'-'`. If the count is 0, `alertDialogue()` is called with `"Winner"` passed as a parameter and `level` increments by 1. If the count is greater than 0, `alertDialogue()` is called with `"Loser"` passed as a parameter and `level` gets assigned `'1'`. `generateRound()` is then called and the gameplay repeats itself.

Drawing Board

The Drawing Board logic makes use of the Very-Accurate accuracy level. This game divides the LED matrix in 144 playable LEDs.

User Experience

This game provides the user with a drawing board. Once the user taps an LED, it lights up in green. Tapping it again turns it red. A third tap lights up both green and red LEDs. A final tap switches it off. This allows the user to make a drawing using three different colour variations.

One of the buttons at the top of the screen switches off the ADC polling, to provide a clear view of the image drawn. The other button clears the board.

Logic

The logic triggers the ADC polling and starts predicting every single full matrix array coming from the Gadget Board. If the prediction remains `'0'` nothing happens. If it is not `'0'`, the LED relevant to the number predicted (between `'1'` and `'144'`) will be affected accordingly. Information on the status of each LED is stored in a List with size 144. The value `'1'` indicates the green LED is lit, value `'2'` the red one, value `'3'` both and no value indicates the LED is switched off.

Tapping the 'broom' icon at the top of the screen clears the List. Tapping the photo icon next to it toggles the ADC polling on or off, by sending the serial message `"Adc\0"` to the Gadget Board.

Return to Chapter 8 – Phase V: Porting

Bibliography

[1] Aemics, “PYg Boards,” [Online]. Available: https://www.aemics.nl/pyg/. [Accessed 11 02 2022].

[2] D. Gillbert, “Ruis Onderzoek - 3,” Oldenzaal, 2022.

[3] Scrum Explainer, “Scrum Methodology: Breaking Down the Scrum Framework,” [Online]. Available: https://scrumexplainer.com/scrum/scrum-methodology/. [Accessed 10 02 2022].

[4] File Format, “Comma-Separated-Values,” [Online]. Available: https://docs.fileformat.com/spreadsheet/csv/?msclkid=d6410bf6cf8e11ec804ae4c43f499887. [Accessed 05 09 2022].

[5] D. Haughey, “MoSCoW Method,” [Online]. Available: https://www.projectsmart.co.uk/tools/moscow-method.php. [Accessed 09 02 2022].

[6] Wikipedia, “Simon Says,” [Online]. Available: https://en.wikipedia.org/wiki/Simon_Says?msclkid=1abef993ae9d11eca9ddb482c596eb29. [Accessed 28 03 2022].

[7] Jupyter, “Jupyter Notebook Architecture,” [Online]. Available: https://docs.jupyter.org/en/latest/projects/architecture/content-architecture.html. [Accessed 10 05 2022].

[8] Python, “Unit Tests,” [Online]. Available: https://docs.python.org/3/library/unittest.html. [Accessed 10 05 2022].

[9] Scikit Learn, “Naive Bayes,” [Online]. [Accessed 10 05 2022].

[10] WIkipedia, “Logistical Regression,” [Online]. Available: https://en.wikipedia.org/wiki/Logistic_regression. [Accessed 10 05 2022].

[11] Data Analytics, “Dataframes vs Numpy Arrays,” [Online]. Available: https://vitalflux.com/pandas-dataframe-vs-numpy-array-what-to-use/. [Accessed 05 09 2022].

[12] Scikit-Learn, “Data Pre-Processing,” [Online]. Available: https://scikit-learn.org/stable/modules/preprocessing.html?msclkid=f7be371ccf9b11ecafd752b00784c293. [Accessed 09 05 2022].

[13] Science Direct, “Confusion Matrix,” [Online]. Available: https://www.sciencedirect.com/topics/engineering/confusion-matrix?msclkid=2d1e68ebd03a11ec85789226f4162eec. [Accessed 10 05 2022].

[14] MLK, “Machine Learning AI,” [Online]. Available: https://machinelearningknowledge.ai/different-types-of-keras-layers-explained-for-beginners/. [Accessed 10 05 2022].

[15] TensorFlow, “TensorFlow Lite,” [Online]. Available: https://www.tensorflow.org/lite/. [Accessed 05 05 2022].

[16] Pub.dev, “Tflite_flutter,” [Online]. Available: https://pub.dev/packages/tflite_flutter. [Accessed 10 05 2022].

[17] TensorFlow, “TFlite's Performance,” [Online]. Available: https://www.tensorflow.org/lite/performance/measurement. [Accessed 01 06 2022].

[18] Analog Devices, “LED as a Light Sensor,” [Online]. Available: https://wiki.analog.com/university/courses/electronics/electronics-lab-led-sensor#:~:text=For%20example%2C%20a%20red%20LED,light%20from%20a%20blue%20LED.. [Accessed 17 03 2022].

[19] Electronic Design, “LED On Detecting Duties,” [Online]. Available: https://www.electronicdesign.com/markets/lighting/article/21777096/single-led-takes-on-both-lightemitting-and-detecting-duties. [Accessed 17 03 2022].

[20] Sparkfun, “Using LEDs as Light Sensors,” [Online]. Available: https://www.sparkfun.com/news/2161. [Accessed 17 03 2022].

[21] Electronics Tutorials, “Analogue To Digital Converter,” [Online]. Available: https://www.electronics-tutorials.ws/combination/analogue-to-digital-converter.html. [Accessed 06 05 2022].

[22] Tutorial's Point, “Machine Learning with Python - Methods,” [Online]. Available: https://www.tutorialspoint.com/machine_learning_with_python. [Accessed 20 02 2022].