

# Embedded Machine Learning

Identifying People Based on Movements in Table Football

**Group:**  
SW512e17

**Supervisor:**  
Giorgio Bacci



March 15, 2018



Department of Computer Science  
Aalborg University  
<http://cs.aau.dk>

## AALBORG UNIVERSITY

### STUDENT REPORT

**Title:**

Embedded Machine Learning

**Theme:**

Identifying People Based on Movements in Table Football

**Project Period:**

Fall 2017

**Group:**

SW512e17

**Participants:**

Morten Rask Andersen  
Anton Christensen  
Christian Mønsted Grünberg  
Steffan Riemann Hansen  
Mathias Ibsen  
Mathias Rohde Pihl

**Supervisor:**

Giorgio Bacci

**Pages:**

81

**Date of Completion:**

March 15, 2018

**Number of Copies:**

1

**Abstract:**

The goal of this project is to design and implement an embedded system for identifying people based on movements in table football. The embedded system should be able to train a neural network, and use it to make predictions based on movement data collected during a game of table football by sensors attached to the handles of the football table.

To collect data, we developed an embedded system consisting of batteries, a microcontroller, and a sensor, which are all attached to a football table. The sensor consists of an accelerometer and a gyroscope. From the raw sensor data an algorithm was devised for extracting movements and splitting each movement into features. To implement the model on an embedded system we created an architecture consisting of two data collection nodes, a relay node, and a DataHub front-end application. The result of this project is an embedded system that is able to collect movements from handles on a football table, use these movements to train an artificial neural network, and accurately identify who is playing. Training and labelling for players are configured on the DataHub website, where live classification results are displayed after the model is trained.

Morten R.A.

---

Morten Rask Andersen

Anton Christensen

---

Anton Christensen

Christian Grünberg

---

Christian Mønsted Grünberg

Steffan

---

Steffan Riemann Hansen

Mathias Ibsen

---

Mathias Ibsen

Mathias Rohde Pihl

---

Mathias Rohde Pihl

## Preface

This report has been written by the software group sw512e17 during the fifth semester at Aalborg University. The project has spanned from September to December 2017. The theme of the project is recognition and analysis of human movement using machine intelligence. The project has been supervised by Giorgio Bacci.

## Reading Guide

The illustrations and figures used throughout this report have been made by group sw512e17 unless otherwise stated. Excluded sections of code have been replaced with ellipsis ([. . .]). Citations will be made using the number notation where the number refers to a source in the bibliography.

Chapter 5 gives a brief introduction to the basics of machine learning and neural networks in particular, this chapter can be skipped, with the exception of section 5.2.4, if the reader already understands the basics of machine learning and neural networks. The entire code base for the final solution including code for analysing different machine learning models can be found at the project group's public repository by navigating to <https://github.com/AAUSoftwareStudentGroup/P5-Embedded>. The source code and additional files have been handed in together with this report and will be referenced within the report as the electronic appendix.

## Abbreviations

Listed below are abbreviations used in the project along with their meaning.

Abbreviation	Meaning
AI	Artificial Intelligence
ANN	Artificial Neural Network
ML	Machine Learning
NN	Neural Network
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SL	Supervised Learning
SVM	Support Vector Machine
USL	UnSupervised Learning
ERD	Entity Relation Diagram
API	Application Programming Interface
LSTMs	Long Short-Term Memory
Player	Person playing the game

Table 1: Abbreviations used throughout the report

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Problem Analysis</b>	<b>2</b>
2.1	Definition of Movement . . . . .	2
2.2	Table Football . . . . .	3
2.2.1	Movements in Table Football . . . . .	3
<b>3</b>	<b>Problem Statement</b>	<b>5</b>
3.1	Milestones . . . . .	5
<b>I</b>	<b>Milestone 1</b>	<b>6</b>
<b>4</b>	<b>Data Collection</b>	<b>7</b>
4.1	Choice of Sensor . . . . .	7
4.1.1	Translation . . . . .	7
4.1.2	Rotation . . . . .	9
4.2	Embedded Platform . . . . .	11
4.3	Hardware Design for Data Collection . . . . .	13
4.3.1	Mounting and Positioning of Device . . . . .	13
4.3.2	Board Layout . . . . .	14
4.3.3	Device Enclosure . . . . .	15
4.4	Implementing Data Collection Software . . . . .	16
4.5	Data Sampling Rate Limit . . . . .	17
4.6	Test of Milestone I . . . . .	19
<b>II</b>	<b>Milestone 2</b>	<b>23</b>
<b>5</b>	<b>Introduction To Machine Learning</b>	<b>24</b>
5.1	Supervised and Unsupervised Learning . . . . .	24
5.2	Machine Learning Models . . . . .	25
5.2.1	Support Vector Machines . . . . .	26
5.2.2	Neural Network . . . . .	26
5.2.3	Logistic Regression . . . . .	27
5.2.4	Selecting the Machine Learning Model Type . . . . .	27

5.3	Neural Networks . . . . .	28
5.3.1	Training a Neural Network . . . . .	31
5.4	Recurrent Neural Networks . . . . .	33
<b>6</b>	<b>Data Processing</b>	<b>35</b>
6.1	DataHub . . . . .	35
6.1.1	Architecture . . . . .	35
6.1.2	Data Modelling . . . . .	36
6.1.3	Uploading Data . . . . .	36
6.1.4	Visualising Data . . . . .	37
6.1.5	Labelling Data . . . . .	38
6.2	Grouping Data into Movements . . . . .	39
<b>7</b>	<b>Analysing Machine Learning Models</b>	<b>42</b>
7.1	Model Analysis . . . . .	42
7.1.1	Test Suite . . . . .	42
7.1.2	Artificial Neural Network Analysis . . . . .	45
7.1.3	Recurrent Neural Network Analysis . . . . .	50
7.2	Choice of Neural Network Model . . . . .	55
<b>8</b>	<b>Embedded Classification</b>	<b>57</b>
8.1	Overview of Milestone II . . . . .	57
8.2	Implementing Embedded Classification . . . . .	58
8.3	Test of Milestone II . . . . .	60
<b>III</b>	<b>Milestone 3</b>	<b>63</b>
<b>9</b>	<b>Embedded Learning</b>	<b>64</b>
9.1	Overview of Milestone III . . . . .	64
9.2	Memory Analysis . . . . .	65
9.3	Implementing Embedded Learning . . . . .	65
9.4	Interface for Embedded System . . . . .	67
9.5	Test of Milestone III . . . . .	68
<b>10</b>	<b>Discussion</b>	<b>73</b>
10.1	General Concerns . . . . .	73
10.2	Data Collection . . . . .	73
10.3	Data Processing . . . . .	74
10.4	Embedded Learning and Classification . . . . .	74

<b>11 Conclusion</b>	<b>76</b>
<b>Bibliography</b>	<b>78</b>

# List of Figures

2.1	4
4.1 Illustrates position of accelerometer directly on the handle.	8
4.2 Shows how to measure translation using a sliding potentiometer attached to the handle	8
4.3 Shows how to measure translation using linear encoding of the handle	9
4.4 The chosen position of the NodeMCU and MPU-6050. The x, y, and z direction of the MPU-6050 are illustrated by the arrows from the MPU-6050 where x is in the direction of translation of the handle	14
4.5 Schematic of the Data-collection device	15
4.6 Device enclosure	16
4.7 Flow diagram for node in milestone 1	17
4.8 Gyroscopic data shown for a single shot	18
4.9 frequency domain of the data	19
4.10 Pictures showing the setup for the test of milestone 1	20
4.11 Data from each box compared	21
4.12 Data from Logger Pro	21
4.13 Two points in figure 4.11 with data loss	22
5.1 Underfitting and overfitting	25
5.2 Illustration of how adding a new dimension may help dividing points	26
5.3 NN with two layers and bias nodes (dark green and dark blue nodes)	28
5.4 The step function	29
5.5 The Sigmoid function	30
5.6 The ReLU function	30
5.7 Sketch of the error as a function of a particular weight	31
5.8 ANN with errors E1 and E2 in the output layer	32
5.9 Illustration of a RNN	34
6.1 DataHub architecture	36

6.2	Entity relation diagram for the database component in the DataHub . . . . .	36
6.3	DataHub upload page . . . . .	37
6.4	DataHub visualisation page showing the data collected during a game of table football	38
6.5	DataHub label page . . . . .	38
7.1	Example of an ANN model created on the DataHub . . . . .	43
7.2	Example of an ANN model created with values . . . . .	43
7.3	Shows the three test for the analysis of the ML models . . . . .	44
7.4	Average accuracy across the three tests for 1-hidden-layered models 1-4, and 7 compared to 2-hidden-layered models 8-11 . . . . .	47
7.5	Average accuracy across the four tests with a varying number of hidden neurons . .	47
7.6	Accuracies from using test 1 (three training matches per player) and 3 on different models (one training match per player) . . . . .	48
7.7	Accuracies of test 1 (two players) and test 2 (three players) . . . . .	49
7.8	Average accuracies from tests using Sigmoid, Softmax, and ReLU . . . . .	49
7.9	Simple LSTM model . . . . .	50
7.10	A simplified illustration of the layers in the stacked LSTM model . . . . .	51
7.11	Training history for model 3 with windows size 10 on test 1 showing the models accuracy on training data and validation data for each training iteration (epoch) . .	53
7.12	Training history for model 1 on test 1 showing the models accuracy on training data and validation data for each training iteration (epoch) . . . . .	54
8.1	Overview of the system to be developed in the second milestone . . . . .	57
8.2	Flowchart of actions repeated in the main loop . . . . .	58
8.3	Flowchart of data collection and processing performed each 2ms . . . . .	59
9.1	Overview of the system to be developed in the third milestone . . . . .	64
9.2	Flow diagram for relay node . . . . .	66
9.3	Picture of the Monitor page of the DataHub . . . . .	68
9.4	Test setup milestone 3 . . . . .	69
9.5	ANN output for each movement performed by player 2 and detected by the sensor attached to the red team's handle in the second match of test 3. The two lines show the current average ANN output for player 1 and player 2 since the beginning of the match. Data is parsed from the relay log file which can be found in electronic appendix. . . . .	71
9.6	Heap size during test 1, test 2, and test 3 (see section 9.5) . . . . .	71



# Chapter 1

## Introduction

Machine learning (ML) has been applied to many different areas such as image analysis, recommender systems, and activity recognition. As such, ML has also been applied to identify people in areas such as mobile device authentication, where data gathered by biometric scanners has been used by various ML models. Examples of such applications include face and eye detection [1].

During our everyday lives, we carry out different tasks which require some sort of movement e.g. walking from point a to point b, handwriting, typing and so forth. As such the project group is interested in examining whether it is possible to use ML for recognising individuals based on their movement, as it could be applied to many areas within the task of identification.

In order to apply ML for identifying people based on movements, we will need to gather a lot of data about such movements. To keep it simple and in a controlled environment, where we know we can gather a lot of data, we have chosen to limit the problem to identifying people based on their movements in table football. The advantage of this includes the possibility of performing data gathering at the university, as there are available football tables, as well as making data gathering more fun, as it requires us to play table football.

The study curriculum states that the project solution, or at least part of it, has to be implemented on an embedded system. The task at hand is to combine the identification of people based on movements using ML and creating an embedded system.

# Chapter 2

## Problem Analysis

The purpose of this chapter is to come up with clear definitions of the different aspects of this project. First, movement is defined, also showing the correlation between different kinds of measurements, helping us make a choice of which kind of data we want to gather. This is followed by a description of table football, allowing us to get a better understanding of the problem domain.

### 2.1 Definition of Movement

According to the Collins Dictionary, movement “... involves changing position or going from one place to another.” [2]. As no human is able to move instantaneously, we define movement as a change of position over a given time period. Thereby, we have time and position as information. A movement of an object can be described as a function  $p : \mathbb{R} \rightarrow \mathbb{R}^3$  expressed as

$$p(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}$$

Where  $x$ ,  $y$ , and  $z$  are functions describing the x, y, and z coordinate of the object at time  $t$ . To describe the object’s change in position we can determine the velocity of the object, which based on  $p$ , can be expressed as a function  $v : \mathbb{R} \rightarrow \mathbb{R}^3$  as follows

$$v(t) = \begin{bmatrix} v_x(t) \\ v_y(t) \\ v_z(t) \end{bmatrix} = p'(t) = \begin{bmatrix} x'(t) \\ y'(t) \\ z'(t) \end{bmatrix}$$

In order to obtain velocity from a standstill, an object must accelerate. We can describe the change of an object’s velocity as the acceleration of the object given by a function  $a : \mathbb{R} \rightarrow \mathbb{R}^3$  as follows

$$a(t) = \begin{bmatrix} a_x(t) \\ a_y(t) \\ a_z(t) \end{bmatrix} = v'(t) = \begin{bmatrix} v'_x(t) \\ v'_y(t) \\ v'_z(t) \end{bmatrix}$$

The correlation between the three properties of movement makes it possible to describe both the velocity and acceleration based on the position as:

$$v(t) = p'(t) \tag{2.1}$$

$$a(t) = v'(t) = p''(t) \tag{2.2}$$

Without additional information we cannot necessarily derive the exact  $p(t)$  based on either  $v(t)$  or  $a(t)$  because this depends on whether there is a unique solution for  $p$  to the differential equations 2.1 and 2.2.

An example is, if we measure the velocity, we cannot obtain the information about the starting position, as it is only a measure of the changes in the position over a given time period. The same goes for acceleration. Given acceleration, we cannot obtain knowledge about the starting velocity. Therefore, it would be preferable to measure position over time, and given this, calculate (if necessary) velocity and acceleration. On the other hand, if we measure e.g. acceleration, we need to assume that additional information such as starting position and velocity is not required to identify who performed the movement.

Another issue is that movements with constant speed cannot be measured based on only acceleration. However, performing a movement with constant speed in a domain such as a table football is unlikely.

## 2.2 Table Football

Table football is defined as “*a game based on soccer [football], played on a table with sets of miniature human figures mounted on rods [handles] allowing them to be tilted or spun to strike the ball . . .*” [3].

A typical game of table football is played by 2-4 players, and is finished once one of the teams reaches the score limit. For competitive table football, specific rules have been defined, as can be seen in [4]. For hobby play, the rules vary, but often include rules such as no spinning of the handle is allowed, meaning turning the handle more than 360 degrees in quick succession. Limiting a certain kind of movement such as spinning allows us to refrain from having to recognise this kind of movement.

### 2.2.1 Movements in Table Football

As previously mentioned; when playing table football, the players are manipulating the game through movements of table football players attached to the handles. There are two kinds of movement through the handles in table football: translation and rotation. This is illustrated in figure 2.1. Combining these movements makes each player able to hit the ball once it is in reach of a table football player or position a table football player to block an incoming shot from an opponent.

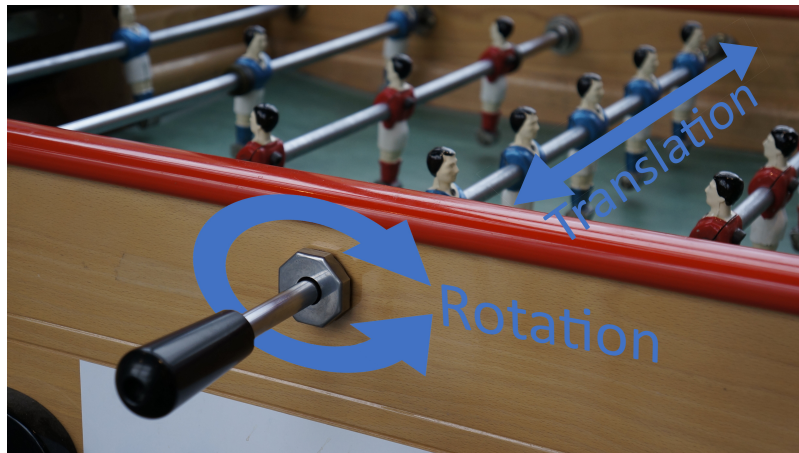


Figure 2.1

# Chapter 3

## Problem Statement

Based on the definitions of human movement and table football in combination with the specified requirements for the semester, as stated in the study curriculum [5], a clear definition of the problem this project aims to solve, can now be presented.

**Problem statement** *How can an embedded system be implemented to identify individuals based on the movement of handles in a table football game using machine learning?*

To help answer this problem, the following four research questions are established:

- *How can data about movements of handles in a table football game be collected?*
- *How can data be processed to use it for training machine learning models to identify people playing table football?*
- *Which machine learning models can be used to identify players?*
- *How can such a machine learning model be implemented on an embedded system?*

### 3.1 Milestones

Based on the problem statement, we define the following requirements as three milestones for the embedded system to be developed in this project. The three milestones are concerned with data collection, embedded classification, and embedded training respectively. The list below is a more detailed description of each of the three milestones.

1. The embedded system must be able to collect data about a player's movements of handles in table football
2. The embedded system must be able to classify which of two players is playing based on a pre-trained model
3. The embedded system must be able to train a model and use this model to classify players

The report is composed of three parts, each concerning their respective milestone.

**Part I**

**Milestone 1**

# Chapter 4

## Data Collection

As described in section 3.1 the first milestone is to measure a player's movements of handles. To be able to do this, we need a sensor to measure the translation and rotation of a handle which is described in section 4.1. The rest of the chapter considers the choice of embedded platform, by comparing different alternatives. Hereafter, we will choose a design for the hardware by considering e.g. size and position of the mounting system used to secure the sensors to the table. The software needed will then be described, followed by a consideration of how fast to sample data. The last section of the chapter will test milestone 1.

### 4.1 Choice of Sensor

In this section, we present different sensors to measure translation or rotation and select which of these we will use to collect data.

#### 4.1.1 Translation

First, sensors for translation are considered.

##### **Accelerometer**

Some accelerometers, such as the MPU-6050, measures acceleration in three orthogonal directions  $x$ ,  $y$ , and  $z$  [6, p. 25]. Based on these accelerations it is possible to construct a three-dimensional acceleration vector that describes the acceleration direction relative to the orientation of the accelerometer, as well as the total acceleration, as the length of the vector.

To measure the acceleration when translating a handle, a possible solution will be to place an accelerometer directly on the handle as shown in figure 4.1.

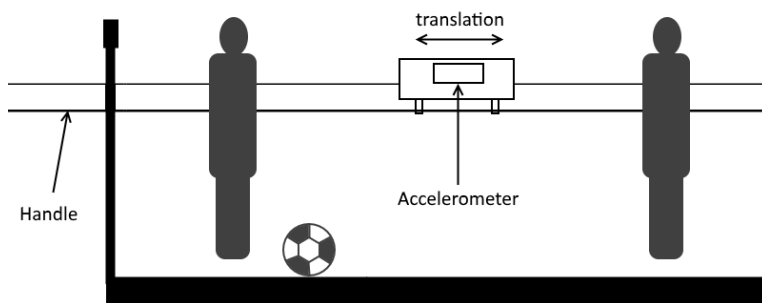


Figure 4.1: Illustrates position of accelerometer directly on the handle.

The cost of a MPU-6050 accelerometer is approximately 1.18 USD [7] which also includes a three-axis gyroscope [6] and is therefore capable of measuring the rotation of the handle as well.

### Sliding Potentiometer

The sliding potentiometer is an electronic component which has a variable resistance controlled through the position of the slider. This component can be used to measure the position of the handle during translation as shown in figure 4.2.

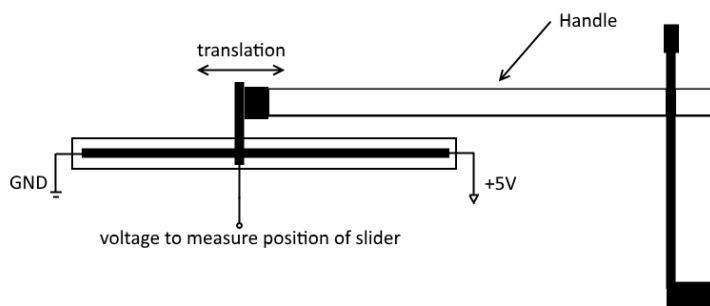


Figure 4.2: Shows how to measure translation using a sliding potentiometer attached to the handle

There are different sizes of sliding potentiometers with different prices. A price example is 3.87 USD for five pieces of 75mm sliding potentiometers [8]. It is difficult to find sliding potentiometers with a travel distance above 100mm. This is a problem because the handles for the table football used in this project can move between a maximum of 125mm to 350mm, depending on which table we are using.

### Linear Encoder

A linear encoder is a digital device used to measure linear movements [9]. There are different types of linear encoders using either absolute or incremental positioning feedback and optical or magnetic encoding [9]. Figure 4.3 shows a way to integrate a linear encoder using incremental feedback and optical encoding.



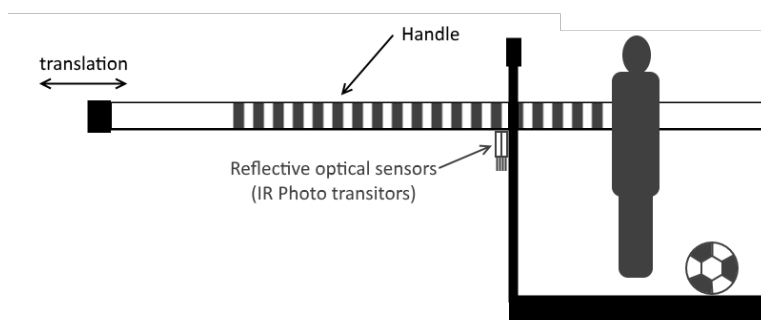


Figure 4.3: Shows how to measure translation using linear encoding of the handle

The linear encoder in figure 4.3 detects movements using reflective optical sensors made of an infrared (IR) diode emitting IR light detected by an IR phototransistor. The phototransistor is activated when the light is reflected by the white stripes on the handle. The phototransistor is deactivated when the light is absorbed by the black stripes on the handle. With this technique, we can measure a change in the position of the handle. To detect which way the handle is moving, two sensors are placed with an offset corresponding to the width of half a stripe, so when it starts moving, one of these sensors will activate/deactivate before the other and the direction is known. The cost of reflective optical sensors is approximately 0.99 USD for ten pieces of the model TCRT5000 each consisting of an IR diode and an IR phototransistor [10].

### 4.1.2 Rotation

We will now consider three different sensors for measuring rotational movements.

#### Gyroscope

A gyroscope detects rotational velocity. A rotational velocity is a change in orientation over time. This can be expressed in units of radians per second. To measure rotational velocity we can use the MPU-6050 which also includes an accelerometer [6].

#### Rotary Potentiometer

A rotary potentiometer, like the sliding potentiometer, is an analogue device with a resistance depending on the position of the knob. A price example for five pieces of rotary potentiometers is 1.18 USD for five pieces [11]. This method can be used if the base of the potentiometer is fixed, but on a moving handle, it may be difficult to attach the rotary potentiometer on the handles of the football table we are using.

#### Rotary encoder

The rotary encoder follows the same technique as the linear encoder. To measure rotation instead of translation we can colour the black and white stripes along the handle instead of around the handle as shown in figure 4.3 and therefore use the same sensor-model. We can use a rotary encoder to measure changes in rotary position when the IR phototransistor is triggered by the black and white stripes. As the TCRT5000 is also able to measure change in rotational position, the price example is the same as for the linear encoder.

## Sensor Discussion and Choice

Based on the description of the six sensors, we will now discuss which sensors will be used in this project. Table 4.1 summarises the characteristics for the different sensors.

Sensor Type	Movement	Data	Price example / component
Accelerometer	Translation	acceleration	1.18 USD / MPU6050
Sliding potentiometer	Translation	position	0.75 USD / 75mm sliding
Linear encoder	Translation	$\Delta$ position	0.1 USD / TCRT5000
Gyroscope	Rotation	Rotary velocity	1.18 USD / MPU6050
Rotary potentiometer	Rotation	Rotary position	0.24 USD / Rotary potentiometer
Rotary encoder	Rotation	$\Delta$ Rotary position	0.1 USD / TCRT5000

Table 4.1: Summary of the different sensors' characteristics where  $\Delta$  signifies change in position instead of absolute position

We need to measure both the translation and rotational movements. One approach would be to use a combination of multiple sensors to measure movements. The advantage of this is that it will add redundancy of the measured data that can be used to compare data from different sensors to increase reliability and accuracy of the measurements. The disadvantage of using multiple sensors is that we will have to spend a lot of time mounting the different sensors on the football table. Because we do not own the football table used in this project we need to mount and remove all sensors each time we are testing. This makes the use of multiple different sensors impracticable.

The advantage of using the sliding and rotary potentiometers is that they provide position data which is more preferable than velocity or acceleration data as explained in section 2.1. The disadvantage of using potentiometers is that because they measure position they need a reference point i.e. the base or the knob of the potentiometer must be at a fixed position while the other part of the potentiometer is moved by the handle. This makes it difficult to mount on the football table without making modifications to the table.

The strip for the linear and rotary encoder could be constructed and mounted on the handle using paper and Duct tape. The disadvantage of the linear and rotary encoders is that the resolution of the encoder is limited by the size of the black and white stripes which depends on the size of the optical sensor detecting the stripes. The width of the optical sensor TCRT5000 is 5.8mm [12] which will result in a low resolution of the data from measuring the rotation because the circumference of the handle is only 44.6 mm, which gives us a resolution of 7.58 points per rotation.

An advantage of using the MPU-6050 is that it can both measure translation and rotation of movements using its internal accelerometer and gyroscope. Another advantage of the MPU-6050 compared to the other sensors is that it is less mechanical as it does not have any moving parts. Because the MPU-6050 measures acceleration and rotational velocity, it does not require a reference point like the other sensors, which enables the entire sensor to be mounted directly on the moving handle. The disadvantage of the MPU-6050 is that we cannot determine the exact position of the handle, since we are only measuring acceleration and rotational velocity as discussed in section 2.1.

We choose the MPU-6050 because it can be used to measure both translation and rotation without any mechanical parts. Another advantage is that mounting it to the table is possible without making modifications to the table itself. By making said choice, we assume that it is possible to identify people based on translational acceleration and rotational velocity.

## 4.2 Embedded Platform

In this section we will choose which embedded platform we will use for data collection. Before we can do this we need to consider what is required of the embedded platform. From the three milestones in section 3.1 and the choice of sensor in section 4.1, we have the following requirements for the embedded platform: The platform . . .

1. must be able to collect data about handle movements, which includes:
  - (a) Reading data from a MPU-6050
  - (b) Transferring data to a PC
2. must be able to evaluate a ML model to classify between two players
3. must be able to train a ML model to classify between two players

The second and third requirement depend on which ML model we choose. Initially, we will have to solve the first requirement in order to collect data that we can use to analyse different ML models on a more computationally powerful system like a PC. We will then update the embedded platform to be able to execute and train the model we choose. In section 4.1, we chose the MPU-6050 to collect movement data. The MPU-6050 sends its readings via the I<sup>2</sup>C protocol [6]. From this we know that the embedded platform should be I<sup>2</sup>C compatible. Another requirement for the first milestone is that the embedded system must be able to send the data to a computer for further analysis. Based on requirement 1a and 1b we have selected three embedded platforms to choose between: Arduino, NodeMCU, and Raspberry Pi.

**Arduino (ATmega328P)** The first embedded platform we will consider is Arduino which is described as: “. . . an open-source electronics platform based on easy-to-use hardware and software.” [13]. There exists several different Arduino development boards [14]. In this section we will consider the Arduino Nano board [15]. We consider this board because we already have several of these available but also because they are small, meaning they will be less in the way when attached to the football table. We have experience with the Arduino platform from a previous project where we created a new programming language for Arduino called *Tang* [16].

The Arduino Nano has an ATmega328P microcontroller with a clock speed of 16MHz enabling up to 16 million instructions per second (MIPS) [15]. It has 2kB of SRAM, and 32kB of flash memory for storing the compiled program [15]. The dimensions of the Arduino Nano are 18mm x 45mm and it weighs 7 grams [15]. The Arduino Nano microcontroller has an I<sup>2</sup>C compatible interface which we can use to communicate with the MPU-6050. The Arduino Nano also features a serial USB connection [17] which we can use to send the collected data to a PC.

**NodeMCU (ESP8266)** The second embedded platform we will consider for the data collection is NodeMCU which is described as: “An open-source firmware and development kit that helps you to prototype your IOT product within a few Lua script lines” [18]. We consider this embedded platform because we already have one NodeMCU devkit 1.0 (from now referred to as NodeMCU) board available and it is possible to program it through the Arduino environment which we also have experience with.

The ESP8266 used in the NodeMCU has built-in support for WiFi and a clock speed of up to 160MHz ( $\approx 20\%$  throughput used by WiFi stack) [19]. The NodeMCU has 4MB of flash where the compiled program is stored [20, 21]. The unused flash space can be used for non-volatile storage. The dimensions of the board are 25mm x 48mm [21]. The NodeMCU microcontroller has an I<sup>2</sup>C

compatible interface [19] which we can use to read data from the MPU-6050. To send data to a PC we can use the built-in WiFi.

**Raspberry Pi (ARM-processor)** The third embedded platform we will consider is the Raspberry Pi, which is described as: “*A small and affordable computer that you can use to learn programming*” [22]. There are different Raspberry Pi boards. We will consider the Raspberry Pi Zero as it is small in size with dimensions of 30mm x 65mm [23]. There are two variants of the Raspberry Pi Zero; one with WiFi and Bluetooth and one without.

We will consider the variant with WiFi and Bluetooth support called Raspberry Pi Zero W. The Raspberry Pi Zero W has a 1GHz ARM processor and 512MB of RAM [24] and is capable of running a full GNU/Linux-based operating system or a limited version of Windows [25].

## Embedded Platform Discussion and Choice

The Arduino Nano has no built-in wireless capabilities and therefore must be connected to our computer using a USB-cable or combined with a wireless module of some kind, which adds complexity and costs to the project. This is impractical for our purposes and as such we chose not to use Arduino Nano for this project.

As seen in table 4.3, both the Raspberry Pi and the NodeMCU have a higher throughput than the Arduino Nano and have built-in WiFi, with the Raspberry Pi also supporting Bluetooth. If we were to choose the Raspberry Pi, we would be able to use high-level languages like Python and we could worry less about problems regarding speed and memory consumption. The NodeMCU on the other hand is cheaper than the Raspberry Pi, physically smaller and consumes less power as seen in table 4.2, but we may have to come up with a more creative solution in order for the product to work under the restrictions presented by its limited resources. This would however lead to learning more about working under said restrictions, which is of interest to the group. Considering these arguments, we choose the NodeMCU.

	Estimated price	Worst-case power consumption	Parameters for worst-case
<b>Arduino Nano(ATMega328P)</b>	2.38 USD [26]	21.12mA@5V, 9.46mA@3.3V [27]	20MHz, Full swing crystal oscillator
<b>NodeMCU (ESP8266)</b>	3.26 USD [28]	320mA@3.3V [29]	Peak current at startup
<b>Raspberry Pi Zero W</b>	14.14 USD [30]	230mA@5V [31]	Shooting 1080p video

Table 4.2: Worst case power consumption and price ranges for the processors

	Clock speed	RAM size	Non volatile storage	Width x Length
<b>Arduino Nano</b>	16 Mhz	2 KB	1 KB EEPROM + 32 KB Flash	18mm x 45mm
<b>NodeMCU</b>	160 Mhz	96 KB	4 MB (QSPI Flash chip)	25mm x 48mm
<b>Raspberry Pi Zero W</b>	1 Ghz	512 MB	Plenty (SD CARD)	30mm x 65mm

Table 4.3: Comparison of specifications for the different embedded platforms

## 4.3 Hardware Design for Data Collection

Based on our choice of sensor and embedded platform, we will now design how these can be combined into a movement-measuring device.

### 4.3.1 Mounting and Positioning of Device

In this section we will discuss how the different parts of the device will be connected and mounted on the football table. We will prefer a solution that is easy to mount as we will have to reattach the device each time we perform tests. We will first discuss the position of the sensor.

**Sensor Position** The placement of the sensor is important, since the data will change drastically depending on the position relative to the movement. The sensor needs to measure the rotation and translation of the handle and therefore needs to move with it, to measure these movements.

**Inside Game** The sensor will be placed on the handlebar inside the playing area.

The cons of this placement are that it might obstruct the vision of the playing area and depending on the rest of the device it might also interfere with the ball. The box might become damaged as a result of being hit by the ball.

**Outside game** The sensor will be mounted on the end of the handlebar, on the grip, which is not inside the playing area.

The does not interfere with the actual game, but limits the size of the device significantly since it must not get in the way of the player. This might also cause problems with mounting the device securely enough, to prevent it from falling of the bar in the middle of a game.

Since the electronics are placed outside the game, they will not be at risk of damage from the ball.

**Choice** With these two options, we choose to place the sensor inside the game since we do not suspect it will interfere greatly with the game compared to placing the sensors outside the game on the handles where it might take up space otherwise used by players. As the box is made of high impact polystyrene (HIPS), and since the box is placed above the board, it will rarely be hit by the ball and will not be damaged, if it is hit.

**NodeMCU and Battery Position** The NodeMCU can also be mounted in multiple different ways. It is important here to consider that the NodeMCU will need a wired connection to the MPU-6050 as well as a power source.

**Complete Device on Bar** The complete device with NodeMCU, batteries, and the sensor will be mounted on the bar and rotate with it. This might cause the bar to become unbalanced and interfere with the game, but it will be easy to mount.

**Only MPU-6050 on Bar** Only the MPU-6050 will be mounted on the bar and the rest of the device will be mounted elsewhere on the table and connected with wires to the sensor.

This will only put minimal weight on the bar, but the wires connecting the sensor and microcontroller might easily get tangled. This approach will also limit rotational movement as the bar cannot be rotated more than once or twice before it needs to rotate the other way to untangle the wires.

**Choice** We choose to put the complete device on the bar, since we think that its size and weight will interfere less with the game than having wires and needing to untangle these during a game.

Figure 4.4 illustrates the chosen position of the NodeMCU and the MPU-6050.

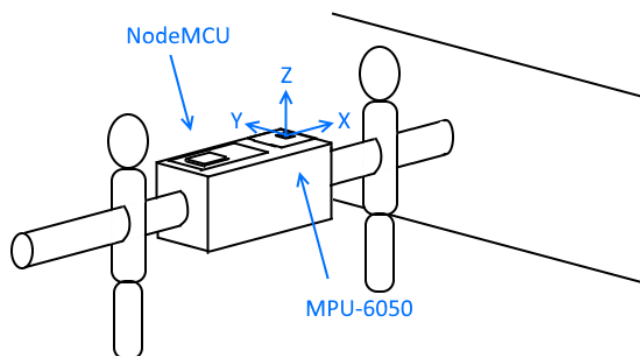


Figure 4.4: The chosen position of the NodeMCU and MPU-6050. The x, y, and z direction of the MPU-6050 are illustrated by the arrows from the MPU-6050 where x is in the direction of translation of the handle

### 4.3.2 Board Layout

The components need to be connected in a way so they will not disconnect when playing.

**Components** The components needed are as follows.

- On/Off switch
- Status LED
- Batteries
- MPU-6050
- NodeMCU

We need some way of turning the device on and off without unmounting the mechanism from the handlebar and a way to easily restart the device while mounted during tests.

We add a status LED which will be used to indicate different states of the system.

We need to power the devices. We have chosen to use type 18650 batteries because they are rechargeable and the ones we already own have a capacity of 1000mAh per cell. We have measured the current used by our ESP8266, connected to WiFi and an MPU-6050, as 85mAh at 8.01V (two fully charged type 18650 cells in series), which means that two cells can supply enough power for  $1000/85 \approx 12$  hours of data collection.

We will of course also need the sensor (MPU-6050) and the microcontroller (NodeMCU) which will send the data to a receiver.

**Schematics** The sensor needs 3.3V power, ground, and a pin pulled either high or low (this indicates the I<sup>2</sup>C address of the sensor) as well as an I<sup>2</sup>C data- and clock-line.

The status LED is connected to pin D0 of the NodeMCU and needs a current-limiting resistor. The NodeMCU is connected to the batteries in series with the on/off switch.

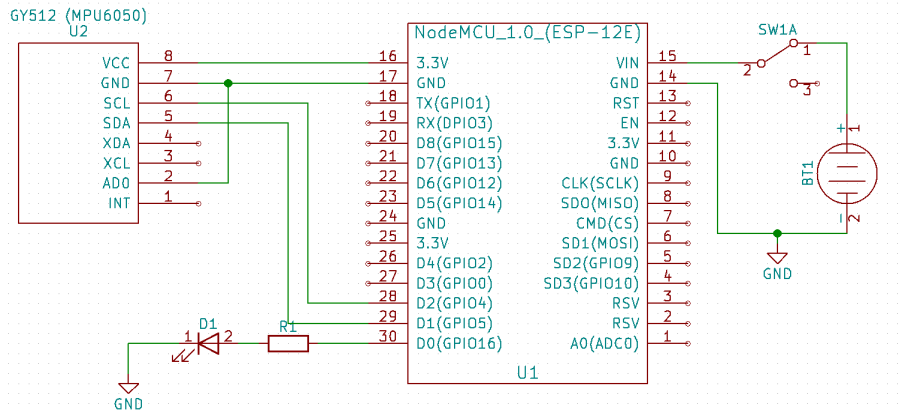


Figure 4.5: Schematic of the Data-collection device

**Board Type** We have three board types we can use to create this circuit:

- Solderless breadboard
- Perfboard
- Custom PCB

The breadboard works by holding the wires by friction. Since the board will experience rapid movements, we do not think this will be sturdy enough. The other two options will involve soldering the components together and will work about equally well in our case, so we choose the perfboard, as it is the cheaper and simpler option for very small quantities.

### 4.3.3 Device Enclosure

The enclosure needs to reliably hold the sensor in the same position relative to the bar every time it is attached, but still be disassemblable, so the sensor can be swapped out and the batteries recharged. The enclosure also needs to house the other components in a secure way, so they will not be damaged by the ball.

The enclosure also needs to be mounted in a way that it will not rotate or move on the bar when playing.

Considering this, we want to build some kind of box capable of being mounted sturdy on a cylindrical metal bar and protecting the components, while not being so big that it can hit the ball as the ball passes beneath it on the table during a game.

**Final Design** The designed box is shown in figure 4.6. The electronic components are placed in one side of the box, while the batteries and on/off switch are placed in the other to keep the balance. The box is mounted with threaded rods passing through each corner, and M4 nuts are tightened at each end, squeezing the box onto the bar and holding it tight.

This proved to be an effective way of mounting the box.

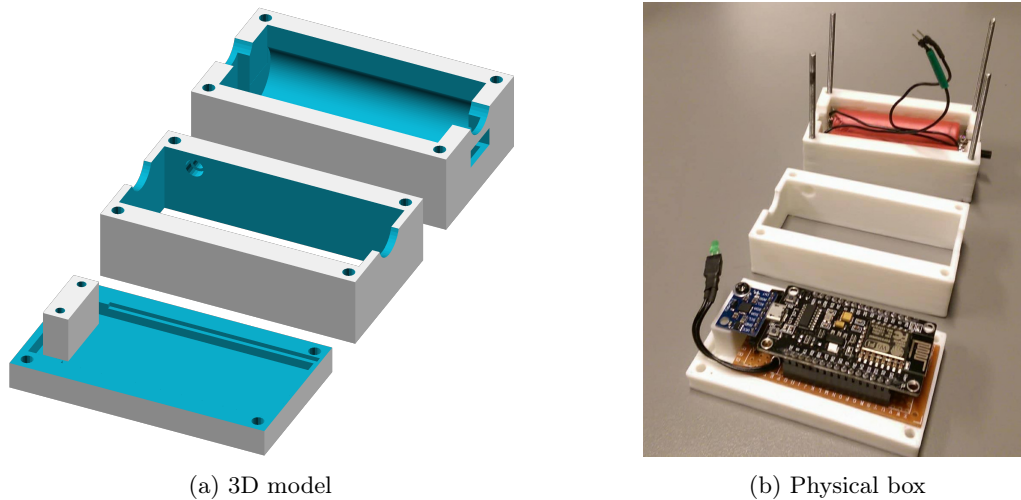


Figure 4.6: Device enclosure

## 4.4 Implementing Data Collection Software

Figure 4.7 shows a flow diagram of the firmware for the data collection node implemented for reaching Milestone I. Initially, the *setup* method is run where the MPU-6050 is configured. The *loop* is then entered where we first check if we have received a package. If a package has been received from a PC we save the IP of the PC so that we later can send information back to the PC. A check is then made to see if it is more than 2ms since the ESP8266 last read data from the MCU and if more than 2ms has elapsed, data is read and sent to the PC's IP over UDP. The loop is then repeated ad infinitum.



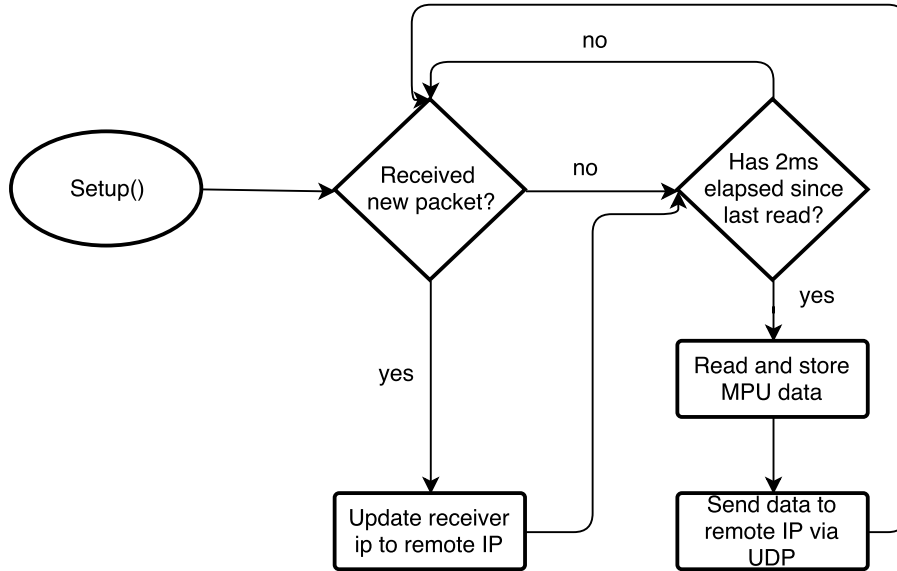


Figure 4.7: Flow diagram for node in milestone 1

When choosing which communication protocol to use for sending data to a computer over WiFi, we consider the two primary options: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). The two protocols are different in several ways. TCP is more reliable but with slower speed as a trade-off in comparison with UDP. By performing a handshake, TCP enables two devices to ensure communication between them, allowing them to check that the data being sent is received in the right order and with intact data [32], but is slow compared to UDP. Even though UDP is unreliable, it still checks for errors in the sense that it has a checksum that needs to match the package and if it doesn't match, it's dropped, as opposed to TCP which would ask the device to re-send the incorrect package [32].

Initially, TCP was used as the communication protocol for this project, but it was realised that in order to collect data to represent human movement, we needed a sampling rate too fast for TCP to handle, as described in section 4.5.

As UDP allows us to send packages rapidly, this means that even if data is lost, we will still be gathering more data because of the higher sampling rate.

## 4.5 Data Sampling Rate Limit

To achieve the best result possible, when the data collected is used for prediction, the maximum amount of information should be preserved. Any information lost from the collected data, could result in the loss of an important feature, which may mean a less precise prediction. We will only consider shots as we in chapter 6 will process the data collected from the chosen sensors to extract features for ML models.

The obvious way to preserve all the necessary information, is to gather data with the highest fidelity possible, given the chosen hardware, and use all the information as input to the ML model. The problem with this approach is, that, as stated in the problem statement, the ML model will eventually run on an embedded system. This means that there are limited resources to both store and run the ML model. For this reason, this section will analyse the data collected to determine

the frequency of the relevant data when playing table football, and thereby give an estimate for the necessary frequency of data collection, to preserve the relevant information.

The first step in determining an estimate of the minimum frequency for data collection is analysing the frequencies of the relevant data. To do this, a preliminary test will be conducted, to analyse the data. The recorded gyroscopic data of a single shot can be seen plotted below:

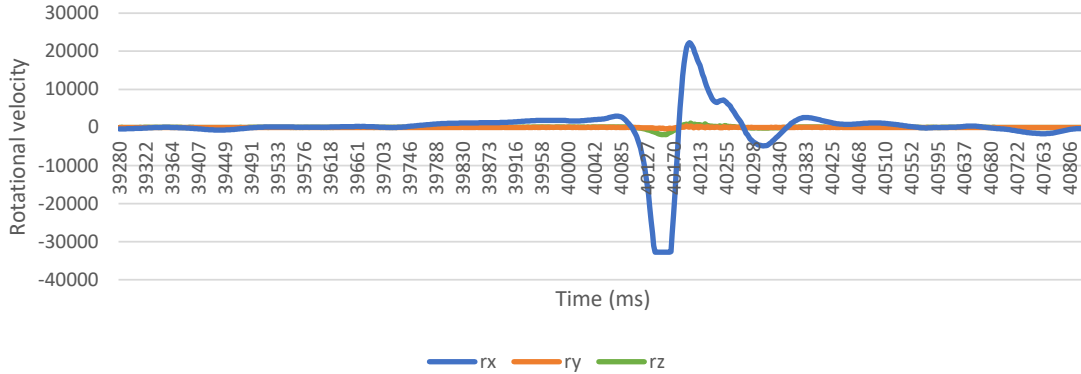


Figure 4.8: Gyroscopic data shown for a single shot

The data is collected with an average frequency of  $500Hz$ . This means that data points are collected approximately every  $2ms$  (see section 4.4). The bottleneck for the frequency of data collection is the speed of communication between the sensor and the embedded processor. A sample rate of at least  $2000Hz$ , or every  $0.5ms$  can be achieved, but would make it impractical to wirelessly transfer the data, since it would take up most of the processing power.

The graph in figure 4.8 shows the data collected for a single shot, but for the actual frequency analysis, all shots in the dataset are used.

The shot seen in figure 4.8 creates large fluctuations in the rotation around the x axis, which is what represents the relevant data for a single shot. Each shot is extracted, and appended to an array. The relevant data has a range of frequencies it appears within, which can be helpful to determine, at what frequency the sensors need to collect data. As the relevant data in the shots reach higher values than the average values before and after the shot, the frequency of the shot will reach higher amplitude as well. This can be used to determine the frequency of the relevant data contained in a shot. To plot the frequency domain, the discrete-time Fourier transform of the dataset needs to be calculated. A mathematical library for Python was used, called NumPy [33]. The resulting graph showing the shot in the frequency domain can be seen in figure 4.9.

What is seen in figure 4.9, is the discrete Fourier transform of the 47 shots identified in the dataset. To successfully perform the Fourier transform on a dataset, the data has to be equally spaced in terms of time. Since the interval varies for the collected data, linear interpolation is used to correct this. As can be seen in figure 4.8, the only significant axis of rotational velocity is the x-axis. For this reason, this is the only data selected for the Fourier transform.

The frequencies observed in figure 4.9 shows that the majority of information is contained in frequencies ranging from  $0Hz$  to  $20Hz$ . This indicates, that the large fluctuations visible on the time domain graph 4.8, is ranging in the same frequency range. The remaining frequencies observed, is considered noise, and is not taken into consideration, when choosing the lower bound of data collection sampling rate.

To calculate the lower limit of the sampling rate to get a valid output, we can make use of the

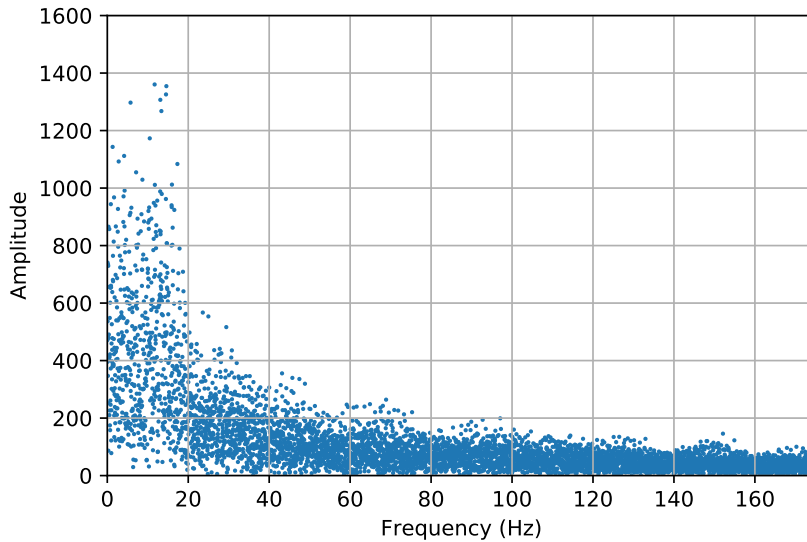


Figure 4.9: frequency domain of the data

Nyquist-Shannon sampling theorem [34]. The Nyquist-Shannon sampling theorem presents a sampling rate that is acceptable in regards to capturing the necessary samples without losing important information. According to [34], the theorem may be stated as “*The sampling frequency should be at least twice the highest frequency contained in the signal. Or in mathematical terms:  $f_s \geq 2 \cdot f_c$  where  $f_s$  is the sampling frequency . . . and  $f_c$  is the highest frequency contained in the signal.*”.

However, according to [35][p. 13], the sampling rate may be a lot higher than double the highest frequency contained in the signal: “*It is not unreasonable to sample the data at 5 times the Nyquist limit to ensure the integrity of the data in both the frequency and time domains.*”. This depends on multiple factors, especially the constraints of disk space.

Based on the Nyquist-Shannon sampling theorem [34], it can be concluded that a lower limit for the sampling rate should be twice that of the frequency of the relevant data.

$$f_s = 2 \cdot 20Hz = 40Hz \quad (4.1)$$

The lower limit to the sampling rate of  $40Hz$  is equivalent to a maximum spacing of the data points of  $25ms$ . This means that our frequency of  $500Hz$  leaves us at more than ten times the Nyquist-Shannon, which ensures the integrity of the collected data.

## 4.6 Test of Milestone I

**Purpose** In the first milestone our goal is to make an embedded system that is able to collect data from the movement of players in a table football game, as specified in section 3.1. The purpose of this test is therefore to validate the embedded system created for this milestone and check if we are able to collect data from the chosen sensors and that the collected data conforms to reality.

**Test Setup** The test setup can be seen in figure 4.10. In figure 4.10a, we see two computers set up next to the football table, with the purpose of receiving and saving the sensor data sent from the microcontrollers in the boxes. Figure 4.10b shows the two boxes (nodes) attached to the same handle of the table. The boxes are identical in regards to measurements and components and are both facing the same direction.



(a) Picture showing the football table setup with the computers receiving sensor data (b) Picture showing how two boxes are attached to a single handle to verify that the data we receive is consistent

Figure 4.10: Pictures showing the setup for the test of milestone 1

**Test Procedure** Start filming two nodes and then turn on the two nodes simultaneously to synchronise the sensor data with the film. Turn on the data collection script from two different computers at the same time (one for each node). Attach the camera to a stationary position to film the boxes. Now perform the following movements to test the sensors:

1. Four slow translations
2. Four quick translations

Stop the data collection scripts and the video recording.

**Test Result** To compare the results of the sensors with respect to each other and the recorded movie we plotted the sensor data and loaded the video into LoggerPro [36]. The comparison of sensor A (orange) and B (blue) is shown in figure 4.11 while the result of analysing the video is shown in figure 4.12.

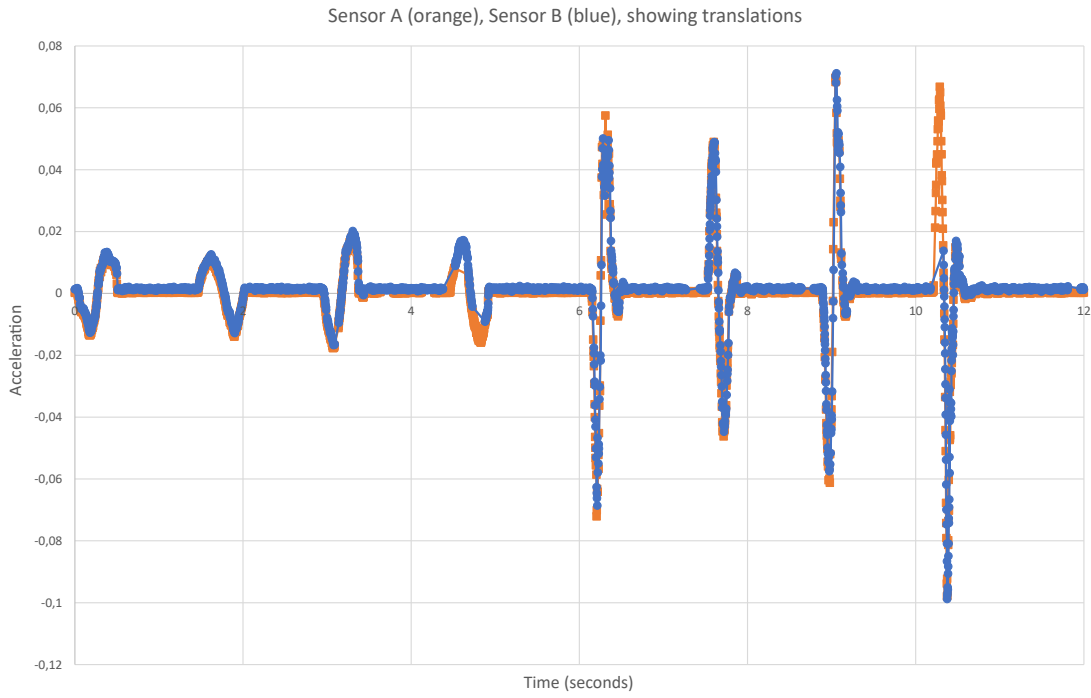


Figure 4.11: Data from each box compared

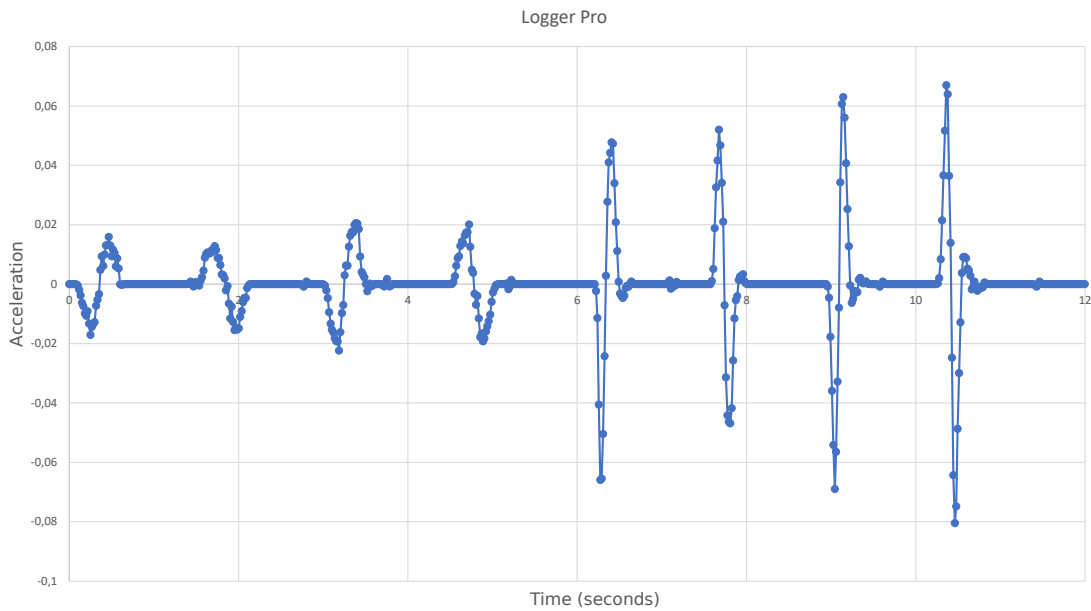
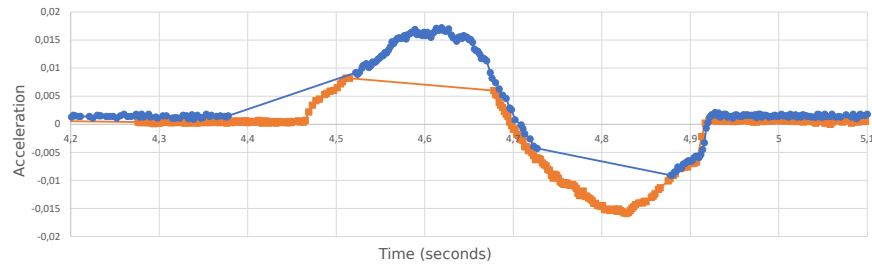
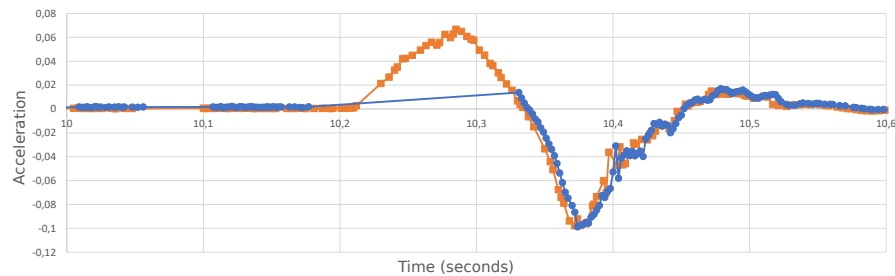


Figure 4.12: Data from Logger Pro



(a) Data from each box compared for the time period from 4.2 seconds to 5.1 seconds



(b) Data from each box compared for the time period from 10 seconds to 10.6 seconds

Figure 4.13: Two points in figure 4.11 with data loss

**Test Discussion** As seen in figures 4.11 and in 4.12, the graphs seem alike with minor differences. However, looking closely at the movements between 4-5 seconds and at 10-11 seconds, major inequalities between the two graphs are seen. These differences are enlarged in figure 4.13a and 4.13b respectively. A straight line instead of a thick line (composed of several dots indicating a measurement has been made) implies a period of time where no packages have been received. The reason for this data loss could be our choice of communication protocol. UDP does not confirm that a package has been received, which means that after sending a package, interference on the network or connection issues may result in the computer not receiving the package.

As UDP is quicker than TCP, we either have to accept a slower sampling rate to ensure less unexpected data loss or accept loss of data. Another solution, based on the assumption that the loop will run slower the more data it has to process, we could try to minimise the amount of data. This could e.g. be done by refraining from using ASCII encoded data, as this requires more bytes to contain the same information as a binary encoding.

We have now reached the first milestone as we have developed an embedded system to collect data about movements of the handles.

Part II

Milestone 2

## Chapter 5

# Introduction To Machine Learning

In order to accomplish the milestone: “*The embedded system must be able to classify which of two players is playing based on a pre-trained model*”, we first need to select and pre-train a model which we will integrate into the embedded system. Before we do this, we will give a brief introduction to ML and select which type of ML we will use in this project.

### 5.1 Supervised and Unsupervised Learning

ML is considered to be a concept within artificial intelligence (AI) and is concerned with machines that are able to learn by themselves based on e.g. past experience and observations [37, 38]. ML techniques can be divided into two subcategories of ML: supervised learning (SL) and unsupervised learning (USL) [38].

**Supervised Learning** In SL, data is generally split into a set of features. SL then works by first using a training set to fit a model. The training set contains an arbitrary number of objects and for each of these objects, a result. Based on the training set, SL can use different algorithms such as support vector machines (SVM) or k-Nearest-Neighbour with the goal of e.g. establishing a decision boundary which makes the machine able to determine the result of new input data. An example of SL could be a classification in which objects are grouped into classes based on some features.

Consider an example where we have to determine whether a fruit is an apple or an orange. In this case, data would be collected for both types of fruits and depending on the data collected, possible features of the fruits could be shape and colour. Each combination of data is then classified according to whether they are associated to an apple or an orange.

According to [38], SL algorithms are often fast and precise but they require proper training and validation. Furthermore, SL algorithms must be able to generalise new data that can accurately predict previously unseen data.

A model that is unable to generalise has been overfit or underfit during training. Underfitting is when the model is not able to determine the proper relation between the features and the output labels and thus will not generalise new data [39, 40].

Overfitting, on the other hand, may occur if the model has been trained in such a way that the prediction function accommodates for noise by fitting the function on the test set such that all training data are labelled correctly. Examples of overfitting and underfitting are illustrated in



figure 5.1b and figure 5.1a, respectively where the blue and orange colours denote different classes and where the circles and squares are train and validation data, respectively. A validation set is a set of data that is not used to train the model, but to check how well the model generalises using unseen data.

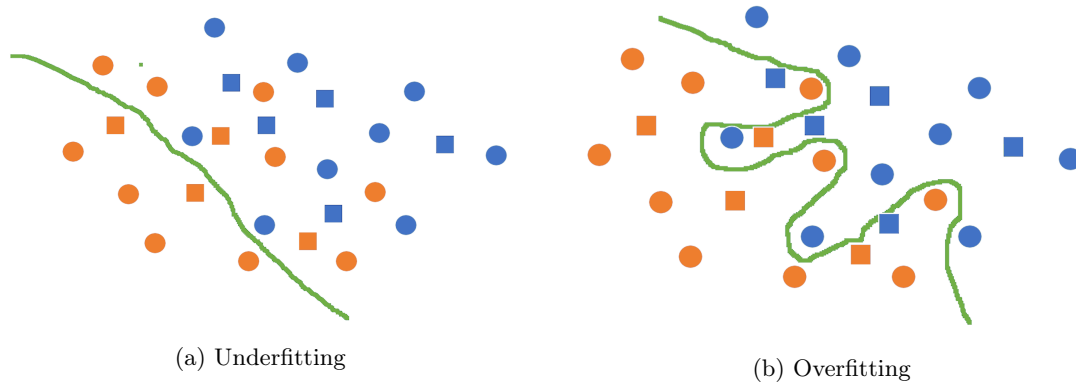


Figure 5.1: Underfitting and overfitting

**Unsupervised Learning** In USL the models are not trained with pre-determined result labels and thus are not told during training what the desired result is supposed to be [37]. An example of USL is clustering where unlabelled data is categorised into clusters based on their similarities [41]. The purpose of USL algorithms differs from the SL algorithms in the sense that USL is not concerned with a simple goal such as making a prediction, but rather discovering interesting relations in the data like dividing the data into subgroups [42].

An advantage of USL is that it is easier to gather unlabelled data, as labelled data often requires human interaction [42]. Because USL can be used to understand and find information about data it can be useful to use USL techniques combined with SL techniques for e.g. feature generation [42].

## 5.2 Machine Learning Models

As described in section 5.1, ML models can be divided into two sub-categories, namely SL and USL. For this project, we will be using SL as the goal is to classify or identify people playing table football. However, USL could be suitable to use if the goal was to cluster data to e.g. tell which players had a similar playing style. In the following subsections, we will analyse the performance of different SL models in relation to identifying people playing table football.

To determine which ML model to use, we will compare the following three model types:

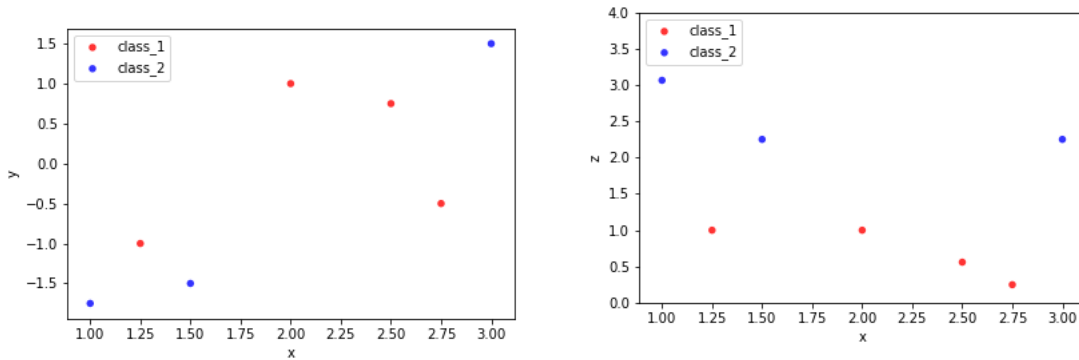
- Support Vector Machines (SVM)
- Logistic Regression (LR)
- Neural Network (NN)

We will discuss each of these models including some of the pros and cons of these models. Based on these pros and cons we will discuss which model type to use in the remainder of the project.

### 5.2.1 Support Vector Machines

The SVM model works by searching for the hyperplane, that best divides a dataset [43]. The distance between the hyperplane and the nearest point from the training set is referred to as the margin, our goal is to find a solution to our problem, that has the largest margin possible.

The idea behind an SVM is its ability to transform a non-linearly separable problem to one that is linearly separable [44] by raising the dimensionality of the feature space. Sometimes it is difficult, with one line, to completely separate the points, e.g. the red and blue dots (see figure 5.2a). However, changing the y-axis to  $z = y^2$ , we are able to easily divide the blue and red dots, as seen in figure 5.2b. SVM models are able to consistently reach a global minimum of a problem, whereas other models often only reach a local minimum in the feature space [43]. Another advantage of an SVM, which e.g. a NN does not benefit from, is the possibility of reproducing the results if the same parameters and data are used. A NN is typically initialised with random weights, making it very difficult to reproduce the same results every time [44]. Disadvantages of using SVM include its computational costs. SVMs are concerned with finding optimal hyperplanes and for non-linear classification problems this “... is a complicated and computationally expensive task.” [45]. Furthermore, SVMs perform badly when there is a lot of training data and when the data is noisy [46].



(a) Feature space with non-linear separable problem (b) Same data as in 5.2a, but with a new y-axis equal to  $y^2$

Figure 5.2: Illustration of how adding a new dimension may help dividing points

### 5.2.2 Neural Network

Neural networks (NN) consist of neurons organised in an input layer, an output layer, and some number of hidden layers between these. Based on certain weights between neurons, a NN is able to predict which output is more likely given a certain input.

A disadvantage of the NN models is that for a network with hidden layers it is difficult to tell what exactly feature representations these hidden layers represent. When training the weights in a NN a technique known as backpropagation is often used, however, when using backpropagation we “... need a very large number of training samples and need a lot of time to gradually approach good values of the weights. Adding new information requires retraining and this is computationally very expensive for Backpropagation Neural Nets ...” [47]. Furthermore, for NNs we are not guaranteed that we find a global minimum for the error and can thus converge to a local minimum instead [43], as opposed to SVMs.

### 5.2.3 Logistic Regression

Logistic regression (LR) is a classification algorithm used for predicting binary outcomes. Suppose a linear function:  $Y = w_0 + w_1 \cdot x_1 + \dots + w_n \cdot x_n$  where  $w$  are weights and  $x$  are numeric input values. Then logistic regression is concerned with minimising the error of the Sigmoid of this linear function [48, sec 7.3.2]. Logistic regression can be trained to predict any linearly separable class; given two classes there exist a decision boundary (linear function) that separates the two classes [48]. For predicting multiple classes multiple LRs can be trained and their results combined to make a single prediction. An LR model can be implemented as a simple artificial neural network (ANN) with an input layer and output layer and a Sigmoid activation function.

### 5.2.4 Selecting the Machine Learning Model Type

The choice of ML model is based on a set of criteria, which fit the platform we have chosen. Since the embedded system chosen in this project has comparatively lower capabilities, regarding computational speed and memory than modern computers, an important criterion is to have fast prediction speed and small model size. Since the input data has high entropy, the model should be able to accommodate for this. The aforementioned criteria can be seen below:

- The model should have a low model size
- The model should allow input with high entropy

The data collected by the sensors is noisy and has high entropy and it is therefore likely that there is not a linear relationship between the input features and the output features meaning that an LR model would be imprecise. A problem with SVMs, as explained earlier, is that it can be difficult to find a decision problem between classes if the data is noisy, which is the case for our project, as we use accelerometers. Since we want to be able to make classification and training on an embedded system, we need to be able to manually adjust the model size. This can be accomplished for a NN by increasing or decreasing the number of e.g. neurons, layers, or labels, however this is not the case for a SVM: “*In contrast to neural networks SVMs automatically select their model size (by selecting the Support vectors).*” [43].

Based on an analysis conducted in the book *Handbook of Research on Modern Systems Analysis and Design Technologies and Applications* [45], table 5.1 shows the average accuracy and computational performance of SVMs and NNs measured on 100 classification problems. A rank close to 1 is the best performing models and a rank close to 0 is the worst performing models. As can be seen in the table the NN has higher average prediction accuracy than SVMs when tested on 100 classification problems, however, the computational time of training and testing is larger for the NN model when compared relative to the models used by [45].

Classifier 2	SVM	NN
Average accuracy	0.585	0.621
Execution time	0.5	0.015

Table 5.1: Average accuracies and execution times of SVM and NN based on analysis in [45]

Since a one-layer NN with Sigmoid activation is equivalent to LR, we can simply choose to design a simple NN if we determine that a simple model is sufficient to identify people based on movements

in table football, while also having the flexibility to increase the model complexity. Furthermore, we can manually select the model size when using NN (by adjusting the number of neurons and layers) which is handy when we work with embedded systems as we have a low amount of memory. Additionally, the analysis in [45] indicates that artificial neural network (ANN) has good average accuracy compared to other models. For these reasons, we choose NNs as the type of models to analyse and implement in the remainder of the project.

## 5.3 Neural Networks

NNs are a specific type of ML models used within SL and are inspired by how the human brain works and learns. NNs consist of a set of neurons which are divided into a set of layers where each layer has connections to the subsequent layer. A simple one-layer NN consists of an input layer and an output layer. Any layer between the input layer and output layer is called a hidden layer. The connections between neurons in the different layers have a weight associated with them which tells how large an impact any particular input-value has on the output of a neuron.

Some NNs also have a type of node known as bias nodes. Bias nodes are similar to neurons in the sense that each bias node associates weights to neurons in a particular layer. The difference between the bias node and ordinary neurons is that the input to a bias node is always one. The bias node is therefore independent of the input to the NN and since the weights of the bias nodes are adjusted during training the NN is, therefore, able to learn to have a bias.

Figure 5.3 shows a simple illustration of a fully connected NN with four input neurons and a bias neuron, one hidden layer with three neurons and a bias neuron, and one output neuron.

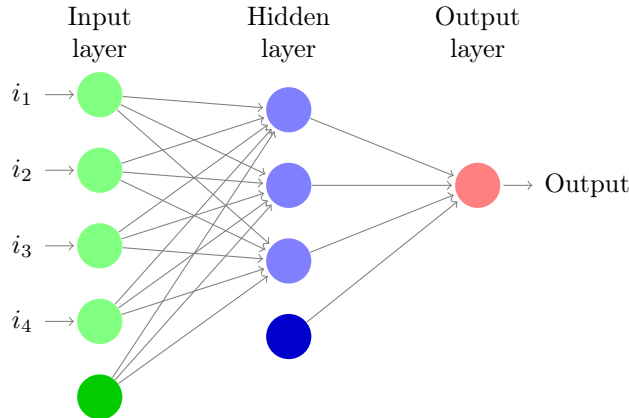


Figure 5.3: NN with two layers and bias nodes (dark green and dark blue nodes)

To calculate the value of a neuron in a hidden layer or output layer, the weighted sum of inputs to that neuron is calculated by summing each input multiplied with their associated weight plus the weight of the bias node. Equation 5.1 formalises this where  $w_{qj}$  is the weight between a neuron  $q$  and a neuron  $j$ ,  $i_j$  is the  $j$ 'th input value, and  $b_q$  is the weight value from the bias node to the neuron  $q$ .

$$x_q = \sum_{j=1}^n (i_j \cdot w_{qj}) + b_q \quad (5.1)$$

Now, the value of the neuron,  $x_q$ , has a value between positive and negative infinity which is fine for regression problems, but assume we instead want to figure out which class a particular input belongs to. Having a continuous output value would not help. To fix this we can use activation functions.

An “... **activation function**, is a function from the real line  $[-\infty, \infty]$  into some subset of the real line such as  $[0,1]$ ” [48, sec. 7.3.2]. An activation function takes as input the weighted sum of input of a neuron which is given by equation 5.1. The activation, “... transforms the signal non-linearly and it is this non-linearity that allows us to learn arbitrarily complex transformations between the input and the output.” [49]. In future sections we will refer to the output of a neuron as the output of applying an activation function.

A simple activation function is the step function which, based on a threshold value, indicates whether a neuron should fire [50, 51]. The step function is shown in equation 5.2 and visualised in figure 5.4, where  $t$  is the threshold value and  $x$  is the weighted sum of input given by equation 5.1 [51].

$$f(x) = \begin{cases} 1 & \text{if } x > t \\ 0 & \text{if } x \leq t \end{cases} \quad (5.2)$$

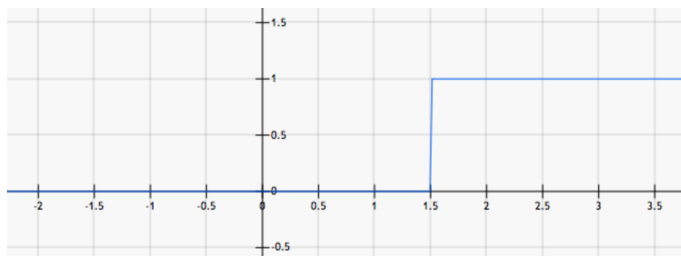


Figure 5.4: The step function

A problem of the step function is that if multiple neurons fire in the output layer, we cannot tell apart which class we are most confident is the correct class, as multiple neurons will have the same output value [50]. Another problem of the step function is that it is not differentiable which makes it difficult to use when making gradient descent (5.3.1) [48, sec. 7.3.2]

The Sigmoid activation function is an example of a differentiable function and is given by:  $f(x) = 1/(1 + e^{-x})$  [48, sec. 7.3.2]. Figure 5.5 shows a plot of the Sigmoid function. The Sigmoid squashes the input into a value between 0 and 1 where a value of 0 represents a neuron not firing and a value of 1 represents the maximum confidence, indicating that a neuron is firing [52]. However, a problem with the Sigmoid function is that when output values come very close to 0 or 1 the gradient is almost zero (see figure 5.5). This gives rise to what is known as the vanishing gradient problem where the network will not be able to learn [50, 52].

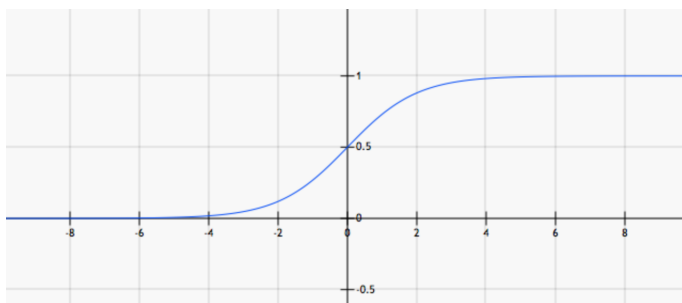


Figure 5.5: The Sigmoid function

The rectified linear unit (ReLU) illustrated in figure 5.6 is another example of a differentiable activation function. The ReLU is defined as  $f(x) = \max(0, x)$  which is evidently faster to compute than the Sigmoid function. However, as can be seen on the plot for the ReLU function the gradient can for some input values go towards 0 resulting in dead neurons which of course means that the output of that particular neuron will not contribute to the evaluation of the network. This problem can be fixed using a modified version of the ReLU known as leaky ReLU [50, 52]. Leaky ReLU is different from ReLU in that for negative input values there is a small slope.

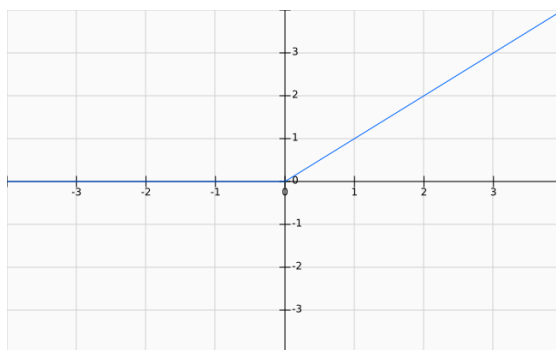


Figure 5.6: The ReLU function

The Softmax activation function has the property that when applied to a layer, it outputs a probability distribution [53], and can, therefore, be convenient to use when we want to determine with what probability our NN gives a specific output or how certain we are that our predictions are correct. The Softmax function is shown below in equation 5.3 where  $a_j^L$  is the output of applying the Softmax activation function to the  $j$ 'th neuron in the  $L$ 'th layer and  $x$  is the weighted sum of inputs to a neuron as given by equation 5.1 [53].

$$a_j^L = \frac{e^{x_j^L}}{\sum_{k=1}^n e^{x_k^L}} \quad (5.3)$$

As explained in [50] and [52], there is no single correct activation function to use, however ReLU is often used instead of Sigmoid due to the vanishing gradient problem. For classification problems the Softmax activation function is convenient to use in the last layer to give a probability distribution [53].

### 5.3.1 Training a Neural Network

We will now briefly describe some of the techniques used for training a NN. Assume that all the weights in the NN are randomly initialised; this will, of course, produce random outputs. To improve the prediction accuracy we need to train the NN to weight each connection accordingly to how much they matter for the desired output. In order to adjust the weight, we use a technique called gradient descent which, based on training examples, tries to minimise the output error by taking the negative gradients of some function with respect to its weights [54]. Consider figure 5.7 which shows the error  $E$  with respect to some weight  $w$  for some error function in the NN. We want to adjust the weights in the NN to minimise error. By repeating this process the cost will become lower as training progresses, and at some point, when the error converges to a minima or if a stopping criterion is reached, training is stopped.

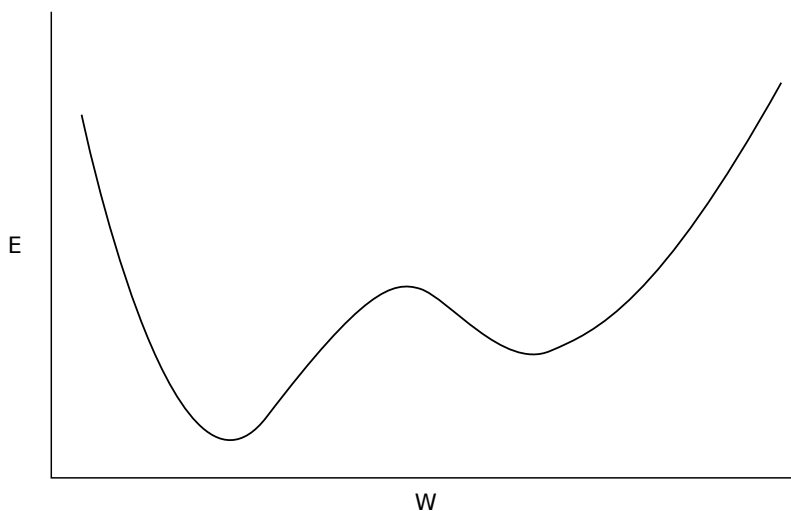


Figure 5.7: Sketch of the error as a function of a particular weight

To update the weights in a NN with more than one layer, backpropagation is used where each weight in the NN is updated relative to how much influence they have on the error [54]. To find this influence, the chain-rule of differentiation is applied by going through the NN from the output layer and backwards, hence the values of the new weights are propagated backwards from the output layer [55, sec: 7.5].

Let  $C$  be the result of some cost function (also called error function). We are interested in finding how much a small change in the weights of the NN affects the cost of the NN. We want to find values for the weights in the NN such that the cost is minimised [56]. To do so, we first want to find out how much the cost changes with respect to the output of a neuron as follows:  $\frac{\partial C}{\partial a}$ , where  $a = f(x)$ ,  $f$  is some activation function like ReLU or Sigmoid, and  $x$  is given by equation 5.1 [54]. Furthermore, we also need to find out how much the output of a neuron changes in relation to its input, thus we want to find the rate of change of the activation function with respect to the sum of weighted inputs for that neuron,  $x$  [54]. Equation 5.4 gives the error of an output neuron with respect to how much that neuron influences the cost function where  $\delta_j^L$  is the error for some neuron  $j$  in the output layer  $L$ ,  $a_j^L$  is the output of a neuron  $j$  in the output layer and  $f'(x_j^L)$  is how fast the activation function is changing in  $x_j^L$  where  $x_j^L$  is the weighted sum of input to a neuron  $j$  in the output layer  $L$  [54]. Equation 5.4 is the result of applying the chain rule of differentiation which we need to do as, the output of a neuron is a composite function  $f \circ g$ , where  $f$  is some activation function and  $g$  is the function for calculating the weighted sum of input for some neuron in the NN,

as given by equation 5.1.

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \cdot f'(x_j^L) \quad (5.4)$$

However, as explained earlier, we need to propagate this error through the network as we only know the cost at the output layer and we are interested in finding the error in each layer so that we, at a later time, can find the rate of change for the weights with respect to the cost function. The error in the hidden layers is again a result of applying the chain rule and is given by equation 5.5. In equation 5.5,  $\delta_i^l$  is the error of neuron  $i$  in layer  $l$  which is given by summing the weighted error for each connection  $w_{ij}$  to a neuron in the subsequent layer and multiplying the result with the rate of change of the activation function  $f$  given  $x_i^l$ .

Figure 5.8 shows a NN with three highlighted neurons (coloured). Assume that errors E1 and E2 have been calculated for the purple and teal output neurons using equation 5.4 then we backpropagate the error to the blue neuron as follows:  $f'(x) \cdot (w1 \cdot E1 + w2 \cdot E2)$  where  $x$  is the sum of weighted input to the blue neuron.

$$\delta_i^l = \left( \sum_{j=1}^n \delta_j^{l+1} \cdot w_{ij} \right) \cdot f'(x_i^l) \quad (5.5)$$

Now that we know how to propagate the error through the neurons of the network, we can find the partial derivative of the cost function with respect to each individual weight in the NN. This tells us how changing each weight in the NN affects the cost of the NN. Suppose that we want to find  $\partial C / \partial w_{ij}$  which is the partial derivative of the cost function with respect to some weight  $w_{ij}$  where  $i$  is some neuron in layer  $l$ ,  $j$  is some neuron in layer  $l + 1$  and  $w^{ij}$  is the weight between these two neurons. Then we can find the partial derivative of a particular weight in the NN by simply multiplying  $a_i^l$  with  $\delta_j^{l+1}$  which is the error of the  $j$ 'th neuron in the  $l + 1$  layer [54]. This is formalised in equation 5.6 [54]. Looking at figure 5.8 we can calculate the partial derivative of the cost function with respect to the weight  $w1$  (orange connection) as follows:  $f'(x) \cdot E1$

$$\frac{\partial C}{\partial w_{ij}} = a_i^l \cdot \delta_j^{l+1} \quad (5.6)$$

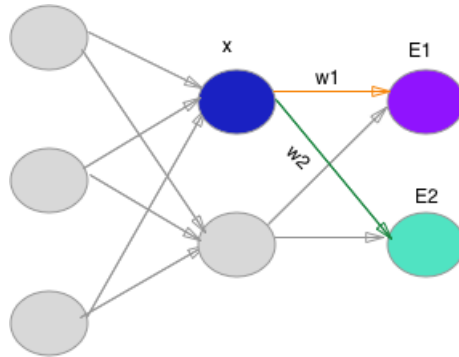


Figure 5.8: ANN with errors E1 and E2 in the output layer

Similarly, we can calculate the impact that a particular bias has on the cost of the network as given by equation 5.7 [54], where  $b$  is the bias node in layer  $l$  and  $w_{ib}$  is the connection between the  $i$ 'th



neuron and the bias node in this layer. Note that we do not have to multiply with the output activation as the input to a bias node always is one.

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \quad (5.7)$$

When we have calculated the partial derivative of the cost function with respect to a specific weight,  $\partial C/\partial w$ , we can adjust that weight according to some constant learning weight as given by equation 5.8, where  $new_w$  is the new weight,  $old_w$  is the old weight, and  $\eta$  is the constant learning factor [48, sec. 7.3.2].

$$new_w = old_w - \eta \cdot \frac{\partial C}{\partial w} \quad (5.8)$$

The explanation above illustrates learning where the weights are updated after each training example, also known as stochastic gradient descent [57]. An alternative to this is batch gradient decent where weights are updated after each batch of examples [48, sec. 7.3.2].

**Cost Functions** play an important role in training a NN as explained earlier. An example of a cost function is the squared cost function (equation 5.9), where the error of a single training example is calculated by taking the difference between the output and the expected result, squared.

$$C_{result,expected} = (expected - result)^2 \quad (5.9)$$

There exist different cost functions and two assumptions that should hold for cost functions used in backpropagation are: They should be able to be written as a function of the output of the NN and the cost should say something about the actual output of the NN relative to the expected output [54, 53]. This means that when the actual output is close to the expected output the cost should be close to zero [54, 53]. The second assumption is that the cost function should be able to be written as an average over cost functions of the training examples [54].

## 5.4 Recurrent Neural Networks

Recurrent neural networks (RNN) are a modification of artificial neural networks (ANN) and are used for making predictions given sequence data [58]. RNNs differ from ANNs by not assuming that all inputs and outputs of a NN are independent of each other [59]. RNNs are by definition recurrent in that they use unrolling to apply some function,  $f$ , over an entire sequence. During this sequence, the output of each element in the sequence is dependent on previous computations. Simple RNNs, as those described so far, are difficult to train due to the vanishing gradient problem [60]. Figure 5.9 shows a simple illustration of a RNN where a network is being unrolled  $t$  times. Note that  $I_1 \dots I_t$  and  $O_1 \dots O_t$  are input and output vectors, respectively for each network from  $1 \dots t$  and  $s$  is the hidden state at each time step [59].

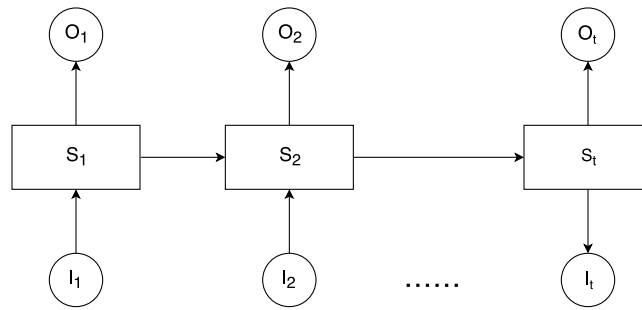


Figure 5.9: Illustration of a RNN

To make training the RNN easier, we will be using a more sophisticated class of RNNs called Long Short-Term Memory networks (LSTM). LSTMs are a modified version of the previously described RNNs and is designed to avoid the problems introduced by the exploding and vanishing gradient problem which makes learning over a long time span difficult [61]. The key part of the LSTM is that information is stored in a cell from which data can be read, written and updated. The cell is structured by different gates that open and closes which determine which data is removed, updated, and other decisions [62]. We will not go into detail explaining LSTMs, however, one should think of LSTMs as a type of RNN with memory cells. In the remainder of this report, we will use the names LSTMs and RNNs interchangeably. LSTMs are further explained in [61] and [62].

# Chapter 6

## Data Processing

In this section we describe how we can manage and process the raw sensor data to use it for training and testing different ML models. The data processing involves labelling, scaling, filtering and grouping the data.

### 6.1 DataHub

Given the collected data we want to ...

- keep track of the collected data
- visualise the data to make it easier for the group to see values instead of raw text
- label the data i.e. associating data with the person who played during the data collection
- process the data

To test and analyse results of different ML models we also want a way to ...

- manage ML models
- manage tests
- compare the accuracy of different models given a test

To do this, we develop a system called DataHub to facilitate these tasks.

#### 6.1.1 Architecture

We choose to develop the DataHub as a web application to make it available to all group members on any device and at any location. For this, we use a client-server architecture where the user interface is presented in a browser on a client computer and the data management is handled centralised with an API and a database on a server. The architecture for the DataHub is shown in figure 6.1.

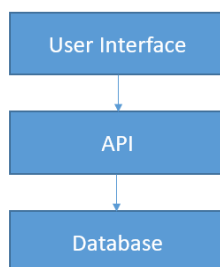


Figure 6.1: DataHub architecture

The database component stores datasets, labels (players), LM models, tests, and relations between these. The API component offers an interface of functions to manage the entities in the database component, and also allows querying. Finally, the user interface component makes the functions in the API component accessible for users through a graphical interface. The DataHub architecture is an example of a closed-strict architecture meaning that each component is only dependent on the component just below.

### 6.1.2 Data Modelling

To model the structure of the domain, we can construct an entity relation diagram (ERD) showing the different entities in the database component and how these are related. The ERD is shown in figure 6.2.

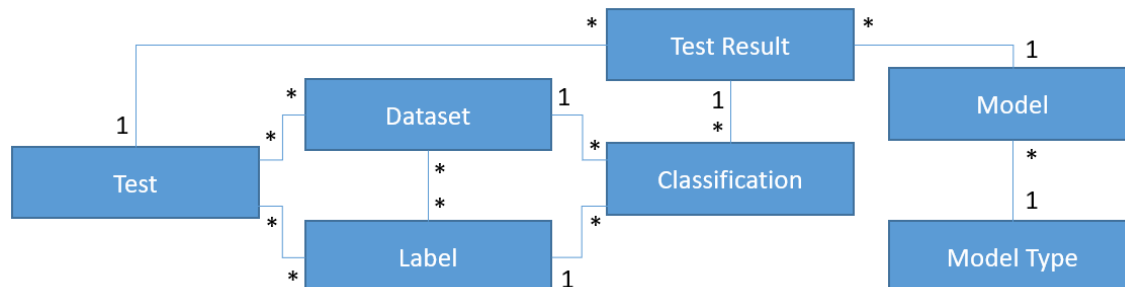


Figure 6.2: Entity relation diagram for the database component in the DataHub

The boxes in figure 6.2 represent the entities and the lines represent relations between entities. The type of relation is represented using 1 and \* where e.g. there is a one-to-many relationship between Test and Test Result meaning that each test is related to 0 or more test results while each test result is always associated a single test. Another example is the many-to-many relation between Test and Dataset indicated with \* and \*, meaning that each dataset may be associated with zero or more tests and vice versa.

### 6.1.3 Uploading Data

After receiving the data from the embedded system and saving this into a comma separated file (csv-file), we want to be able to upload this file to the DataHub to represent it as a dataset entity.

To do this, we have a function in the API component which handles the upload process. This function is then used by the user through the page shown in figure 6.3.

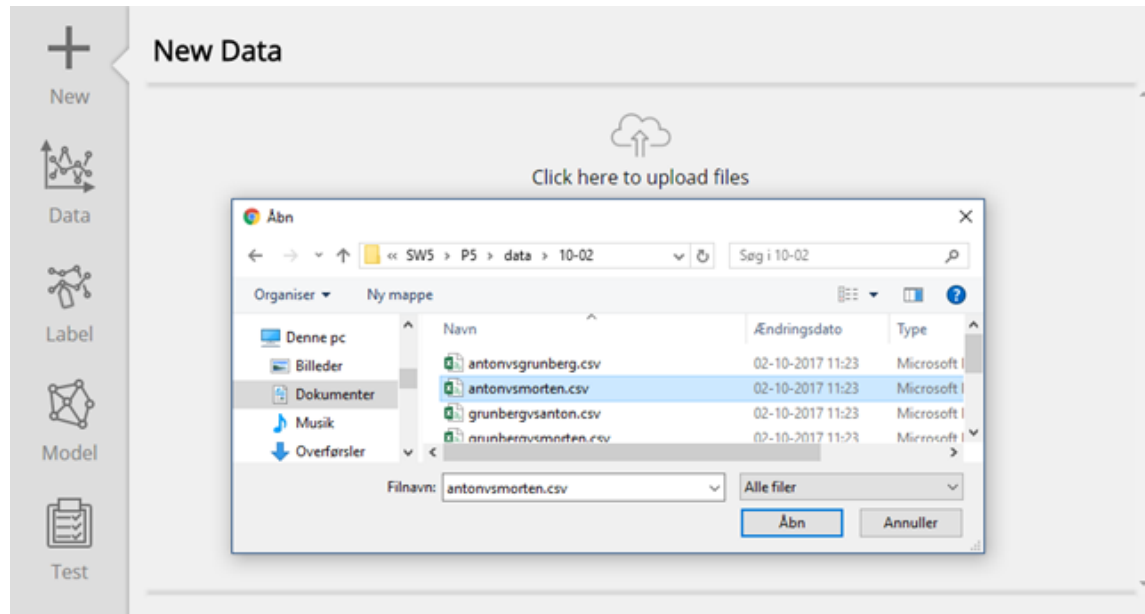


Figure 6.3: DataHub upload page

#### 6.1.4 Visualising Data

To visualise the uploaded data, we have a function in the API component which returns the data points to be visualised as shown on the page in figure 6.4. The charting API used is called Highcharts [63].

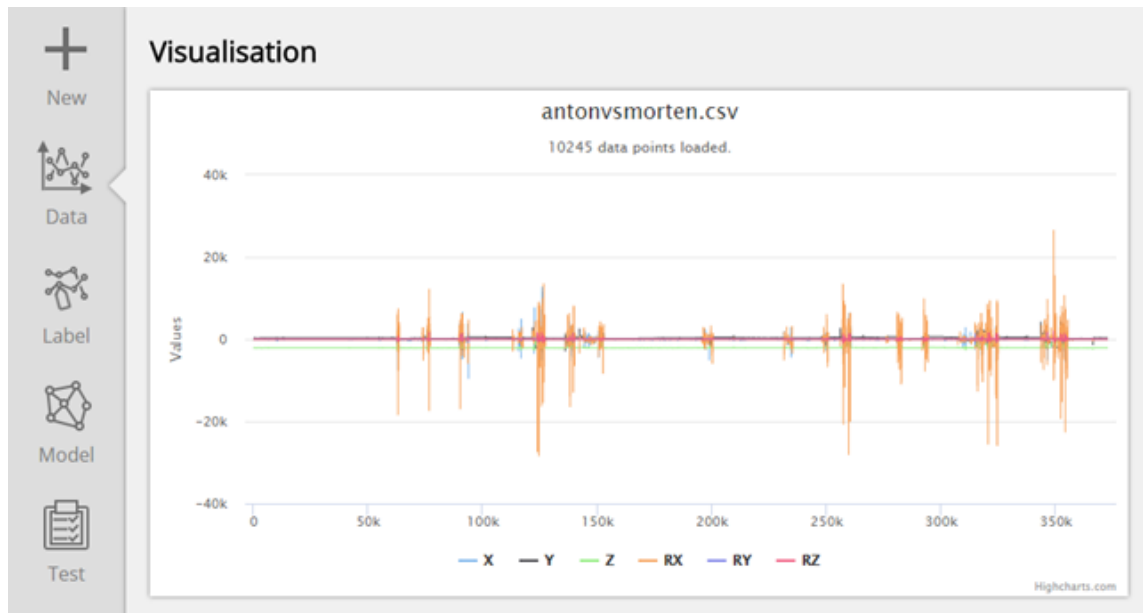


Figure 6.4: DataHub visualisation page showing the data collected during a game of table football

### 6.1.5 Labelling Data

To manage the relationship between datasets and labels we create the page shown in figure 6.5. At this page, we can select each dataset and afterwards select which labels are associated with it. For player classification, we only select one label that is the player who played during the data collection.

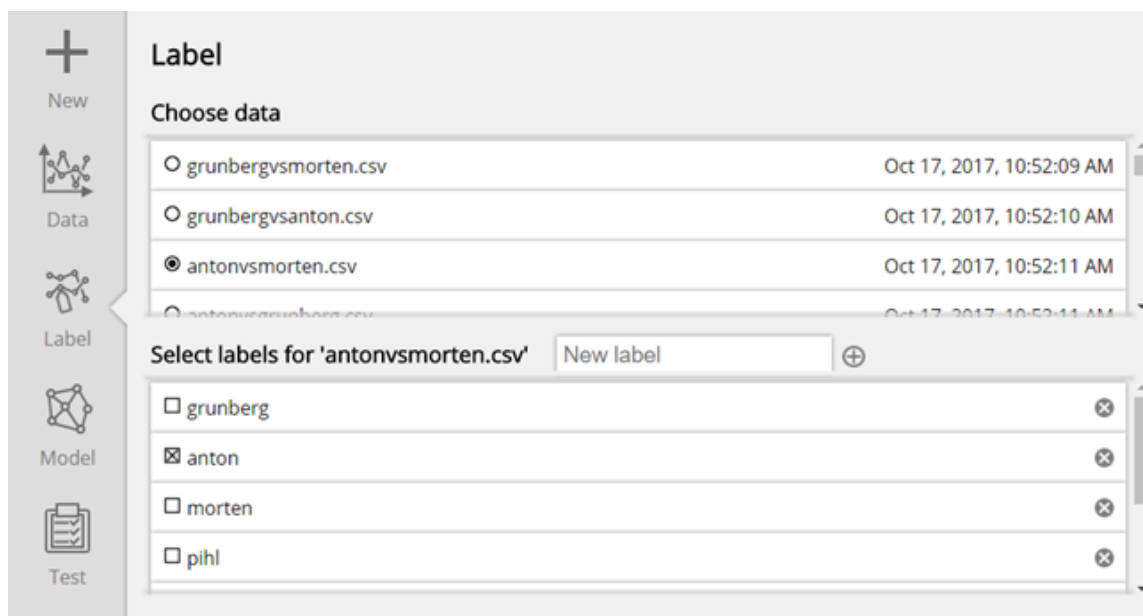


Figure 6.5: DataHub label page

The model and test management in the DataHub will be described later in relation to the test of the different ML models. Before we describe this we will now look at how we can create a more structured representation of the data from an entire table football match.

## 6.2 Grouping Data into Movements

We now have functionality to upload, manage, and label data in the DataHub. To test different ML models we need to find a way to simplify and structure the data. The reason for this is that if we sample data from the MPU-6050 with an interval of 2ms then the amount of data being collected during an entire table football match of e.g. 5 minutes will be  $5 \cdot 60s \cdot (\frac{1}{2ms}) \cdot 6 \cdot 2 \text{ bytes} = 1800000$  bytes. Storing this amount of data is expensive, especially considered the low amount of memory of most embedded systems. For this reason the data points we collect are downscaled in size by taking the average over 20 data points as shown in listing 6.1.

```

1  [...]
2  List<Data> downScaled = new List<Data>();
3  t = 0; x = 0; y = 0; z = 0; rx = 0; ry = 0; rz = 0;
4  int counter = 0;
5  int scale = 20;
6  foreach (var datapoint in data)
7  {
8      t += datapoint.Time;
9      x += datapoint.X;
10     y += datapoint.Y;
11     z += datapoint.Z;
12     rx += datapoint.RX;
13     ry += datapoint.RY;
14     rz += datapoint.RZ;
15     counter++;
16     if (counter == scale)
17     {
18         downScaled.Add(new Data() { Time = t / scale, X = x / scale, Y
           ↪ = y / scale, Z = z / scale, RX = rx / scale, RY = ry /
           ↪ scale, RZ = rz / scale });
19         t = 0; x = 0; y = 0; z = 0; rx = 0; ry = 0; rz = 0;
20         ccounter = 0;
21     }
22 }
23 [...]
```

Listing 6.1: The code converting sets of 20 data points into an average downscaled data point

Looking at the plot in figure 6.4, we see that there is a clear distinction between silent moments and movements, by looking at the size of the fluctuations. As the goal of this project is to identify people based on movements, we want to group the data into movements and filter out the silent data. This is done by filtering the data by rotation around x (RX) which is an indicator of movement (see figure 6.4). A movement consists of at least ten downscaled data points, where we start grouping (measuring) when detecting an RX value that is above 5000 or below -5000. We then keep

measuring, until we have encountered ten 'silent' downsampled points, i.e. ten downsampled points in a row, all with an RX value between -5000 and 5000. The code for grouping movements is shown in listing 6.2.

```

1 public class ShotIdentifier : IGroupIdentifier
2 {
3     public List<Group> Identify(List<Data> data)
4     {
5         bool measuring = false;
6         int zeroStreak = 0;
7         Stack<Data> groupData = new Stack<Data>();
8         List<Group> groups = new List<Group>();
9         foreach (var point in data)
10        {
11            if (measuring)
12            {
13                groupData.Push(point);
14            }
15            if (point.RX < 5000 && point.RX > -5000)
16            {
17                zeroStreak++;
18                if (zeroStreak == 10)
19                {
20                    groups.Add(new Group() { Data =
21                        ↪ groupData.Reverse().ToArray() });
22                    groupData = new Stack<Data>();
23                    measuring = false;
24                }
25            }
26            else
27            {
28                measuring = true;
29                zeroStreak = 0;
30            }
31            return groups;
32        }
33    }

```

Listing 6.2: The class ShotIdentifier.cs grouping data into movements (shots)

The code in listing 6.1 and 6.2 are implemented in the DataHub to downscale and extract movements in the datasets which we can use in chapter 5 to train and test different ML models. The classification task is now reduced to classification of a player based on individual movements instead of classification based on data from an entire table football match. A limitation of the algorithm is that it may remove data which can be used to identify a player.

As we sample at a rate of 500Hz, downscaling by a factor of 20, the resulting frequency is 25Hz. Since this is less than the Nyquist-Shannon limit set in section 4.5, there is a possibility, that the data for some movements do not represent reality accurately. For this reason, we must take into



account that we might need to revise this algorithm later if we encounter low accuracy of the ML models.

## Chapter 7

# Analysing Machine Learning Models

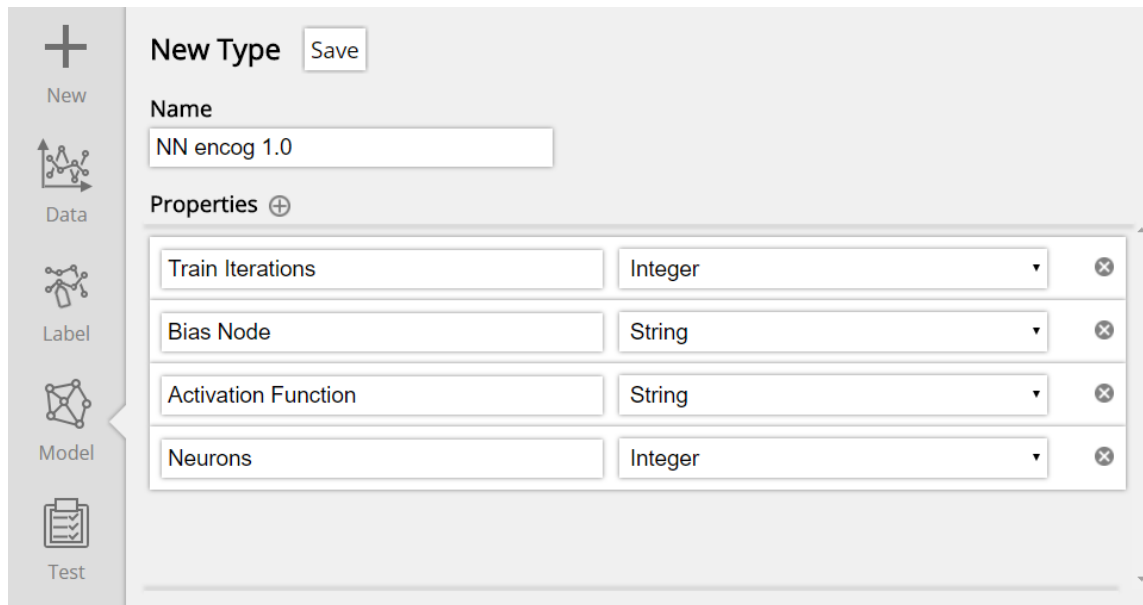
Based on the pros and cons of the models analysed in chapter 5, we will select a subset of these models which will be further analysed to see how well they perform for this problem. In order to analyse the performance of the models, we construct a series of tests that each model will be evaluated against. Based on these tests we then analyse the performance of the different models. Finally, we compare the different models, and from that analysis, we select the model to be used in this project.

### 7.1 Model Analysis

In section 5.2.4, we chose NNs as the type of ML model to implement for this project. The purpose of this section is to analyse different NN models relative to this project and determine which NN model to use in the remainder of the project.

#### 7.1.1 Test Suite

In this section, we will describe the test suite used for analysing different ML models in the remainder of this section. As a tool for testing different ML models we use the DataHub as described in chapter 6. Figure 7.1 shows an example of a model for a particular kind of ANN with properties created on the DataHub.

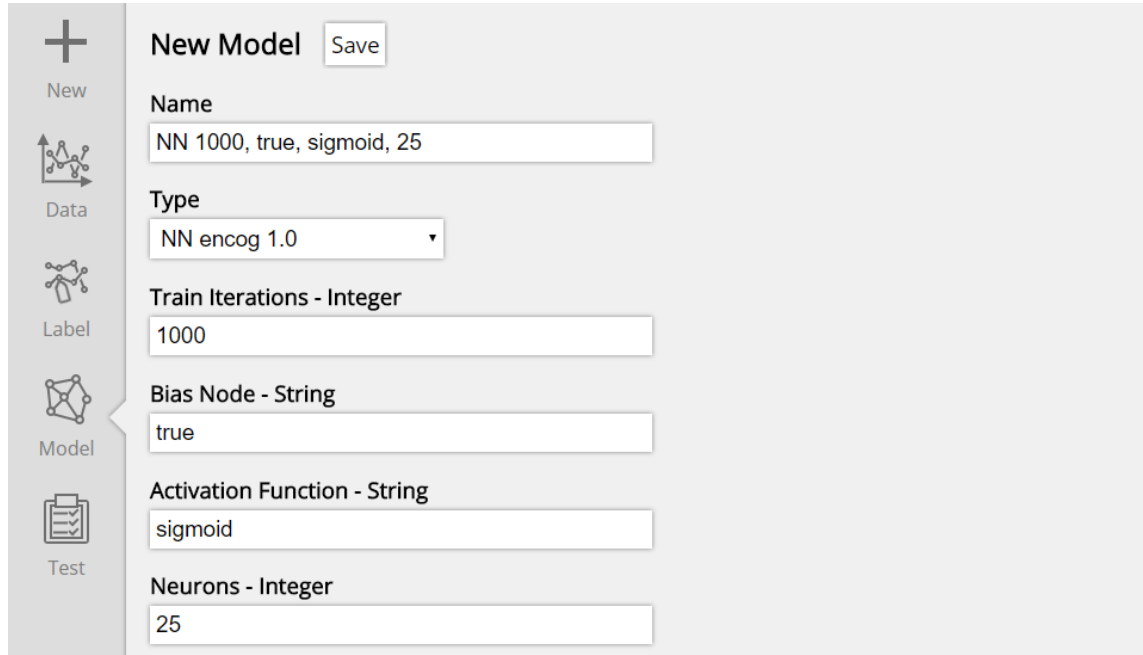


The screenshot shows the 'New Type' interface in DataHub. On the left is a vertical sidebar with icons for 'New', 'Data', 'Label', 'Model', and 'Test'. The main area is titled 'New Type' with a 'Save' button. The 'Name' field contains 'NN encog 1.0'. Below it is a 'Properties' section with a plus icon, containing four rows of parameter definitions:

Property Name	Property Type
Train Iterations	Integer
Bias Node	String
Activation Function	String
Neurons	Integer

Figure 7.1: Example of an ANN model created on the DataHub

The created models can then later be instantiated with different values for the parameters, as shown in figure 7.2.



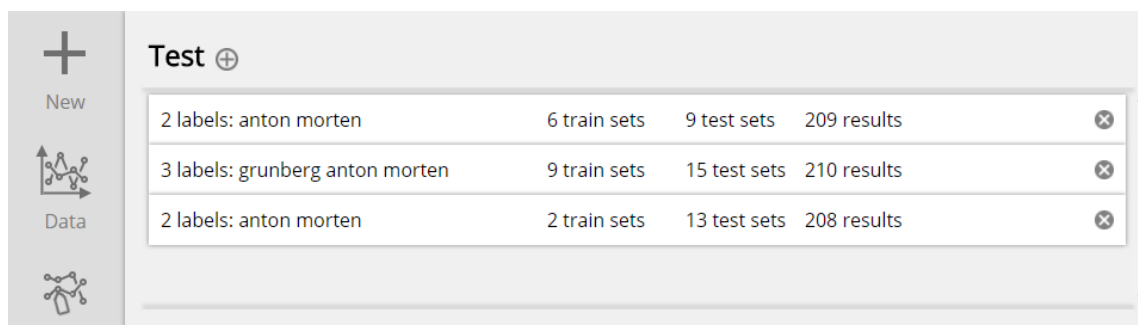
The screenshot shows the 'New Model' interface in DataHub. On the left is the same vertical sidebar as in Figure 7.1. The main area is titled 'New Model' with a 'Save' button. The 'Name' field contains 'NN 1000, true, sigmoid, 25'. The 'Type' dropdown menu is set to 'NN encog 1.0'. Below are five parameter fields with their values:

Parameter Name	Value
Train Iterations - Integer	1000
Bias Node - String	true
Activation Function - String	sigmoid
Neurons - Integer	25

Figure 7.2: Example of an ANN model created with values

As described earlier in section 5.2.4, we have chosen to focus on analysing different NNs. Specifically, we have chosen to test different variations of ANNs and RNNs which we do in section 7.1.2 and 7.1.3,

respectively. To test these models, we have gathered data from three different people within the project group by using the product described in chapter 4. To test the performance, of the different variations of the RNNs and ANNs created on the DataHub, a test suite has been established from the collected data and contains the tests shown in figure 7.3. In the remainder of the project, the tests shown in the figure will be denoted test 1, test 2, and test 3 according to the order in which they appear in the figure. The purpose of the test suite is to check the performance of the ML models on the different subsets of the collected data. More precisely, we want to check how the models perform when giving respectively two labels (test 1 and 3) and three labels (test 2). Furthermore, we want to check how well the models perform when having more training data (test 1) in contrary to having less training data (test 3). To perform the tests an API has been created for the DataHub where clients can request to get the data for the datasets contained in the test suite. The datasets used in the tests are collected using the embedded system described in milestone 1 from matches between two players playing a match to ten goals.



Test				
2 labels: anton morten	6 train sets	9 test sets	209 results	✕
3 labels: grunberg anton morten	9 train sets	15 test sets	210 results	✕
2 labels: anton morten	2 train sets	13 test sets	208 results	✕

Figure 7.3: Shows the three test for the analysis of the ML models

## Encoding the Labels

When we create datasets on the DataHub each dataset labelled with the player's name as a string. Thus two possible strings could be Alice and Bob. In order to feed these labels to a NN, we will need some kind of numerical representation. Thus Alice could be given the value 1 and Bob 2. However, this will encode that there is a distance of 1 between Alice and Bob. This becomes a problem when we increase the number of players. To avoid this problem we will encode our labels using a technique called *one hot encoding* where each player is given its own value and assigned to 1 or 0 to indicate which category amongst all the others we are referring to. Lets look at a short example for the Alice and Bob strings. In this case, we have two categories meaning that the total numbers of distinct categories are exactly two, thus we have two values. Assume that we order our labels lexicographical then Alice would be encoded as  $[1, 0]$  and Bob as  $[0, 1]$ .

## Calculating Accuracy

In order to compare different ML models, we need a measure of how accurate the models are at classifying players correctly on test datasets. To do this we run each model ten times for each test. We do this in order to capture that the error during training of NNs might converge to different local minima depending on the initial random values assigned to the weights of the network. Each test contains multiple test datasets. For each test dataset, there is an expected player to be classified. Each dataset contains a list of movements that each model will classify. To determine a model's classification of a player based on a test dataset, we average the output of the model's classification for all the movements in the test dataset and select the class with the highest average value. To

measure the accuracy we compare the expected player with the player classified by the ML model. We can then sum all the correct classifications  $N_{correct}$  for the test datasets in a test and divide it by the total number of test datasets in the test  $N_{total}$  to calculate the accuracy of the ML model for a given test as follows.

$$A_{testrun} = \frac{N_{correct}}{N_{total}}$$

This calculates the model accuracy based on a single run of a given test. Because we run the model ten times for each test, we need a value to represent the combined accuracy of all test runs. To do this, we average the accuracies  $A_{testrun_1}, A_{testrun_2}, \dots, A_{testrun_{10}}$  for the ten test runs.

$$A_{test} = \frac{1}{10} \sum_{i=1}^{10} A_{testrun_i}$$

To get a measure of a model's accuracy across all three tests we introduce what we will refer to as the average accuracy as the average of the three test accuracies.

$$Avg_{model} = \frac{A_{test1} + A_{test2} + A_{test3}}{3}$$

### 7.1.2 Artificial Neural Network Analysis

In this section, we will analyse the accuracy of different models of artificial neural networks (ANNs). We construct and test models to be able to determine if ANNs can be used to classify players and to test the following hypothesis constructed by the group members which will give us more information about the construction of ANNs. Hypotheses for different models given the same amount of training iterations and tests:

1. Two hidden layers implies higher average accuracy than one hidden layer
2. For models with one hidden layer, increasing the number of hidden neurons leads to higher average accuracy
3. More training data implies higher average accuracy
4. Fewer players to classify between implies higher average accuracy
5. Rectified linear unit (ReLU) as activation function implies higher average accuracy compared to Sigmoid and Softmax

In this section, the average accuracy refers to the average of the accuracies from the three tests on datasets which are not part of the training set.

#### Feature Selection

All the ANNs compared in this section are trained with acceleration data in the sensor's  $x$ ,  $y$ ,  $z$ -direction, as well as the rotation velocity around the  $x$ -axis, which we call  $rx$ . Rotation velocities around the  $y$ - and  $z$ -axis are not included as these are not influenced by movements of the handles because of the position of the sensor as shown in figure 4.4.

### Constructing Artificial Neural Networks

Each ANN compared in this section has 40 input neurons. This is based on the minimum number of data points in each of the groupings described in section 6.2 which all contains at least ten data points. Because each data point has four features  $x$ ,  $y$ ,  $z$ , and  $rx$  then we end up with  $10 * 4 = 40$  inputs. The number of output neurons corresponds to the number of players to classify between. Each output neuron represents a player. The output value of the neuron represents the confidence ranging between 0 and 1 that the input to the ANN was performed by the corresponding player. The output layer of each tested ANN has the Sigmoid activation function to ensure values ranging between 0 and 1. Given the configuration of the input layer and output layer, we can now test different configurations of the hidden layer(s) of the ANNs to test if it is possible to classify players and to test the hypotheses stated in the beginning of section 7.1.2.

### Testing Artificial Neural Networks

To test ANNs we need an implementation of the structure and training behind ANNs. For this, we use the Encog library. We choose this library because it supports the C# programming language and allows a variety of different models including NNs [64]. The Encog library also supports persisting a trained model which is useful if a model should be used later or integrated into an embedded system. The table below shows the configurations of the different models used during the test including the accuracy of each test. The different configurations are based on the five hypothesis presented at the beginning of section 7.1.2.

Model	Hidden layers	Training iterations	Has bias node	Activation function	Neurons	$A_{test1}$	$A_{test2}$	$A_{test3}$
1	1	1000	True	Sigmoid	5	97%	84%	90%
2	1	1000	True	Sigmoid	25	100%	93%	88%
3	1	1000	True	Sigmoid	100	100%	95%	98%
4	1	1000	True	Sigmoid	250	100%	97%	95%
5	1	1000	True	ReLU	250	100%	73%	91%
6	1	1000	True	Softmax	250	99%	81%	89%
7	1	1000	True	Sigmoid	500	100%	77%	88%
8	2	1000	True	Sigmoid	100, 25	100%	67%	89%
9	2	1000	True	Sigmoid	25, 100	94%	75%	76%
10	2	1000	True	Sigmoid	50, 50	98%	79%	79%
11	2	1000	True	Sigmoid	5, 5	98%	67%	85%

Table 7.1: Shows the tested models and their accuracy on each test

**Hypothesis 1:** *Two hidden layers implies higher accuracy than one hidden layer.*

To test two hidden layers versus one hidden layer we compare models 1-4, and 7 against models 8-11.

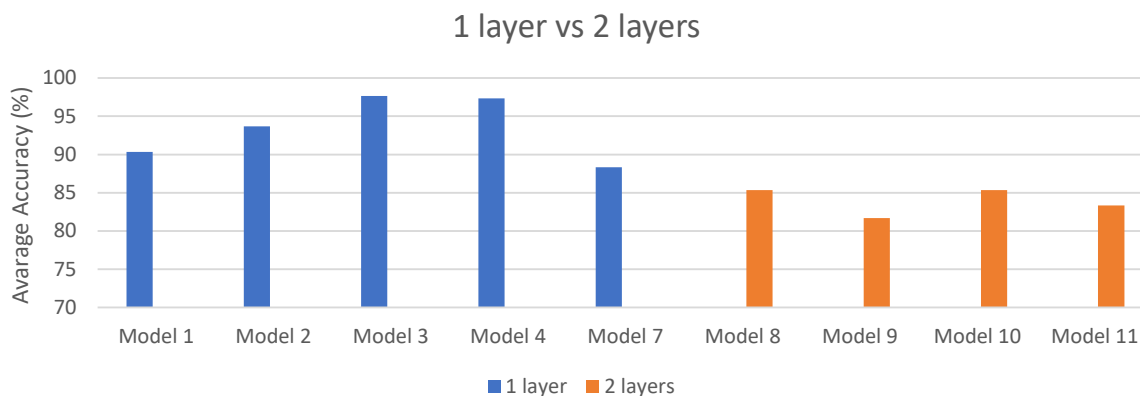


Figure 7.4: Average accuracy across the three tests for 1-hidden-layered models 1-4, and 7 compared to 2-hidden-layered models 8-11

Figure 7.4 shows that the 1-hidden-layered models 1-4, and 7 have a higher average accuracy across the three tests than the 2-hidden-layered models 8-11. Therefore, hypothesis 1: “Two hidden layers implies higher accuracy than one hidden layer” does not hold true for the models and tests analysed in this section.

As explained in [52], one hidden layer with non-linear activation functions is enough to approximate any function, but that in practice having three layers instead of two is often better. Even though that it is the case that adding depth to a NN often increase prediction accuracy these networks are also harder to train as new feature representations are created and the relationship between each of these new features and the output has to be learned [52, 65].

**Hypothesis 2:** *increasing the number of hidden neurons leads to higher average accuracy for 1-layered models.*

To test how the number of hidden neurons influences the accuracy we use the models 1-4, and 7. These models have the same configuration except the number of neurons in the hidden layer.

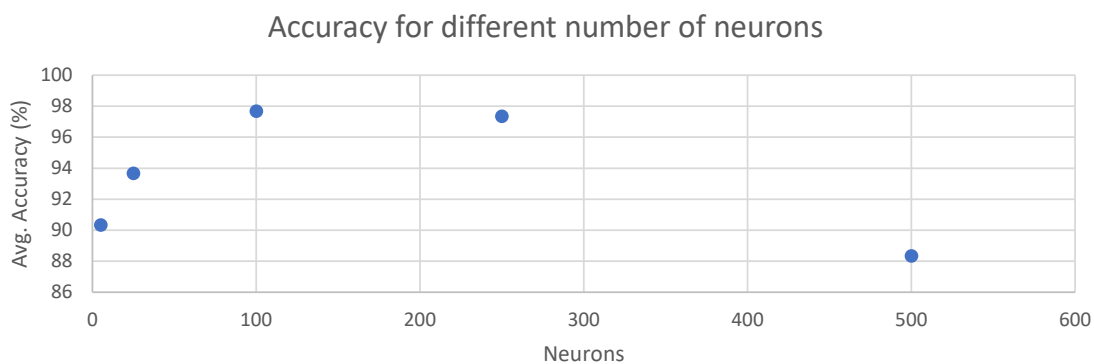


Figure 7.5: Average accuracy across the four tests with a varying number of hidden neurons

Given the results shown in figure 7.5 then the hypothesis: “More hidden neurons implies higher accuracy for 1-layered models” does not hold because the models with 250 and 500 hidden neurons

have a lower average accuracy than the model with 100 hidden neurons.

The reason why increasing the number of neurons in a hidden layer does not necessarily increase the accuracy is that even though we can model more complex functions, we are more likely to overfit the models [52].

**Hypothesis 3:** *More training data implies higher average accuracy.*

To test this hypothesis, we use test 1 and 3. Test 1 and 3 contains data for the same two players. For each of the two players test 1 has training data from three matches and test 3 has training data from one match. Given hypothesis 3 we would expect that the models would get a higher accuracy for test 1 than test 3.

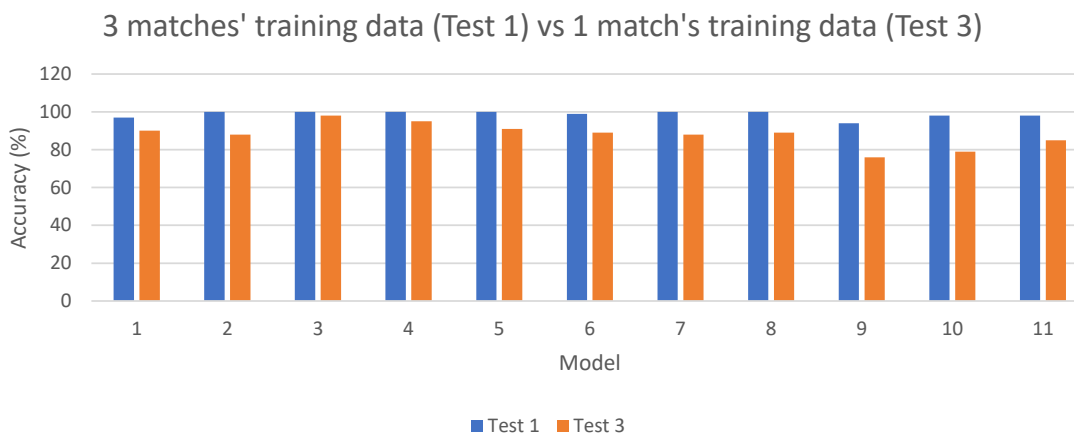


Figure 7.6: Accuracies from using test 1 (three training matches per player) and 3 on different models (one training match per player)

Figure 7.6 shows that for each model the accuracy from test 1 is higher than the accuracy from test 3. Because test 1 contains more training data than test 3 then the hypothesis holds true for these models and tests.

More training data allows the NN to perform a better generalisation of the data as it allows the NN to more accurately learn which weight values, on many different inputs, gives a low error. For this reason more training data provides a greater accuracy in our case of training a NN, except if new data e.g. has a lot of noise.

**Hypothesis 4:** *Fewer players to classify between implies higher average accuracy.*

To test this hypothesis, we use test 1 and 2. Test 1 contains data for two players and test 2 contains data for three players.



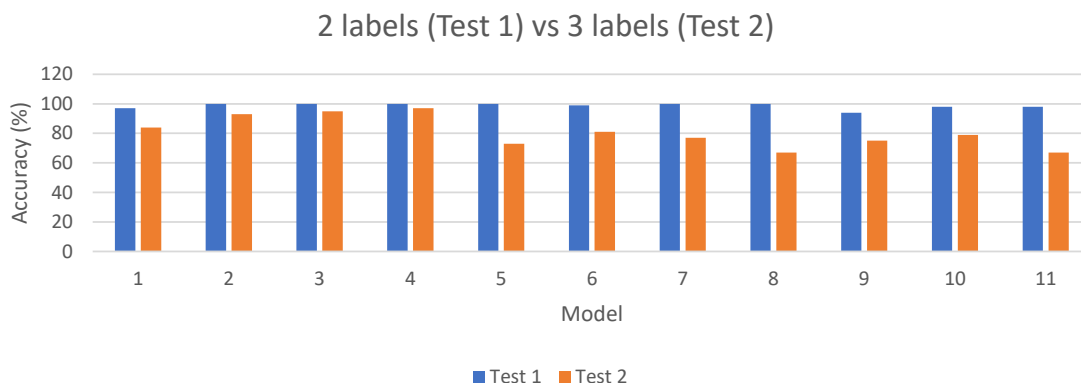


Figure 7.7: Accuracies of test 1 (two players) and test 2 (three players)

Figure 7.7 shows that for each model the accuracy from test 1 is higher than the accuracy from test 2. Because test 2 requires the model to classify between one more player than the two players in test 3 then the hypothesis holds true for these models and tests.

With fewer possible outputs (as the number of output neurons is equal to the number of players to classify between) the prediction accuracy naturally increases as there is a greater chance to ‘guess correctly’. Even for a random prediction method the accuracy will increase the fewer players are involved.

**Hypothesis 5:** *Rectified linear unit (ReLU) as activation function implies higher average accuracy compared to Sigmoid and Softmax.*

To test this hypothesis, we compare three models with the same configuration except for the activation function for the hidden layer.

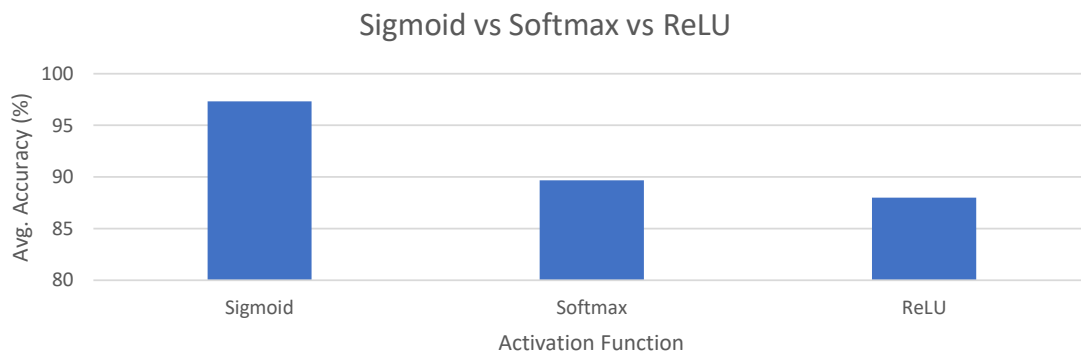


Figure 7.8: Average accuracies from tests using Sigmoid, Softmax, and ReLU

Figure 7.8 shows that the model with Sigmoid as activation function has a higher average accuracy than the models with Softmax and ReLU as the activation function. Based on the three tests then the model with ReLU as activation function has the lowest accuracy of the three activation functions compared in this test. Therefore, the hypothesis: *“Rectified linear unit (ReLU) as activation*

*function implies higher average accuracy compared to Sigmoid and Softmax*” does not hold true for these models and tests.

When applying the Softmax activation function in the hidden layer we create a linear relationship between the neurons in that layer as the outputs of each neuron sum to one [53]. Ideally we want non-linear activation function in the hidden layers of a NN as we are able to learn problems where there is a non-linear relationship between input and expected output values [50].

An explanation for ReLU actually having the worst accuracy of the three tested functions might be because of the dying ReLU problem. As ReLU ranges from 0 to infinity in a linear manner, the gradient can go towards 0, meaning some weights will not be adjusted. The more neurons that die, the more passive the network becomes [50]. As mentioned in 5.2.2, this problem can be fixed by using a modified version of ReLU called Leaky ReLU.

The reason that the Sigmoid activation function is the best performing in our case could be because we have a shallow NN where it, for our problem, does not encounter 'vanishing gradients'.

### Classifying Players Using Artificial Neural Networks

Based on the analysis and test results in this section, we can conclude that the accuracy of ANNs tested in this section depend on different factors such as the number of layers, neurons, players, and training data. Table 7.1 shows that it is possible to classify between two players with accuracies between 94-100% given three matches of training data and accuracies of 76-98% given one match of training data. Classification between three players in test 2 was done with accuracies between 67-97%. The model with the best average accuracy of 98% across the three tests is model 3 with 1 hidden layer, 100 hidden neurons + bias node, Sigmoid activation function, and 1000 training iterations.

### 7.1.3 Recurrent Neural Network Analysis

We will now analyse the performance of different RNN models in correlation to this project. To do this we will start out with designing a simple RNN model and gradually change it to see how this affects the performance of the model. This section assumes that the reader has read the chapters: data collection and data preprocessing explained in chapter 4 and 6, respectively, along with the ML theory in chapter 5.

#### Model 1

The first model we will be designing is a simple RNN with an input layer, a LSTM layer, and an output layer. The connection between layers in this model is illustrated in figure 7.9.

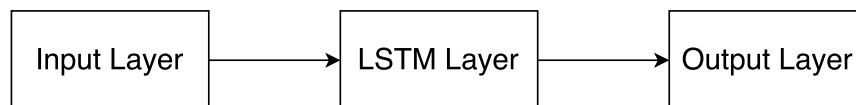


Figure 7.9: Simple LSTM model

As explained in chapter 6, the data from the sensors has been processed into a series of shots where each data point in the shot contains acceleration in x, y, z, and rotation in x. Furthermore, the Datahub API will ensure that all shots have at least a length of ten data points. To transform the data into a series of timesteps that can be used as the recurrent elements in our RNN, one approach

is to let each data point be the recurrent elements. Therefore the idea is to rescale the input-shape used for the ANN models in section 7.1.2 from  $[n, 40]$  into  $[n, 10, 4]$  where  $n$  is the number of shots, 10 is the number of timesteps and 4 is the number of features for each timestep.

To implement the LSTM model for this model we use the Keras library [66]. We will not go into detail regarding how this RNN was implemented, but the code can be found on the public repository (see reading guide) for those interested.

Table 7.2 shows the performance accuracy of the model in relation to the test suite described in 7.1.1.

Model 1	2-3 neurons	25 neurons
Test 1	99%	98%
Test 2	73%	86%
Test 3	85%	57%

Table 7.2: Accuracy for model version 1. 2-3 neurons is based on the number of labels.

As we can see from the table above the model has a good performance on the three tests and is far better than any random model (recall that test 2 has three labels while test 1 and 3 have 2). Test 1 and test 3 both have two labels, and it is notable that the accuracy of test 3 is lower than the accuracy of test 1. Looking closer at the test suites we see that test 3 has two train sets while test 1 has six train sets which indicates that an increase in training data increases the performance of the RNN. Furthermore, the results in the table show that having 25 neurons in the hidden layer instead of only a few neurons increases the accuracy which could indicate that the more output labels the more complex our model has to be. Furthermore, the results of test 3 indicate that the more complex the NN, the more training data is required to obtain good prediction accuracy.

## Model 2

The second RNN model we will be analysing is a RNN with an input layer, two LSTM layers (stacked LSTM) and an output layer. The model is visualised in figure 7.10

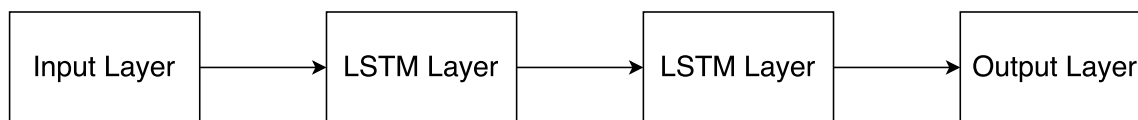


Figure 7.10: A simplified illustration of the layers in the stacked LSTM model

Similarly to adding extra hidden layers in an ANN adds extra levels of abstraction and give new feature abstractions that can help predict complicated problems, adding extra LSTM modules after each other (stacked LSTM) can be used to predict more completed sequence predictions. The effect of adding more depth to a network is explained in the following quote from [67]: "Deep learning is built around a hypothesis that a deep, hierarchical model can be exponentially more efficient at representing some functions than a shallow one [...]". Figure 7.10 shows the layout for the stacked LSTM where the output of the first LSTM layer is sequence data which is fed to the second LSTM layer.

To test the model we will run two tests one with 25 neurons in the second LSTM layer and one with 250 neurons which will both test the model but also how the model performs when we increase or

decreases the number of neurons in the second LSTM layer.

Model 2	25 neurons	250 neurons
Test 1	97%	80%
Test 2	89%	71%
Test 3	69%	57%

Table 7.3: Accuracy for model version 2

The results in table 7.3 indicate that for 25 neurons in the HLs, the stacked LSTM model has high prediction accuracy and that increasing the number of neurons does not increase the performance of the model, at least not when increasing the neurons by a factor of ten. Furthermore, when comparing this model to model 1, we see that this model performs better for test 2, but worse for test 3. Since there are only a few train sets in test 3 this indicates that a larger NN needs more training data than a smaller NN in order to get appropriate weights for making an accurate prediction which makes sense.

### Model 3

So far we have only considered RNNs where the timesteps used for the LSTM layers were data points. However, one could imagine that instead of using data points as sequence data we could instead use sequences of shots as timesteps to predict who is playing. As model 1 is the simplest RNN considered so far, we will modify the preprocessing of this model to use sequences of shots instead of sequences of data points. When requesting data from the Datahub API we receive a number of shots where each shot has at least ten data points. Similar to model 1, we will only look at the ten first data points giving each shot a total of 40 features. Now we have to turn our data into sequences of data with  $t$  timesteps. We can do so by choosing a window size,  $n$ , and then slice the original array of shots into segments of size  $n$  by taking the first  $n$  shots and grouping them into a segment then the next  $n$  shots and so on until the remaining elements are less than  $n$ . However, by doing so we assume that each window starts at a specific position, but what we really want is for our model to be able to pick up at any position and make an accurate prediction. For this reason, we will be using rolling windows instead where we given a start index,  $i$ , first will take the elements from  $i$  to  $i + n$  then from  $i + 1$  to  $i + 1 + n$  and so forth.

Model 3	window size 3	window size 10
Test 1	52%	50%
Test 2	33%	35%
Test 3	54%	51%

Table 7.4: Accuracy for model version 3

As seen in table 7.4, the model seems to perform poorly, independent of the tested window sizes, indicating that the RNN fails to capture the relationship between sequences of shots (if there is any). This means that if a RNN model is to be used, the data points which a shot consists of seem to be the best thing to use as the recurrent elements.

## Visualising the Training of Models

In this section, we will visualise the training history for some of the previously described models in order to determine why some models perform poorly and others have high accuracy. Figure 7.11 shows the training of model 3 with windows size 10 on test 1. As explained earlier, the validation set is a subset of the training set and as we can see, the accuracy for the validation set does not improve as the number of epochs (training iterations) increases which indicates that the model is being underfitted. Because the model is underfitted it fails to describe the relationship between input and output of the model which explains the poor performance of the model.

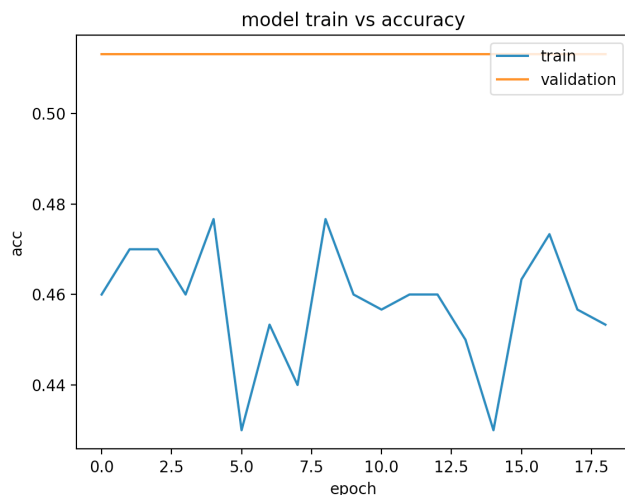


Figure 7.11: Training history for model 3 with windows size 10 on test 1 showing the models accuracy on training data and validation data for each training iteration (epoch)

On the other hand, we saw that model 1 had high accuracy for test 1. Figure 7.12 shows the training history for model 1 on test 1 and as we can see this model is far better, in fact, we see that the accuracy of the validation set and test set seem to follow closely. However, we also see that after 40 epochs the accuracy starts to differ more and more which indicates that the model is being slightly overfitted. The accuracy is still close and the model performs well on test 1 as seen earlier. Currently we try to avoid overfitting by using an early stopping function in our models where the model will stop training if it has not improved within ten epochs. To improve this we could, for instance, change the stopping criteria used or introduce dropout layers as explained in the Keras documentation [66]. However, this comes with the risk that the model will stop too early.

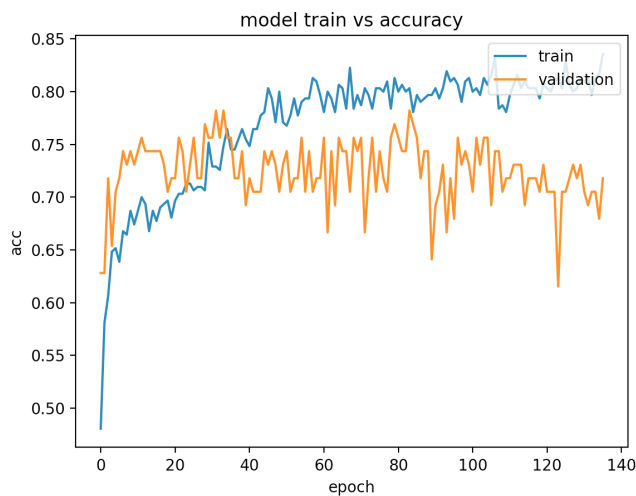


Figure 7.12: Training history for model 1 on test 1 showing the models accuracy on training data and validation data for each training iteration (epoch)

### Discussing the RNN Models

The analysis indicates that it is possible for RNN models to have high prediction accuracy when identifying table football players based on movements. Furthermore, the analysis indicates that using data points as the recurrent elements in the RNN yields a better performance than using sequences of shots. As illustrated in table 7.5, it seems that using a stacked LSTM model generally has better performance than model 1, but it is clear that this model requires more training data than having a single LSTM layer. In fact, assuming that each weight requires 8 bytes of memory, a model with 25 neurons requires a total of 3052 weights or 24kB of memory and a stacked LSTM with 25 neurons in each LSTM module requires a total of 8152 weights which is approximately 64kB of memory assuming each weight is stored as 8 bytes.

Model	Hidden layers	Max Training iterations	Window size	Has bias nodes	Activation	Neurons	$A_{test1}$	$A_{test2}$	$A_{test3}$
1	1	1000	10	True	ReLU + Softmax	nr. of labels	99%	73%	85%
1	1	1000	10	True	ReLU + Softmax	25	98%	86%	57%
2	1	1000	10	True	ReLU + Softmax	25	97%	89%	69%
2	1	1000	10	True	ReLU + Softmax	250	80%	71%	57%
3	1	1000	3	True	ReLU + Softmax	nr. of labels	52%	33%	54%
3	1	1000	10	True	ReLU + Softmax	nr. of labels	50%	35%	51%

Table 7.5: Shows the tested models and their accuracy on each test

## 7.2 Choice of Neural Network Model

In this section we will choose a specific ML model to use in the remainder of the project, based on the analysis in section 7.1.3 and 7.1.2 .

In section 7.1.2 it was concluded that the performance of ANN depends on many factors such as training data, activation function, and the structure of the ANN; e.g. how deep the network is and the number of neurons in each layer. Despite this, the analysis concluded that the best-tested model was the ANN model 3 with 100 hidden neurons + bias node and a Sigmoid activation function. Assuming that each weight requires 8 bytes of memory this model requires at least  $41 * 100 + 101 * 2 \approx 34\text{kB}$  memory. In the RNN analysis in section 7.1.3 it was found that having a stacked LSTM network with 25 neurons in the HUs had the best overall performance of the tested RNN models, but that it required a total of approximately 64kB of memory.

Table 7.2 shows the average predictions accuracy and memory consumption of the tested ANN and RNN models with the best accuracy on the performed tests. As it is shown, the ANN model has the highest prediction accuracy relative to the test suit. Another advantage of the ANN model over the RNN is that it requires less memory and that the model is simpler to implement than a LSTM model. Even though we cannot conclude that the ANN model, in general, has better performance than the RNN model, since the results depend on the chosen test suits, the findings in section 7.1.2 and 7.1.3 indicate that an ANN model with 100 hidden neurons and Sigmoid as activation function works well for identifying people based on movements.

Furthermore, the ANN model is simple to implement, relative to a RNN model, and this, in addition to consuming less memory are the reasons why we choose ANN.

---

	<b>Accuracy</b>	<b>Memory consumption</b>
RNN	85%	≈ 64kB
ANN	98%	≈ 34kB

---

Table 7.6: Table comparing the ideal RNN and ANN models. The memory consumption is based on the number of weights in the two networks where each weight is set to use 8 bytes of memory



# Chapter 8

## Embedded Classification

From section 5.2.4 we know that we can classify between two players using an ANN with an input layer of 40 neurons, a hidden layer of 100 neurons, and an output layer of 2 neurons. In this section, we will describe how we can implement a function which evaluates such a model in order to reach the second milestone of this project: *The embedded system must be able to classify which of two players is playing based on a pre-trained model.*

### 8.1 Overview of Milestone II

For milestone 2 we introduce a new architecture. An overview of which can be seen in figure 8.1.

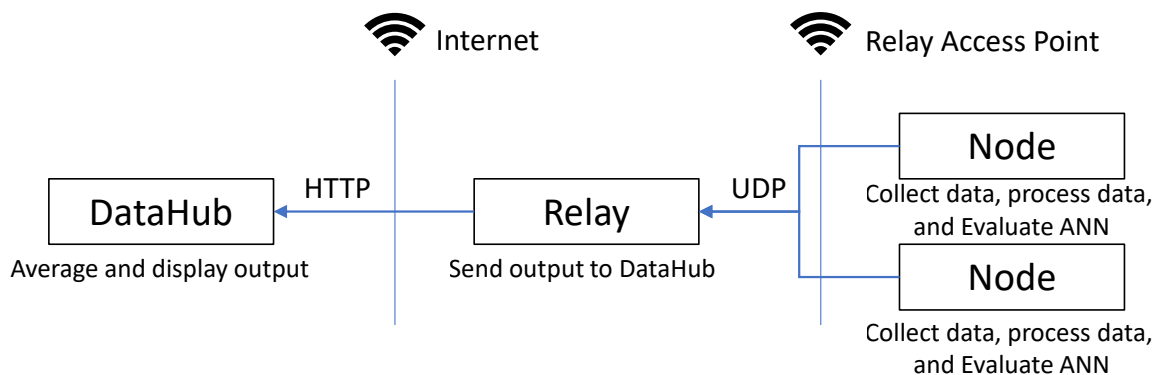


Figure 8.1: Overview of the system to be developed in the second milestone

We now want to run a NN on a NodeMCU and send it directly to the DataHub which stores and visualises the data through the relay. This means that the nodes can focus on collecting data and evaluate the network, while the relay focuses on talking to the server and collecting the results from the nodes. This is necessary since the HTTP requests to the DataHub are pretty slow since it uses TCP and works through the internet. This causes the relay to *hang* for up to hundreds of milliseconds at a time, which is unacceptable for the nodes. If the relay receives data in this time it will simply be put into a buffer and processed later. The nodes can then communicate to the relay over a private network using UDP which is fast and somewhat reliable, since the private network does not have any interference on the network itself.

## 8.2 Implementing Embedded Classification

In this section, we will describe how to implement embedded classification on the NodeMCU and show the implementation details for evaluating a trained ANN. To perform classification, we use a pre-trained model from the Encog library similar to the model selected in section 5.2.4. Figures 8.2 and 8.3 describe the software implementation of milestone 2. In figure 8.2, the main loop is shown. If input is ready, we evaluate the ANN with the input received and send the result of the evaluation as a UDP packet. Every 2ms this loop is interrupted to call the data collection method, which sets *inputReady* to true. This method is shown in figure 8.3.

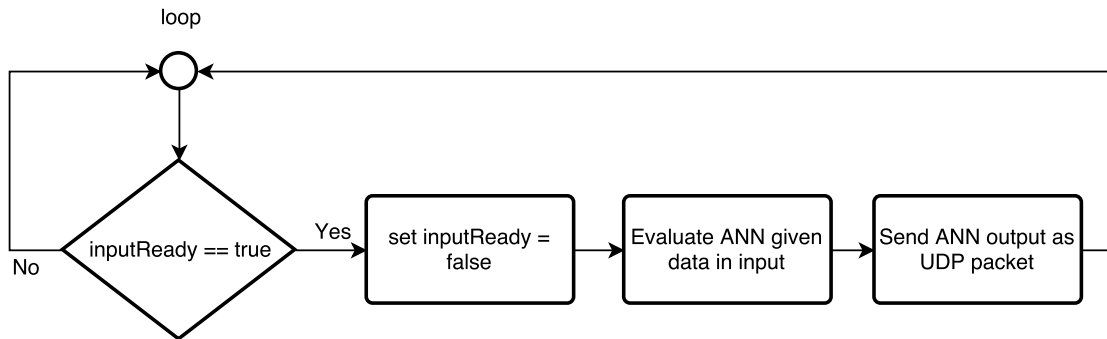


Figure 8.2: Flowchart of actions repeated in the main loop

Each 2ms, we measure data collected by the sensors. The data measured is then converted to a data point which is added to buffer1. When this buffer contains 20 data points, we take the average of these values and clear the buffer. The state *measuring movement* refers to a boolean value. If this value is set to true, which means we are actively logging shot information, we add this average data point to buffer2. When buffer2 contains ten elements, we check if the absolute value for the last ten data averaged data points is below 0.5, as this means we have seen enough movement with a rotational acceleration below a given threshold, meaning we classify the shot as finished. When a shot is finished, we send it to an *input* variable and reset buffer2.

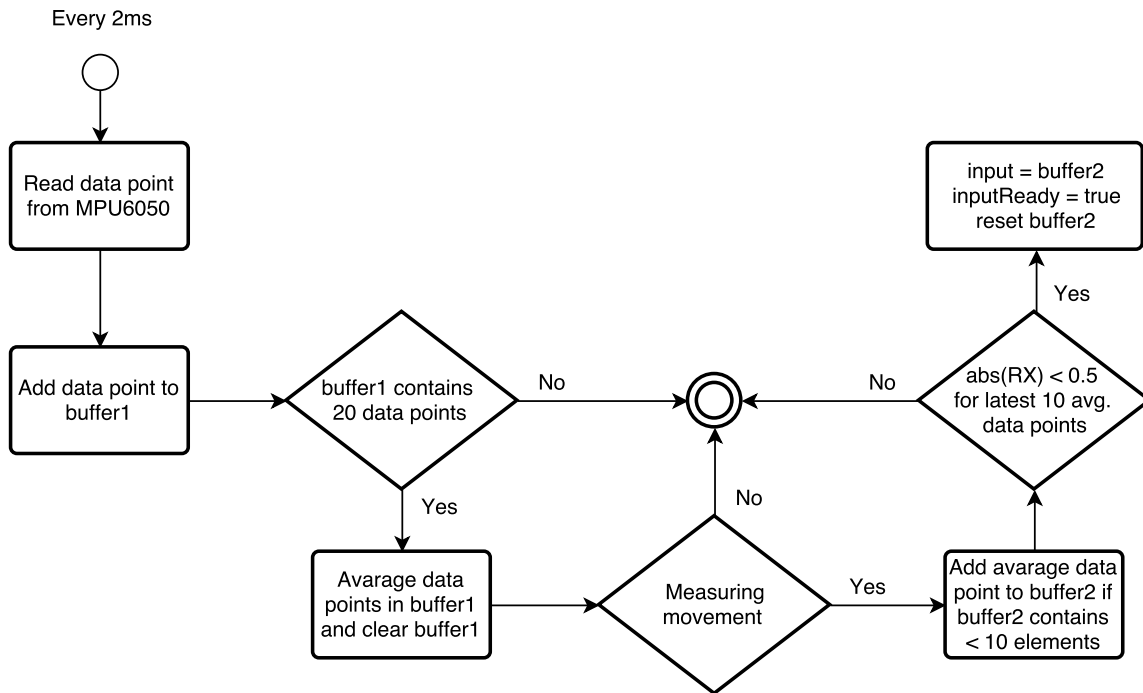


Figure 8.3: Flowchart of data collection and processing performed each 2ms

**Evaluate ANN on Embedded** To represent the data gathered from the sensors we have created a data point struct that contains fields for the three-dimensional acceleration data gathered by the accelerometer and for the three-dimensional rotation data gathered by the gyroscope. Furthermore, data structures have been created for representing nodes, layers, NN, groups of data points, and to store the result of the ANN.

To make the embedded classification a model is trained on a separate system and the weights and bias nodes of the trained model are stored and loaded into the memory on the embedded system. The method *EvaluateNetwork* in listing 8.1 receives a group of data points. In line 16 - 21 in listing 8.1 the input layer is created producing a total of 40 input neurons ( $4 \times 10$ ) in the input layer. In line 24 - 29 we calculate the output of each neuron which is our implementation of equation 5.1. The way we implement it is by looping through each layer, except the last, and for each neuron in this layer we calculate the neurons contribution (weighted input) to each neuron in the next layer which can be seen on line 27 in 8.1. After the output of each neuron in a layer has been calculated the Sigmoid activation function is applied to the output of each of the neurons and the output values in the last layer correspond to the output of the NN.

```

1 networkResult EvaluateNetwork(group g) {
2   // fill input layer
3   layer* l;
4   node* current;
5   node* next;
6   int nextNodeIndex;
7
8   // init node values to zero
9   for(l = ann.layers; l < ann.layers + ann.n_layers; l++) {

```

```

10     for(current = l->nodes; current < l->nodes+l->n_nodes; current++) {
11         current->val = 0;
12     }
13 }
14
15 l = ann.layers;
16 for(int i = 0; i < 10; i++) {
17     l->nodes[i*4 + 0].val = g.datapoints[i].X;
18     l->nodes[i*4 + 1].val = g.datapoints[i].Y;
19     l->nodes[i*4 + 2].val = g.datapoints[i].Z;
20     l->nodes[i*4 + 3].val = g.datapoints[i].RX;
21 }
22
23 // loop through layers
24 for(l = ann.layers; l < ann.layers + ann.n_layers-1; l++) {
25     for(current = l->nodes; current < l->nodes+l->n_nodes; current++) {
26         for(nextNodeIndex = 0, next = (l+1)->nodes; nextNodeIndex < (l+1)
↪ ->n_nodes; nextNodeIndex++, next++) {
27             next->val += current->val*current->weights[nextNodeIndex];
28         }
29     }
30     for(nextNodeIndex = 0, next = (l+1)->nodes; nextNodeIndex < (l+1)->
↪ n_nodes; nextNodeIndex++, next++) {
31         next->val += l->bias.weights[nextNodeIndex];
32         next->val = sigmoid(next->val);
33     }
34 }
35
36 // copy final layer node values to result vector
37 l = ann.layers+ann.n_layers-1;
38 for(int i = 0; i < l->n_nodes; i++) {
39     ann.lastResult.results[i] = l->nodes[i].val;
40 }
41
42 return ann.lastResult;
43 }

```

Listing 8.1: The EvaluateNetwork function implemented on the embedded system

## 8.3 Test of Milestone II

In the second milestone, we train a given model using the Encog framework [64]. We want to test that our implementation of the NN and the Encog framework will give similar results for equal inputs. To test this, we will feed data that was collected in milestone I, to both the Encog framework and the embedded system. The same model is then used for classification on both a computer using the Encog framework for classification like in milestone I (see section 4.6) and it is also transferred to the microcontroller.

**Purpose** The purpose is to test if the logic used to evaluate a NN on the embedded system is working correctly by comparing inputs and outputs of our implementation to that of the Encog library. We also want to test if single precision floats will give the same results as using double precision down to the precision of 1/1000th. This is done since floats use 32 bits less than doubles and since the majority of used RAM on the ESP8266 is taken up by values of type double, this will save us a lot of memory and allow us to use bigger networks or use other memory-demanding features.

**Test Setup** The test setup consists of a PC and a ESP8266, connected with a USB-cable to allow serial communication.

### Test Procedure

1. Construct a NN model using Encog with 40 input neurons, 100 hidden neurons, and 2 output neurons.
2. Construct a test dataset containing sets of 40 double precision floating point numbers.
3. Evaluate the test dataset on the ESP8266 with the constructed model using single precision floats
4. Evaluate the test dataset on the ESP8266 with the constructed model using double precision floats
5. Evaluate the test dataset in Encog with the constructed model
6. Save the output of each test to a file

**Test Result** The results of the test are shown in table 8.1 with each system in a column and nine results from each system running the same sensor-output. As we can see, using doubles is practically as accurate as Encog, since the errors are likely just rounding errors. We then repeated the test using floats instead of doubles on the ESP8266 and the results became more inaccurate, but still no more than than a  $\pm 10^{-6}$  difference, which is still accurate enough for our purposes and since we are primarily interested in which output neuron is the largest by a distinguishable amount (who the network thinks is playing), this is of no concern.

Encog output	ESP8266 using Doubles	ESP8266 using Floats
0.534176096421199	0.534176096421199	0.534176111221314
0.680407498449225	0.680407498449226	0.680407524108887
0.827025915406907	0.827025915406907	0.827025949954987
0.238695802160103	0.238695802160103	0.238695859909058
0.646675655305976	0.646675655305976	0.646675705909729
0.29626519071482	0.296265190714820	0.296265065670013
0.906369857570365	0.906369857570365	0.906369864940643
0.0610164419409884	0.0610164419409880	0.061016447842121
0.998362617607717	0.998362617607717	0.998362600803375

Table 8.1: A subset of the results of the test using a NN with 40 input neurons, 100 hidden layers . Differences are marked in **bold**

**Test Discussion** The test results in table 8.1 shows that the embedded system developed to solve milestone II is able to produce similar prediction results as the Encog library which was used in 7.1.2 to analyse different ANN models and in this milestone to train the ANN. Furthermore, the test shows that it is possible to decrease the precision of variables used for weights from double-precision to single-precision with only a small decrease in accuracy which means that in the next milestone we can decrease the memory used by the network with 50% which enables us to have a more complex NN or store more data on the ESP8266.

## Part III

# Milestone 3

# Chapter 9

## Embedded Learning

In this chapter, we will describe how we programmed an embedded system for training a NN and further used this trained NN to predict who is playing. To do this we will first describe the different components that constitute the solution for the third and final milestone of our product for identifying people based on movements in table football. Hereafter, we will describe how they interact with each other to make embedded learning and prediction. We then conduct a memory analysis of the memory required for implementing the chosen ANN model in order to determine if the ANN fits and how much memory we have left for local variables. Finally, we will describe the implementation of stochastic gradient descent to update the weights of the neural network.

### 9.1 Overview of Milestone III

In this section, the changes from milestone 2, going into milestone 3, will be described. The overview of milestone 2 can be seen in chapter 8. As seen in figure 9.1, the nodes are no longer responsible for evaluating the ANN. Instead, their only purpose is to send individual shots to the relay. The relay now has the responsibility to both train and evaluate the network. The relay can be configured by the DataHub to either train or evaluate the network, by choosing a label for each node. This means that future data the relay receives from those nodes will be used to train the NN towards that label. If no label is selected for the nodes the data which the relay receives from the sensors will simply be evaluated with the current model and the output sent on to the DataHub.

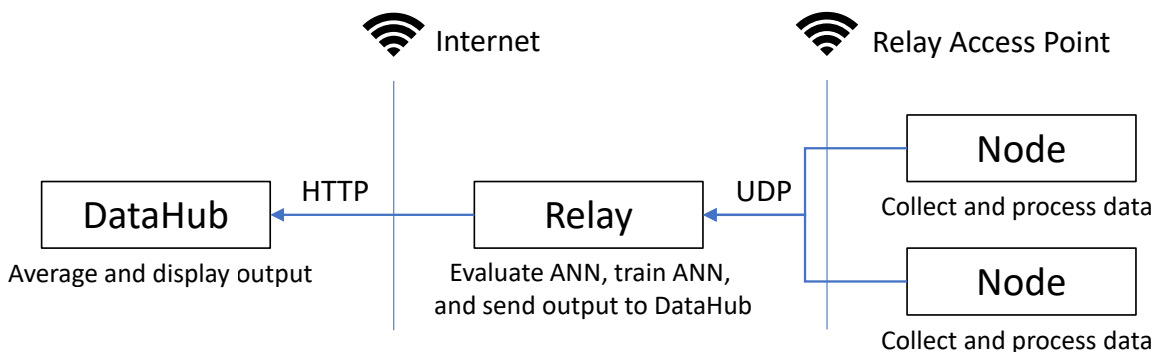


Figure 9.1: Overview of the system to be developed in the third milestone



## 9.2 Memory Analysis

Memory shortage is a concern since the device will crash if we run out of memory. On the other hand, if there is memory left over, we can use this to increase the size of our data buffers. We will do some calculations to ensure that we do not exceed the memory limit of the ESP8266.

The ESP8266 has 81920 bytes of user data RAM [68] of which only a subset is usable as dynamic heap memory. To determine how much heap we have available for our program, we can run `ESP.getfreeheap()`, which returns how many bytes are left in the heap for us to use. This function returns 46864 bytes when run as the first thing in an otherwise empty program, which means that we have about 46.6KB memory for our heap, in which we need to store a neural network and a set of input data for training and verification of the network along with other global and static variables.

In table 9.1 we have listed the major memory uses in the relay and calculated our expected heap usage and found that we should have around 11 KB left when the cache is full. An example of this is our network, which has 40 input nodes, 100 hidden nodes and two output nodes along with a couple of bias nodes, we need  $(40 + 1) \cdot 100 + (100 + 1) \cdot 2 = 4302$  floats to represent the weights, which equals to 17208 bytes.

We found that malloc has a 4 byte overhead[69] each time it is invoked, which we have included in our calculations.

In figure 9.6 we can see there's a little under 10KB heap left when the cache is full, which means our calculation is not far off. There is, however, 1.3 KB we cannot account for. We assume this is because some libraries like the network are using some heap, which we cannot control along with some small variables which we have not included in our calculation.

Memory Used for...	Heap usage in bytes
Node Input-buffer	3000
Dathub Output-buffer	1024
NN-Weights	17208
NN-nodes	2272
NN-layers	84
Training-data array	960
Training-data	9600
Shuffle buffer	640
Malloc overhead	752
RAM available	46864
Free heap	11324

Table 9.1: Heap usage of the relay

## 9.3 Implementing Embedded Learning

In figure 9.1 we presented an overview of the components used for this milestone. We will now look closer at the relay node to describe how we implemented learning on an embedded system.

Figure 9.2 shows the flow of execution in the relay. First, a setup function is called where the NN structure is created and WiFi is set up. Then the loop is entered where we first check if there is new data from one of the sensors connected to the relay's network. If there is new data the data is read from the UDP package. If no label is set for the node then it means the NN should be evaluated. To do this a forward parse of the NN is conducted and the output of the node in the last layer is saved in an output buffer. If the labels are set, the shots are inserted into the buffer created for that label. When all buffers have been filled the NN is trained. Lastly, the predictions held in the output buffer are sent to the server and displayed on the monitor page and the loop is repeated.

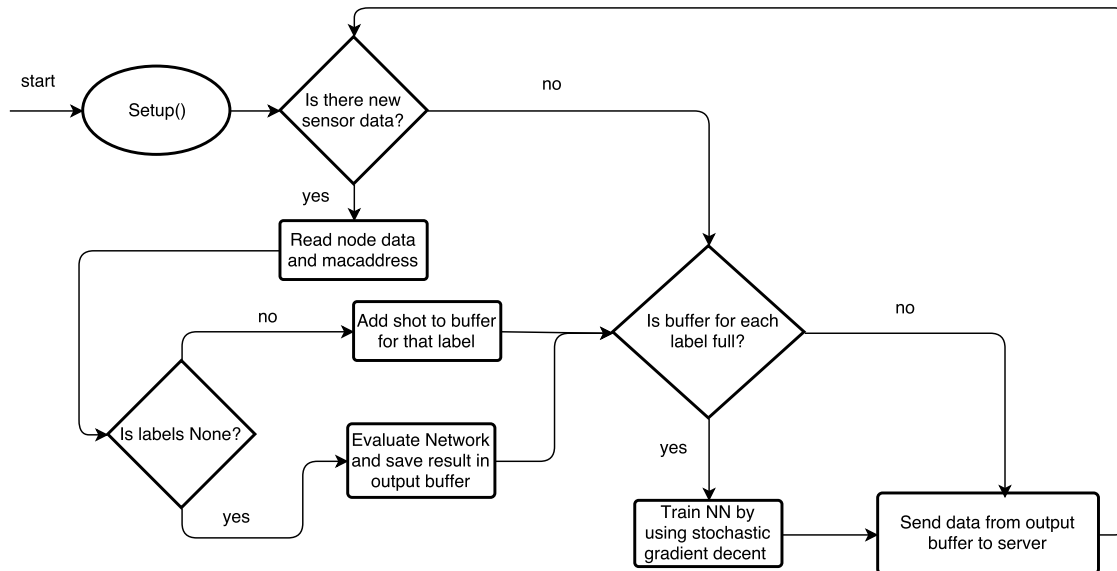


Figure 9.2: Flow diagram for relay node

To train the NN on the embedded platform we use stochastic gradient descent. Listing 9.1 shows the code for implementing stochastic gradient descent on an embedded system. The theory behind training a NN is described in section 5.3.1.

```

1 networkResult trainNetwork(network* n, example* examples, int
  ↪ n_examples){
2 networkResult res;
3 // for each example
4 for(int i = 0; i < n_examples; i++) {
5 // feed it though the network
6 res = EvaluateNetwork(n, examples[i].input);
7
8 // calculate squared errors on the output
9 _calculateOutputError(n, examples[i].output);
10
11 // back propogate error
12 _backpropogateErrorValues(n);
13
14 // update weights based on errors
15 _updateWeights(n, LEARNING_RATE);

```

```
16 }  
17  
18 return res;  
19 }
```

Listing 9.1: Source code for training the network on the embedded system

During training of the NN, we use a validation set to find out when the error has converged. We do this by splitting the data into a test set and a validation set and then calculate the mean squared error for the validation set before each training. If the training does not improve within n number of steps by at least some constant value then training is stopped and the data for each label is flushed from the buffers. If the NN is not converged we keep training and before each training, the training data is shuffled to avoid creating a bias in the training as explained in [70].

## 9.4 Interface for Embedded System

The DataHub Monitor page is the confidence-communication page of the DataHub. It has two functions. If the label is seen on the top left of each chart in figure 9.3 is set to a player, it will communicate to the relay that it should use the sensor data to train for this specific player and the chart will not update. If the label is set to 'none', this indicates to the relay that it should start predicting who is playing between the labels associated with the sensor. The chart will then begin live-updating with the average value of the confidences for each player. Whether to perform training on a person or to predict who is playing is communicated from the DataHub back to the relay through an API call. This call includes information about the MAC address and could e.g. look like what is shown in listing 9.2. This string communicates that there are two sensors (as we have one sensor for each MAC address), on the first sensor, player1 currently has a confidence of 0.2 and player2 has a confidence of 0.9.

```
"2C3AE833EC15#Player1:0.2;Player2:0.9;2C3AE8340553\#Player1:0.2;  
↪ Player3:0.4;"
```

Listing 9.2: Example of how MAC addresses are associated with labels and confidences

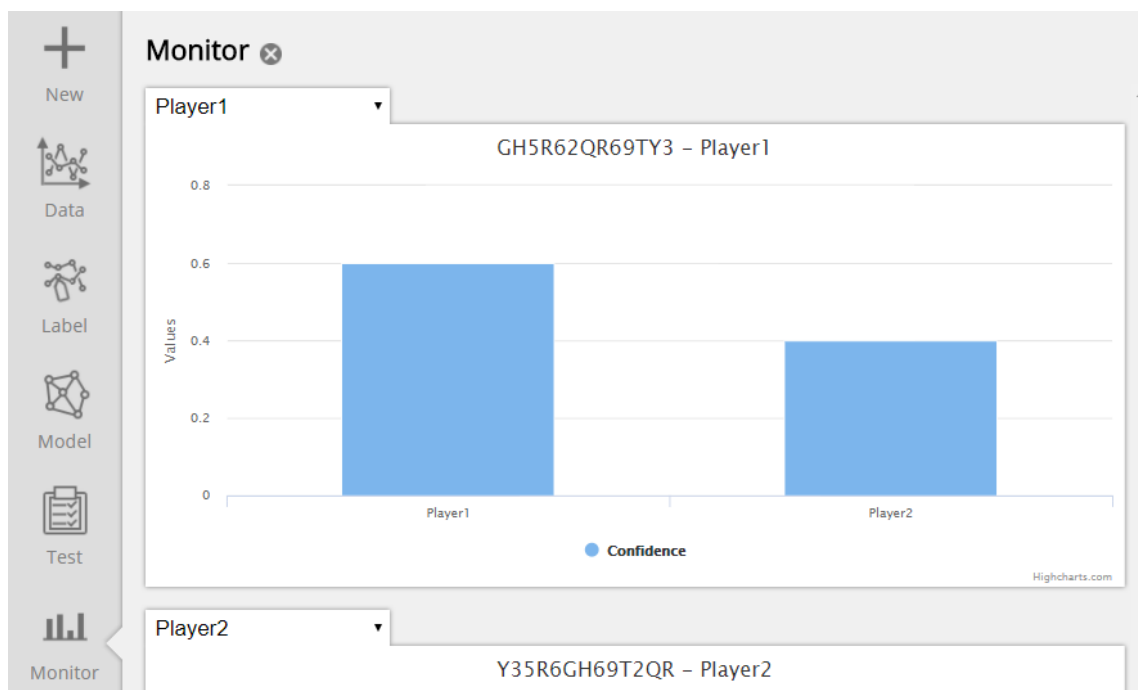


Figure 9.3: Picture of the Monitor page of the DataHub

## 9.5 Test of Milestone III

In the third prediction, we assume that we have already tested functionality from the two first milestones in 4.6 and 8.3 respectively. In addition to having this functionality tested, in this milestone, we also need to test that the embedded system is able to train the weights of the NN so the NN can be used to identify people based on movements in table football.

**Test Setup** Figure 9.4 shows the test setup used for testing the embedded system implemented for the third and final milestone (see figure 9.1 for hardware overview for milestone 3). As seen in the figure two nodes are attached to the frontmost handles of each team. Furthermore, the nodes are attached such that the sensors points in opposite direction of each other. The Relay node is connected to the computer and outputs test information such as input and out values of the ANN which is sent serially to the laptop that writes this to a log file. Furthermore, the laptop screen is recorded during an entire test including training where the DataHub monitor page and the serial output is displayed.



Figure 9.4: Test setup milestone 3

**Test Procedure** The test is performed with the following test procedure:

1. Attach node 1 to the blue team's handle and node 2 to the red team's handle (see test setup image)
2. Power on the relay, node 1, and node 2
3. Set current label of node 1 to player 1
4. Set current label of node 2 to player 2
5. Players start playing until 30 movements have been collected for each player
6. Wait until ANN training session is done (train:done appears in the console)
7. Set current label of node 1 to player 2
8. Set current label of node 2 to player 1
9. Players switch side and plays until 30 movements have been collected for each player
10. Wait until ANN training session is done (train:done appears in the console)
11. The DataHub monitor page is cleared and labels are set to none
12. Play a match to ten goals where each shot is classified
13. Reset data at the DataHub monitor page
14. Players switch side and play a match to ten goals where each shot is classified
15. Save Results and clear Datahub monitor page
16. Power off Relay, node 1 and node 2

**Test Result** A total of 3 tests were performed among three project members of the group where a test was conducted for each possible combination of the three project members playing against each other. Table 9.2 shows the group members corresponding to Player1 and Player2 of each test as well as the start time for the two matches played in each test.

Test	Player 1	Player 2	Start time (Match 1)	Start time (Match 2)
Test 1	Anton	Morten	730963ms	1538426ms
Test 2	Anton	Ibsen	696589ms	1385825ms
Test 3	Ibsen	Morten	685264ms	1079378ms

Table 9.2: The group members representing player 1 and player 2 during the three tests and the starting times for the two matches played corresponding to the time in the relay log which can be found in electronic appendix.

The results of the first and second match (step 12 and 14 in the test procedure) are shown for each test in table 9.3.

Case	Test	Match	Team	Player 1	Player 2	Classified	Actual
1	Test 1	Match 1	Blue	0,45	0,55	Player 2	Player 2
2	Test 1	Match 1	Red	0,54	0,43	Player 1	Player 1
3	Test 1	Match 2	Blue	0,56	0,43	Player 1	Player 1
4	Test 1	Match 2	Red	0,45	0,60	Player 2	Player 2
5	Test 2	Match 1	Blue	0,38	0,61	Player 2	Player 2
6	Test 2	Match 1	Red	0,60	0,41	Player 1	Player 1
7	Test 2	Match 2	Blue	0,53	0,48	Player 1	Player 1
8	Test 2	Match 2	Red	0,45	0,55	Player 2	Player 2
9	Test 3	Match 1	Blue	0,48	0,57	Player 2	Player 2
10	Test 3	Match 1	Red	0,55	0,46	Player 1	Player 1
11	Test 3	Match 2	Blue	0,64	0,46	Player 1	Player 1
12	Test 3	Match 2	Red	0,41	0,66	Player 2	Player 2

Table 9.3: The results from the three tests of the third milestone. The table shows for each sensor (one for each team) the average output of the embedded ANN given the data from the corresponding match in each of the three tests as well as the classified player (the player with the highest average output) and the actual player

To illustrate how the average is derived from individual classifications of handle movements we can illustrate the ANN output for each of the movements in case 12 of table 9.3 as shown in figure 9.5.

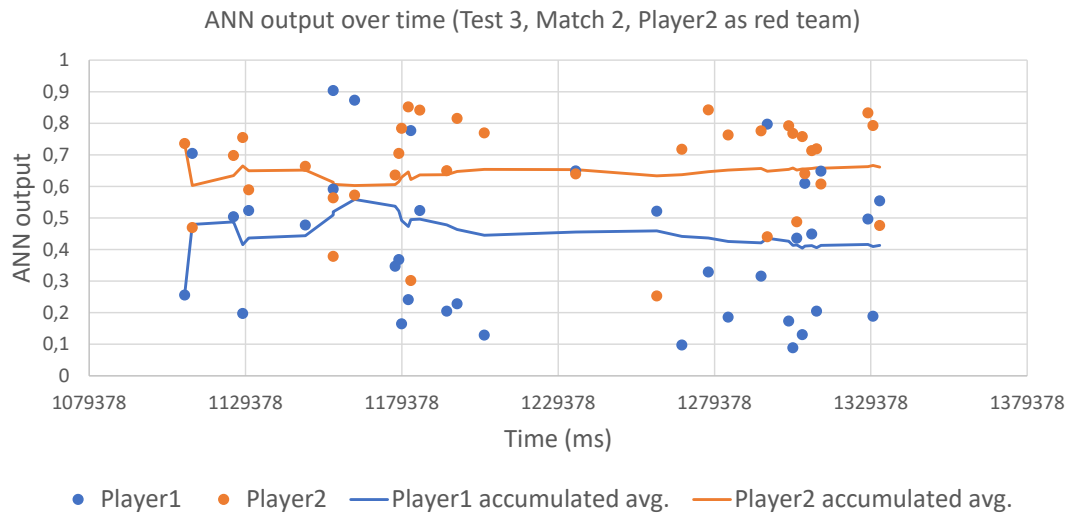


Figure 9.5: ANN output for each movement performed by player 2 and detected by the sensor attached to the red team's handle in the second match of test 3. The two lines show the current average ANN output for player 1 and player 2 since the beginning of the match. Data is parsed from the relay log file which can be found in electronic appendix.

**Heap size during test 1, 2 ,3** Furthermore we made a test to ensure that we do not run out of heap size which can be seen in figure 9.6, where we can see that at its lowest point the ESP8266 has just under 10 KB of memory on the heap left which is not far of our rough calculation of 12 KB.

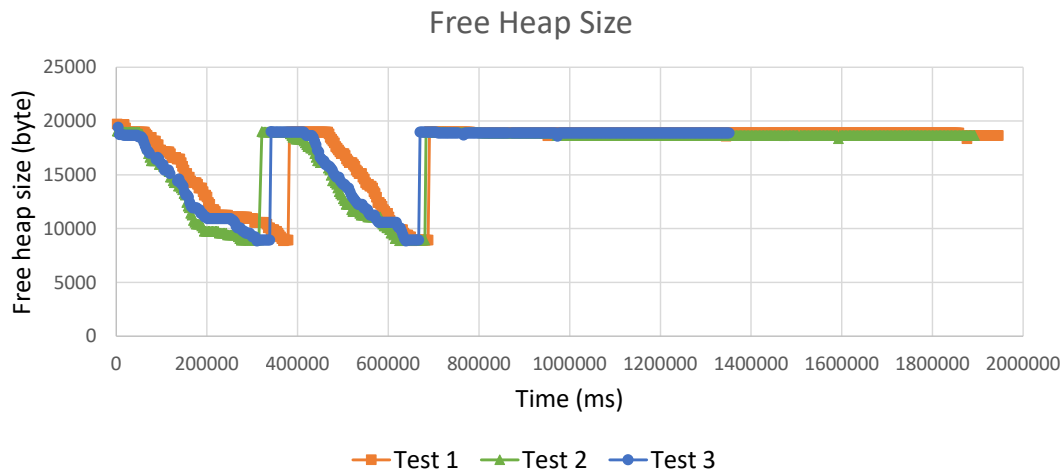


Figure 9.6: Heap size during test 1, test 2, and test 3 (see section 9.5)

**Test Discussion** As seen in table 9.3, the player of each match was classified correctly for all 12 cases based on the average output of the ANN. This shows that the model trained on the embedded system is able to evaluate data for individual movements to output values for player 1 and player 2 such that the player with the highest average output value of the entire match corresponds to the actual player. Even though the average output of the ANN classifies the correct player, there might be individual movements which are classified wrong. This can be seen in figure 9.5 where some movements performed by player 2 are classified as player 1 (Blue dot is above the orange dot at a given point in time). A reason for this can be that when the model is trained with 30 movements from each player at each side of the table then not all characteristics of each player's movements are captured in the model. Another reason can be that player 2 performs movements that are similar to the movements that the model was trained to classify as player 1. As shown in the analysis of ANNs in section 7.1.2, adding more players to classify between can decrease the accuracy of the model. In this test, we have only tested classification between two players at a time. This means that the probability of classifying the correct player by randomly selecting one of the two players is 0.5. This would be a problem if we only got a single test case as we would not be able to conclude if the model is performing better than random. Because we have 12 test cases then the probability that the model would select the correct player randomly in all twelve cases is  $0.5^{12} \approx 0.000244$ . Because this is very unlikely then we conclude that the model is performing better than random.



# Chapter 10

## Discussion

### 10.1 General Concerns

In chapter 7 we analysed different ANN and RNN models based on data collected using the embedded system created in milestone 1. During this analysis, the ANN and RNN models were implemented using two different ML libraries Encog and Keras, respectively. It is uncertain whether using different libraries would have impacted the performance of the different ML models. Furthermore, during the analysis of the ML models, we only considered a subset of possible NN structures and hyper-parameters as the purpose of the ML model was not to find an optimal model, but rather find a model that had high performance on the test suite, and was simple enough to implement, given the time frame available for this project, while also taking into account that the memory usage of the NN should be able to fit on an embedded system with low memory. It has also become clear for the project group that the results of testing different ML models, to some degree, depend on the test suite they are tested against. For this reason, it can not be concluded that ANNs is more suitable models for this particular kind of problem than RNN models as further internal testing has shown that, over multiple test suits, the prediction accuracies of our best RNN model and ANN model are similar. However, the findings of chapter 7 still hold, as the ANN models are simpler to implement than RNN models and require less memory. Another concern is that we do not know how well the chosen ANN model scales beyond three labels, as we have not had prioritised to spend time on collecting, testing, and analysing data with more than three classes. For this reason, it is, therefore, possible that a deeper NN would be more sufficient for classifying amongst more than three classes. It is also uncertain how well the model works when used to predict over more than one football match as it is possible that players gradually change playstyle. However, one can address this issue by retraining the NN, which *is* possible with the current solution.

### 10.2 Data Collection

In section 4.6 we tested the embedded system's requirements specified by milestone 1 for its ability to collect data, and concluded that several data points were lost during data collection and transmission, which could be a caused by the use of UDP. In iteration 2 and 3, the nodes responsible for collecting data were changed, so that they no longer send each data point individually, but instead grouped the data points into shots, and sent each of those as a single package to the relay node. This means that for the solution for milestone 2 and 3, data points contained within a shot is only lost during data transmission if the entire shot is lost, as the entire shot is sent as one package.

However, this is from a ML perspective preferable, as we do not lose data within a shot used for learning and predictions. Furthermore, losing a shot will not affect the training of the NN as the implemented ANN does not take into account the sequence or order in which shots appear. As such the result of losing shots is that it will take longer to fill the buffer during training, and during classification it would just mean that the particular shot, which was lost, would not contribute to the overall prediction result.

For data collection, we chose to use MPU-6050 which is a chip with accelerometer and gyroscope sensors. This choice of sensors means that we are only able to gather information about the translation and rotation of handles and not able to get the position of handles without knowing the start position and orientation of the sensors. It is uncertain how the use of other types of sensors would have affected the findings in this project and also uncertain whether or not it would have been better to use multiple types of sensors for measuring the same data. The reason the project group did not consider the use of other sensors is that we were able to determine that it was possible using the MPU-6050 to isolate movement data from the data, while also being able to make NN models, which could use this data to predict, with high accuracy, who was playing.

### 10.3 Data Processing

In chapter 6 we described how the collected data was processed into features that could be used for ML models. During the pre-processing, we down-scaled the collected data by averaging over 20 data points. However, after analysing the frequency domain of the data in section 4.5, we concluded, by using the Nyquist-Shannon sampling theorem; taking the average of 20 data points could mean that some information of a shot would be lost and that we probably should have chosen to average over fewer data points. However, the reason for not changing this approach, is that it was discovered after the project group already had analysed and determined, that it was possible to use the current approach of averaging over 20 data points, and create models that could, with a high accuracy, identify among two or three persons playing table football. During data processing data were also grouped into shots based on an algorithm developed by the project group which is described in chapter 6. However, it is likely that this algorithm does not capture all shots or that it captures information about other movements than 'shots'. Another approach we could have taken was to also include data about movements, i.e. not only take data points where the rotation exceeds some threshold, but also take into account movements without rotation. Furthermore, the current approach only consider the first ten average data points of each shot and use this data to make a classification meaning that if a shot contains more than ten average data points, this information is not used. The reason for choosing this approach is that NN models need a fixed number of input features, however, one could instead have used other approaches for making features, where all the data points would have been taken into account, like using statistic features. However, we are unsure if this approach would have been better for the final and chosen model implemented in chapter 9.

### 10.4 Embedded Learning and Classification

In chapter 9 we implemented an ANN on an embedded system for learning and classifying people, based on movements in table football. The chosen model is based on the NN models analysed in chapter 7 and tries to follow the same approach, used for these models. However, due to memory limitations on the embedded system, we had to limit the amount of data that could be stored in memory, during training. To accommodate for this, we designed the training of the NN, such that it is possible to train the NN multiple times with new data. Furthermore, we chose to use the

Sigmoid instead of using a combination of ReLU and Softmax for activation function. The reason for this is that we analysed that the Sigmoid function worked well for our problem.

As described in section 9.5 we tested the third and final milestone of the embedded system on every combination of three project members, and during each test the model had to classify between two of the group members. The test showed, that the system was able to correctly identify amongst the project members in each of the three tests, giving a total of 12 correctly predicted cases. However, ideally we would prefer to test the system amongst more than two players at a given time, as it is unsure how well the current solution works for more than two players. Additional tests were not conducted due to time constraints as it currently takes a lot of time to set up the embedded system, analyse the results, and document it.

# Chapter 11

## Conclusion

In this project we have, based on analysis of different ML models, designed and implemented an embedded system for training an ANN with the purpose of identifying players amongst a set of two players playing table football. This system was implemented as an answer to the problem:

**Problem statement** *How can an embedded system be implemented to identify individuals based on the movement of handles in a table football game using machine learning?*

In order to answer the problem statement, we devised four research questions (see chapter 3). We will now describe how we solved each of these research questions by answering them separately before making the final conclusion.

**Research question 1** *How can we collect data about movements of handles in a table football game?*

We designed an embedded system for collecting data in chapter 4. The system consists of two 3D printed *boxes* which each contain batteries, a NodeMCU, and a MPU-6050. Every two milliseconds, the NodeMCU processor reads acceleration and rotation of the three axes: x,y, and z, and groups this data into what we call a data point. Based on the gathered data points, the NodeMCU is responsible for grouping the data points into shots and sending the shots via UDP to another NodeMCU module (the relay). When collecting data, each of these boxes were attached to the football table as described in the test setup in section 9.5.

**Research question 2** *How can data be processed to use it for training machine learning models to identify people playing table football?*

This research question is answered in chapter 6, by first down-scaling the data, by taking an average over 20 data points and then grouping the down-scaled data into shots, based on a threshold value for the rotation in the x-axis. For each shot, the first ten data points are extracted, and from each data point the acceleration in x, y, and z and the rotation in x is extracted, giving a total of 40 features. These features are then used for training a ML model for classifying people, based on movements in table football.

**Research question 3** *Which machine learning models can be used to identify players?*

To answer this research question we designed and tested different machine learning models in chapter 7, and found that an ANN model with 40 input neurons, a hidden layer with 100 neurons, bias nodes, Sigmoid activation function, can correctly classify all 12 cases based on the average output of the ANN for a given match. During classification of shots then some of these may be classified wrong, but the overall average is correct in the 12 cases seen in table 9.3.

**Research question 4** *How can such a machine learning model be implemented on an embedded system?*

We answered this final research question by creating an embedded system consisting of two data collection nodes, one relay node, and a front-end web application with an API (DataHub). The data collection node gathers data from the gyroscope and accelerometer sensor, groups the data into shots, and sends the shots to a relay node. On the relay node, we implemented the previously described ANN model and trained the model by storing shots for each player in separate buffers. When the buffers are filled, the shots are separated into training and validation sets, and then the ANN is trained on the training set until the error on the validation set does not drop by at least some constant value for five training iterations. The buffers are then flushed, and the process is started over until an API response from the DataHub indicates, that training should stop. When training has been stopped, the relay node will make a forward pass of the trained ANN for each shot it receives from the data collection nodes. Each forward pass will yield an output value for each node in the output layer (one output neuron for each player). The output values of each forward pass is sent to the DataHub via HTTP, and will also contain the MAC address for the data collection node from which the data was collected. The DataHub then keeps track of which data belongs to which data collection node, and accumulates and averages the data for each node and displays the results for each node on the DataHub monitor page.

As discussed in chapter 10, we have developed an embedded system for identifying people based on movements. The system has been tested on three project members of the group and was able to accurately identify amongst the three members in all of the 12 cases. The probability for a random model to gain the same results is  $0.5^{12}$ , which is approximately 0.00024 or 0.024%, making it unlikely that these results were made by a random model.

We chose to implement the ML model on the embedded system as a challenge to ourselves, as it would give us the opportunity to learn how to implement an ANN from scratch, beyond what could simply have been learned using libraries.

In conclusion, we have designed and implemented an embedded system for classifying amongst two players playing football table. The embedded system is able to train and predict players based on movements, and the prediction results can be displayed live on the DataHub monitor page. We can, therefore, confirm that it is possible to classify between people based on their movements in table football.

# Bibliography

- [1] Hadid et al. *FACE AND EYE DETECTION FOR PERSON AUTHENTICATION IN MOBILE PHONES*. English. <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4357512&tag=1>.
- [2] Collins Dictionary. *Definition of 'Movement'*. English. <https://www.collinsdictionary.com/dictionary/english/movement>.
- [3] Dictionary.com. *table football*. <http://www.dictionary.com/browse/table-football>.
- [4] BRITISH FOOSBALL ASSOCIATION. *RULES OF THE GAME*. <https://britfoos.com/guides/rules/>.
- [5] Aalborg Universitet. *Studieordning for Bacheloruddannelsen i software*. Danish. [http://www.sict.aau.dk/digitalAssets/203/203759\\_software-bachelor-e16.pdf](http://www.sict.aau.dk/digitalAssets/203/203759_software-bachelor-e16.pdf). 2016.
- [6] InvenSense Inc. *MPU-6000 and MPU-6050 Product Specification Revision 3.4*. English. <https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf>. 2013.
- [7] ebay. *MPU-6050 6DOF 3 Axis Gyroscope+Accelerometer Module for Arduino DIY*. English. <https://www.ebay.com/itm/MPU-6050-6DOF-3-Axis-Gyroscope-Accelerometer-Module-for-Arduino-DIY/201415045005?hash=item2ee545af8d:g:OSwAAOSwDwtUm4WP>.
- [8] ebay. *5Pcs 75mm Double Original A10K Behringer Fader Straight Sliding Potentiometer*. English. <https://www.ebay.com/itm/5Pcs-75mm-Double-Original-A10K-Behringer-Fader-Straight-Sliding-Potentiometer/232366561765?epid=20002809537&hash=item361a2059e5:g:C6IAAOSwXY5Z01uD>.
- [9] Danielle Collins. *FAQ: What types of linear encoders are there and how do I choose?* English. <http://www.motioncontroltips.com/faq-what-types-of-linear-encoders-are-there-and-how-do-i-choose/>. 2015.
- [10] ebay. *10pcs TCRT5000L TCRT5000 Reflective Photoelectric Switch Infrared Optical Sensor*. English. <https://www.ebay.com/itm/10pcs-TCRT5000L-TCRT5000-Reflective-Photoelectric-Switch-Infrared-Optical-Sensor/381375105733?epid=1231476045&hash=item58cbba4ac5:g:yQEAMXQleBT1wnc>.
- [11] ebay. *5pcs-10K-OHM-3-Terminal-Linear-Taper-Rotary-Audio-B-Type-Potentiometer*. English. <https://www.ebay.com/itm/5pcs-10K-OHM-3-Terminal-Linear-Taper-Rotary-Audio-B-Type-Potentiometer/172618234352?epid=2113996503&hash=item2830d939f0:g:-6sAAOSwLZY6iOA>. 2017.
- [12] Vishay Semiconductors. *Reflective Optical Sensor with Transistor Output*. English. <https://www.vishay.com/docs/83760/tcrt5000.pdf>. 2017.
- [13] Arduino. *What is Arduino?* English. <https://www.arduino.cc/en/Guide/Introduction>. 2017.
- [14] Arduino. *Arduino Products*. English. <https://www.arduino.cc/en/Main/Products>. 2017.
- [15] Arduino. *Arduino Nano*. English. <https://store.arduino.cc/arduino-nano>. 2017.
- [16] Morten Rask Andersen et al. "Tang: A Programming Language for Arduino". English. In: (2017).
- [17] Atmel. *ATmega48A/PA/88A/PA/168A/PA/328/P*. English. [http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P\\_datasheet\\_Complete.pdf](http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf). 2015.

- [18] NodeMcu Team. *NodeMcu*. English. [http://www.nodemcu.com/index\\_en.html](http://www.nodemcu.com/index_en.html). 2017.
- [19] Espressif. *ESP8266EX Datasheet Version 5.6*. English. [http://espressif.com/sites/default/files/documentation/0a-esp8266ex\\_datasheet\\_en.pdf](http://espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf). 2017.
- [20] AI Thinker. *ESP-12E WiFi Module*. English. <http://www.kloppenborg.net/images/blog/esp8266/esp8266-esp12e-specs.pdf>. 2015.
- [21] NodeMCU Team. *nodemcu-devkit-v1.0*. English. [https://github.com/nodemcu/nodemcu-devkit-v1.0/blob/master/NODEMCU\\_DEVKIT\\_V1.0.PDF](https://github.com/nodemcu/nodemcu-devkit-v1.0/blob/master/NODEMCU_DEVKIT_V1.0.PDF). 2017.
- [22] RASPBERRY PI FOUNDATION. *RASPBERRY PI*. English. <https://www.raspberrypi.org/>. 2017.
- [23] lady ada. *Introducing the Raspberry Pi Zero*. English. <https://cdn-learn.adafruit.com/downloads/pdf/introducing-the-raspberry-pi-zero.pdf>. 2017.
- [24] RASPBERRY PI FOUNDATION. *RASPBERRY PI ZERO W*. English. <https://www.raspberrypi.org/products/raspberry-pi-zero-w/>. 2017.
- [25] Lifehacker. *The best operating system for your Raspberry Pi projects*. <https://lifehacker.com/the-best-operating-systems-for-your-raspberry-pi-projec-1774669829>.
- [26] Ebay-user worldchips. *Nano V3.0 CH340G/FT232 ATmega328P MINI USB 5V 16M Micro-controller Board Arduino (CH340G DIY Kits no Cable)*. <https://www.ebay.com/itm/Nano-V3-0-CH340G-FT232-ATmega328P-MINI-USB-5V-16M-Micro-controller-Board-Arduino/322368047776?hash=item4b0ea20aa0:m:mmqB9TXZX04CrH1sXf-wqKg>.
- [27] avrProgrammers. *ATmega328p Power Consumption*. <https://www.avrprogrammers.com/howto/atmega328-power>.
- [28] Ebay-user diybox. *ESP8266 ESP-12E CH340G WIFI Network Development Board Module For NodeMcu Lua*. <https://www.ebay.com/itm/ESP8266-ESP-12E-CH340G-WIFI-Network-Development-Board-Module-For-NodeMcu-Lua/112230225390?epid=581455279&hash=item1a217155ee:g:ShQAAOSwImRYSTVS>.
- [29] Espressif Kelly. *ESP8266 Power Consumption*. <http://bbs.espressif.com/viewtopic.php?t=133>.
- [30] RaspberryPi.dk. *Raspberry Pi Zero W (Wireless)*. <https://raspberrypi.dk/produkt/raspberry-pi-zero-w/>.
- [31] RasPi.TV. *How much power does Pi Zero W use?* <http://raspi.tv/2017/how-much-power-does-pi-zero-w-use>.
- [32] Diffen. *TCP vs. UDP*. [https://www.diffen.com/difference/TCP\\_vs\\_UDP](https://www.diffen.com/difference/TCP_vs_UDP).
- [33] Numpy developers. *numPy mathematical library for python*. <http://www.numpy.org/>.
- [34] Bruno A. Olshausen. "Aliasing". English. In: (2000).
- [35] Kurt T. Manal and Thomas S. Buchanan. *BIOMECHANICS OF HUMAN MOVEMENT, CHAPTER 5*. English. [http://www.unhas.ac.id/tahir/BAHAN-KULIAH/BIO-MEDICAL/NEW/HANBOOK/0071449337\\_ar005-Biomechanics\\_Of\\_Human\\_Movement.pdf](http://www.unhas.ac.id/tahir/BAHAN-KULIAH/BIO-MEDICAL/NEW/HANBOOK/0071449337_ar005-Biomechanics_Of_Human_Movement.pdf). 2014.
- [36] Vernier. *Vernier - Logger Pro*. <https://www.vernier.com/products/software/lp/>.
- [37] Bernard Marr. *What Is The Difference Between Artificial Intelligence And Machine Learning?* English. <https://www.forbes.com/sites/bernardmarr/2016/12/06/what-is-the-difference-between-artificial-intelligence-and-machine-learning/#1a9e48e52742>. 2016.
- [38] Ciro Donalek. "Supervised and Unsupervised Learning". English. In: (2011).
- [39] John A. Bullinaria. *Bias and Variance, Under-Fitting and Over-Fitting*. English. <http://www.cs.bham.ac.uk/~jxb/INC/19.pdf>. 2015.

- [40] Scott Fortmann Roe. *Understanding the Bias-Variance Tradeoff*. English. <http://scott.fortmann-roe.com/docs/BiasVariance.html>. 2012.
- [41] Shimon Ullman et al. “Unsupervised learning Clustering”. English. In: (2014).
- [42] Standofrdl. *Unsupervised Learning Clustering*. English. <https://lagunita.stanford.edu/c4x/HumanitiesScience/StatLearning/asset/unsupervised.pdf>.
- [43] David Olson. *Advanced data mining techniques*. Berlin: Springer, 2008. ISBN: 978-3-540-76916-3.
- [44] Kristin Bennett and Colin Campbell. “Support Vector Machines: Hype or Hallelujah?” In: *SIGKDD Explorations* 2 (2000).
- [45] Mahbubur Syed. *Handbook of Research on Modern Systems Analysis and Design Technologies and Applications*. Hershey, PA: Information Science Reference, 2009. ISBN: 978-1-59904-887-1.
- [46] Sunil Ray. *Understanding Support Vector Machine algorithm from examples (along with code)*. <https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/>.
- [47] MINDS@UW. *Neural Networks*. [https://minds.wisconsin.edu/bitstream/handle/1793/7779/ch1\\_3.pdf](https://minds.wisconsin.edu/bitstream/handle/1793/7779/ch1_3.pdf).
- [48] David L. Poole and Alan K. Mackworth. *Artificial Intelligence - Foundations of Computational Agents*. English. Second Edition. Cambridge University Press. ISBN: 9781107195394.
- [49] Aditya Sharma. *Understanding Activation Functions in Deep Learning*. English. <https://www.learnopencv.com/understanding-activation-functions-in-deep-learning/>.
- [50] Avinash Sharma V. *Understanding Activation Functions in Neural Networks*. English. <https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>.
- [51] Michael Nielsen. *Using neural nets to recognize handwritten digits*. English. <http://neuralnetworksanddeeplearning.com/chap1.html>. 2017.
- [52] Stanford. *CS231n Convolutional Neural Networks for Visual Recognition*. <http://cs231n.github.io/neural-networks-1/#actfun>.
- [53] Michael Nielsen. *Improving the way neural networks learn*. English. <http://neuralnetworksanddeeplearning.com/chap3.html>. 2017.
- [54] Michael Nielsen. *How the backpropagation algorithm works*. English. <http://neuralnetworksanddeeplearning.com/chap2.html>. 2017.
- [55] David Poole. *Artificial intelligence : foundations of computational agents*. Cambridge, United Kingdom: Cambridge University Press, 2017. ISBN: 9781107195394.
- [56] Matt Mazur. *A Step by Step Backpropagation Example*. <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example>.
- [57] José M. Vidal. *Incremental (Stochastic) Gradient Descent*. <http://jmvidal.cse.sc.edu/talks/ann/stockgd.html?style=White>.
- [58] Jozefowicz et al. “An Empirical Exploration of Recurrent Network Architectures”. English. In: (2015).
- [59] Denny Britz. *Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs*. English. <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>. 2015.
- [60] Pascanu et al. “On the difficulty of training recurrent neural networks”. English. In: (2013).
- [61] Deeplearning4j Development Team. *A Beginner’s Guide to Recurrent Networks and LSTMs*. English. <https://deeplearning4j.org/lstm.html>.



- 
- [62] Christopher Olah. *Understanding LSTM Networks*. English. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>.
- [63] Highcharts. *Highcharts*. [www.highcharts.com](http://www.highcharts.com).
- [64] Jeff Heaton. “Encog: Library of Interchangeable Machine Learning Models for Java and C#”. In: *Journal of Machine Learning Research* 16 (2015), pp. 1243–1247. URL: <http://jmlr.org/papers/v16/heaton15a.html>.
- [65] Adriana Romero et al. “FITNETS: HINTS FOR THIN DEEP NETS”. English. In: (2015).
- [66] François Chollet et al. *Keras*. <https://github.com/fchollet/keras>. 2015.
- [67] Razvan Pascanu et al. *How to Construct Deep Recurrent Neural Networks*. <https://arxiv.org/pdf/1312.6026.pdf>.
- [68] Max Filippov. *Memory Map of ESP8266*. <https://github.com/esp8266/esp8266/wiki/wiki/Memory-Map>.
- [69] Unknown. *umm\_malloc - Memory Manager For Small(ish) Microprocessors*. [https://github.com/esp8266/Arduino/tree/master/cores/esp8266/umm\\_malloc](https://github.com/esp8266/Arduino/tree/master/cores/esp8266/umm_malloc).
- [70] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. <http://ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent>.