Delft University of Technology
Master of Science Thesis in Embedded Systems

Real-time Ball-touch Classification on an Insole Sensor using Neural Networks

Jacobus Habte

Embedded Networked Systems

Delft University of Technology

Real-time Ball-touch Classification on an Insole Sensor using Neural Networks

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Jacobus Habte
j.habte@student.tudelft.nl
kooshabte@live.nl

1 June 2022

Author
Jacobus Habte (j.habte@student.tudelft.nl)
(kooshabte@live.nl)

Title
Real-time Ball-touch Classification on an Insole Sensor using Neural Networks

MSc Presentation Date
14 June 2022

Abstract

Monitoring and analyzing human movement is used in many fields, ranging from healthcare and industrial applications to sports analytics. To provide a football player or their coach with insight into their performance during a game, or their technical development over time, many methods are available such as camera setups and smart vests. However, to provide a more direct and biomechanical approach, this thesis utilizes sensors placed in the insoles of a player's shoe. These sensors are equipped with an Inertial Measurement Unit (IMU), providing very direct insight into the accelerations and rotations of the player's feet. Additionally, external impact on the feet can be detected, such as after a jump, a tackle by an opponent or a ball touch. To provide the player with information about covered distance, speed and ball touches and possibly many more metrics, the raw data from the sensor needs to be processed. Various methods can be used to provide the different metrics, ranging from custom algorithms to Machine Learning approaches.

This thesis provides an approach using neural networks to detect ball touches in real-time on these insole sensors. This approach combines the challenge to classify the ball touches correctly with the additional challenge to do so on a constrained embedded device, an Arm Cortex-M4 microcontroller. A development flow is set up using the TensorFlow framework. Recorded data can be used to develop and train a neural network model, after which the model is converted and optimized to be deployed to the sensor. Several neural network models and settings have been benchmarked both on classification performance and resource usage. The final approach to this classification task samples the data from the IMU at 500 Hz, detects ball-touch candidates and creates a window of samples, which is then fed to a one-layer Convolutional Neural Network (CNN) model. This model achieves an accuracy of 95.8%, while the implementation on the sensor uses 64.9 KB Flash, 5.0 KB RAM and has an execution time of 8 ms per classification.

Preface

This thesis marks the end of my years of study in Delft, both at The Hague University of Applied Sciences and Delft University of Technology. I have enjoyed studying both Electrical Engineering and Embedded Systems, which led me through many different study, internship and work experiences. Applying Embedded Machine Learning in my thesis, provides an interesting end to these years, as it is a rapidly developing field with many promising applications. I am grateful for all good memories, and also for the challenges, which show that one shouldn't rely too much on their own strength. Finally, I am looking forward to put everything I have learned to good use in my future career.

I would like to thank my colleagues at JOGO, and especially Michaël, for the introduction to and support with my exploration of Embedded Machine Learning. Also many thanks to my thesis supervisor, Koen Langendoen, who pointed me in the right direction when I lost overview and provided countless helpful comments on my thesis.

Furthermore, I want to thank my friends, for showing interest in my work, joining me for lunch walks during Corona lockdowns and enabling me to relax after a week of hard work. My girlfriend, for supporting me during my whole studies, in different ways throughout my bachelor and master. And finally, my parents, to whom I am grateful for encouraging and motivating me unconditionally throughout all these years.

Koos Habte

Delft, The Netherlands
1st June 2022

Contents

Chapter 1

Introduction

Classifying human movement is a challenge encountered in many forms. As humans, we subconsciously classify someone who runs as going slow or fast. When we see someone lift an object, we estimate how heavy the object is. While these classifications are often quite coarse, they give an indication of the activity.

However, often the human classification capabilities are not sufficient. Imagine trying to monitor the actions of all twenty-two players during a game of football. Some interesting metrics would be covered distance, maximum sprinting speed, leg distribution and the number of different kinds of ball touches. Impossible for a human to keep track of. This challenge can be partly solved by recording the game and applying automated video analysis. However, this still does not give the detailed insight into an individual player's performance we would expect from modern-day technology.

A better approach would be to be able to detect a person's movements, not based on a fixed camera setup, but using a wearable device. A common way to detect human motion with a wearable, is using an Inertial Measurement Unit (IMU). Such a wearable, consisting of sensors that detect accelerations and rotations, could give a more detailed insight into the biomechanical behaviour of a human. This is applied to our use case of monitoring football players. The gathered data can provide information about their current performance and can be used to monitor their gait to prevent injuries. The same technology could be applied in industrial applications to monitor a mechanic's physical status and in healthcare to monitor elderly people or to provide long-term monitoring during a medical rehabilitation process.

1.1 Context

JOGO[1], a Dutch startup focused on football player development, has taken up the challenge to build a sensor that can provide insight into the performance of a football player during a training session or a match. This sensor is to be placed in the shoe insole of a player and contains an IMU to measure the movements of the foot. Using this approach, the acquired measurements can provide information about each individual player. The sensor should be able to

[1]`https://www.jogo.ai/`

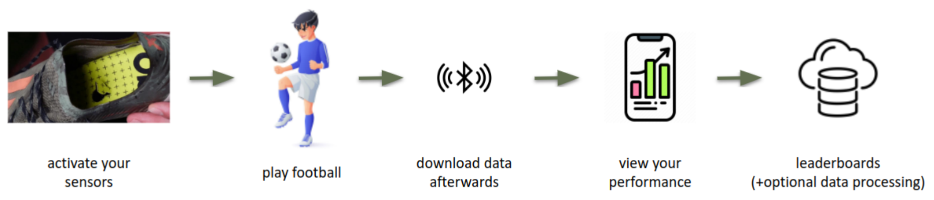


Figure 1.1: **Usage flow of the JOGO sensor.**

provide metrics such as covered distance, speed, acceleration, ball touches and more. Additionally, smart features such as automatic starting and stopping of a session need to be implemented.

To view and analyze the results of a session, a smartphone with the JOGO app is required. Many use cases can be realized: showing the results of your most recent session, comparing your performance over time and even competing with others through online leaderboards. The user flow, illustrated in Figure 1.1, is as follows:

1. Activate the sensors, either manually through the app or with an optional automatic start algorithm.
2. Play football.
3. Download the data from the sensor to the smartphone after the session is stopped.
4. Display the player's performance in the app.
5. Upload the data to the cloud, for further cumulative statistics and leaderboards.

An important part of the user experience is that the sensor should be able to store multiple match or training sessions without the need to download them immediately, and secondly, when the user decides to download the data, the download time should not be too long. An analysis of the user experience, as described in Section 3.1, reveals that the available storage and download time are limiting factors. When storing raw sensor data on the sensor, the duration of a session would be limited to only 20 minutes, and the resulting download time would be about 5 minutes. Since a football match or training often lasts up to two hours, this limitation is not desirable. And even if the available storage would allow storing two hours of data, downloading the data would take roughly 30 minutes. Clearly, a user does not want to wait that long for the results after a session. This shows the need for processing the raw data on the sensor itself.

With all this in place, JOGO can provide insight into your football performance similar to popular sports tracking apps like Strava[2]. Within such an online environment, personalized challenges and leaderboards can be set up, encouraging younger and older people to go outside, play football and improve their skills or just adopt a healthier lifestyle.

[2]https://www.strava.com/features

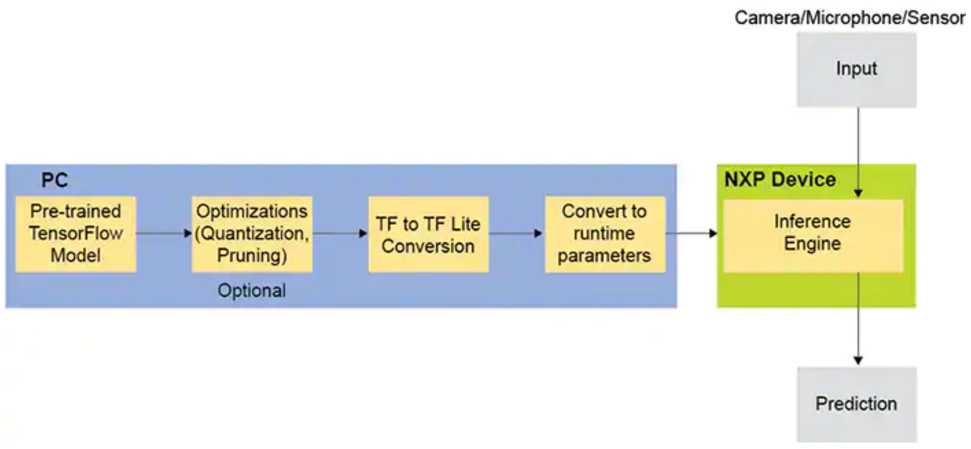


Figure 1.2: **Flow from a trained TensorFlow model to running on an embedded device.**

A second approach to using the sensor and the data provided by the sensor is the use by football clubs. If every player of a (youth) team is equipped with a sensor, the coach can be provided with detailed insight into every player's development over time. This can help in providing personal training schedules for each player, tailored to their strong and weak skills. In the long term, based on the gathered data of many players, it might even be possible to predict which characteristics are signs of a talented player.

1.2 Research Question

As described in Section 1.1, the sensor should be able to calculate multiple generic movement metrics as well as specific football-related statistics. Calculating acceleration, speed and distance accurately is a problem often solved with a biomechanical approach, using the IMU, a Kalman filter and integration of the accelerations to speed, and the speed to distance [1, 2, 3]. The other aspect is the football-specific movements. For this we need to be able to detect when a player touches a ball, and also classify the corresponding type of ball touch. From an Embedded Systems perspective, classifying football movements in real-time offers an interesting challenge to apply Embedded Machine Learning, which is a quickly emerging field. Therefore, the subject of this thesis will be focused on classifying ball touches such as dribbles, passes and shots, bringing together sports science and Embedded Machine Learning.

Machine Learning is a broad subject and has many subdomains. One of those subdomains is neural networks, which is an approach based on how the human brain works. Multiple studies suggest that the self-learning capabilities of neural networks result in better accuracy for classifying motions and ball touches than conventional machine learning methods [4, 5]. Considering that the classification needs to be performed on an embedded device, there are multiple constraints and challenges since the memory and computing performance of the targeted kind of embedded device are limited.

To target such a constrained embedded device, a neural network model can be designed and trained in a regular way, with a framework like TensorFlow. However, before it can be deployed to an embedded device, the model needs to be optimized to be more lightweight, as illustrated in Figure 1.2.

Taking into account the desired classification task, the promising performance of neural networks and the challenging context of Embedded Machine Learning, we arrive at the following research question:

How can neural networks be applied for embedded real-time classification of ball touches in football?

1.3 Contributions

This thesis provides an analysis of the possibilities and challenges of running neural networks on an ARM Cortex-M4 microcontroller, a Nordic nRF52832, in the context of an existing application with IMU and BLE connectivity. To explore the possibilities and limitations we use the problem of classifying ball touches as dribbles, passes and shots. This results in three contributions:

1. Exploration of technical possibilities and limitations of running neural networks in TensorFlow Lite for Microcontrollers on an ARM Cortex-M4 microcontroller.
2. A development flow to design, train, test and deploy neural networks on the nRF52832 within JOGO's application.
3. An accurate approach to classifying ball touches with neural networks on an embedded sensor using IMU data.

1.4 Thesis Structure

Throughout this thesis, the following structure is followed:

- Chapter 2 describes some background theory on sports science and Embedded Machine Learning, and discusses relevant related work.
- Chapter 3 describes the design and implementation process, including data analysis and neural network design choices.
- Chapter 4 describes the experiments conducted to analyze the classification performance as well as the resource performance in the embedded context.
- Chapter 5 presents the conclusions, an answer to the research question and recommendations for future work.

Chapter 2

Background and Related Work

The subject of this thesis, classifying ball touches with neural networks, brings two research areas together: sports science and neural networks. This chapter covers the necessary background theory and related work on both these topics.

2.1 Sport Movement Classification

A well-known way of analyzing football matches is using video recordings [6, 7, 8]. The camera-based approach can provide match highlights, tactical analysis and a summary of team and personal statistics. While some of these video-based analyses can provide individual ball-touch information, the actual information is limited by the camera resolution, frame rate and field of view. Some of these restrictions can be avoided by placing multiple cameras, however, this obviously adds up in cost and setup time. Also note that for every training or match location, a camera setup is required, or the setup needs to be moved between locations. Another important drawback is that a lot of manual actions are required to extract information, while still not providing very detailed insights on player level as noted by Fischer *et al.* [6].

Another common way of monitoring sports activity is to have players wear a smart vest equipped with GPS and sometimes an IMU, as analyzed by Theodoropoulos *et al.* [9] and Neville *et al.* [10]. This method mainly provides insight into player velocity, acceleration, and covered distance. This way the vest can distinguish different activities such as walking, running and jumping. In case the vest features an IMU, limited information can be provided on events. Reliably detecting ball touches is not possible, since the sensor is located at the upper body, but other events such as tackles, jumping for a header and impact from another player can be detected.

A third method, which lacks the drawbacks of the first two methods, is to have players wear a sensor with an IMU in their shoes. Cust *et al.* [11] analyze the application of IMUs in many different sports. A few of the advantages of using IMU sensors are the relatively low cost, small form factor, and scalability to multiple players and multiple locations. Another very relevant feature is the option to place the sensor at the relevant position for the sport in question, in JOGO's application the shoes of a football player. This specific placement provides additional biomechanical insight into the player's movements and impact of the ball on the foot. Similarly, when applying this to basketball, the sensor can be attached to the arms of a player, or in the case of hockey even incorporated into the hockey stick.

When using an IMU, two kinds of classification can be applied: continuous and event-based. In Section 2.1.1 continuous classification is described. With this approach, a window size is selected, and every window will be classified to determine the current activity. This could be used to classify whether a player is standing still, running, dribbling or performing another activity. Section 2.1.2 describes event-based classification. This approach is more focused on detecting possible interesting events, such as a dribble or a shot, and then performing the classification. To do this, a "simple" peak detection algorithm selects candidate windows, which will then be classified.

2.1.1 Continuous Classification

An example of continuous classification is shown by Taghanaki *et al.* [5]. The authors propose a method to classify running styles using five IMUs, with a 500 Hz sampling frequency, placed on the feet, legs and lower back. By analysing the running features, such as heel and toe strikes, stride length and the space between both feet, suggestions can be given to runners to improve their stance and prevent injuries. The authors propose a Convolutional Neural Network (CNN) and a Long Short Term Memory (LSTM) model. The CNN model achieves an 86% accuracy and the LSTM model achieves a 93% accuracy in classifying eight different running styles correctly. Unfortunately, while the paper uses data from wearable sensors, the presented neural networks are not implemented on the sensors, but on a development PC.

The implementation focuses on detecting running style, using ten-second windows as input for the various neural networks. While this might be applied to football to detect whether a player is walking, running, dribbling or juggling for a consecutive time period, it is not optimal for detecting a single event like a pass or shot. The ball touch might occur right at the end of one window and the beginning of the next window, thus missing a part of the data when classifying the windows individually. Another issue with this continuous classification is that, even with a low false-positive percentage, the absolute number of false-positive ball-touch classifications might get large over time. This shows the need for a method that only selects windows that meet certain requirements for classification, as described in the next section.

2.1.2 Event-based Classification

Hollaus *et al.* [12] proposed a method to detect successful catches in American Football. The system consists of two devices, one for each wrist. Multiple sensors are used, including an accelerometer, gyroscope, magnetometer and a microphone. The authors propose a three-layer CNN, which processes the raw data of the four sensors. The accelerometer and magnetometer show the most significant contribution to the classification accuracy. On a dataset with 541 catches and 218 drops, an accuracy of 93% is achieved with true positives rates of 83.3% and 96.3% for drops and catches, respectively. It should be noted that the authors do not propose an actual event-detection mechanism for possible touches, thus the performance would presumably be lower if the training set would also contain samples that are not a catch or a drop.

Kautz *et al.* [13] proposed a CNN model to perform activity recognition in beach volleyball using an accelerometer located on the wrist. Data is gathered from actual beach volleyball game situations, and can be grouped into ten different action classes, including a null class, which indicates a significant motion not related to beach volleyball. A high-pass Butterworth filter is applied to detect peaks in the acceleration data, and when it crosses a certain threshold, the peak is considered a relevant event. This event-detection algorithm has an accuracy of 99%. A two-layer CNN is used, which first applies convolution to each accelerometer axis, and then in the second layer, applies convolution to the three axes at once. This approach achieves an accuracy of 83.2% on a dataset with 4242 actual events and 7938 false-positive events. Several other machine learning approaches were analyzed, all achieving lower accuracy than the CNN.

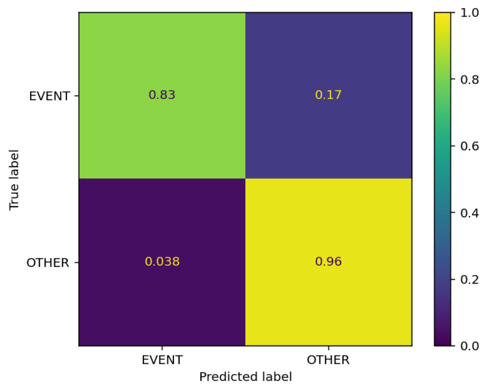
2.1.3 Ball-touch Classification in Football

The event-based classification method is the most relevant approach to the ball-touch classification problem. Schuldhaus *et al.* [14] have proposed a very relevant approach to classifying ball touches of football players. The authors use a 6-axis IMU with a sampling frequency of 1000 Hz, located in the insole at the back of both shoes. Data has been gathered from two studies. The first study consists of single passes and shots, varying in intensity. The second study consists of passes and shots taken from a dynamic situation in an 11 vs. 11 match.

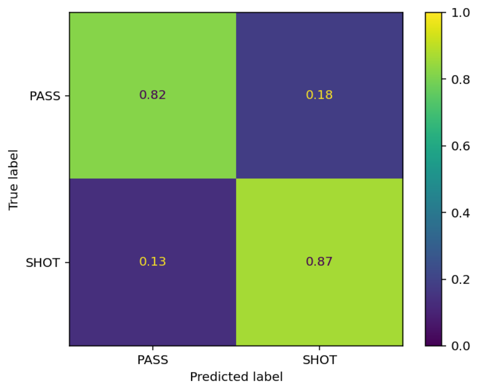
The accelerometer data from both feet is filtered using a high-pass Butterworth filter. The high-pass filter highlights the events with a big, sudden impact on the feet, while removing smooth, low-frequency movements like walking. To find the peaks that could indicate a ball touch, the magnitude of the accelerometer axes is calculated for both feet separately. Subtracting both magnitudes, and taking the absolute value results in the signal for peak detection. This subtraction uses the assumption that when one leg is used to touch the ball, the other leg is the supporting leg, thus having a low signal intensity. After the subtraction, every remaining peak above a certain threshold will be considered for classification. Contrarily, when the magnitudes of both feet are similar, it is evident there can be no ball touch.

To classify the peak, a one-second window around the peak is taken, four statistical features are calculated for each axis and fed into several Machine Learning classifiers, of which the Support Vector Machine (SVM) gives the best results. Figure 2.1a shows the results for classifying a pass or shot event versus other high-intensity events, giving an 89.5% mean classification accuracy. Figure 2.1b shows the result of classifying passes versus shots, giving an 84.2% mean classification accuracy.

Stoeve *et al.* [4] compared the ball-touch classification performance of the well-performing SVM to multiple neural network implementations, which were not considered by Schuldhaus *et al.* [14]. While this approach records the data at 200 Hz, the other details are very similar. Data is recorded from both single ball touches and real matches, possible ball touches are provided by a peak detection algorithm, and the windows are either 1 or 2 seconds. From the comparison between the SVM, CNN and LSTM, the CNN shows the best results.



(a) **Relevant event vs. other.**



(b) **Pass vs. shot.**

Figure 2.1: **Confusion matrices showing classification results from [14, Table 3 and 4]. (a) Event samples: 354. Other samples: 3581. (b) Pass samples: 278. Shot samples: 15.**

2.2 Embedded Machine Learning

While self-developed algorithms, like the peak detection in Section 2.1.3, are often a good and lightweight approach to detect events in data, they might not suffice when dealing with more complex, multidimensional data. Considering the six axes of data the IMU provides, and the many small differences there are in the movements from person to person, it appears that classifying ball touches should be approached with machine learning. As mentioned in Section 2.1.3, classification can be performed continuously, but for ball touches the focus will be on event-based classification. A common approach for classifying IMU data is to extract statistical features from the selected window, and feed those into a machine learning model, such as Naive Bayes, k-nearest neighbor, SVM or XGBoost [4, 12, 13, 15, 16].

An often used and well-performing conventional Machine Learning method is XGBoost, a Gradient Boosting framework [17]. Gradient Boosting is considered a state-of-the-art approach to many classification challenges, which is achieved by using multiple decision trees to boost the accuracy of a prediction. McGrath *et al.* [15] and van den Tillaar *et al.* [16] show that Gradient Boosting is the best performing method in classifying cricket and handball touches. Therefore this method will be used as an example of conventional Machine Learning approach to compare to the proposed neural network implementations in Section 4.1.5.

Stoeve *et al.* [4] and Zhao *et al.* [18] mention that conventional machine learning approaches require a manual selection of features. However, when dealing with time-series data such as IMU data, using neural networks would be beneficial [11]. Neural networks can basically extract deep features from the data [18], these deep features perform better at highlighting significant details in the raw data than manually selected statistical features.

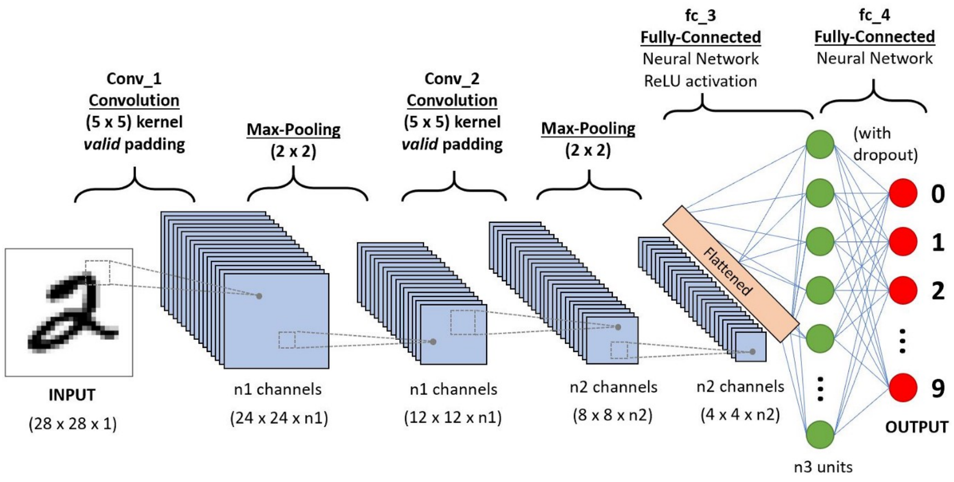


Figure 2.2: **A simple CNN, illustrating the use of kernels on an image.**

2.2.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a special type of neural network that features a convolutional layer, as illustrated in Figure 2.2. This layer performs convolution on an image (or IMU data) by going over the data with a kernel of a certain size. By using this kernel, features such as circles, lines and curves can be extracted from an image. Usually, when going over an image with the convolution, multiple kernel configurations are used, to be able to detect different features, resulting in an increasing number of channels after each convolutional layer, as seen in Figure 2.2.

For example, a 5x5 kernel takes 25 pixels of an image and calculates one value from this. By shifting the kernel over the picture, a new picture is created with highlighted features, different in each channel. A convolutional layer is often combined with a max-pooling layer, which also goes over the picture with, for example, a 2x2 filter, but now keeping just the maximum value out of the four pixels, resulting in a smaller image. This smaller image has the advantage that the next layer has to process fewer pixels, but thanks to the convolution also the most predominant information is preserved, basically highlighting the desired feature. This combination of convolution and max-pooling can be repeated on the resulting smaller image, as seen in Figure 2.2, and is finally flattened into an array of values and fed into two fully-connected layers producing the actual classification. These fully-connected layers consist of neurons, which apply a function to each of the inputs. The applied function depends on the weight of the neuron, which is selected through the training process of the neural network. By connecting all these inputs and outputs, each value from the convolution will have an impact on the classification output. While not shown in Figure 2.2, often one or more additional dropout layers are applied during training, which prevent or reduce overfitting by randomly leaving some nodes out [19]. This approach is used in related applications, for example by Hollaus *et al.* [12] and Kautz *et al.* [13].

Translating this concept from an image to IMU data, convolution can be used to detect peaks and slopes in the data that are indicators of a certain type of ball touch. This is commonly applied in processing IMU data for detecting human and sports movements [12, 13, 20].

Faraone and Delgado-Gonzalo [21] show the use of CNNs implemented on a Nordic nRF52 microcontroller, very similar to the one used in this thesis, as noted in Section 1.3. With an input size of 1x256 and seven convolutional layers, the resulting model size is 200 KB, and the execution time is 94.8 ms. The most demanding layers are the convolutional layers, contributing most to the memory usage, execution time and power consumption.

2.2.2 Recurrent Neural Networks

Another type of neural network is a Recurrent Neural Network (RNN). RNNs have a special feature to remember information. While regular neural network layers only know about their current input, RNNs also remember information from the previous input [22]. Figure 2.3 shows an example of an RNN. When analyzing the first input x_0, there is no additional information, however when processing the second input $x_!$, there is information available from the previous input. This is a quite naive notion of memory, considered short-term memory. There is only information available from the last input, and there is no indication whether the information was actually meaningful or not.

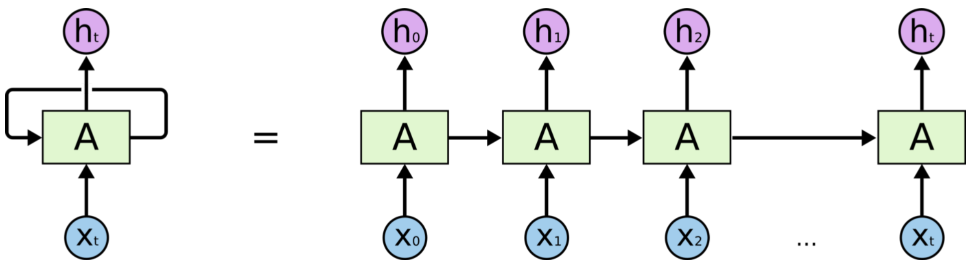


Figure 2.3: **On the left an RNN with the memory loop, on the right its unrolled equivalent [22].**

An improved, or special kind of RNN are Long Short Term Memory (LSTM) networks, which provide the desired long-term memory. LSTMs have been introduced by Hochreiter and Schmidhuber [23], an example is shown in Figure 2.4. This kind of network provides a solution to remember relevant information for the long term. To achieve this, an LSTM uses three gates, a *forget gate* (represented by the first sigmoid layer), an *input gate* (represented by the second sigmoid layer and the tanh layer), and an *output gate* (represented by the final sigmoid layer). The forget gate is used to forget irrelevant information, the input gate selects the relevant information and finally, the resulting output is retrieved through the output gate.

Since this approach provides information over time, it seems to be a relevant addition to ball-touch detection with IMU data, where the classification of a ball touch could be performed better with information from the past. For example, when a player passes a ball, it could be that he tilts his foot in a certain manner a few milliseconds before the ball touch, this information can then be used at the

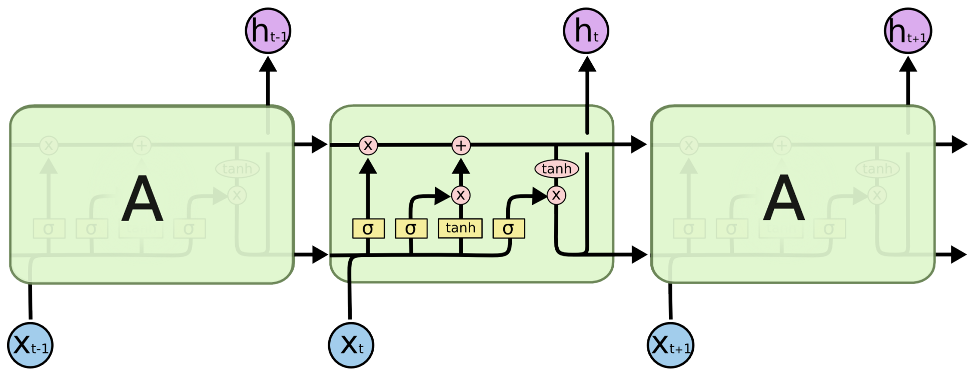


Figure 2.4: **Illustration of an LSTM implementation [22].**

moment of the actual ball touch. LSTMs have been applied by Sherratt *et al.* [24] to detect human motions, by Taghanaki *et al.* [5] to detect running styles, and by Stoeve *et al.* [4] to detect ball touches. An LSTM approach provides good results for detecting human movement styles, while it is outperformed by a CNN when detecting ball-touch events in research by Stoeve *et al.* [4].

2.2.3 TensorFlow

While the classification performance and resource requirements are very important, the development flow is also a significant factor, as indicated in Section 1.3. Therefore, a Machine Learning framework to design, train and test the models is preferred over a custom implementation tailored to the experiments in this thesis. Many options exist such as Embedded Learning Library [25] and Caffe [26], but TensorFlow Lite for Microcontrollers is a well-suited candidate that shows some relevant benefits over other options [27]:

- The development environment of the regular TensorFlow framework can be used to design, train and test models.
- The framework is under active development.
- Hardware-specific optimizations, such as CMSIS-NN [28], can be easily integrated.

2.2.4 Performance Optimizations

Since running neural networks on embedded devices is subject to several resource limitations, such as execution time, memory size and battery capacity, many approaches can be found on how to overcome these issues. A few relevant methods are highlighted.

Integer Quantization

To optimize a neural network model for an embedded device, a technique called integer quantization can be used in TensorFlow, as described by Krishnamoorthi [29]. Instead of using 32-bit float values for the parameters of a model, these values can be quantized to 8-bit integers. This reduces the accuracy, but with less than a 2% decrease. The model size however is reduced by a factor of four, and the speedup can be up to three times.

CMSIS-NN

To enhance to execution of Neural Networks on Arm Cortex processors, Lai *et al.* [28] presented CMSIS-NN. This library provides optimized software implementations of certain functions for neural networks, such as matrix multiplication, convolution, pooling layers and activation functions. The first step in the optimization is also using quantization. The commonly used 32-bit floating-point data is converted to 8-bit or 16-bit integers. This has a minimal impact on accuracy, while having the mentioned advantages of quantization. In the case of CMSIS-NN the quantization brings another advantage, it enables the main optimization of using Single Instruction Multiple Data (SIMD) instructions. The quantized 8-bit or 16-bit values can be fit in 32-bit registers, thus being able to apply a CPU instruction to multiple values at the same time. By rearranging the data to use SIMD instructions, matrix multiplication (the cornerstone of neural networks) can be heavily optimized. The implementation shows a 4.6x improvement in execution time and a 4.9x improvement in energy efficiency on a set of baseline implementations, without a notable impact on classification performance.

2.3 Discussion

Section 2.1 describes some of the available methods to classify sports movements, such as camera analysis, smart vests and IMU sensors. To be able to provide detailed insight into the performance of a single player, the related work shows that a wearable sensor equipped with an IMU is the best solution. Since the goal is to detect ball touches, which occur irregularly, the event-based approach makes the most sense. The approach therefore needs to include an event detection algorithm to select ball-touch candidates in the IMU data.

Section 2.2 shows that using neural networks yields better classification results than conventional machine learning algorithms, with CNNs being the most promising approach. To also ensure easy deployment on the embedded device, TensorFlow Lite for Microcontrollers should be considered, as it includes support for hardware-specific performance optimizations and the models can be trained and tested using the regular TensorFlow framework.

Chapter 3

Design and Implementation

This chapter describes the design and implementation of neural networks for ball touch detection on the JOGO sensor. To arrive at a neural network design, we need to consider many aspects, such as the data collection, the training process and especially the restrictions of the embedded device. We want to process the IMU data in real-time, so the available resources of the sensor have to be explored, to see what restrictions the neural network designs have to adhere to. The following subjects provide the necessary context and building blocks for the designs in this chapter and the experiments in Chapter 4:

- Requirements in the context of JOGO's existing hardware and firmware.
- An overview of the system from data collection to real-time inference.
- Analysis and preprocessing of the IMU data and neural network design.
- The use of TensorFlow for training and deployment on the sensor.

Section 3.1 covers the given constraints of the existing hardware and firmware. Section 3.2 introduces an overview of the system. Section 3.3 gives some insights into the data gathering, how the data looks, and the necessary preprocessing of the data. Section 3.4 explains the design choices towards the neural networks that will be used for the experiments. Section 3.5 and Section 3.6 describe the use of TensorFlow both for training on the PC and inference on the sensor.

3.1 Integration in JOGO's Sensor Platform

The core of JOGO's sensor platform consists of a Nordic nRF52832, a 64 MHz Cortex-M4 microcontroller with a Floating Point Unit (FPU) equipped with 512 KB Flash and 64 KB RAM storage. A 6-axis IMU (accelerometer + gyroscope) records the measurement data, 16 MB external Flash is available for storing raw or processed data and an 85 mAh battery provides power to the system. A summary of the specifications is shown in Table 3.1.

To show the restrictions of the system that have to be taken into account, this subsection covers the relevant components and provides some example calculations. As a reference for the sampling frequency, we use 1000 Hz as used in prior research [14] and commercial applications like Playermaker [1], as well as 200 Hz as used by Stoeve *et al.* [4].

Component	Type	Details	
Processor	Nordic nRF52832 64 MHz Cortex-M4F	Flash	512 KB
		RAM	64 KB
IMU	ICM-42686-P	Accelerometer	±32g (18 bits)
		Gyroscope	±4000dps (19 bits)
External Flash	-	16 MB	
Battery	-	85 mAh	

Table 3.1: **Details of the sensor.**

3.1.1 Storage

Considering the storage, we first show why processing on the sensor is necessary, as opposed to storing raw data on the sensor and offloading it after a football session. An IMU sample, containing raw data of the six axes, is stored as a 23-byte object in the current firmware. With the available external Flash memory of 16 MB, we get the maximum session duration depending on the sampling frequency using the following equation:

$$session\_duration(s) = \frac{16\,MB}{data(bytes)} * \frac{1}{f_s(Hz)} \tag{3.1}$$

Using Equation (3.1) with a sampling frequency of 1000 Hz results in a maximum session duration of only 12 minutes, and even with a sampling frequency of 200 Hz, we end up with only 60 minutes. This maximum duration is clearly not sufficient, as a regular session, match or training, could last up to two hours. Additionally, with the current firmware implementation, downloading this 16 MB of data to an Android phone via BLE takes about half an hour. While the theoretical BLE throughput of the nRF52832 is 2 Mbps, the actual throughput depends on many factors, such as packet size, phone model and the environment, and is currently limited to approximately 10 KB/s [30]. Both the required storage and downloading time show why we actually need to do (some of) the processing on the sensor, instead of downloading everything and processing it on a phone or in the cloud.

If we instead process the raw IMU data on the sensor, we can for example reduce the update rate to 10 Hz, and compress the data to only contain covered distance and ball-touch information, resulting in a 12-byte object. Using Equation (3.1) with a data size of 12 bytes and a sample rate of 10 Hz, we get a maximum of about 40 hours, which is more than enough to store multiple sessions. Modifying Equation (3.1) to show the data size based on the session duration, we get:

$$session\_size(KB) = \frac{session\_duration(s) * f_s(Hz) * data(bytes)}{1024} \tag{3.2}$$

Using Equation (3.2) for a session of 2 hours, 12 bytes data and 10 Hz, we get a data size of 844 KB, which can be downloaded in approximately one and a half minute. Considering that firmware changes might improve the download time, and the sensor could automatically start offloading the data when the player's phone is nearby, this results in an acceptable and user-friendly storage usage and download time.

3.1.2 Execution Time

There are two restrictions concerning the execution time. The first is the duration of the event to be analyzed. The classification of an event should not take longer than the duration of the event itself. If the execution time would be longer, a new event could be triggered before the previous classification is finished. This will result in a lag in event handling, and eventually in missing events.

The other restriction is the buffer size of the IMU. While the neural network inference is in progress, the microcontroller can not process incoming IMU data, so the IMU has to store it in its internal buffer. When the microcontroller has finished the classification, it can again retrieve new samples from the IMU. With the selected high resolution data mode, the maximum number of samples that can be stored in the buffer is 100 samples [31]. Depending on the sampling frequency this results in a maximum execution time, as shown in Equation (3.3).

$$execution\_time_{max}(s) = \frac{100}{f_s(Hz)} \tag{3.3}$$

To summarize, there are two restrictions on the execution time, which should both be met, therefore the smallest value of the two restrictions is leading:

1. The execution time should be smaller than the duration of the event.
2. The execution time should be smaller than the time it takes to fill the IMU buffer with new samples, see Equation (3.3).

To illustrate these two restrictions, we use Equation (3.3) with a sampling frequency of 1000 Hz. This results in a maximum allowed execution time of 100 ms, while when using a sampling frequency of 200 Hz, the resulting maximum execution time is 500 ms. These values show the strict maximum execution time based on the given sampling frequency, which could be further restricted if the event duration is actually shorter. So when using an event duration of 50 ms at 1000 Hz, the resulting maximum execution time is 50 ms, while when using an event duration of 150 ms at 1000 Hz the execution time is still limited to 100 ms.

3.1.3 Memory Usage

The current firmware on the sensor, including IMU processing, storage and Bluetooth communication takes approximately 388 KB Flash and 44 KB RAM, respectively 76% and 69% of the total available memory. This leaves 124 KB Flash and 20 KB RAM for the neural network implementation, as shown in Table 3.2.

	Flash		**RAM**	
	KB	**%**	**KB**	**%**
Total	512	100	64	100
Used	388	76	44	69
Available	**124**	**24**	**20**	**31**

Table 3.2: **Overview of memory used by the firmware, and the amount available for neural network implementation.**

3.2 System Overview

Figure 3.1 shows an overview of the system, and the separate sections of data processing, training and real-time classification. The data section starts with recording the raw IMU data on the sensor, where the sampling frequency is the main parameter of importance. The event detection and window creation are performed on the PC during the training process, but are also implemented on the sensor for real-time use. The event detection has fixed parameters, as determined by JOGO's data scientists, while the window size can be optimized based on the use case and resource limitations.

The training section happens completely on the PC. The data in the created windows is normalized, split into training and test sets, and the label distribution is calculated to be able to perform a balanced training. The model that will be trained can have various designs, featuring dense layers, convolutional layers and LSTM layers, which is described in detail in Section 3.4. Also, every kind of layer has its own parameters such as the number of neurons, the convolution kernel size, and the number of kernels. The training process in TensorFlow is described in Section 3.5, and after this process is finished, the trained model is converted to a TensorFlow Lite model, which can be transferred to the sensor.

Finally, the inference section runs completely on the sensor. The settings for window creation are set in the firmware, the trained and converted neural network model is stored in the memory, and the classification can be performed whenever the event detection is triggered. The output of the classification will then be stored in the external Flash memory until it is downloaded to the player's phone.

Given this system overview, in the context of the restrictions described in Section 3.1, the remainder of this chapter describes the different design and implementation steps.

3.3 IMU Data Analysis and Preprocessing

This section covers the data coming from the IMU, and the necessary analysis and processing of this data. The IMU data consists of three accelerometer axes and three gyroscope axes. The accelerometer measures accelerations in the range ± 32g, and the gyroscope measures angular velocity in the range ± 4000 degrees per second.

3.3.1 Event Detection

The event detection approach is not within the scope of this project, as it was already provided by JOGO's data scientists, but is briefly covered for completeness.

To detect events that are possibly a ball touch, the gyroscope is used. The approach to event detection, as shown in Figure 3.2, is to calculate the magnitude of the gyroscope axes, then a second-order Butterworth 200 Hz high pass filter is applied, using a similar approach as Schuldhaus *et al.* [14] and McGrath *et al.* [15]. Using the high pass filter, the low-frequency parts of for example walking and running are removed, while the sudden and rough impact of possible ball touches is highlighted and can be found using a simple peak finding

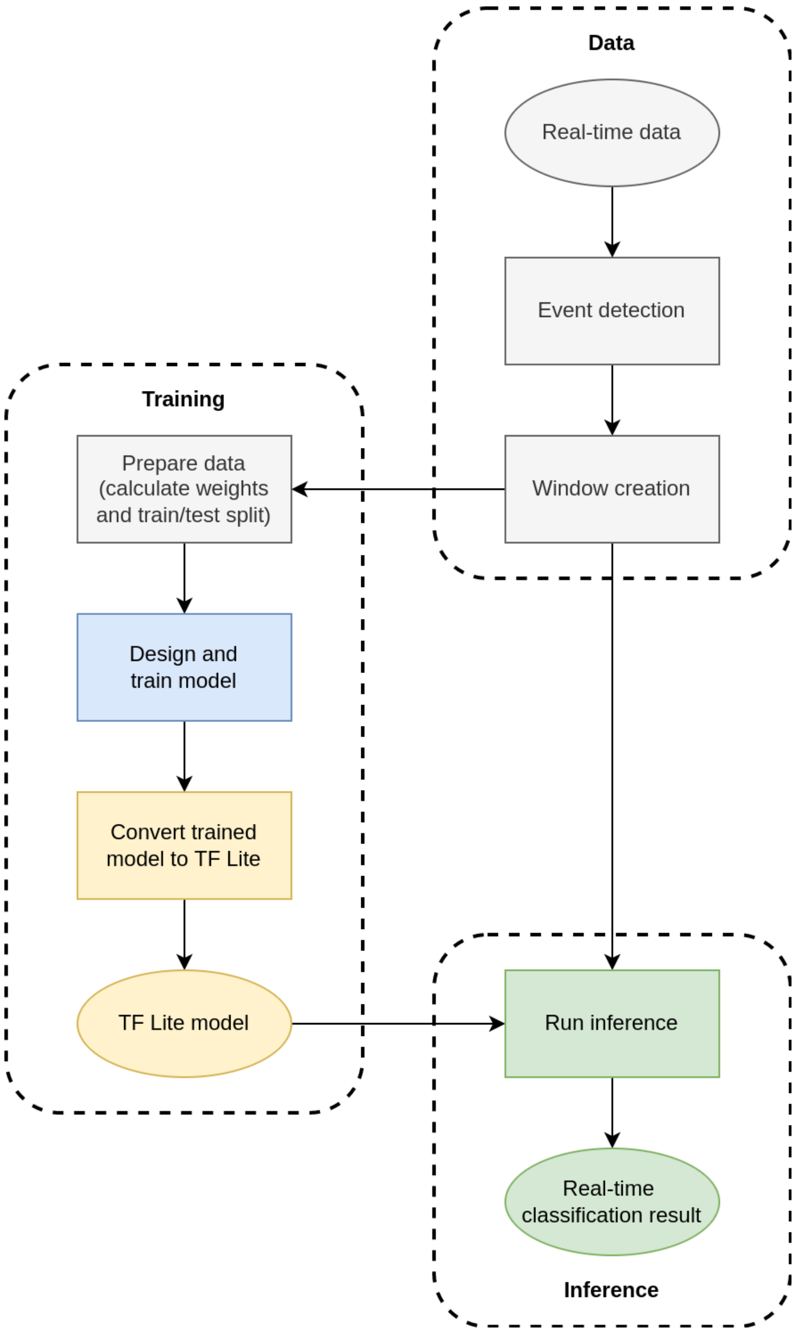

Data
Real-time data
Event detection
Window creation
Training
Prepare data (calculate weights and train/test split)
Design and train model
Convert trained model to TF Lite
TF Lite model
Run inference
Real-time classification result
Inference

Figure 3.1: **Overview of the use of TensorFlow. On the left the training process, on the right the process on the sensor. Grey elements are custom implementations. Blue objects use regular TensorFlow. Yellow objects use TensorFlow Lite, and green objects use TensorFlow Lite for Microcontrollers.**

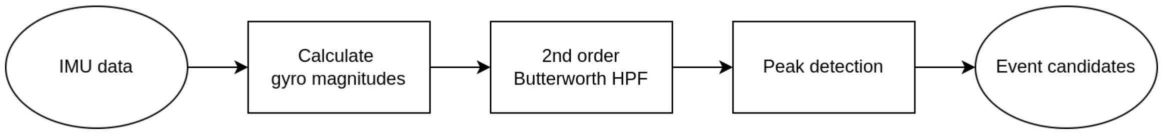


Figure 3.2: **Event detection flow.**

algorithm. As these peaks indicate when the event has happened, they can be used to create a window around it with a user-preferred length, to perform the classification.

3.3.2 Ball-touch Features

Before we go into the details of each type of event, it needs to be clear how the axes of the IMU correspond to the orientation of the sensor in the shoe, which is shown in Figure 3.3. The x-axis is in the forward/backward direction, the y-axis in the left/right direction, and the z-axis in the up/down direction. Note that this illustrates the right shoe. To handle the changed physics for the left shoe, the accelerations on the y-axis and the rotations around the x-axis and z-axis are inverted in the firmware. By inverting these axes, the same data processing and models can be applied to both the left and right foot.

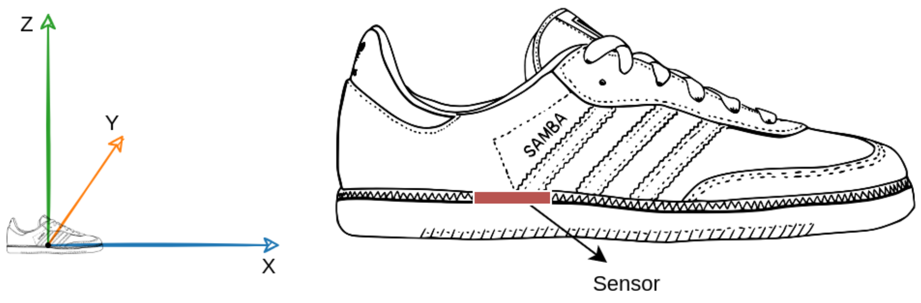
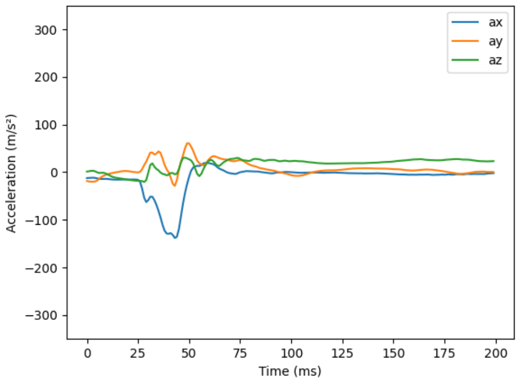


Figure 3.3: **Illustration of the sensor location and the axes orientation of the right shoe.**

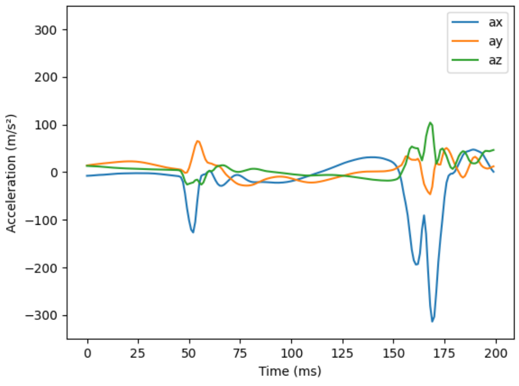
After a window is selected as a ball-touch candidate, the actual classification needs to take place. To make sure no ball touches are missed, the event detection algorithm settings can not be too strict, as a consequence there will also be false positives among the selected candidates. To distinguish the false positives (or so-called noise samples) from the actual ball touches, it needs to be clear what the distinctive features are. These features can then also be used to classify the various ball touches.

Figure 3.4 shows an overview of noise, dribble, pass and shot samples. For a clear overview in the plots, only the accelerometer values are shown. A first look at the data already shows clear differences between the different ball touches, but we will go over them one by one to explain the relation between the data and the physics of the ball touch.

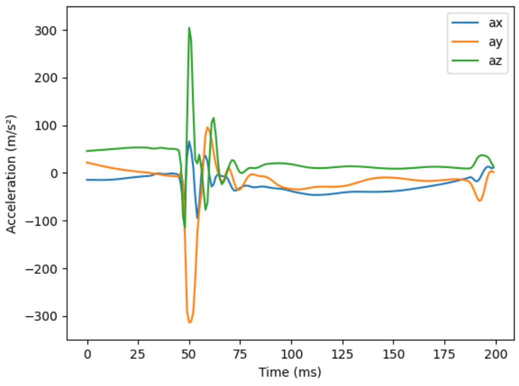
Figure 3.4a shows a noise sample, showing some accelerations that indicate movement. However, compared with the other ball touches, we notice that the



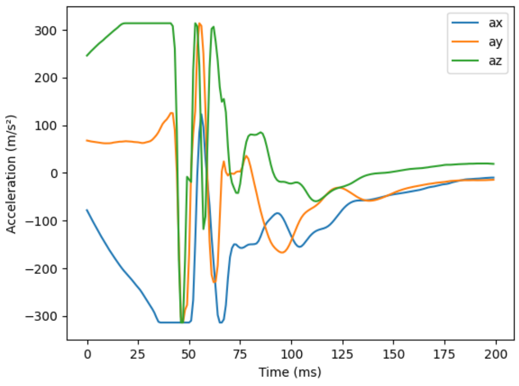
(a) **Noise sample**



(b) **Dribble sample**



(c) **Pass sample**



(d) **Shot sample**

Figure 3.4: **Overview of accelerations per type of ball touch.**

accelerations do not follow a clear pattern. These might be steps, foot flicks or other sudden movements or impacts.

Figure 3.4b shows a dribble sample. Note the two peaks in the *ax* axis. The first peak indicates the ball touch, the second peak indicates the foot hitting the ground after the dribble. While the combination of the two peaks is very distinctive for the dribble event, it is important to note that the magnitudes of the first peak are very similar to a noise sample, thus hard to distinguish.

Figure 3.4c shows a pass sample. To perform a pass, the foot is rotated outwards to touch the ball with the inside of the foot, and the movement is slightly upwards to give power to the pass. The distinctive peak in *ay* indicates the use of the inside of the shoe, while the peak in *az* represents the slight upwards motion. Interesting to note is that the accelerations of the foot during a pass have a short impact, but are coming close to the maximum range in amplitude.

Finally, Figure 3.4d shows a shot sample. The shot is performed on the laces, thus acting both in the forward/backward direction as well in the up/down direction, which is visible through the large values of *ax* and *az*. Note that due to the power of the shot, the acceleration measurement is clipped at the maximum reading.

Looking at these accelerometer values, it is apparent that the different ball touches each have their own specific pattern. The amplitudes of the different axes, the width of the peaks, and the relation between the different axes are all indicators of the specific ball touch. Additionally, there is similar information coming from the gyroscope readings, however for visualizing this, the gyroscope is less suitable.

3.4 Neural Network Design

By now we know that the six axes of IMU data provide relevant distinctive information on the ball-touch events, and we have an event detection algorithm to trigger when a candidate event happens. Before the raw IMU data is ready to be used as input to a neural network, we need to take one more step to prepare the data, which is to normalize the data. This is necessary because of the different scales of the accelerometer and gyroscope data. In such a case normalizing is considered common practice when working with neural networks, as it improves the training and optimization process [32]. The accelerometer data is provided in the range of -314 to 314 m/s^2, while the gyroscope data ranges from -4000 to 4000 rad/s, and is normalized to values between 0 and 1.

3.4.1 Dense Layer Approach

The simplest approach is to use a neural network with just a dense layer, as shown in Figure 3.5. This approach takes all six IMU axes, appends them into a single array of numbers and then feeds it to the dense layer. This is a very lightweight approach, but the dense layer now considers the data without any sense of the six different axes. The output of the dense layer is then forwarded to the output dense layer for the final classification outcome.

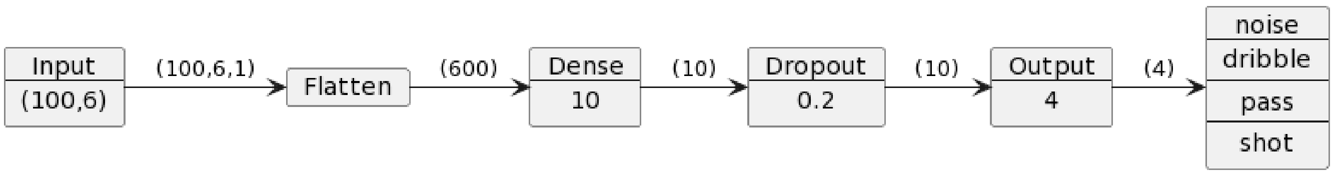


Figure 3.5: **Network design with a dense layer.**

3.4.2 One-layer CNN Approach

The second, more interesting approach, is to use a convolutional layer to process the six axes from the IMU, as shown in Figure 3.6. This convolutional layer is used to extract the distinctive features of the different ball touches, as described in Section 3.3.2. After moving over the data with a set number of different kernels, the data size is reduced by a pooling layer. This helps both to reduce any chance on overfitting, as well as to keep execution size and model size small. After flattening the output, a dropout layer is applied to reduce the risk of overfitting, then finishing with the output dense layer, which provides the classification result.

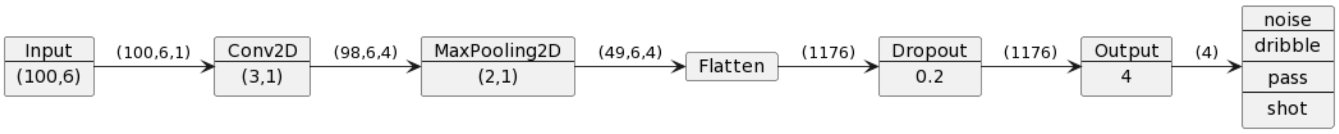


Figure 3.6: **Network design with one convolutional layer.**

3.4.3 Multi-layer CNN Approach

Figure 3.7 shows an extended version of the one-layer CNN model, now with two convolutional layers. This approach is often used to be able to retrieve more complex information from the data. The model can be extended with more convolutional layers in the same manner. As the complexity of the model increases compared to the one-layer CNN model, a multi-layer CNN can optimize better for the training data, thus starting to overfit on the training data. To counter this, the dropout rate is increased to 0.5.

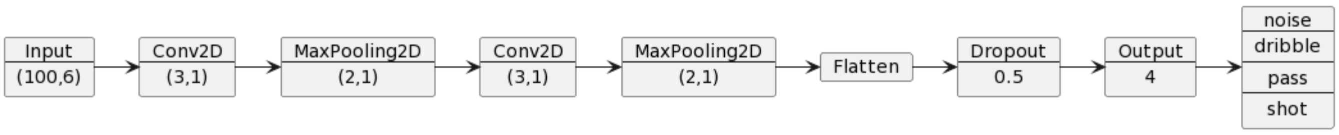


Figure 3.7: **Network design with two convolutional layers.**

3.4.4 LSTM Approach

Since many papers mention the use of LSTMs for the added benefit of looking at historical data, this approach is also applied [4, 5]. The structure of the network, as shown in Figure 3.8, is similar to the one-layer CNN approach, however with an added LSTM unit after the convolutional layer. This LSTM layer processes the six separate IMU axes, and is able to not just go over the data and process the numbers, but also utilize historical information within the window. For example, when looking at Figure 3.4b, the LSTM could ”remember” the peak at 50 ms, and make a more sure classification when it encounters the second peak at 170 ms.

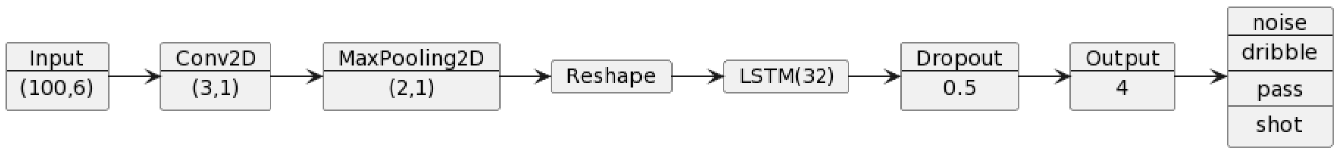


Figure 3.8: **Network design with LSTM layer.**

3.5 TensorFlow Training Process

The training process in TensorFlow is straightforward. The candidate window data is imported, after which the dimensions of each window are checked to ensure all axes are present and every window has the correct window size. To compensate for an over-representation of a specific event type in the data, the occurrence of the different classes is counted and a weight is calculated. TensorFlow takes this weight into account during training, to prevent overfitting on an over-represented event type. This is relevant since there are significantly more noise samples in the dataset than actual shot samples.

Finally, the data is split into a training and test set by using an 80%/20% split, after which the training set is split into training and validation data again using an 80%/20% split, and the training process can start. The training is performed using the TensorFlow framework with the models covered in Section 3.4, using the Adam optimizer, since it is a very common and effective optimizer [33, 34]. During the training process, the optimal parameters for the neurons in the neural network are determined using the optimizer, which tries to minimize the error of the classification by applying small changes to those parameters in every training iteration. After a set number of iterations, the training process is stopped, resulting in a trained neural network.

Until now, every step was performed using the regular TensorFlow framework. However, after the training is finished we switch to TensorFlow Lite. Using the built-in tools, the trained model can be converted to a TensorFlow Lite model. The TensorFlow Lite model is optimized compared to a regular model, such that it only contains strictly necessary operations. The model size can be reduced further by applying quantization, as explained in Section 2.2.4. This uses only 8-bit integers for the model instead of 32-bit floats, reducing both the model size as well as the computational load for the sensor. The final step is to convert the TensorFlow Lite model to a byte array, which can then be incorporated into the TensorFlow Lite for Microcontrollers implementation on the sensor.

3.6 TensorFlow on the Sensor

Finally, using the byte array representation of the model, we can move to the implementation on the sensor. TensorFlow Lite for Microcontrollers provides a C++ based implementation to run models with only a select subset of operations. One of the restrictions is that the LSTM model, as described in Section 3.4.4, can not be executed. Since the nRF52832 is an Arm Cortex-M model, the CMSIS-NN library with optimized kernels can be included in the framework, as described in Section 2.2.4. This comes at a slight cost in Flash usage, but improves the execution time, as shown in Section 4.2.6.

The implementation on the sensor consists of two parts, handling setup and inference. The setup has to be executed once, this loads the model, checks its properties and allocates the required amount of RAM for the inputs and the model parameters. The inference part can then be triggered every time the event detection algorithm selects a candidate. The input window is checked to see if it corresponds to the model input parameters, after which the inference starts. After finishing the inference, the result is returned as a probability per class, which can then be used to select the predicted class.

Chapter 4

Experiments

This chapter describes several experiments based on the models described in Chapter 3. Section 4.1 highlights a selection of interesting classification differences between different models and settings. Section 4.2 shows how different models and settings contribute to the resource usage.

4.1 Classification Experiments

Based on the different neural network models as described in Chapter 3, we first look at the impact of various parameters on the classification performance. Also, based on the related work described in Chapter 2 we compare the neural network approach to an implementation with XGBoost, both in terms of performance and development approach. The dataset used for the classification consists of candidates provided by the event detection algorithm described in Section 3.3.1, containing noise, dribble, pass and shots samples. The distribution of the different classes is shown in Table 4.1.

Type	**Number**
Noise	1200
Dribble	565
Pass	435
Shot	101

Table 4.1: **Dataset distribution.**

4.1.1 Sampling Frequency

As mentioned in Chapter 2, classifying sports movements with IMU-based sensors has been applied before, and others use sampling frequencies ranging from 200 Hz to 1000 Hz [1, 4, 14]. To find the optimal sampling frequency, we evaluate multiple sampling frequencies to compare their performance, using the one-layer CNN model described in Section 3.4.2. The window size is for each sampling frequency set to match a window duration of 200 ms, which contains all relevant characteristics of the various ball touches, as explained in Section 3.3.2.

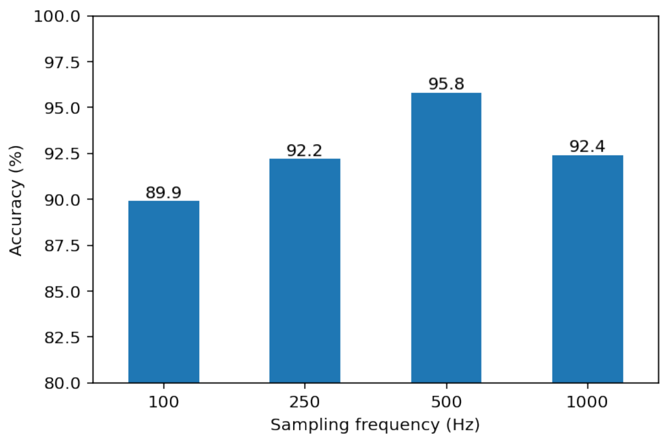
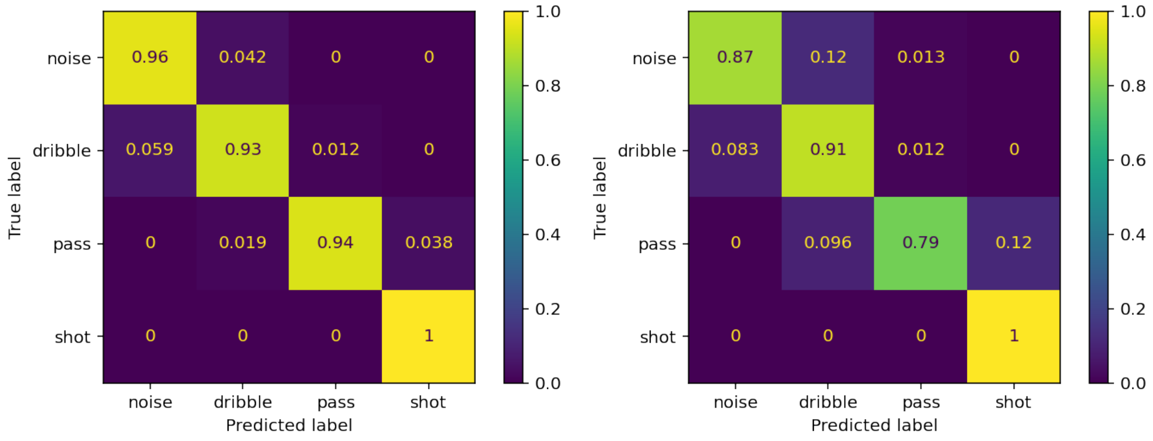


Figure 4.1: **Classification accuracy at different sampling frequencies.**



(a) **Sampling frequency at 500 Hz.** (b) **Sampling frequency at 100 Hz.**

Figure 4.2: **Confusion matrices showing the difference in classification performance at a sampling frequency of (a) 500 Hz and (b) 100 Hz.**

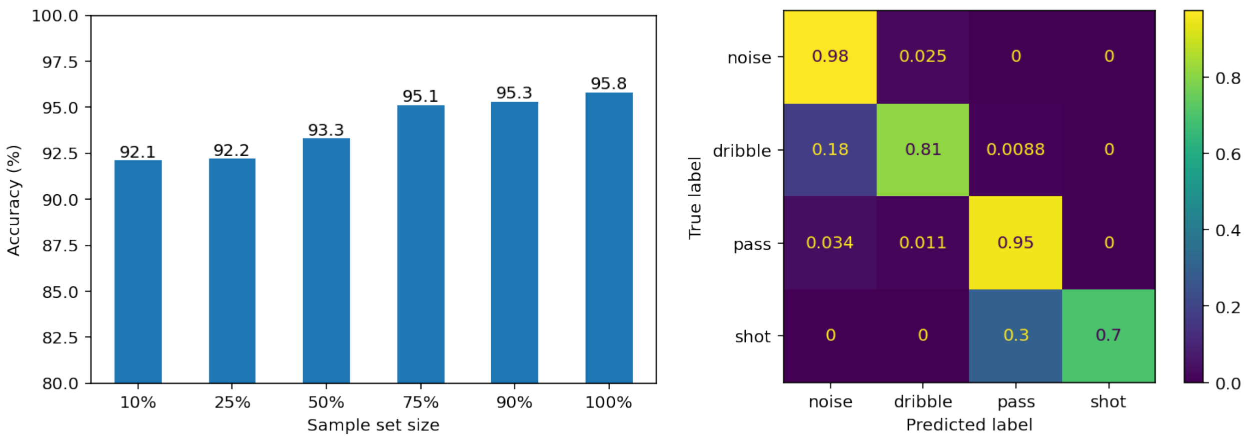
The accuracy results are shown in Figure 4.1. While the decreasing performance for the lower sampling frequencies makes sense, the lower performance at 1000 Hz is unexpected, since more information on the movement would be expected to yield better results. However, in the case of processing with a neural network, the most probable explanation is that there is too much information in the windows. Therefore the model overfits on the detailed windows, which do not properly match the slightly different windows in the test set.

Figure 4.2 compares the classification results at a sampling frequency of 500 Hz to the results at a sampling frequency of 100 Hz. We note that all shots are being classified correctly at both sampling frequencies. Figure 4.2a shows that at 500 Hz each class has a satisfactory true positive rate of more than 0.9. If we compare this to Figure 4.2b, which shows the 100 Hz results, we notice clearly lower true positive rates. We note that shots are still properly classified, as they show a clear pattern of multiple samples at maximum sensor readings, which is retained at 100 Hz. However, for the shorter and more subtle peaks in the noise, dribble and pass samples, we note that the distinguishing patterns vanish due to the lower sampling frequency, leading to a lot of misclassifications between these classes.

Based on this experiment we continue to use a sampling frequency of 500 Hz.

4.1.2 Training Set Size

When performing data collection it is relevant to know when there is enough data available to develop a well-performing model. Figure 4.3 shows the classification results when reducing the number of training samples. We notice that when only dropping 10% of the training the performance already drops, therefore we can conclude that more training could be beneficial to improve the accuracy. When using 10% of the training set, we still get an accuracy higher than 90%, due to the large number of noise samples in the dataset. However, we note in Figure 4.3b that it results in clear misclassifications of dribbles as noise, which makes sense, since the noise samples can contain many random movements. Since the model can not properly generalize yet what a dribble is supposed to look like, it often considers it as noise. Similarly for the shots, since there are only a few examples of shots in the training set, and a soft shot can be very similar to a pass, they might be misclassified as a pass.

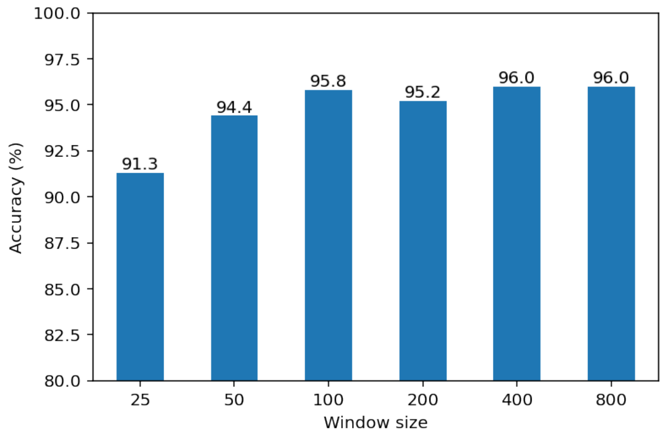


(a) **Accuracy at different training set sizes.** (b) **10% of the training set size.**

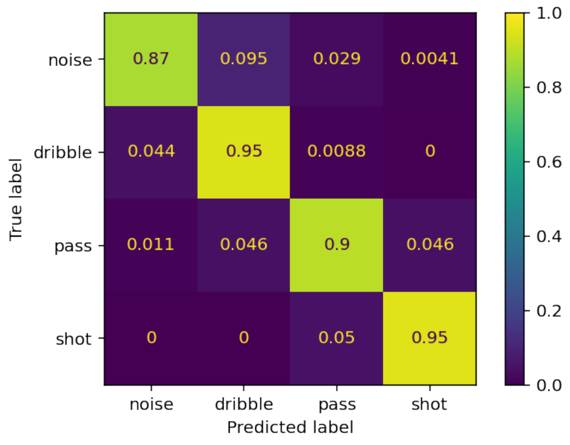
Figure 4.3: **Classification results with varying training set sizes.**

4.1.3 Window Size

Varying the window size we find that using 100 samples, which translates to 200 ms at 500 Hz, is the optimal setting. This corresponds to the findings in Section 3.3.2, that show a helpful characteristic peak in the dribble window which clearly distinguishes between noise and a dribble. Looking at Figure 4.4a, we notice that using larger windows provides a similar performance, showing that all relevant information is in the 200 ms window. Since the resource use increases significantly with larger window sizes, as shown in Section 4.2.1, there is no added value to utilize them. When lowering the window size to 25 samples and looking at the corresponding confusion matrix in Figure 4.4b, a few values stand out. As explained, the model can not properly distinguish between noise and a dribble, thus many noise samples are classified as dribbles. Similarly, the window size limits the capability to distinguish the short peak of a pass from the longer time of the impact of a shot. Finally, we notice the spread of misclassifications. While misclassifying some noise samples as dribbles is unavoidable in machine learning, a noise sample should never be classified as a shot.



(a) **Accuracy at different window sizes.**



(b) **Window size of 25.**

Figure 4.4: **Classification results with varying training set sizes.**

4.1.4 Neural Network Performance Comparison

As shown in Section 3.4, we compare a dense layer model, a one-layer CNN model, a multi-layer CNN model and an LSTM model. Additionally, we analyze the impact of the following parameters: window size, number of convolutional layers, kernel size, number of kernels and number of dense layers. Figure 4.5 shows the accuracy of the mentioned different neural models, all run with a window size of 100 at 500 Hz.

The classification performance of the one-layer CNN is covered already in Section 4.1.1, and the multi-layer CNN models unfortunately show no improved results. Especially with three convolutional layers, we notice the accuracy actually going down again. While the multi-layer CNN models show no statistically relevant higher accuracy, with more insight into the training data and network design, it might be possible to utilize the additional layers to better extract the relevant features.

The LSTM model described in Section 3.4.4 does not show improved classification performance compared to the CNN models, while the model size is more than 15 times larger. Additionally, LSTM layers are currently still under development to be supported in TensorFlow Lite Micro[1]. While the LSTM approach does not show added value for ball-touch detection, corresponding to the findings by Stoeve *et al.* [4], when the implementation is available and the resource usage can be benchmarked, other applications can be considered that are better suited for the use of LSTMs, as shown by Taghanaki *et al.* [5] and Sherratt *et al.* [24]. The dense layer approach clearly performs worse than the CNN models, but has still a reasonable accuracy, considering that it just looks at the raw data as numbers. Due to this way of processing the data it has a hard time distinguishing the difference between noise and pass samples.

The experiments not yet covered, varying the kernel size and the number of kernels, show a non-significant difference in accuracy varying between 94.0% and 95.8%, and are therefore not extensively covered. Again, while they show no added value for in classification performance, the resource usage analysis of these parameters in Section 4.2 is still relevant, as it might be useful for future different classification tasks.

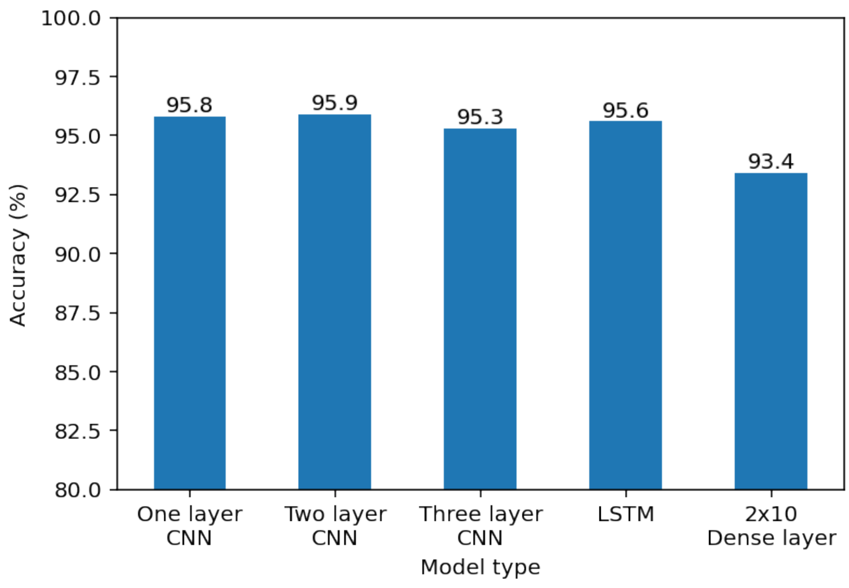


Figure 4.5: **Accuracy of various models.**

4.1.5 XGBoost

As mentioned in Section 2.2, XGBoost is a state of the art machine learning algorithm, used in many classification problems. Compared to neural networks, we can not feed the raw IMU data to train an XGBoost model. Instead, statistical features need to be selected and calculated, which can then be used for training. As a comparison, an XGBoost model has been trained with the minimum, maximum, mean, kurtosis, skew and standard deviation values of the six separate axes. Figure 4.6 shows the classification result. The first conclusion we can draw is that it is hard for the model to distinguish a shot from a pass.

[1]https://github.com/tensorflow/tflite-micro/issues/920

This makes sense if we look back at Figure 3.4c and Figure 3.4d, where we see that a strong pass and a shot both have peaks at the maximum accelerometer reading, with the difference that the shot sample has multiple of those readings. This approach takes just a minimum and maximum value for the whole window, and therefore misses the fact that multiple samples in a shot window are at the minimum and maximum value. Similarly, when differentiating between noise and a dribble, the pattern throughout the window is important, which is missed by this approach.

Of course when calculating the statistical features for the XGBoost model, we could divide the event window into multiple subwindows, and calculate the statistical features for each subwindow. However, that would require a lot of manual fine-tuning to select the right number of subwindows, and select the relevant features which actually add information. Due to this downside, the neural network approach shows to be a proper improvement, considering the self-learning capabilities and the good results.

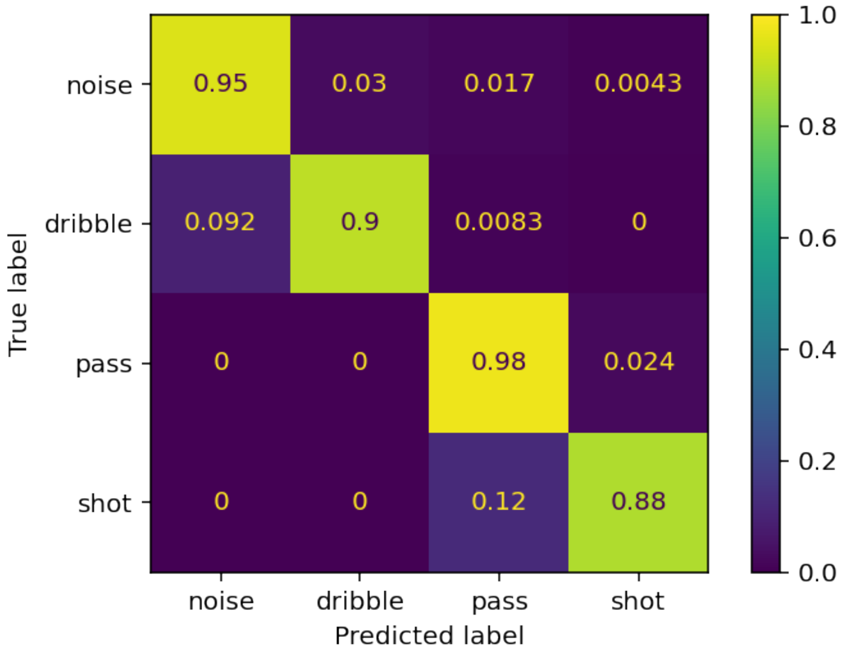


Figure 4.6: **Confusion matrix of classification with XGBoost.**

4.2 Resource Experiments

In addition to the classification experiments, this section covers a selection of resource experiments. To understand how design decisions will impact the performance and resource usage, parameters such as the window size, number of network layers, kernel size and number of kernels are changed and benchmarked. The results are visualized in graphs in this section, while the details can be found in Appendix A. The experiments are performed with the one-layer CNN model

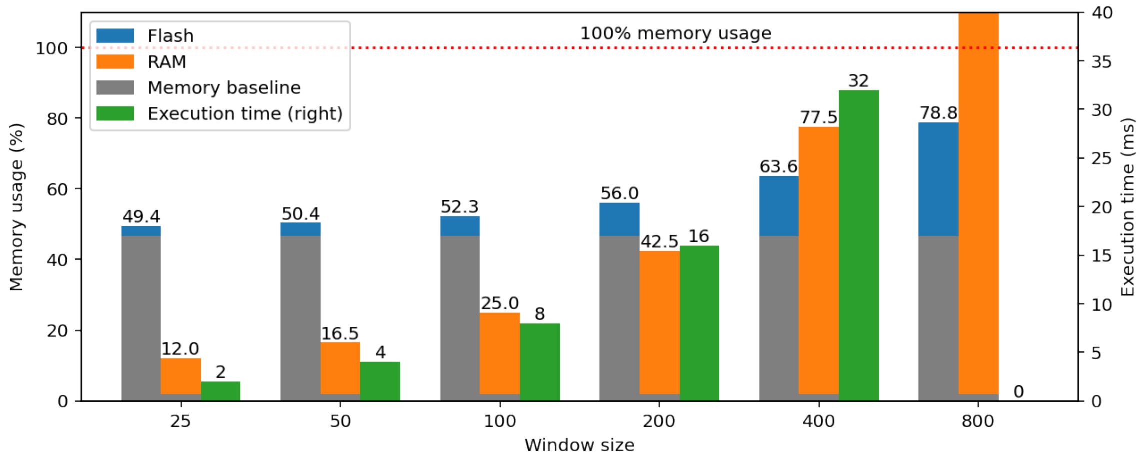


Figure 4.7: **Results of resource experiment with varying window size.**

as described in Section 3.4.2, using four kernels, a 3x1 kernel size and a window size of 100 at 500 Hz, unless specified otherwise.

Before going into details, it is important to note the purpose of the Flash and RAM when using TensorFlow. The Flash is used to store the complete model, information about the network design, the number and kind of layers, and fixed values such as the weights for the neurons in the dense layers. The RAM is used to store the input buffer to the network, but also convolution values, which are calculated on-the-go during the execution of the model.

As mentioned in Section 3.1.3, there is 124 KB Flash and 20 KB RAM available. The baseline memory usage for the TensorFlow Lite for Microcontrollers implementation on the sensor is 41.3 KB Flash when using just dense layers, and 57.8 KB Flash when also using convolutional layers. The RAM usage is in both cases 0.4 KB. This baseline usage is displayed in the figures in this section as a percentage of the total memory usage.

4.2.1 Window Size

Figure 4.7 shows the results of the window size experiment. The window size setting starts at 25 and goes up to 800. The impact on Flash usage is caused by the general increase of parameters in the model. While the increase at smaller window sizes is still small due to overhead, the increase gets more significant as the window size gets larger. The RAM usage increases exponentially, so at a window size of 800 the limit of the RAM is reached, and the model can not be run on the sensor anymore. This increase is caused by the doubling of input values, but also the additional convolution values which have to be calculated, since there are more input values. Finally, the execution time doubles every time the input window size doubles, which is a clear linear relation with the number of values the neural network has to process.

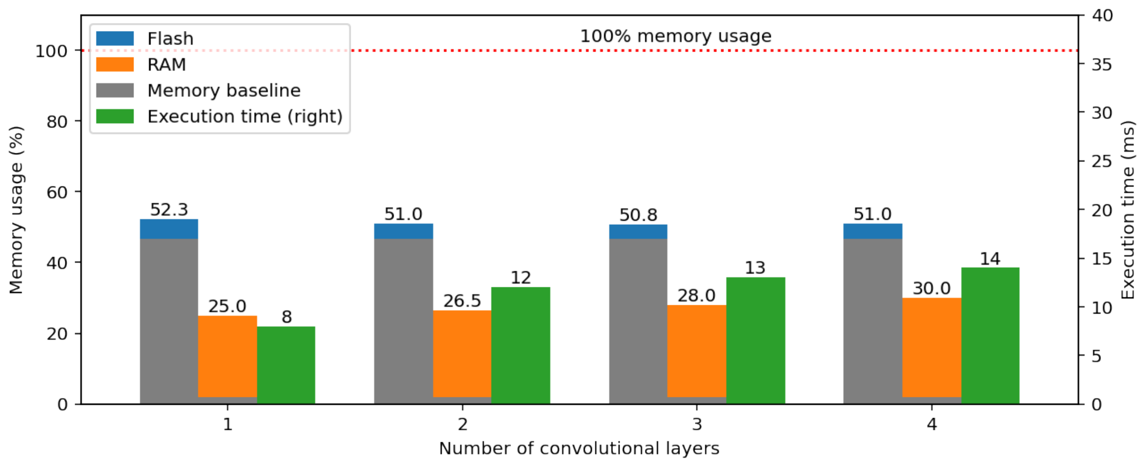


Figure 4.8: **Results of resource experiment with varying number of convolutional layers.**

4.2.2 Number of Convolutional Layers

Figure 4.8 shows the results of the convolutional layer experiment. When adding more convolutional layers, the convolution process has to be performed multiple times. However, as explained in Section 3.4.3, between every additional convolutional layer, a pooling layer is added. This pooling layer halves the number of input values to the next convolutional layer, resulting in fewer input values for the final dense layer. Since the input size to the final dense layer is an important contributor to the model size, we see a slight decrease in Flash size as a result. On the other hand, since every convolutional layer requires values to be calculated, additional RAM is needed to store those values. The execution time increases as expected with every added convolutional layer. The decaying increase is also expected, again because of the pooling layers. For example with the four convolutional layers, the input to the final convolutional layer is only a 10x6 window, compared to an input window of 100x6 to the first convolutional layer, thus having a smaller contribution to the total execution time.

4.2.3 Kernel Size

Figure 4.9 shows the results of the kernel size experiment. When using a bigger kernel size, the output of the convolutional layer will be smaller, since the kernel needs more spacing at the borders. This results in a smaller model size and less Flash usage, since the final dense layer has to process fewer values. Similarly, the RAM usage is also smaller, since fewer convolution result values have to be stored. Contrary to the very small decrease in memory usage, the execution time does in fact increase. This increase is expected since the matrix multiplication as part of applying the convolution kernel to the data, gets more complex the bigger the kernel is.

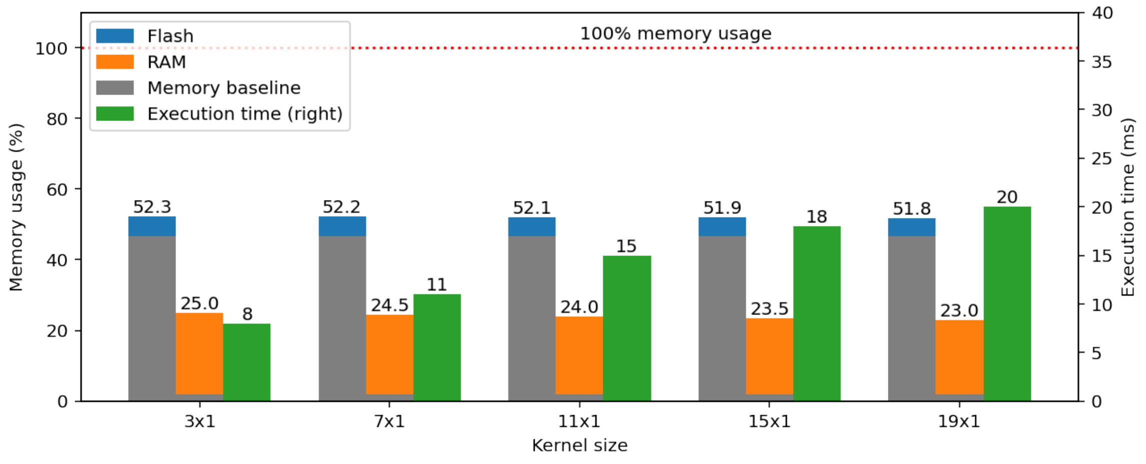


Figure 4.9: **Results of resource experiment with varying kernel size.**

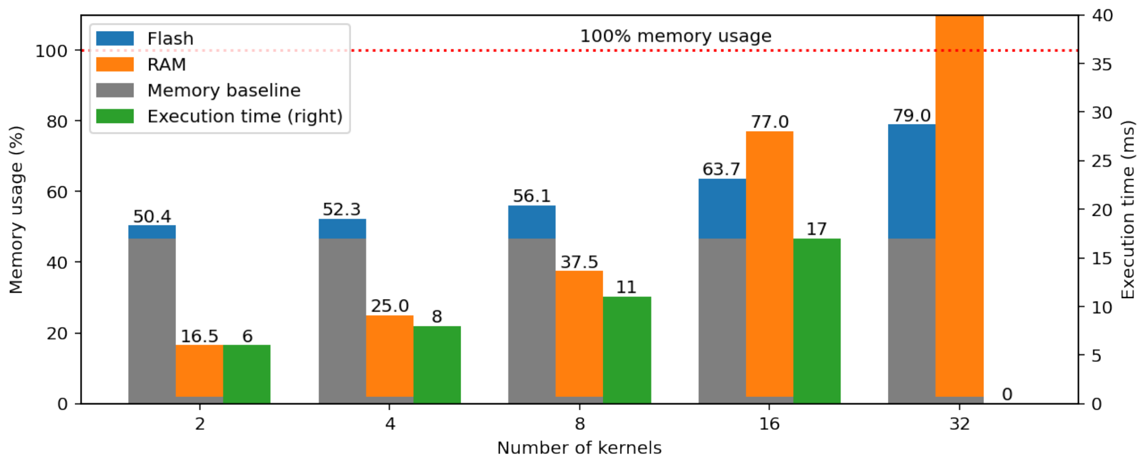


Figure 4.10: **Results of resource experiment with varying number of kernels.**

4.2.4 Number of Kernels

Figure 4.10 shows the results of the kernel number experiment. With this experiment, all metrics get bigger when the number of kernels increases. This makes sense, since this method is strictly adding information to every convolution step. This adds up to the total number of parameters that will be fed to the final dense layer, thus adding to the Flash memory usage. Since more buffers are needed to store all these values, which are calculated during the execution of the network, we notice that while there is some overhead at smaller numbers of kernels, the RAM usage doubles when using 16 or more kernels. This results in a maximum feasible number of kernels of 16. Finally, the execution time increases as expected.

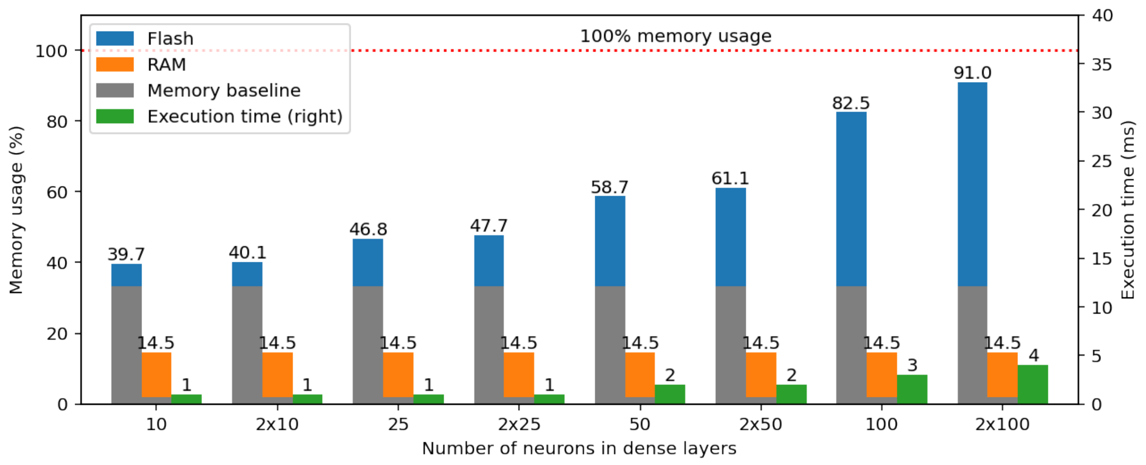


Figure 4.11: **Results of resource experiment with varying number of neurons per dense layer.**

4.2.5 Number of Dense Layers

Figure 4.11 shows the results of the number of neurons experiment. This experiment uses the dense model as described in Section 3.4.1, with one or two dense layers. The numbers on the horizontal axis indicate how many neurons are used and whether the model uses one or two dense layers. This experiment clearly shows the difference between the convolution, which is a mathematical processing step, and applying a dense layer. The dense layer consists of neurons with a certain fixed weight as a result of the training process, and therefore significantly impacts the Flash memory as the number of neurons increases. Following the same reasoning that there is just an increase in fixed numbers, the RAM usage does not significantly increase when adding more neurons. Finally, the execution times are quite small compared to a convolutional model, corresponding to the network analysis by Faraone and Delgado-Gonzalo [21], mentioned in Section 2.2.1.

4.2.6 Optimizations

Figure 4.12 and Table A.6 show a comparison of the resource usage of different optimizations of the one-layer CNN model described in Section 3.4.2. It appears that, as a result of quantization, the model size is decreased by a factor 2.9, which decreases the Flash and RAM requirements significantly, as they now do not have to store float values, but just 8-bit integer values. The speedup from using CMSIS-NN brings the execution down significantly, however increases the Flash usage by 8 KB due to the necessary source code for the implementation. An odd observation is that the execution time of the network when applying quantization while not using CMSIS-NN results in a higher execution time than when using the non-optimized network. Further investigation should be conducted to find the cause, since it seems to be a bug.

Comparing the model with quantization and CMSIS-NN enabled to the model without these optimizations results in an 8% decrease in Flash usage, 67% de-

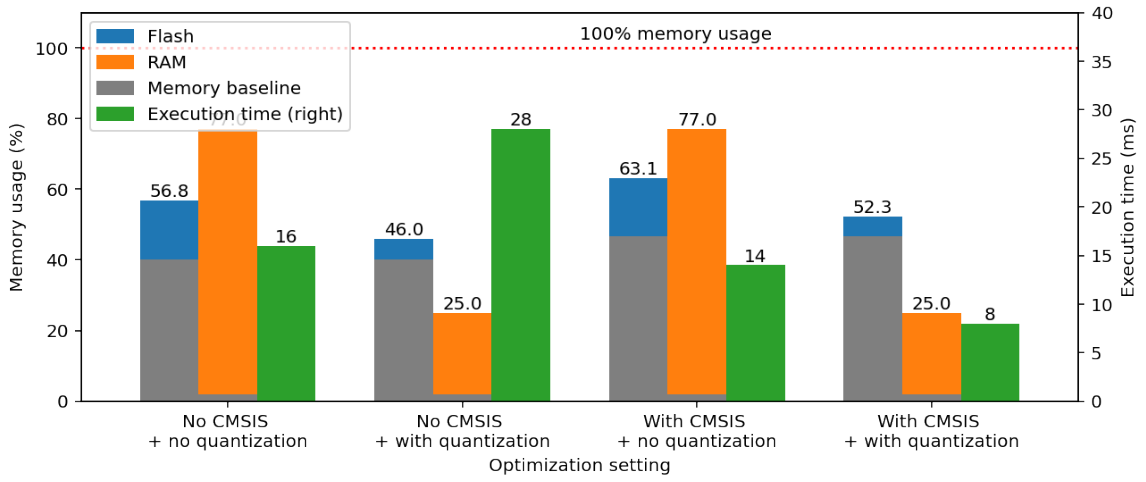


Figure 4.12: **Results of resource experiment with varying optimization settings.**

crease in RAM usage and a 2x improvement in execution time. While the authors of CMSIS-NN advertise a higher improvement in execution time of 4.6x, this was achieved on a higher-end microcontroller using a larger network, consequently with smaller overhead and more room for improvement. Still, the improvements both in memory usage and execution time, make these optimizations very valuable and allow to deploy more complex neural networks on such constrained devices.

4.3 Discussion

This chapter described the performance of different models and their parameters, explaining both classification performance and resource performance. Based on the experiments we conclude that a one-layer CNN model, as described in Section 3.4.2, with 4 kernels and a 3x1 kernel size and a window size of 100 at 500 Hz is best suited for the classification problem of distinguishing noise, dribble, pass and shot samples with an accuracy of 95.8%.

The LSTM approach appeared not to be of added value to the ball-touch classification problem, but should still be considered in the future for other classification tasks on the JOGO sensor, such as running style as shown by Taghanaki *et al.* [5] and Sherratt *et al.* [24].

Section 4.2 shows the various parameters and their impact on the resource usage. This provides a reference for future development of the neural network approach for the JOGO sensor.

Chapter 5

Conclusions and Future Work

The goal of this thesis project was to explore the possibilities and limitations of running neural networks on a constrained Cortex-M based sensor platform to accurately classify ball touches in football. Multiple papers are available proposing the use of neural networks to process IMU sensor data for sports movement classification, however they only use the sensor to gather the data, while the classification is done on a PC [5, 12, 13, 14]. While this thesis proposes a similar approach, the approach itself is also actually implemented on the embedded device, highlighting the distinguishing challenge in this work. This challenge has been summarized in the following research question in Section 1.2:

How can neural networks be applied for embedded real-time classification of ball touches in football?

5.1 Conclusions

A development flow has been set up, which consists of some preparatory steps, the actual training of the network, and deployment to the sensor. The preparatory steps include importing the labeled data, preprocessing and network design setup. The actual training can then be performed, with options to modify the relevant parameters, after which the training results are visualized. If the model performance is satisfactory, the model can be converted to a byte array and included in the modified firmware.

With the neural network designs provided in Chapter 3 several experiments have been conducted, as described in Chapter 4. We find that we can use IMU data sampled at 500 Hz, combined with a one-layer CNN to achieve the result as shown in Figure 4.2a, with a classification accuracy of 95.8%. The implementation of this neural network on the sensor uses 64.9 KB Flash and 5.0 KB RAM, and takes 8 ms to execute, which is well within the restriction described in Section 3.1.

In addition to a satisfactory classification performance, Section 4.2 explores the resource impact of different parameters. This provides insight into the possibilities when JOGO's data scientists continue development with neural networks, either to improve the current approach, or to solve other classification problems.

To summarize, we have shown how we can apply neural networks for real-time classification of ball touches, by proposing an approach which fits well within the resource limitations of the sensor, and offers similar or better classification compared to other sport classification research [12, 14], which did not offer real-time classification.

5.2 Future Work

An important footnote is that this work has been conducted mainly from an embedded perspective. While JOGO's data scientists have shared their knowledge during the development, there are aspects that they can further improve when integrating neural networks in the next version of the sensor. One of the limitations during the design was the lack of insight into what happens between multiple convolutional layers, improving this combined with more training data and optimizing the use of convolution, might actually unlock the full potential of using multiple convolutional layers instead of just one. Similarly, the LSTM approach might have more to offer. While it shows no added value to classify ball touches, Taghanaki *et al.* [5] and Sherratt *et al.* [24] show the benefit of LSTM for specific insight into walking and running styles. Therefore, it might be suited for other football-related classification tasks, which can be added to the sensor in the future. Possible examples are fatigue monitoring, or an automatic start and stop of a session that can detect when a person starts playing football, by distinguishing the gait of a person during a match compared to regular activities.

Two methods to improve the event detection and the ball-touch classification are worth mentioning. The first is matching the IMU data from both feet in real-time. Schuldhaus *et al.* [14] applied this concept, while not in real-time, to enhance the event detection. This works by comparing the peak in the data of the foot that hits the ball with the data of the standing leg, which is still on the ground. This could be a beneficial approach to, for example, lower the number of false positives in the event detection or to find a relation between the two feet for each ball touch. However, it comes at an additional cost of processing time, battery usage and complexity of the communication between the two sensors.

The second method is using piezoelectric material as an additional sensor. This kind of sensor comes in different forms, for example as proposed by Mao *et al.* [35] and de Fazio *et al.* [36]. Such a sensor can provide pressure information on multiple points of the foot, which can be used as additional inputs for ball-touch classification, or for gait and fatigue monitoring. In addition to the provided sensor data, piezoelectric sensors can be used for energy harvesting. This is quite promising as a football player is constantly moving and thus providing energy to the sensor. Combined with further research into the power consumption of the different neural network implementations, this approach could be beneficial to prolong the battery life or to extend the power budget for the sensor to allow more functionality.

Bibliography

[1] Mark Waldron, Jamie Harding, Steve Barrett, and Adrian Gray. A New Foot-Mounted Inertial Measurement System in Soccer: Reliability and Comparison to Global Positioning Systems for Velocity Measurements During Team Sport Actions. *Journal of Human Kinetics*, 77(1):37–50, 2021.

[2] Hongyu Zhao, Zhelong Wang, Sen Qiu, Yanming Shen, Luyao Zhang, Kai Tang, and Giancarlo Fortino. Heading Drift Reduction for Foot-Mounted Inertial Navigation System via Multi-Sensor Fusion and Dual-Gait Analysis. *IEEE Sensors Journal*, 19(19):8514–8521, 2018.

[3] Maoran Zhu, Yuanxin Wu, and Shitu Luo. f^{2}IMU-R: Pedestrian Navigation by Low-Cost Foot-Mounted Dual IMUs and Interfoot Ranging. *IEEE Transactions on Control Systems Technology*, 30(1):247–260, 2021.

[4] Maike Stoeve, Dominik Schuldhaus, Axel Gamp, Constantin Zwick, and Bjoern M Eskofier. From the Laboratory to the Field: IMU-Based Shot and Pass Detection in Football Training and Game Scenarios Using Deep Learning. *Sensors*, 21(9):3071, 2021.

[5] Setareh Rahimi Taghanaki, Michael Rainbow, and Ali Etemad. Wearable-based Classification of Running Styles with Deep Learning. *arXiv preprint arXiv:2109.00594*, 2021. Accepted to the 17th IEEE-EMBS International Conference on Wearable and Implantable Body Sensor Networks (BSN).

[6] Maximilian T Fischer, Daniel A Keim, and Manuel Stein. Video-based Analysis of Soccer Matches. In *Proceedings Proceedings of the 2nd International Workshop on Multimedia Content Analysis in Sports*, pages 1–9, 2019.

[7] Carlos Cuevas, Daniel Quilon, and Narciso García. Techniques and applications for soccer video analysis: A survey. *Multimedia Tools and Applications*, 79(39):29685–29721, 2020.

[8] Takamasa Tsunoda, Yasuhiro Komori, Masakazu Matsugu, and Tatsuya Harada. Football Action Recognition Using Hierarchical LSTM. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 99–107, 2017.

[9] John S Theodoropoulos, Jeremy Bettle, and Jonathan D Kosy. The use of GPS and inertial devices for player monitoring in team sports: A review of current and future applications. *Orthopedic reviews*, 12(1), 2020.

[10] Jonathon G Neville, David D Rowlands, James B Lee, and Daniel A James. A Model for Comparing Over-Ground Running Speed and Accelerometer Derived Step Rate in Elite Level Athletes. *IEEE Sensors Journal*, 16(1): 185–191, 2015.

[11] Emily E Cust, Alice J Sweeting, Kevin Ball, and Sam Robertson. Machine and deep learning for sport-specific movement recognition: A systematic review of model development and performance. *Journal of sports sciences*, 37(5):568–600, 2019.

[12] Bernhard Hollaus, Sebastian Stabinger, Andreas Mehrle, and Christian Raschner. Using Wearable Sensors and a Convolutional Neural Network for Catch Detection in American Football. *Sensors*, 20(23):6722, 2020.

[13] Thomas Kautz, Benjamin H Groh, Julius Hannink, Ulf Jensen, Holger Strubberg, and Bjoern M Eskofier. Activity recognition in beach volleyball using a Deep Convolutional Neural Network. *Data Mining and Knowledge Discovery*, 31(6):1678–1705, 2017.

[14] Dominik Schuldhaus, Constantin Zwick, Harald Körger, Eva Dorschky, Robert Kirk, and Bjoern M Eskofier. Inertial Sensor-Based Approach for Shot / Pass Classification During a Soccer Match. In *KDD workshop on large-scale sports analytics*, pages 1–4, 2015.

[15] Joseph McGrath, Jonathon Neville, Tom Stewart, Hayley Clinning, and John Cronin. Can an inertial measurement unit (IMU) in combination with machine learning measure fast bowling speed and perceived intensity in cricket? *Journal of Sports Sciences*, 39(12):1402–1409, 2021.

[16] Roland van den Tillaar, Shruti Bhandurge, and Tom Stewart. Can Machine Learning with IMUs Be Used to Detect Different Throws and Estimate Ball Velocity in Team Handball? *Sensors*, 21(7):2288, 2021.

[17] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[18] Bendong Zhao, Huanzhang Lu, Shangfeng Chen, Junliang Liu, and Dongya Wu. Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics*, 28(1):162–169, 2017.

[19] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[20] Franz MJ Pfister, Terry Taewoong Um, Daniel C Pichler, Jann Goschenhofer, Kian Abedinpour, Muriel Lang, Satoshi Endo, Andres O Ceballos-Baumann, Sandra Hirche, Bernd Bischl, *et al.* High-Resolution Motor State Detection in Parkinson's Disease Using Convolutional Neural Networks. *Scientific reports*, 10(1):1–11, 2020.

[21] Antonino Faraone and Ricard Delgado-Gonzalo. Convolutional-Recurrent Neural Networks on Low-Power Wearable Platforms for Cardiac Arrhythmia Detection. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 153–157. IEEE, 2020.

[22] Christopher Olah. Understanding LSTM Networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015. Last accessed: Mar. 24, 2022.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 9(8):1735–1780, 1997.

[24] Freddie Sherratt, Andrew Plummer, and Pejman Iravani. Understanding LSTM network behaviour of IMU-based locomotion mode recognition for applications in prostheses and wearables. *Sensors*, 21(4):1264, 2021.

[25] Embedded Learning Library (ELL). `https://microsoft.github.io/ELL/`. Accessed: 2021-11-10.

[26] *Caffe Model Development on MNIST Dataset with CMSIS-NN Library.* NXP Semiconductors, 4 2020. Rev. 0.

[27] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, *et al.* TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *Proceedings of Machine Learning and Systems*, 3:800–811, 2021.

[28] Liangzhen Lai, Naveen Suda, and Vikas Chandra. CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs. *arXiv preprint arXiv:1801.06601*, 2018.

[29] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[30] Chris Coleman. A Practical Guide to BLE Throughput. `https://interrupt.memfault.com/blog/ble-throughput-primer`, 2019. Last accessed: May 16, 2022.

[31] *ICM-42688-P Datasheet.* TDK InvenSense, May 2021. Rev. 1.5.

[32] Warren S. Sarle. comp.ai.neural-nets FAQ, Part 2 of 7: Learning. `http://www.faqs.org/faqs/ai-faq/neural-nets/part2/`, 2002. Last accessed: Apr. 17, 2022.

[33] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[34] Jason Brownlee. Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. `https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/`, 2017. Last accessed: May 16, 2022.

[35] Yupeng Mao, Mailun Shen, Bing Liu, Lili Xing, Song Chen, and Xinyu Xue. Self-Powered Piezoelectric-Biosensing Textiles for the Physiological Monitoring and Time-Motion Analysis of Individual Sports. *Sensors*, 19 (15):3310, 2019.

[36] Roberto de Fazio, Elisa Perrone, Ramiro Velázquez, Massimo De Vittorio, and Paolo Visconti. Development of a Self-Powered Piezo-Resistive Smart Insole Equipped with Low-Power BLE Connectivity for Remote Gait Monitoring. *Sensors*, 21(13):4539, 2021.

Appendix A

Resource Experiment Data

Window size	**Model parameters**	**Model size (KB)**	**Flash (KB)**	**RAM (KB)**	**Execution time (ms)**
25	1076	3.6	61.3	2.4	2
50	2324	4.8	62.5	3.3	4
100	4724	7.2	64.9	5.0	8
200	9524	12.0	69.5	8.5	16
400	19124	21.6	78.9	15.5	32
800	38324	40.8	97.7	29.4	xxx

Table A.1: **Results of resource experiment with varying window size.**

Convolutional layers	**Model parameters**	**Model size (KB)**	**Flash (KB)**	**RAM (KB)**	**Execution time (ms)**
1	4724	7.232	64.9	5.0	8
2	2280	5.656	63.3	5.3	12
3	1084	5.328	63	5.6	13
4	560	5.664	63.3	6.0	14

Table A.2: **Results of resource experiment with varying number of convolutional layers.**

Kernel size	**Model parameters**	**Model size (KB)**	**Flash (KB)**	**RAM (KB)**	**Execution time (ms)**
3x1	4724	7.2	64.9	5.0	8
7x1	4548	7.1	64.7	4.9	11
11x1	4372	6.9	64.6	4.8	15
15x1	4196	6.7	64.4	4.7	18
19x1	4020	6.5	64.2	4.6	20

Table A.3: **Results of resource experiment with varying kernel size.**

Number of kernels	Model parameters	Model size (KB)	Flash (KB)	RAM (KB)	Execution time (ms)
2	2364	4.8	39.9	3.3	6
4	4724	7.2	41.6	5.0	8
8	9444	12.1	44.1	7.5	11
16	18884	21.7	52	15.4	17
32	37764	41.0	66	29.4	xxx

Table A.4: **Results of resource experiment with varying number of kernels.**

Number of neurons	Model parameters	Model size (KB)	Flash (KB)	RAM (KB)	Execution time (ms)
10	6054	8.0	49.2	2.4	0.5
25	15129	17.1	58.0	2.4	1
50	30254	32.2	72.8	2.4	2
100	60504	62.4	102.3	2.4	3
2x10	6164	8.6	49.7	2.9	0.5
2x25	15779	18.2	59.2	2.9	1
2x50	32804	35.3	75.8	2.9	2
2x100	70604	73.2	112.9	2.9	4

Table A.5: **Results of resource experiment with varying number of neurons per dense layer.**

CMSIS-NN enabled	Quantization enabled	Model parameters	Model size (KB)	Flash (KB)	RAM (KB)	Execution time (ms)
No	No	4724	20.9	70.4	15.4	16
	Yes	4724	7.2	57.0	5	28
Yes	No	4724	20.9	78.3	15.4	14
	Yes	4724	7.2	64.8	5	8

Table A.6: **Results of resource experiment with varying optimization settings.**