



ASHESI UNIVERSITY

Edge Computing and Machine Learning on Embedded Systems

CAPSTONE PROJECT

BSc. Computer Engineering

Lorraine Makuyana

2022

ASHESI UNIVERSITY

**EDGE COMPUTING AND MACHINE LEARNING ON EMBEDDED
SYSTEMS**

CAPSTONE PROJECT

Capstone project submitted to the Department of Engineering, Ashesi University in partial fulfillment of the requirements for the award of Bachelor of Science degree in Computer Engineering.

Lorraine Makuyana

2022

DECLARATION

I hereby declare that this capstone is the result of my own original work and that no part of it has been presented for another degree in this university or elsewhere.

Candidate's Signature:

.....

Candidate's Name:

.....

Date:

I hereby declare that preparation and presentation of this capstone were supervised in accordance with the guidelines on supervision of capstone laid down by Ashesi University College.

Supervisor's Signature:

.....

Supervisor's Name:

.....

Date:

Acknowledgments

To my supervisor, Dr. Nathan Amanquah, whose encouragement, academic advice, and guidance helped me undertake this project.

Abstract

Running Machine Learning (ML) in embedded systems has fueled the rush for edge computing, where machine learning runs in edge devices. This approach to ML yields many results, such as lower latencies and reduction of network traffic and bandwidth. This project set out to explore machine learning in embedded systems. The Edge Impulse Platform was used to collect data and to a neural network. The neural network created was able to distinguish between five classes of motion. The Neural Network created was tested on two microcontrollers and a desktop. Inferencing on the Arduino Nano BLE took 24ms, and on the desktop, it took 271.8 μ s.

Table of Contents

DECLARATION.....	ii
Acknowledgments	iii
Abstract.....	iv
Table of Contents	v
List of Figures.....	viii
Chapter 1: Introduction and Background	1
1.1 Background	1
1.2 Objective	6
Chapter 2: Literature Review and Related work.....	8
2.1 Machine Learning (ML).....	8
2.2 Neural Networks (NN).....	8
2.3 Tiny Machine Learning (TinyML).....	12
2.4 Strategies for deploying Machine Learning in Embedded Systems	12
2.5 Project Scope.....	15
Chapter 3: System Design and Implementation	17
3.1 Chapter Objectives	17
3.2 Use Case.....	17
3.3 User Requirements	18

3.3.1 Functional and non-functional Requirements.....	18
3.4 Choice of microcontroller	18
3.5 Workflow Overview.....	19
3.6 Implementation Steps in detail.....	21
3.7 Creating a NN machine learning model on Edge Impulse.....	21
3.7.1 Data Acquisition.....	21
3.7.2 Impulse Design: Developing the Neural Network	25
3.7.3 Training	27
3.7.4 Anomaly Detection.....	29
3.7.5 Model Testing.....	30
3.7.6 Model Deployment.....	31
3.8 Building a web application to interface with FRDM-K64F.....	32
3.8.1 Application Technology Stack	32
3.8.2 How it works	32
Chapter 4: Results and Analysis	34
4.1 Chapter Objectives	34
4.2 Running the model	34
4.2.1 Deploying to Arduino Nano 33 BLE.....	34
4.2.2 Deploying to the FRDM-K64F development board.....	36
4.2.3 Deploying on Desktop PC	37

Chapter 5: Conclusion	39
5.1 Chapter Objectives	39
5.2 Limitations	39
5.3 Future Work	40
References.....	42
Appendix.....	45
A. Code for Arduino Nano BLE.....	45
B. Code for FRDM-K64F.....	48
C. Challenges and ways to address them.....	51
1. Undefined references to functions.....	51
2. Data Collection using the Edge Impulse CLI.....	51

List of Figures

Figure 1.1: Steps in embedded ML with inference done on the cloud	2
Figure 2.1: Inputs and outputs of a neural network	9
Figure 2.2: Deep neural network (DNN) [17]	10
Figure 2.3: FPGA in an embedded system [16].....	14
Figure 2.4: ML audio classification model with ARM M4 and M7.....	15
Figure 3.1: FRDM-K64F Development Board with MK64FN1M0VLL12 microcontroller	19
Figure 3.2: Different possible configurations of an embedded system E	20
Figure 3.3: Workflow overview.....	21
Figure 3.4: Data samples for motion classes from Edge Impulse platform.....	23
Figure 3.5: Train and Test splits before and after balancing data sets.....	24
Figure 3.6: Impulse design on EI platform	25
Figure 3.7: Feature explorer for ML model classes	26
Figure 3.8: Neural network architecture	27
Figure 3.9: Confusion matrices for (a) float32 and (b) int8 model.....	28
Figure 3.10: Anomaly explorer for ML model	29
Figure 3.11: Testing sample results with new data for left-right class	30
Figure 3.12: Testing result for an anticlockwise class sample from test set data	31
Figure 3.13: Web application data flow diagram.....	32
Figure 3.14: Screenshots of web application	33
Figure 4.1: Predictions on an Arduino Nano BLE.....	35
Figure 4.2: Classification of anticlockwise (left) and left-right (right) classes on a Desktop PC	37

Chapter 1: Introduction and Background

1.1 Background

Artificial Intelligence (AI) has been on the rise with the technology revolution and has evolved from being used in remote applications to driving applications that are a significant part of our everyday lives [1]. Machine Learning (ML), a subset of AI, is a method designed for data analysis, enabling automation for intelligent systems by identifying data, recognizing data patterns, and making data-driven predictions with little to no human interaction. In some devices, ML has been so carefully developed to run accurately and seamlessly without supervision and has been the reason for scalability and sustainability in several industries across the world [2].

Machine Learning models are sometimes complex, with large data sets and intensive computations that require large memory and expensive computing power. However, extending ML to run on inexpensive hardware has helped make this efficient technology accessible, even in the smallest systems on network edges, by using hardware such as microcontroller units [3]. Running ML models on the edge, i.e., edge computing where data processing in an embedded system is done at the data source and closer to the network edge, has provided many advantages for several applications and created hope for advanced embedded applications that would previously drain network traffic [3]. This is because in some current systems, data collection is done at the data source by the sensors, and inference happens on the cloud, and then the results are sent back to the embedded system via the internet. A pictorial view of this process is shown in Figure 1.1. This drains network traffic and increases bandwidth, introducing delays and possible errors and cyber-attacks. The realization of such advanced systems relies on the ability of devices and processes to analyze large amounts of data, also taking into consideration that edge

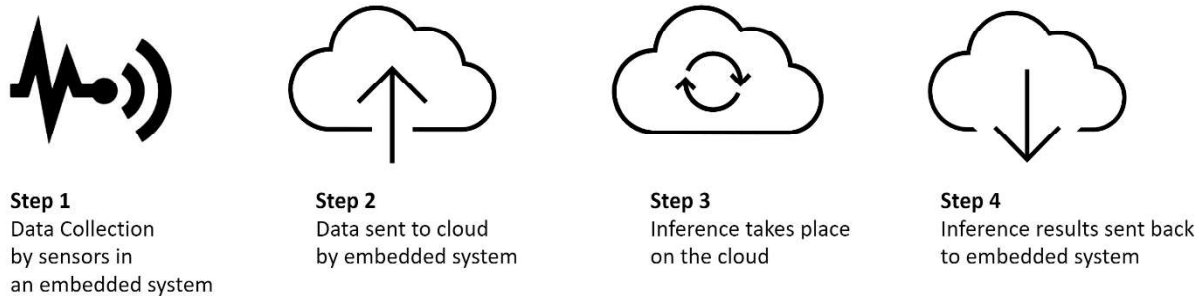


Figure 1.1: Steps in embedded ML with inference done on the cloud

devices usually cannot perform the computations required for data analytics since their sole purpose is to monitor and transmit data to more powerful systems, which usually are cloud-based [4].

Edge computing has many crucial advantages. Lower latencies can be achieved by moving computations closer to network edges and data sources, eliminating several data exchanges between data sources and computation systems, meaning no delay in computations and inferencing [4]. This reduces network traffic while ensuring complex tasks are still executed accurately. In these cases, there is efficient resource usage and effective bandwidth usage. Also, higher accuracies are reached since there are fewer errors from data transfer to data processing centers or systems, owing to the use of raw and not aggregate data [4]. For the same reasons, higher energy efficiency and better privacy are achieved as data is not moved between the source and the cloud [4]. Sustainability is also an added advantage of ML on embedded devices since microcontrollers used for these tasks are power efficient, reducing the carbon footprint. These reasons have advanced the use of edge computing in new applications and its adoption in existing ones. Embedded applications can utilize the data from sensors through low-power devices that can perform intensive computations efficiently on microcontrollers.

ML on embedded devices has several applications, ranging from vibration and motion detection, and voice and sound recognition, to vision and image detection, in order of increasing computing power required for each task. These tasks have been further applied to applications in networked systems in the Internet of Things (IoT), innovative healthcare, and robotics. Generally, ML models are computationally intensive and consume large memory. Owing to this critical drawback, the adoption of ML in embedded and mobile devices is relatively low as advanced models require more resources [5]. Different optimization techniques are adopted to fit computationally and memory intensive models to enable them to run within resource-limited hardware in embedded systems and mobile environments.

Common examples of machine learning algorithms include Naïve Bayes Classifier Algorithm, K-means clustering algorithm, and linear and logistic regression algorithms. Others include random forest and nearest-neighbors algorithms [6]. The complexity in both time and power required by these machine learning models may depend on a variety of factors, including implementation, data properties, and model parameters. Of these algorithms, the Naïve Bayes algorithm is the least computationally intensive one, while the regression and the random forest ones are similarly intensive algorithms, stated in terms of training complexity [7]. Support vector machines (SVMs), hidden Markov models (HMMs), and deep neural networks (DNNs), which are very computationally expensive, even more than the former ones, have been optimized to run in resource-constrained spaces. While ML is still a recent technology, research has already been done to take advantage of embedded ML, to develop efficient hardware structures and to create innovative and intelligent hardware architectures that can handle the requirements of the models even with their high performance.

ML on microcontrollers requires optimization because microcontrollers use less power, have less memory space, have real-time operating systems, and offer a more focused user interface that gives them an advantage over single-board computers (SBCs). SBCs, while very fast, are microcomputers with many peripherals that are not necessary for use in embedded machine learning, and they also require more computing power [8]. On the other hand, microcontrollers are very specialized in their use cases and provide just enough peripherals for embedded systems, therefore are more efficient and do not waste resources. However, SBCs are an excellent alternative to microcontrollers in the embedded world because they can be easily modified to host applications that require intensive computing and extended to have analog peripherals through advanced circuitry [8]. This project will focus on running ML on microcontrollers because of their advantages and how they can be integrated into many projects and applications easily and for the reasons mentioned above.

From classifiers to Bayesian networks and decision trees, several machine learning algorithms have been developed, each having its own specified set of outcomes and application areas [9]. These algorithms can be deployed to microcontrollers in embedded devices on the edge to perform their tasks. The ML algorithm of interest to this project is Neural Networks (NNs), which refers to systems of algorithms trained to realize underlying relationships in huge data sets, while doing so in a fashion that mimics how the human brain works [10]. NNs are mathematically elegant with encapsulated layers connected by links, which connect the input data to the output data in the model from end to end. In these intelligent systems, data propagation happens sequentially per layer. In every layer, each node consists of a function that collates all the information from all nodes in the preceding layer, determines its own output, and passes it out to the nodes of the next layer. The next layer then uses the data to determine its own output, which it

sends to the next, in such a way that the input for each hidden layer is the output of its preceding layer. The computations, usually mathematical operations, happen progressively until the last layer, where the output of the NN is determined. The next chapter will give a more detailed explanation of how NNs work. Consequently, NNs become computationally expensive quickly and require hardware that can handle such computations and keep up with speed.

Fixed point arithmetic has been used in machine learning for computations and to store numbers in memory. Fixed point arithmetic has been used for its advantages in memory reduction and latency, especially when values are represented in low precision in four bits or less [11]. It has also been used owing to its low power consumption and computation time, especially for models such as deep convolutional networks (DCNs), whose complex architecture has been increasing [12]. For some processes requiring compute-intensive synthesis of models in optimization schemes, such as during optimization in HMMs for speech synthesis, fixed-point representation introduces accuracy errors to the synthesis [5]. This has opened a window for floating-point arithmetic to be explored. Recently, ML has taken advantage of floating-point arithmetic, with some systems and processors built with floating-point computation capability and deviating from the usual fixed-point domain. The main advantage floating point has over fixed-point representation is its ability to represent both small and large real numbers in a reasonably small amount of memory, which can be particularly useful in representing a wider range of numbers in the same amount of memory [5]. This is ideal for embedded devices because they are resource-constrained with very little memory to store all their data.

Other advantages of floating-point computations include fitting more numbers implying more training samples and activations, storing more numbers implying more models in system caches, transmitting more numbers per second, and computing faster due to the extraction of more

parallelism in fixed-width registers [13]. In fixed-point arithmetic, however, fewer numbers are stored, fewer training samples can be extracted, and even fewer models are stored in the system cache, saving memory but compromising the effectiveness of the models. With floating-point representation comes the advantage of having an accurate representation of deep learning parameters that are usually non-linear; hence, no optimization or estimation of these is needed as in fixed-point representation [14]. Of course, these pros come with cons, including the limitation of ranges of numbers that can be represented and the introduction of quantization errors when full-precision numbers are stored. These disadvantages can be mitigated by using more exponent bits and using Stochastic rounding, respectively [14]. Most disadvantages of the floating-point domain can be mitigated by using more exponent bits which increases the range of numbers, and stochastic rounding and fine-tuning to address errors introduced in quantization [5]. Furthermore, though fixed-point arithmetic poses a power advantage over floating-point, it is not suitable for representing parameters in non-linear models and cannot carry as many numbers as floating-point representation can. Most disadvantages of the floating-point domain can be mitigated by using more exponent bits which increases the range of numbers, and stochastic rounding and fine-tuning to address errors introduced in quantization [5].

1.2 Objective

An implementation of embedded machine learning was done using an ATmega328P microcontroller interfaced with a Field Programmable Gate Array (FPGA), which acts as a coprocessor [14]. The speed of inferencing of the ML model was found to be twice that of inferencing on a microcontroller only. Also, in [15], an ML library was built to implement neural network algorithms in embedded system hardware, whose final output was VHDL code. This project extends these implementations already explored by deploying the ML model fully on a

microcontroller capable of floating-point computations to change the way the Machine Learning model actuates and increase accuracy. This project aims to determine the speed gain or loss when using such a microcontroller because of the change in operations while also using a neural network ML model. The NN implemented will categorize motion into any one of five classes, i.e., idle, clockwise, anticlockwise, left-right and up-down motions. The results of inferencing by the NN will be used to feed a motion detection application database, whose contents will be displayed on a web application. This embedded system can be adapted for spaces where the recognition and detection of motion types are essential, for example, in sports gyms and physiological training centers to track equipment in use or in alarm systems.

Chapter 2: Literature Review and Related work

2.1 Machine Learning (ML)

Machine Learning describes the several methods by which systems make informed data-driven decisions by learning from already available data and making inferences on new data. ML has three main branches: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the model learns already existing data (labeled data) and then makes predictions on unseen data. Most supervised learning techniques are classification methods. Examples of supervised learning techniques include logistic regression, k-nearest neighbors (kNNs), and artificial neural networks (ANNs) [9]. With unsupervised learning, ML models learn on their own from unlabeled data through the identification of similarities and patterns, and examples include clustering, k-means, and anomaly detection algorithms. Lastly, reinforcement learning is one through which models learn through trial and error, with example techniques such as genetic algorithms and estimated value functions. The training phase is the process through which ML models learn, and the trained model is used to make decisions on new data in the inference phase of the implementation.

2.2 Neural Networks (NN)

Neural networks fall under supervised machine learning techniques and are an example of classification methods. Neural networks (NN) can be trained to do complex event classification that works very well and efficiently in embedded machine learning. In NNs, interconnected neurons rely on each other, and each holds an activation which is just a number. Neural networks work in a way that mirrors the behavior of the human brain. Neurons are nodes that hold numbers for computations when data is passed through them to the next neuron [16]. A number of processes happen inside a node, which include computing weighted sums of inputs from previous layers,

adding bias terms and applying one or more activation functions which introduce non-linearity to the model.

Neural networks consist of many node layers, the first of which is an input layer, followed by hidden layers and an output layer. Computations in a neuron or node are done using data inputs from the preceding layer, weights, and a bias or threshold. A threshold or a bias is a value below which the particular neuron is inactive in a given inference cycle. The weight is used to determine the importance of any given variable in the neuron; hence, larger values contribute more to the output than lower ones [17]. The input data is then multiplied by the respective weight and passed to an activation function which determines the output. The threshold then comes into play to fire or activate the neuron if the output value exceeds it, allowing it to pass its output to the neurons in the next layer [17]. This direction of flow creates a feedforward network. A network in which each neuron is connected to each one in the next layer is called a fully connected network. The output of the final layer in a neural network is a number, which is the probability that the particular class is the one to which the input data belongs; hence the output class is determined by probability.

Neural networks are deterministic, which means that once a model is trained, it always produces the same output for the same input, and there is no randomness in the model as for probabilistic methods [18]. This is explained in Figure 2.1.1 below.

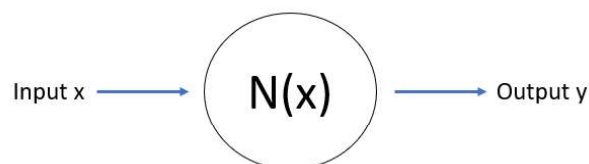


Figure 2.1: Inputs and outputs of a neural network

There is at least one hidden layer in deep neural networks, that is, at least three layers in total. Figure 2.2 below, adapted from [17], shows a fully connected deep neural network (DNN) with many hidden layers.

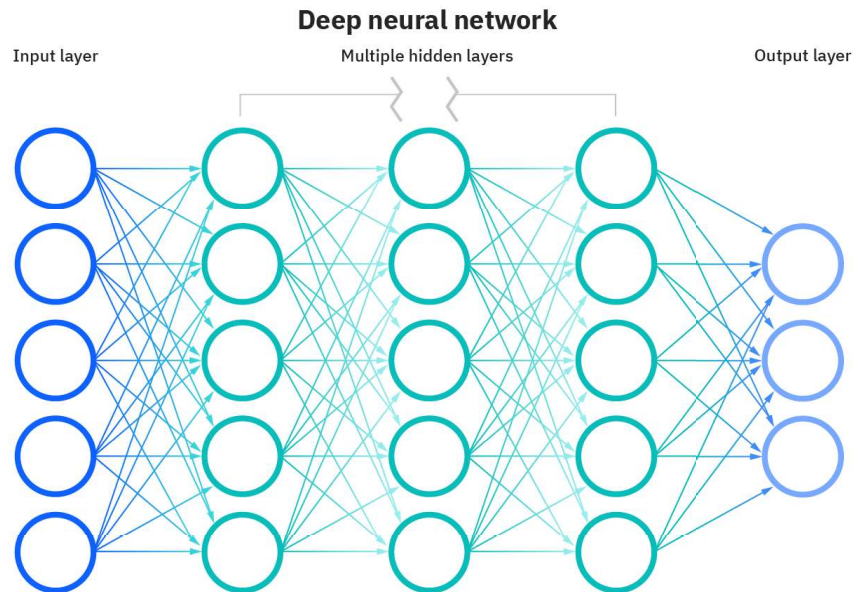


Figure 2.2: Deep neural network (DNN) [17]

Neural networks have been used in signal processing applications to get better accuracy (sometimes over 10x) in predicting the next sample of signals using sequential deep neural networks while using less memory space. They can also be applied in computer vision, natural language processing, and other applications that require even more intensive computations and better inferencing patterns.

Interfacing a microcontroller and an FPGA has already been demonstrated using Arduino and FPGA to determine how much faster inferencing will be if the FPGA acts as an accelerator [16]. In [6], the speed was found to be twice that of machine learning on microcontrollers only, which is very efficient and useful for embedded devices, especially on the edge, because of reduced delays in providing outputs to users. However, in [16], the FPGA accelerator was not used for

training purposes which did not allow real-time inferencing of input data. In addition, the board of choice for the project only allowed a limited number of DSP tiles which was 90; hence, more extensive networks could not be tested to determine how the speed changes when networks become wider. Also, the use of the FPGA implies more power losses introduced to the system, which, usually, engineers do not have control over power optimizations [19].

In [20], a library was built to help with code generation of advanced tasks for developers to help with deploying heavy algorithms on resource-constrained devices. The tool developed was able to create VHDL code for a problem described which is very useful, especially for embedded engineers. However, a Python library was developed for the problem defined. While this can be quite useful, most embedded engineers use the C or C-like programming languages to program their devices; hence, switching to Python can be a challenge. In addition, the library does not contain any network pruning to ensure memory efficiency of the library in the real world. The author also mentions that their use of fixed-point arithmetic is owing to the ease of computation it introduces, ignoring how the limited precision and range can affect the accuracy and performance of the output.

In this project, a high-end microcontroller capable of performing computations in floating-point arithmetic will be used to evaluate any performance differences. This will increase model accuracy and performance, as non-linear parameters can be stored in their raw form without optimization. Though having a drawback of quantization errors that can be introduced, techniques such as stochastic rounding or randomized rounding have been developed to counter them, making floating-point representation the next stage in ensuring the development of more accurate and efficient ML models to be deployed on embedded hardware.

2.3 Tiny Machine Learning (TinyML)

Tiny Machine Learning (TinyML), a machine learning technique that shrinks ML models to run on small, low-powered devices, adds the ability to leverage deep learning algorithms to train and adapt their size without sending data back to the cloud [5]. On the cloud, compute-intensive models are trained in high-end data centers using supercomputers equipped with the necessary build to sustain such complicated models, while with TinyML, mobile machine learning can be carried out with very low power consumption, usually in the 0.1W range [5]. Enabling ML models to run on low-powered hardware offers many advantages that enable applications to improve user experience and enhance applications. This provides a decrease in latency, which decreases the time for data analysis at edge devices. Compared to traditional ML models running on edge devices, TinyML saves computing power and uses less memory, with on-device applications in the 1mW and below power range, allowing edge devices to run unplugged for a long time, sometimes months or even years [21]. TinyML has inspired solutions in different sectors, such as powering smart homes, health, agriculture, avionics, and human-computer interactions.

2.4 Strategies for deploying Machine Learning in Embedded Systems

To take full advantage of the power of edge computing, ML models have to be deployed at the edge of the network to ensure inferencing at those points. Several strategies have been employed to ensure optimum delivery of ML models at the edge. In most cases, ML models are pre-trained on cloud servers where large data sets are initially sent to from sensors for training and validation processes. Due to the memory and power constraints of edge devices, ML models cannot be deployed directly to these devices [11]. This constraint is even more significant if cases of compute expensive models such as deep learning models or models requiring large data sets to maintain their features and accuracy are considered. Because of this, ML models are simplified

through processes such as quantization, of course, taking necessary caution to ensure no loss of accuracy. Quantization approximates a neural network that is dependent on floating-point numbers by one that uses numbers of low bit-width [11].

For ML models developed for Android and microcontrollers running with an Arduino-compatible software, the TensorFlow Lite library [22] is usually utilized to simplify TensorFlow models. TensorFlow Lite is an open-source software library that allows running TensorFlow models on mobile and embedded environments and devices. TensorFlow encapsulates many layers of its processes to ensure seamless use of TensorFlow models, even for beginners in IoT development [22]. TensorFlow example models span from object detection, image recognition, speech recognition, and text classification. These can be used and expanded in various applications to create powerful applications in different fields.

The Edge Impulse Platform is another resource used to deploy ML on embedded devices [23]. The platform helps create robust, quantized ML models through a few clicks and can perform inferencing online by collecting data on any provided development boards, most of which are predominantly ARM processors [23]. Though with encapsulation and optimization of processes, it allows the engineer to define their model on their own through a series of guided steps.

Research [24] shows that machine learning models run on microcontrollers face issues with memory, power, and speeds; hence, this practice is not advisable. However, there can be hardware optimizations such as energy efficiency enhancement, and software optimizations such as architectural and algorithmic enhancement methods can be used to enable ML on microcontrollers [24]. In addition, lightweight applications can be implemented on microcontrollers, for example, signal controllers that awaken a system once it detects activity [24]. One of the strategies used to deploy machine learning in embedded systems is Graphics Processing Units (GPUs), which can

perform high-speed operations in parallel and hence increase the model's performance [24]. They, however, require more power to run ML models, and given that embedded applications usually have to run in power-constrained devices, GPUs are impractical for such applications.

Microcontrollers paired with FPGAs (Field Programmable Gate Arrays) are also used as one of the best strategies for ML development on the edge. The FPGA acts as an accelerator that runs the ML models and delivers its output to the MCU, which can then drive full state applications that require the results of inferencing. FPGAs are easily programmed, do not consume as much power as GPUs, and are usually used in systems where minimal development time and high gains in hardware are required. In previous works [16], a microcontroller was successfully paired with an FPGA, as shown in Figure 2.3. The data from the sensors is fed into the microcontroller unit, which delivers it to the FPGA via General Purpose Input-Output (GPIO) pins [16]. The FPGA performs the necessary computations on its fast hardware and returns the results to the microcontroller which then makes predictions.

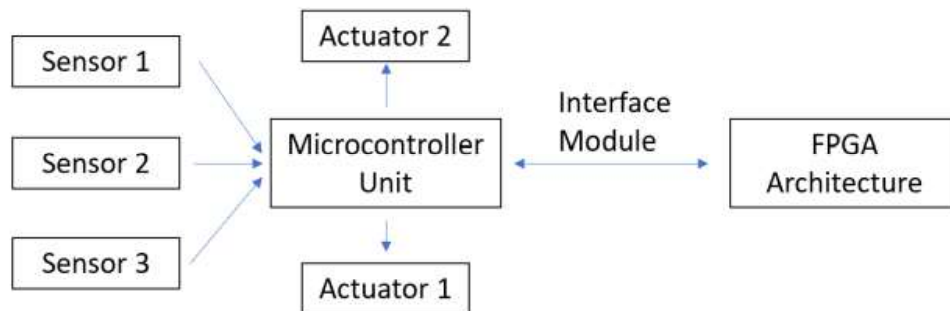


Figure 2.3: FPGA in an embedded system [16]

In some cases, two microcontrollers are used, one with enough resources to run a full-fledged ML model and another whose role is the same as that for a microcontroller paired with a hardware accelerator [18]. An example adapted from [18] is shown in Figure 2.4 for an audio classification neural network. The microphone sends data to the first microcontroller (Arm Cortex M4), which reads input data and performs the inferencing function. Once done, it signals the second microcontroller (ARM Cortex M7) of this event by either toggling a pin or any serial or I²C communication protocol. Though cheaper, this setup requires more power and more code for it to run seamlessly compared to one with only one microcontroller.

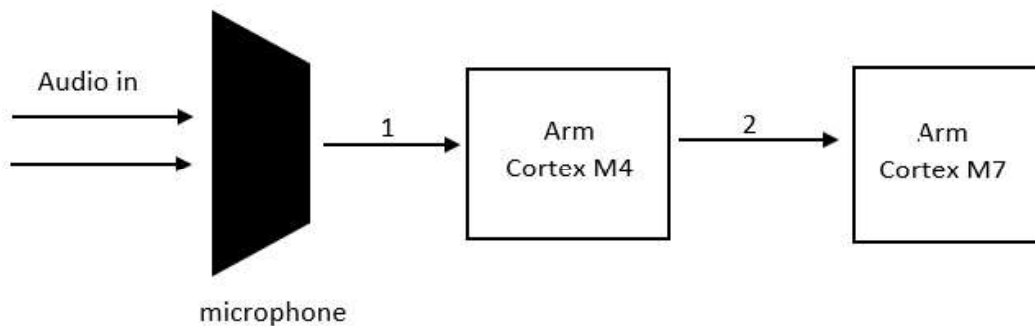


Figure 2.4: ML audio classification model with ARM M4 and M7

2.5 Project Scope

Much work has been done in embedded machine learning on which this work is based and modifies. Intelligent systems, including home/office systems, have been designed and built by leveraging IoT technology, edge computing, and embedded machine learning. However, for advanced tasks such as image recognition, there is a need for very high processing power and high computing capabilities to keep up with the tasks' requirements, which indicates a need for specialized, expensive hardware that is commercially not viable. The focus of most previous projects and work is on ensuring deep learning inferences on low-end microcontrollers such as the

ATMega328P, and sometimes this microcontroller is interfaced with an accelerator. However, in [25], a Jetson TX2 embedded deep learning platform was used to evaluate an approach to embedded ML that determines an appropriate DNN for a given input. This is by employing ML to build a cheap predictive model that selects a previously trained DNN and an optimization constraint. Though this approach significantly increased the speed of inferencing and improved accuracy, there were high memory overheads. In this work, a single pre-trained DNN will be used for inference to ensure the space and time efficiency of the model, which will then be applied to a motion detection system. In addition, the authors in [25] used Python to execute feature extraction, which increased the overhead in their application, and recommended, however, that a more efficient language be used to reduce the overhead. This project will use C and C++ to perform feature extraction, which is expected to reduce these overhead costs.

Also, the use of floating-point computations to alter the actuation of the machine learning model and using a more robust microcontroller such as the K64F microcontroller than the ATMega328P will be implemented in this work as well. Furthermore, a simple application to illustrate a practical application of the motion detection NN will be developed for use in training centers, etc.

Chapter 3: System Design and Implementation

3.1 Chapter Objectives

This chapter will provide clarification on the specific needs this project intends to address in the form of functional requirements, non-functional requirements, and constraints. This outline will then inform the high-level design of the system that will explore the project topic and the various technologies that will be utilized for the project's success. The use case of this project will be discussed in detail in this chapter.

The system proposes a workflow to enable Machine Learning on a microcontroller that can do floating-point computations to change the way the ML model actuates and to increase accuracy. Floating-point computation is key because of its ability to represent both large and small real numbers in a reasonable amount of memory storage. A NN machine learning model will be built and run on the chosen microcontroller to determine the speed gain or loss of inferencing in terms of inference time by comparing the results to those of inferencing for the same model in an optimized fixed-point version.

This system is targeted at embedded system designers, students, and professionals who wish to use ML on microcontrollers in various projects. The requirements stated below determine the choice of the microcontroller to use during this project.

3.2 Use Case

Lisa Tim, a machine learning researcher, has collected data and trained a machine learning model (DNN) to solve her problem and wants to deploy her application in an embedded device at the network edge to reduce her network traffic. Lisa requires high accuracy for her system, higher speed, and accurate representation since her deep learning parameters have non-linear properties.

As a result, she decides to use a microcontroller capable of storing numbers and performing computations in floating-point format.

3.3 User Requirements

3.3.1 Functional and non-functional Requirements

The system should:

- Detect at least three types of motion
- Fetch and receive data from the internet
- Use an ARM processor that supports floating-point computations
- Perform computations in floating-point
- Perform computations in fixed-point (for comparison, as a base for floating-point)
- Be cheap
- Connect to the internet
- Consume minimal power.
- Easy to use and integrate into projects.

3.4 Choice of microcontroller

Two microcontrollers were initially chosen for this project, the K64F, and the STM32F407VG. The specifications for the microcontrollers can be found in Table 3.1. Although the specifications given in the table above make the STM32 a better choice of microcontroller for this project, the MKL64F will be used since it is readily available. The MKL64F meets the project's requirements in frequency, power, cost, and floating-point capabilities. Although the STM32 would outperform the MKL64F given the specifications above, the results will still be valid and indicative. In addition, the MKL64F can be connected to the internet via Ethernet which

makes it a smart choice for the development of an application that requires data from the microcontroller. The board of choice with this microcontroller is shown in Figure 3.1.

Table 3.1: Specifications for the K64F and the STM32F407VG microcontrollers

Specification	MKL64F	STM32F407VG
Operating Frequency	120 MHz	168 MHz
Power consumption	Low power	12.7 mA
Cost	\$6.63	\$15
Computations in floating-point	Yes	Yes
Ease of use	Easy	Easy



Figure 3.1: FRDM-K64F Development Board with MK64FN1M0VLL12 microcontroller

3.5 Workflow Overview

The Edge Impulse Platform will be used to create, train and test the Machine Learning Model to use with this project. The model built will be a simple deep learning motion detection system that determines the type of motion an accelerometer is exhibiting by classification among five different motion classes. The classes are:

- Idle: the accelerometer is considered idle when it is static, i.e., not moving.
- Clockwise: in a clockwise motion, the accelerometer moves in a circular motion forward.
- Anticlockwise: the accelerometer moves in an anticlockwise motion, which is reverse of the clockwise motion, hence backward.
- Left-right: the accelerometer moves in a left to right or right to left pattern.
- Up-down: the accelerometer moves in an up and down or vice versa pattern.

Figure 3.2 below shows the different configurations of a device or embedded system E with arrows showing the motion directions that result in the classes described above. These motions may represent, for example, types of exercise or movement of equipment.

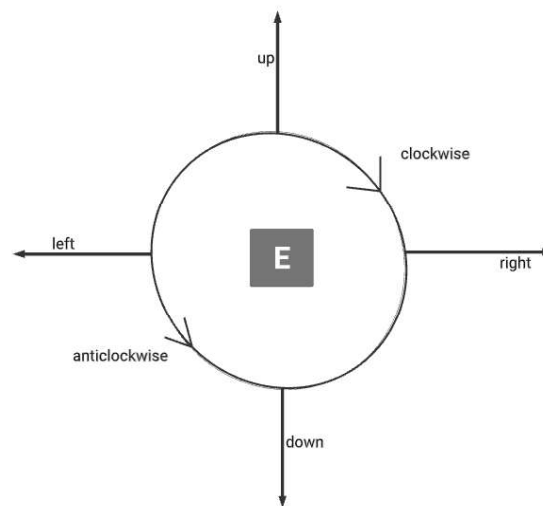


Figure 3.2: Different possible configurations of an embedded system E

After creating, evaluating, and modifying the model, the model is deployed on the K64F microcontroller, and analysis is done. Figure 3.3 below shows the workflow overview, with the work to be done on the Edge Impulse platform and on the microcontroller in an embedded device.

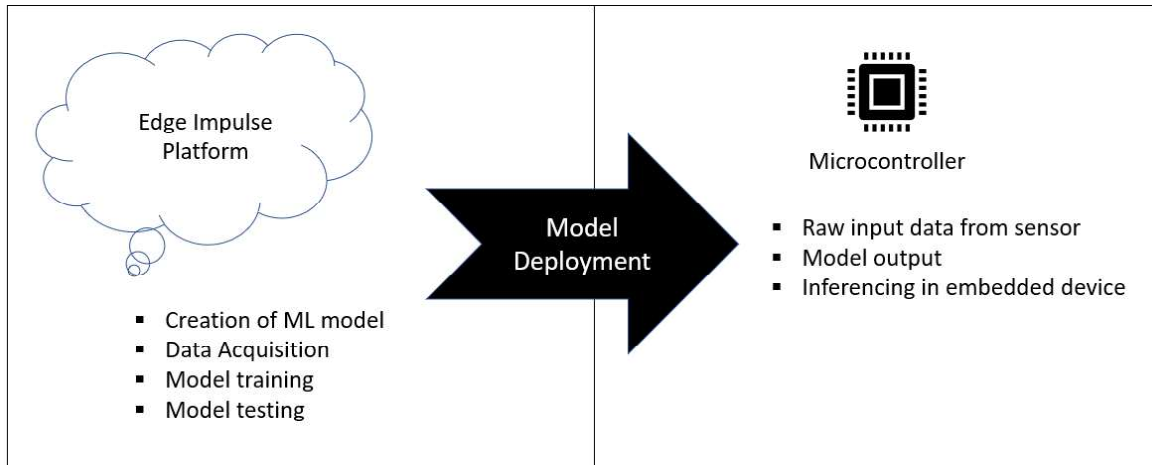


Figure 3.3: Workflow overview

3.6 Implementation Steps in detail

The Edge Impulse (EI) Platform was built to enable the creation and deployment of machine learning models on embedded systems through any microcontroller that can understand the C++ programming language. Several microcontrollers and development boards are supported on the platform, such as the Arduino Nano BLE, Raspberry Pi 4, NVIDIA Jeston Nano, and the Raspberry Pi RP2040 among others. This platform was chosen for this project because of its simplicity and proper documentation for developing the machine learning model on the cloud. Such a platform will speed up the development process of the ML model, thereby having users spend more time on the development of the embedded system overall.

3.7 Creating a NN machine learning model on Edge Impulse

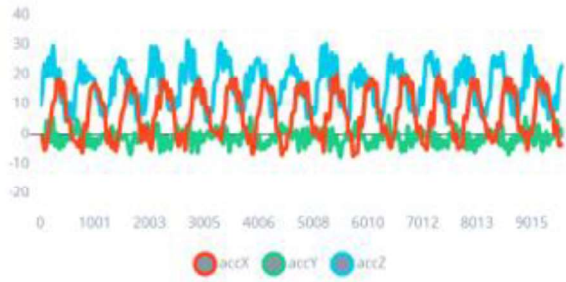
3.7.1 Data Acquisition

The data is collected on the EI platform by connecting a suitable device, preferably the device on which the machine learning model is expected to run. For instance, it is advisable to collect data using the Arduino Nano BLE board if it is the actual board that will be used to deploy

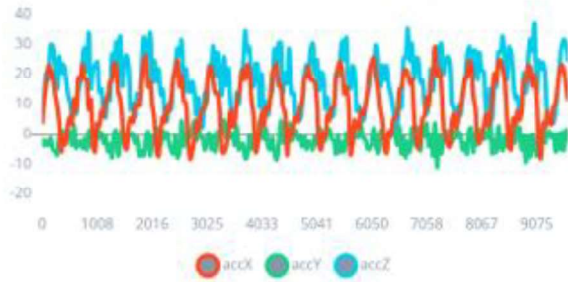
the ML model. This ensures accuracy, reduces anomalies in the data, and helps keep account of inconsistencies across different sensor models on different devices [18]. The device or development board is connected to the EI platform via the internet, or through the Edge Impulse CLI downloaded and installed on the user's machine. In this project, sensor data was collected through a mobile phone because of the ease of access to the platform via the internet, requiring no installations; hence, it was the quickest way to get started. To collect data using a mobile phone, the phone is connected to the EI platform using a QR code provided on the "devices" page of the platform.

Due to requiring multiple data sets to create a robust machine learning model, twenty-five samples, each ten seconds long, were collected for each class of motion described above. Having an equal number of samples per data set is essential in ensuring that the model does not lean towards a particular data set when performing live classification with unseen data; hence it does not become a naïve classifier [18]. The classes are then the labels of the data. Hence, the model here is a supervised machine learning model and belongs to the classification group of the same type of machine learning model. Samples of data representing each class are shown in Figure 3.4, extracted randomly from the data collected on the EI platform. The image captions are self-explanatory.

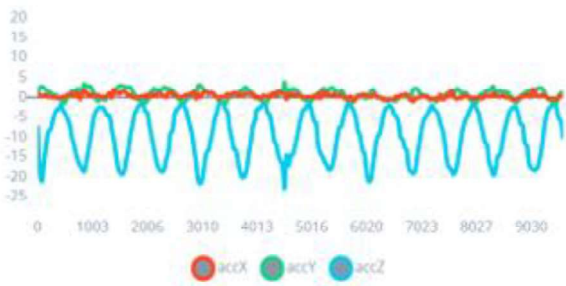
anticlockwise.2v7tpfes



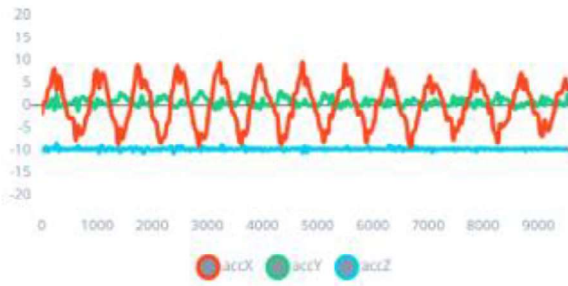
clockwise.2v7tec1d



up-down.2ubmoa97



left-right.2ubmle5j



idle.2ubmiod0



Figure 3.4: Data samples for motion classes from Edge Impulse platform

Following data collection, the data is divided into Training Set and Test/Validation Set, as shown in Figure 3.5. Initially, all data collected exists in the training set. However, there is a need

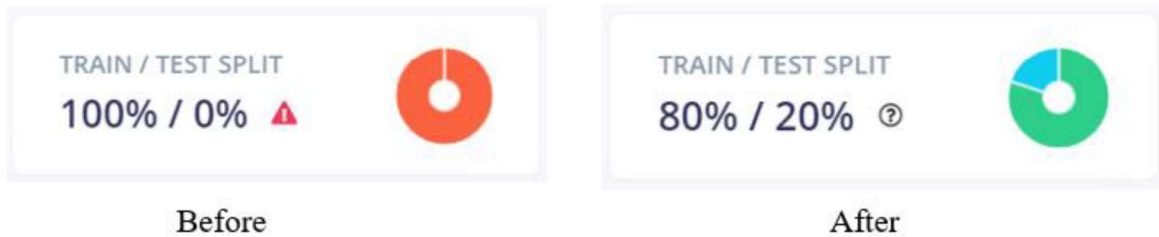


Figure 3.5: Train and Test splits before and after balancing data sets

to split the data to enable model testing on unseen data and make way for parameter modification if required. Also, splitting is essential to measure the model's accuracy on unseen data, ensure effective mapping of inputs and outputs, and prevent model overfitting. Overfitting occurs when the ML model accurately predicts training data but fails to classify unseen data, a trait commonly exhibited by highly complex models. The training set is the actual sample of the data used to fit the model from which the model learns, while the Validation set or development set is the sample used to provide an unbiased evaluation of the model fit on the training set while also tuning the model's hyperparameters [25]. However, the model does not learn from the validation set data. The training and validation sets are implicitly defined in the 80% of data belonging to the Training set, and the remaining 20% belongs to the Test dataset, which evaluates the final model fit of the training dataset once model training is complete. Since twenty-five data samples were collected for each motion class, this split entails that twenty samples per class belong in the Training Set and the remaining five in the Test Set.

3.7.2 Impulse Design: Developing the Neural Network

Feature extraction is also done on the EI platform by creating an impulse by signal processing on raw data. The different blocks contained within this stage are explained below:

- a. Spectral Analysis - this processing block is ideal for repetitive motion, usually from accelerometers, and is responsible for extracting a signal's frequency and power characteristics over time. Figure 3.6(a) shows this block and its inputs.
- b. Classification (Keras) - this is a learning block that learns the patterns from data and applies them to new data. This learning block is ideal for motion or audio recognition. Figure 3.6(b) shows the Keras classification block.
- c. Anomaly Detection - this is also a learning block that helps find outliers in new data, hence determining whether something went wrong during classification. It complements classifiers and is ideal for the recognition of unknown states. Figure 3.6(c) shows the K-means anomaly detection block.
- d. Output features represent the expected outputs from the machine learning model, as shown in Figure 3.6(d).

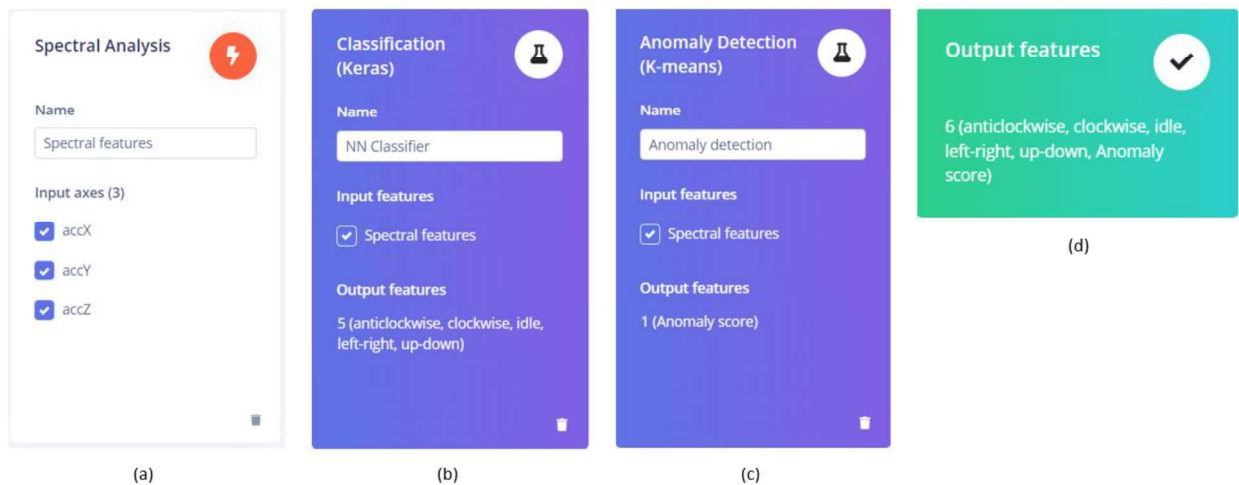


Figure 3.6: Impulse design on EI platform

After impulse design, the data features will be seen in the Feature Explorer, as shown in Figure 3.7, with the left key indicating the distinctive features for the classes. The RMS values of the accelerometer's x, y, and z values are used to plot the points to express the overall changes in the values and for uniformity of error.

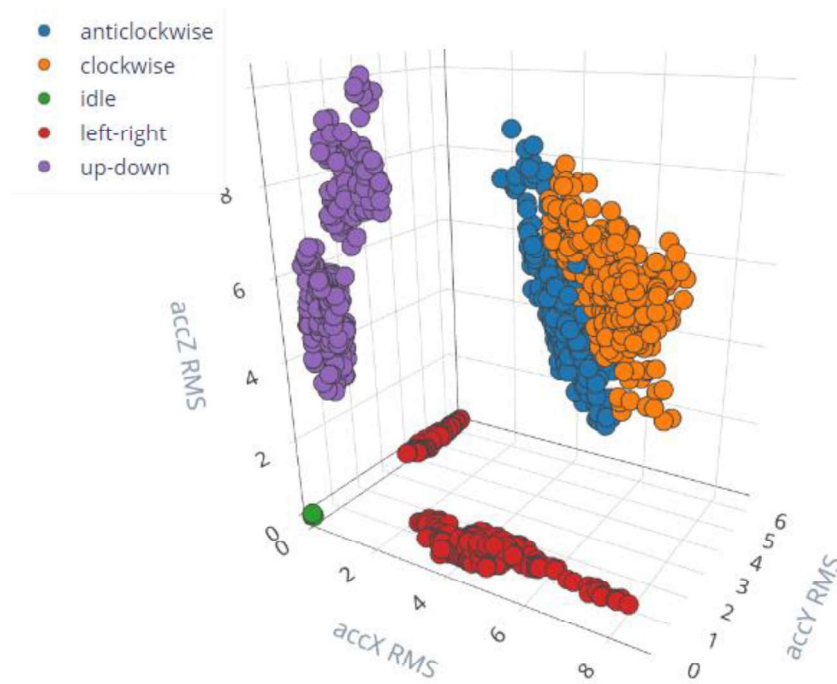


Figure 3.7: Feature explorer for ML model classes

In a neural network architecture, there are input layers, hidden layers, and output layers in the model, as shown in Figure 2.2 in Chapter 2. The number of input nodes corresponds to the number of features extracted from the input data during feature extraction. Depending on the model created for each application, the number of hidden layers is determined.

In this project, five layers were chosen, which maximized the accuracy of the machine learning model. This architecture is broken down in Figure 3.8, with twenty-five, eighteen, and eleven neurons or nodes in the three hidden layers, respectively. The number of hidden layers and

the number of neurons in each hidden layer can be changed, requiring the model to be retrained. With this, there is a need to check the model performance to ensure that it matches the requirements or specifications of the embedded system or the application it has to work in.

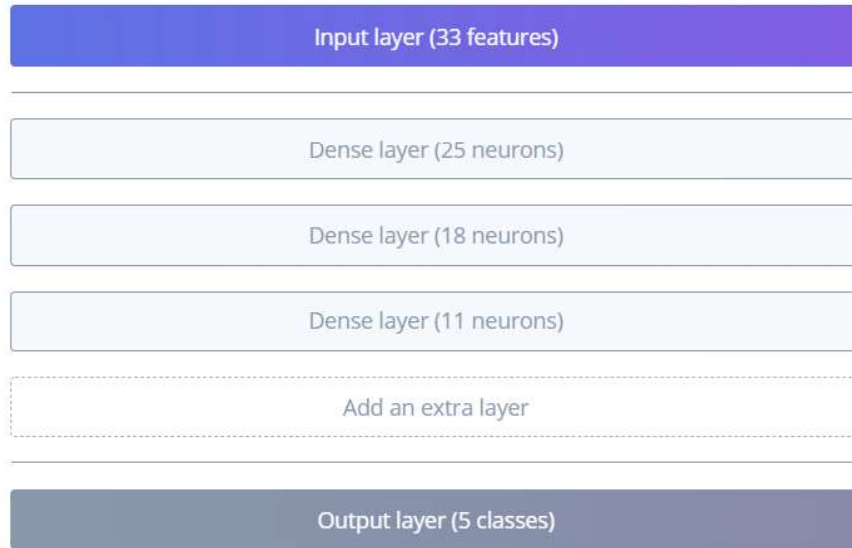


Figure 3.8: Neural network architecture

For this project, the architecture in Figure 3.8 was initially chosen through trial and error with two layers, and one more layer was added while training. The model was retrained with different combinations of nodes per layer with three hidden layers until this architecture was reached where the accuracy was at its maximum.

3.7.3 Training

With the chosen architecture of the neural network, the model is trained. The EI platform trains the model automatically and generates the confusion matrix of the model. A confusion matrix is an N -by- M matrix used to assess the performance of a machine learning model by testing the model against data in its validation set [27]. With this, the accuracy of the model is determined and its loss. The accuracy is determined in the confusion matrix by the equation below.

$$Accuracy = \frac{Correct\ predictions}{Total\ predictions}$$

When training, the EI platform trains and creates two models for the data, an optimized integer version and an unoptimized floating point one. The accuracy of the quantized integer (fixed-point) version of the model is slightly lower than the accuracy of the unoptimized float model, as shown in Table 3.2. This is due to the rounding-off involved in quantization to help ensure minimal accuracy loss. These losses are recognizable in cases where model accuracy is of paramount importance, for example, for tests for a disease. The differences in the accuracies of the integer version and the float version of this project, as shown in Table 3.2, are also because Edge Impulse highly optimizes its processes to ensure the highest performance of ML models.

Table 3.2: Training performances of int8 and float32 model

	Quantized/Optimized int8 version	Unoptimized float32 version
Accuracy	99.2%	99.5%
Loss	0.02	0.02

	ANTICLOCKWISE	CLOCKWISE	IDLE	LEFT-RIGHT	UP-DOWN
ANTICLOCKWISE	99.5%	0.5%	0%	0%	0%
CLOCKWISE	2.1%	97.9%	0%	0%	0%
IDLE	0%	0%	100%	0%	0%
LEFT-RIGHT	0%	0%	0%	100%	0%
UP-DOWN	0%	0%	0%	0%	100%
F1 SCORE	0.99	0.99	1.00	1.00	1.00

(a)

	ANTICLOCKWISE	CLOCKWISE	IDLE	LEFT-RIGHT	UP-DOWN
ANTICLOCKWISE	99.5%	0.5%	0%	0%	0%
CLOCKWISE	3.7%	96.3%	0%	0%	0%
IDLE	0%	0%	100%	0%	0%
LEFT-RIGHT	0%	0%	0%	100%	0%
UP-DOWN	0%	0%	0%	0%	100%
F1 SCORE	0.98	0.98	1.00	1.00	1.00

(b)

Figure 3.9: Confusion matrices for (a) float32 and (b) int8 model

The confusion matrix in Figure 3.9 shows the model prediction results against the actual classification that was expected in a confusion matrix. The values show the percentages of total predictions that was done for each class. The information in Figure 3.9 shows that classification for the “idle,” “left-right,” and “up-down” classes were determined without any errors in both models, hence 100% accuracy. However, the “anticlockwise” and “clockwise” classes were predicted with some errors, hence 99.5% and 97.9% accuracies for the unoptimized float32 model, and the corresponding percentages apply for the optimized int8 model.

3.7.4 Anomaly Detection

The anomalies or outliers in the data are determined through anomaly detection, using the same axes as those in the feature explorer (RMS axes values). Anomaly scores are determined, and an anomaly explorer is established. This process helps determine if something went wrong during classification and is an excellent technique for fraud detection. Figure 3.10 below shows the anomaly explorer for the ML model created.

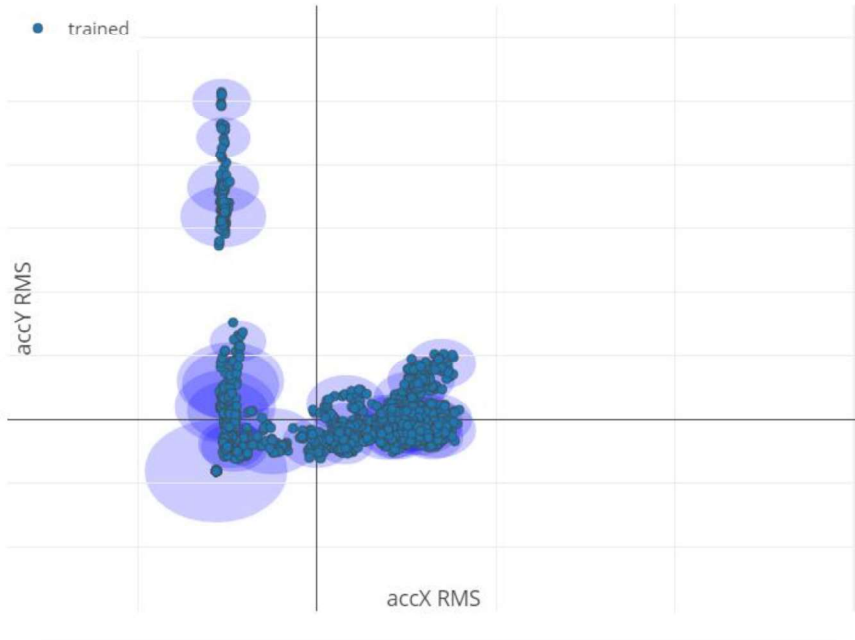


Figure 3.10: Anomaly explorer for ML model

3.7.5 Model Testing

The model is tested using the Test data initially separated when the train/test split was done or new data recorded when live classifying. Figure 3.11 is an example testing the model using a new sample belonging to the left-right class.

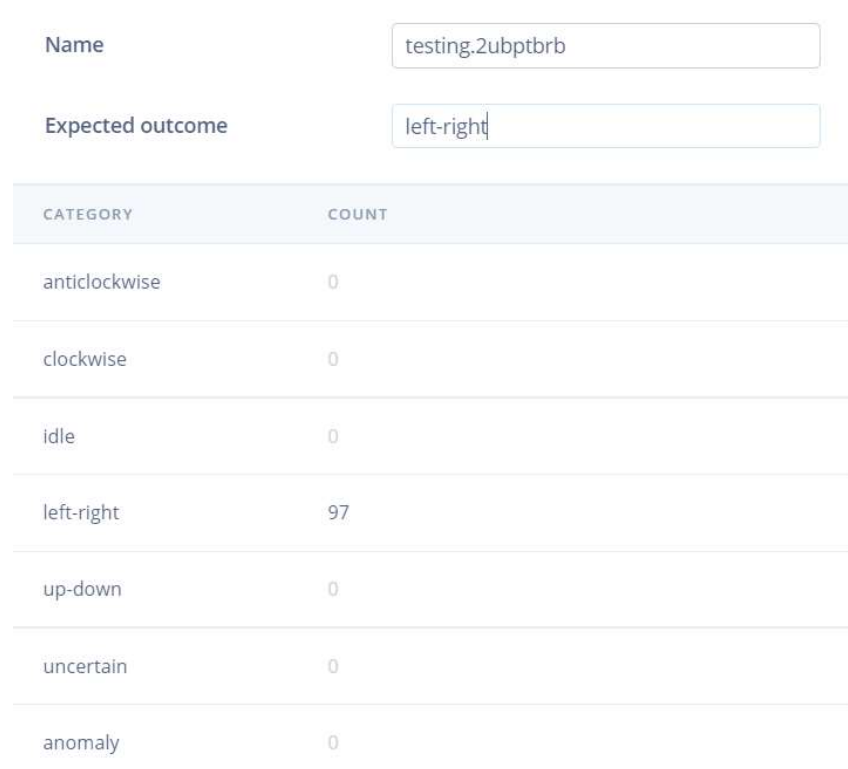


Figure 3.11: Testing sample results with new data for left-right class

In Figure 3.11, the count for the left-right class is the highest; hence, the sample belongs to this class. Since the expected outcome and the actual outcome are both the same, the model accurately predicted that the sample belongs to its actual class.

Another test was also carried out on a sample belonging to the anticlockwise class, and the count for the anticlockwise class was the highest, as shown in Figure 3.12 below, although there were counts greater than zero in the other classes. With this, it can be seen that the model accurately predicted the outcome.

Name	<input type="text" value="anticlockwise.2v7revtl"/>
Expected outcome	<input type="text" value="anticlockwise"/>

CATEGORY	COUNT
anticlockwise	95
clockwise	1
idle	0
left-right	0
up-down	0
uncertain	1

Figure 3.12: Testing result for an anticlockwise class sample from test set data

3.7.6 Model Deployment

The ML model can be deployed on several microcontrollers listed on the Edge Impulse platform. However, since the development board for this project is not directly supported by the platform, the model was downloaded from the Edge Impulse platform as a C++ library that can be integrated into the workspace of the IDE used to program the microcontroller. The library contains three folders with interdependent files for inferencing. For some platforms not supported by the Edge Impulse Platform, some functions had to be implemented. These functions were implemented and added successfully to the library. Edge Impulse also provides other ways to deploy the machine learning model on several other devices aside microcontrollers and development boards.

3.8 Building a web application to interface with FRDM-K64F

An application to interface with the FRDM-K64F microcontroller was built. It can be used in a gym or fitness center, for the center managers to know which systems or equipment is in use and how it is being used. This helps the managers assign equipment to people as they arrive at the gym, given that they already know which equipment is in use and which one is idle.

3.8.1 Application Technology Stack

The code for the application can be accessed on GitHub¹. This application was built using the following programming languages and frameworks:

- Frontend – HTML, CSS and JavaScript
- Backend – Node JS and MySQL

3.8.2 How it works

The workflow of the web application and the embedded system is described in Figure 3.13, with the arrows indicating the direction of data flow from each component of the entire system.

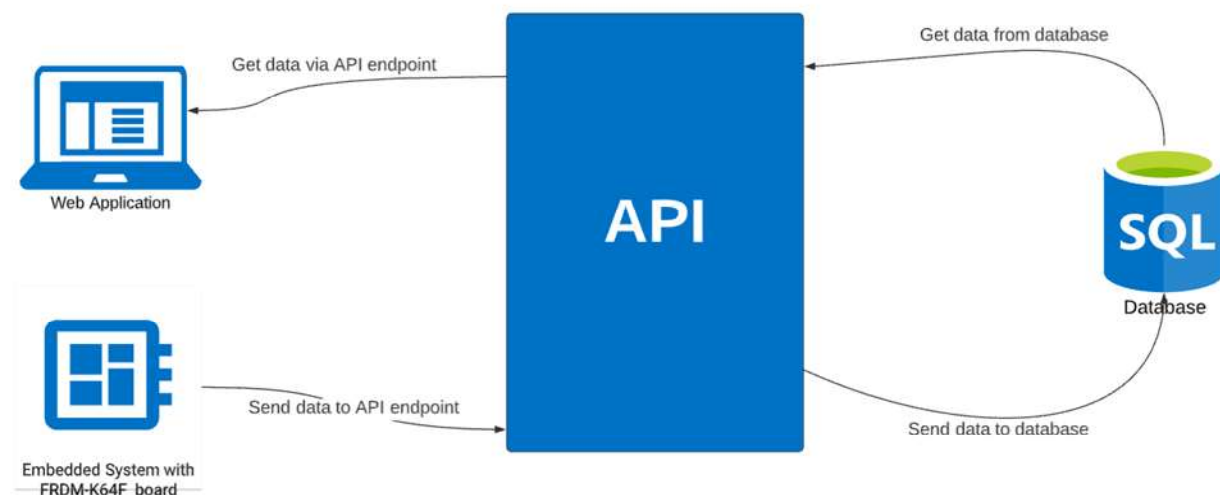


Figure 3.13: Web application data flow diagram

¹ <https://github.com/lorrainemakuyana2022/capstone-application>

The information from the sensors in the embedded system is sent to the database by the microcontroller by sending the results of the ML model classification to a database via an API endpoint built in the application backend. The MCU classifies the motion periodically to update the database on the motion state of the device. The API then sends data and records this data in the database. The frontend application queries the database every three seconds to refresh its data and update the status of the equipment on the dashboard, hence displaying it to the fitness center managers, who will assign people only to unoccupied equipment. This system can also be applied to a monitoring system for the physical activity of a physiotherapy patient remotely or collating the types of exercise done by a person remotely as well.

Figure 3.14 shows screenshots of the web application interface. The one on the left shows the current status as left-right, which shows as green, and the same applies for the other five classes of motion. However, as shown on the right, the current status shows in red if it is uncertain indicating an anomaly.

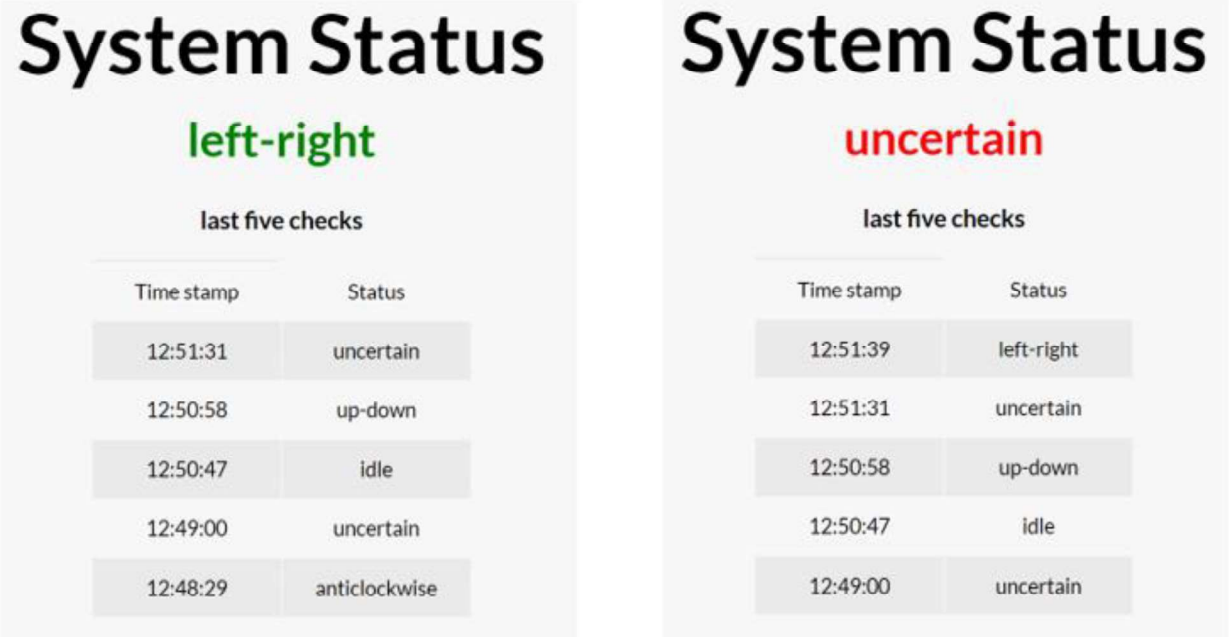


Figure 3.14: Screenshots of web application

Chapter 4: Results and Analysis

4.1 Chapter Objectives

This chapter analyses the inference time of the ML model on both the Arduino Nano BLE board and the FRDM-K64F board. The parameter to determine the speed gain or loss of the system are classification time or inference time. Inference time is the time taken from when the accelerometer sensor on the board gets the data and data is put in the format it should be in to the time when the ML model completes inferencing and classification and displays the results, or the time just before it processes sending the data to the API endpoint for storage in the database.

4.2 Running the model

The programs for this project were uploaded on GitHub for ease of access and can be found in a GitHub repository². The steps outlined below represent the approach taken to deploy the machine learning model on the Arduino Nano 33 BLE, the FRDM-K64F, and on the desktop.

4.2.1 Deploying to Arduino Nano 33 BLE

To test the model, its integer version was first deployed on the Arduino Nano 33 BLE board. The Edge Impulse platform fully supports this board. Inferencing was done using the Nano 33 BLE compatible zip library generated from the Edge Impulse platform. The motion classes clockwise and anticlockwise were classified incorrectly and it seemed the model could not accurately distinguish among the two. The remaining classes were classified correctly without errors. Figure 4.1 shows the classification of the anticlockwise class on the left and on the right, a prediction of the idle class correctly.

² <https://github.com/lorrainemakuyana2022/capstone-ML-models>

```

Prediction: anticlockwise
Time taken is 20 ms

Prediction: anticlockwise
Time taken is 22 ms

Prediction: anticlockwise
Time taken is 22 ms

Prediction: anticlockwise
Time taken is 20 ms

Prediction: anticlockwise
Time taken is 20 ms

Prediction: idle
Time taken is 20 ms

Prediction: idle
Time taken is 21 ms

Prediction: idle
Time taken is 23 ms

Prediction: idle
Time taken is 22 ms

Prediction: idle
Time taken is 20 ms

```

Figure 4.1: Predictions on an Arduino Nano BLE

The accuracy was not as expected because the data collection on which the algorithm is based was collected using a mobile phone and then run on the board. In developing models for commercial use, or in cases where an extremely high degree of accuracy is required, the data collection and the actual inferencing should be carried out on the same device, or at least the same type of device on which it will run live. In this project, data acquisition was not carried using the Arduino Nano BLE because of installation issues with the Edge Impulse CLI. The necessary specifications of the Arduino Nano BLE are shown in Table 4.1.

Table 4.1: Arduino Nano specifications

Specification	Value
Microcontroller	nRF52840
Clock speed	64 MHz

The Edge Impulse Platform creates models and estimates for a device that runs on an ARM Cortex M4 with a clock speed of 80 MHz. However, the Nano BLE runs at a clock speed of 64

MHz; hence the speed of inferencing is expected to be a bit slower as the clock speed is lower. After adding the .zip library to the Arduino IDE, the program in Appendix A was run on the board. For different inferences using data from different output classes, the inference time fluctuated between 22ms and 26ms; hence, the average inference time can be calculated as follows:

$$\text{Inference time} = \frac{22 + 23 + 24 + 25 + 26}{5} \text{ ms}$$

$$\text{Inference time} = 24 \text{ ms}$$

The inference time for the Arduino Nano BLE above is tolerable for an embedded system that detects what type of exercise a person is doing, since it can give accurate results in 0.024 seconds.

4.2.2 Deploying to the FRDM-K64F development board

The float32 version of the machine learning model was tested and run on the FRDM-K64F development board to now test the performance of the floating-point version of the model. To do this, a C++ library of the model was downloaded from the Edge Impulse Platform and added to the MCU Expresso IDE workspace. The specifications of the MK64F microcontroller are shown in Table 4.2.

Table 4.2: MK64F Specifications

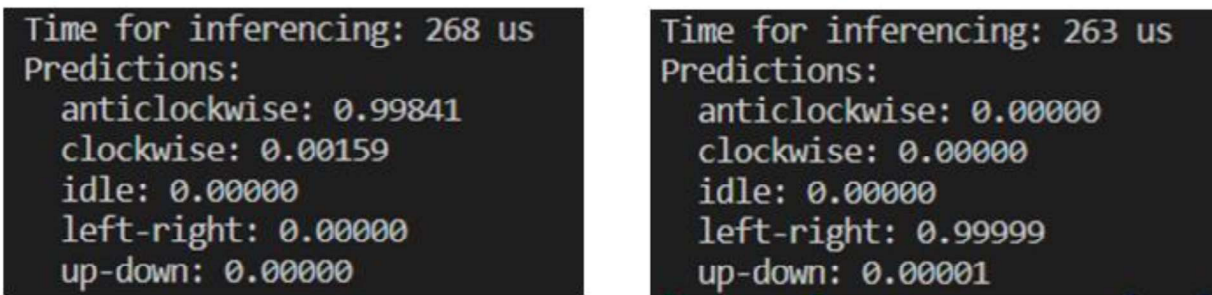
Specification	Value
Microcontroller	MK64FN1M0VLL12
Clock speed	120 MHz

Some challenges faced with deploying the neural network on the MK64F included compilation issues of having too many characters in the compile path filenames passed to the linker

(>32k characters allowed), which is a limitation of the MCU Expresso IDE. This problem persisted even when folder paths were shortened. The code for the neural network could be run on a desktop PC.

4.2.3 Deploying on Desktop PC

The machine learning model was also run on a desktop using the downloaded C++ library. The library content, including the three interdependent folders, was run with the implemented functions that required implementation. Raw features were taken from the data samples initially collected and pasted at the appropriate location in the model's main file. Figure 4.2 shows the classification of a test sample from the anticlockwise (left) and the left-right (right) classes, with the time for inferencing for each class.



```
Time for inferencing: 268 us
Predictions:
anticlockwise: 0.99841
clockwise: 0.00159
idle: 0.00000
left-right: 0.00000
up-down: 0.00000

Time for inferencing: 263 us
Predictions:
anticlockwise: 0.00000
clockwise: 0.00000
idle: 0.00000
left-right: 0.99999
up-down: 0.00001
```

Figure 4.2: Classification of anticlockwise (left) and left-right (right) classes on a Desktop PC

Inferencing on the desktop was much faster than inferencing on the Arduino Nano BLE. A sample from each output class was inferenced, and the time was noted to enable the calculation of an average. The initial run on the desktop took about 34ms to complete due to overheads. After the first run, the inferencing time reduced drastically because of optimizations by the operating system. The first run is not taken into consideration in this calculation as it can be seen as an outlier. Inferencing on the desktop PC was calculated as follows:

$$\textit{Inference time} = \frac{269 + 268 + 264 + 265 + 293}{5} \mu\text{s}$$

$$\textit{Inference time} = 271.8 \mu\text{s}$$

Chapter 5: Conclusion

5.1 Chapter Objectives

This paper explored running a complex deep learning algorithm on a microcontroller that can perform floating-point computations. Data for the neural network created was collected using a mobile phone and the Edge Impulse platform. The neural network was able to detect and learn different types of motion, and the model was built on the Edge Impulse platform as well. Tests were carried out on an Arduino Nano BLE, the FRDM-K64F, and a desktop PC. The model was successfully run on the Arduino Nano BLE and a desktop PC. It was found that inferencing on the Arduino Nano BLE took 24ms, and on the desktop, it took 271.8 μ s, which was much faster than the Arduino Nano BLE inferencing. The motion classes idle, left-right, up-down were classified accurately with no error. While the model was able to classify anticlockwise and clockwise, sometimes, these classes were mixed up owing to the motions being the same with only a change in direction.

This chapter highlights the challenges and limitations faced in developing the ML algorithm and testing the neural network on both platforms. Some areas of development to further this research will also be proposed in this chapter for future researchers to explore.

5.2 Limitations

Some challenges were met while developing this project, and some influenced major project decisions. Listed below are some of the challenges that were faced when this project was being undertaken:

- The Edge Impulse Platform used to develop the neural network for this project optimizes the creation of the neural network, making it challenging to know the nature of the

implementation of the neural network, hence, the difference between the fixed and floating-point representations could not be determined.

- The Edge Impulse platform targets particular hardware such as the Arduino Nano BLE and the STM32 Cube AI. Software and support for other hardware are not provided on the platform, making it challenging to set up for unsupported hardware correctly.

5.3 Future Work

While a highly performant machine learning algorithm was created and deployed for this project to test the speed of inferencing, there are ways to advance the project. This project creates a good baseline for future projects in embedded machine learning with microcontrollers capable of floating-point computations. Recommendations for future work include:

- This work depended on building the neural network on the cloud on the Edge Impulse platform. The neural network development can be taken offline to allow multiple iterations of the model and reduce challenges with installing software to enable seamless inferencing.
- Other Edge Impulse supported microcontrollers can also be used, for example, the STM32 described in Chapter 3, to explore how the algorithm behaves on microcontrollers with different specifications.
- Although continuous learning was done with the Arduino Nano BLE, it would be very useful to carry out real time inferencing on the FRDM-K64F with floating point capability.
- A hardware accelerator such as an FPGA can be designed and deployed in the embedded system to explore the speed of inferencing if the microcontroller has floating point capability.
- To reduce dependency on the Edge Impulse platform, TensorFlow Lite can be explored directly.

- The integer version of the machine learning model can also be tested on the K64F since this was not done in this project due to linker errors.

References

- [1] M. O'Connor, H. Norman and T. Aurora, *Embedded Machine Learning Design For Dummies®*, *Arm Special Edition*. New Jersey: John Wiley & Sons, 2019.
- [2] Business Insider, "Artificial intelligence is changing these 9 industries", *businessinsider.com*, 2017. [Online]. Available: <https://www.businessinsider.com/sc/artificial-intelligence-companies?r=US&IR=T>. [Accessed: 15- Apr- 2022].
- [3] M. Magno, *ETH Zürich at Science & Innovation Day 2019*. 2020. Retrieved 6 October 2021 from <https://www.youtube.com/watch?v=6xKXyj3uGMs>
- [4] A. M. Ghosh and K. Grolinger, "Edge-Cloud Computing for Internet of Things Data Analytics: Embedding Intelligence in the Edge with Deep Learning," in *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 2191-2200, March 2021, doi: 10.1109/TII.2020.3008711.
- [5] T. Ajani, A. Imoize and A. Atayero, "An Overview of Machine Learning within Embedded and Mobile Devices—Optimizations and Applications", *Sensors*, vol. 21, no. 13, p. 4412, 2021. Available: 10.3390/s21134412.
- [6] K. Wakefield, "A guide to the types of machine learning algorithms", *Sas.com*, 2022. [Online]. Available: https://www.sas.com/en_gb/insights/articles/analytics/machine-learning-algorithms.html#:~:text=There%20are%20four%20types%20of,-supervised%2C%20unsupervised%20and%20reinforcement. [Accessed: 15- Apr- 2022].
- [7] The Kernel Trip, "Computational complexity of machine learning algorithms", *The Kernel Trip*, 2018. [Online]. Available: <https://www.thekerneltrip.com/machine/learning/computational-complexity-learning-algorithms/>. [Accessed: 15- Apr- 2022].
- [8] Seed Studio, "All about CPUs: Microprocessor, Microcontroller and Single Board Computer - Latest Open Tech From Seeed", *Latest Open Tech From Seeed*, 2021. [Online]. Available: <https://www.seeedstudio.com/blog/2020/10/27/all-about-cpus-microprocessor-microcontroller-and-single-board-computer/>. [Accessed: 15- Apr- 2022].
- [9] T. Ayodele, "Types of Machine Learning Algorithms", in *New Advances in Machine Learning*, 2021, pp. 19-48.
- [10] J. Zou, Y. Han and S. So, "Overview of Artificial Neural Networks", *Methods in Molecular Biology™*, vol. 19, no. 1, pp. 14-22, 2008. Available: 10.1007/978-1-60327-101-1_2 [Accessed 9 December 2021].

- [11] A. Gholami, Z. Yao, S. Kim, Z. Dong, M. Mahoney and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference", 2021. Available: <https://arxiv.org/pdf/2103.13630.pdf>. [Accessed 28 February 2022].
- [12] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16). JMLR.org, 2849–2858.
- [13] N. Wang, C. Chen and K. Gopalakrishnan, "Ultra-Low-Precision Training of Deep Neural Networks", *IBM Research Blog*, 2019. [Online]. Available: <https://www.ibm.com/blogs/research/2019/05/ultra-low-precision-training/>. [Accessed: 09- Dec- 2021].
- [14] J. Foerster, "Nonlinear Computation in Deep Linear Networks", *OpenAI*, 2017. [Online]. Available: <https://openai.com/blog/nonlinear-computation-in-linear-networks/>. [Accessed: 18- Apr- 2022].
- [15] G. Ho, "Floating-Point Formats and Deep Learning", *Eigenfoo*, 2020. [Online]. Available: <https://www.eigenfoo.xyz/floating-point-deep-learning/>. [Accessed: 09- Dec- 2021].
- [16] O. Amenreynolds, "Using Commercial FPGAs as External Accelerators for Artificial Neural Networks in Embedded Applications," Capstone Document, 2020.
- [17] IBM Cloud Education, "What are Neural Networks?", *Ibm.com*, 2020. [Online]. Available: <https://www.ibm.com/cloud/learn/neural-networks> [Accessed: 24- Feb- 2022].
- [18] S. Hymel & A. Fred-Ojala. *Introduction to embedded machine learning*. Coursera. Available <https://www.coursera.org/learn/introduction-to-embedded-machine-learning>. [Accessed: 25 April 2022]
- [19] Rfwireless World, "Advantages of FPGA | disadvantages of FPGA", *Rfwireless-world.com*, 2021. [Online]. Available: <https://www.rfwireless-world.com/Terminology/Advantages-and-Disadvantages-of-FPGA.html>. [Accessed: 10-Oct- 2021].
- [20] I. Akotey, *Edge Computing and Machine Learning on Embedded Systems*. Capstone Document, 2021.
- [21] *Tinymml.org*, 2021. [Online]. Available: <https://www.tinymml.org/>. [Accessed: 06- Dec- 2021].
- [22] Tensorflow, "TensorFlow Lite | ML for Mobile and Edge Devices" , 2022. [Online]. Available: <https://www.tensorflow.org/lite>. [Accessed: 28- Feb- 2022].

- [23] Edge Impulse, *Edge Impulse*, 2022. [Online]. Available: <https://www.edgeimpulse.com/>. [Accessed: 20- Apr- 2022].
- [24] A'râbi, Mohammad-Ali & Schwarz, Viktoria. (2019). General Constraints in Embedded Machine Learning and How to Overcome Them — A Survey Paper. 10.13140/RG.2.2.14747.21280.
- [25] V. Marco, B. Taylor, Z. Wang and Y. Elkhatib, "Optimizing Deep Learning Inference on Embedded Systems Through Adaptive Model Selection", *ACM Transactions on Embedded Computing Systems*, vol. 19, no. 1, pp. 1-28, 2020. Available: 10.1145/3371154.
- [26] T. Shah, "About Train, Validation and Test Sets in Machine Learning", *Towards Data Science*, 2017. [Online]. Available: <https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7>. [Accessed: 17- Mar- 2022].
- [27] A. Ruddha, "Confusion Matrix for Machine Learning", *Analytics Vidhya*, 2020. [Online]. Available: <https://www.analyticsvidhya.com/blog/2020/04/confusion-matrix-machine-learning/#:~:text=A%20Confusion%20matrix%20is%20an,by%20the%20machine%20learning%20model>. [Accessed: 22- Apr- 2022].

Appendix

A. Code for Arduino Nano BLE

Note: This file is dependent on the Arduino .zip library expected to be part of the user's Arduino libraries. The .zip library can be downloaded from GitHub at this link³. Once downloaded, run the nano_ble33_sense_accelerometer example file, or copy and paste the code below.

```
/* Includes ----- */
#include <motion-detection-final_inferencing.h>
#include <Arduino_LSM9DS1.h>

/* Constant defines ----- */
#define CONVERT_G_TO_MS2    9.80665f
#define MAX_ACCEPTED_RANGE  2.0f

/* Private variables ----- */
static bool debug_nn = false;
static uint32_t run_inference_every_ms = 200;
static rtos::Thread inference_thread(osPriorityLow);
static float buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE] = { 0 };
static float inference_buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE];

/* Forward declaration */
void run_inference_background();

void setup()
{
    Serial.begin(115200);
    Serial.println("Edge Impulse Inferencing Demo");
    if (!IMU.begin()) {
        ei_printf("Failed to initialize IMU!\r\n");
    }
    else {
        ei_printf("IMU initialized\r\n");
    }
    if (EI_CLASSIFIER_RAW_SAMPLES_PER_FRAME != 3) {
        ei_printf("ERR: EI_CLASSIFIER_RAW_SAMPLES_PER_FRAME should be equal to 3 (the
3 sensor axes)\n");
        return;
    }
}
```

³ <https://github.com/lorrainemakuyana2022/capstone-ML-models>

```

    inference_thread.start(mbed::callback(&run_inference_background));
}

float ei_get_sign(float number) {
    return (number >= 0.0) ? 1.0 : -1.0;
}

void run_inference_background()
{
    delay((EI_CLASSIFIER_INTERVAL_MS * EI_CLASSIFIER_RAW_SAMPLE_COUNT) + 100);
    ei_classifier_smooth_t smooth;
    ei_classifier_smooth_init(&smooth, );

    while (1) {
        int start_time = millis();

        memcpy(inference_buffer, buffer, EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE *
sizeof(float));

        signal_t signal;
        int err = numpy::signal_from_buffer(inference_buffer,
EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE, &signal);
        if (err != 0) {
            ei_printf("Failed to create signal from buffer (%d)\n", err);
            return;
        }
        ei_impulse_result_t result = { 0 };

        err = run_classifier(&signal, &result, debug_nn);
        if (err != EI_IMPULSE_OK) {
            ei_printf("ERR: Failed to run classifier (%d)\n", err);
            return;
        }

        ei_printf("Predictions ");
        ei_printf("(DSP: %d ms., Classification: %d ms., Anomaly: %d ms.)",
            result.timing.dsp, result.timing.classification, result.timing.anomaly);
        ei_printf(": ");

        const char *prediction = ei_classifier_smooth_update(&smooth, &result);
        ei_printf("%s ", prediction);
        ei_printf(" [ ");
        for (size_t ix = 0; ix < smooth.count_size; ix++) {
            ei_printf("%u", smooth.count[ix]);
            if (ix != smooth.count_size - 1) {
                ei_printf(", ");
            }
        }
    }
}

```

```

        else {
            ei_printf(" ");
        }
    }
    ei_printf("]\n");
    int time_taken = millis() - start_time;
    ei_printf("\n Time taken is %d ms \n", time_taken);
    delay(run_inference_every_ms);
}
ei_classifier_smooth_free(&smooth);
}

void loop()
{
    while (1) {
        uint64_t next_tick = micros() + (EI_CLASSIFIER_INTERVAL_MS * 1000);
        numpy::roll(buffer, EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE, -3);
        IMU.readAcceleration(
            buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 3],
            buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 2],
            buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 1]
        );

        for (int i = 0; i < 3; i++) {
            if (fabs(buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 3 + i]) >
MAX_ACCEPTED_RANGE) {
                buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 3 + i] =
ei_get_sign(buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 3 + i]) * MAX_ACCEPTED_RANGE;
            }
        }

        buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 3] *= CONVERT_G_TO_MS2;
        buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 2] *= CONVERT_G_TO_MS2;
        buffer[EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE - 1] *= CONVERT_G_TO_MS2;

        uint64_t time_to_wait = next_tick - micros();
        delay((int)floor((float)time_to_wait / 1000.0f));
        delayMicroseconds(time_to_wait % 1000);
    }
}

#if !defined(EI_CLASSIFIER_SENSOR) || EI_CLASSIFIER_SENSOR !=
EI_CLASSIFIER_SENSOR_ACCELEROMETER
#error "Invalid model for current sensor"
#endif

```


B. Code for FRDM-K64F

Large files are needed to be able to run this file on the FRDM-K64F and are available on GitHub at this link⁴. In the frdm-k64f folder, download and extract the library zip file available and place its contents in the source folder in your workspace. Copy and paste the file below in the file you will run on the board.

```
#define ML_ON_K64F

#ifndef ML_ON_K64F

#include "board.h"
#include "peripherals.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "MK64F12.h"
#include "fsl_debug_console.h"

// Include files from Edge Impulse Library
#include <ei/classifier/ei_run_classifier.h>
#include <ei/porting/ei_classifier_porting.h>

#define RED_LED          (22)    // PTB22 red onboard LED
#define MASK(X)          (1UL << X)
#define LED_INTERVAL    (500)  // 500ms restart

// CONFIGURE SYSTICK TIMER TO GET INFERENCE TIME
volatile unsigned long counter = 0;
void configure_SysTick();
unsigned long millis(void);
void SysTick_Handler(void);

// Initialize LED to turn on RED during inference
void initialize_LED();

// Run inferencing
int run_ML_inferencing();

// Callback function declaration
static int get_signal_data(size_t offset, size_t length, float *out_ptr);
```

⁴ <https://github.com/lorrainemakuyana2022/capstone-ML-models>

```

// Raw features copied from test sample (Edge Impulse > Model testing)
// To be always changed by Microcontroller Accelerometer
static float input_buf[] = {
    -0.6550, 2.7881, 19.1040, -3.7558, ... };

int main(){
    initialize_LED();
    configure_SysTick();
    unsigned long start_time = 0u;
    unsigned long end_time;

    while (1){
        run_ML_inferencing();
        unsigned long end_time = millis();
        printf("Time for inferencing is %d ms", end_time);
        end_time = 0u;
    }
}

int run_ML_inferencing() {
    signal_t signal;           // Wrapper for raw input buffer
    ei_impulse_result_t result; // Used to store inference output
    EI_IMPULSE_ERROR res;     // Return code from inference

    size_t buf_len = sizeof(input_buf) / sizeof(input_buf[0]);

    if (buf_len != EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE) {
        printf("ERROR: The size of the input buffer is not correct.\r\n");
        printf("Expected %d items, but got %d\r\n",
            EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE,
            (int)buf_len);
        return 1;
    }
    signal.total_length = EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE;
    signal.get_data = &get_signal_data;

    res = run_classifier(&signal, &result, false);

    printf("run_classifier returned: %d\r\n", res);
    printf("Timing: DSP %d ms, inference %d ms, anomaly %d ms\r\n",
        result.timing.dsp,
        result.timing.classification,
        result.timing.anomaly);
    printf("Predictions:\r\n");
}

```

```

for (uint16_t i = 0; i < EI_CLASSIFIER_LABEL_COUNT; i++) {
    printf("  %s: ", ei_classifier_inferencing_categories[i]);
    printf("%.5f\r\n", result.classification[i].value);
}

#if EI_CLASSIFIER_HAS_ANOMALY == 1
    printf("Anomaly prediction: %.3f\r\n", result.anomaly);
#endif

return 0;
}

static int get_signal_data(size_t offset, size_t length, float *out_ptr) {
    for (size_t i = 0; i < length; i++) {
        out_ptr[i] = (input_buf + offset)[i];
    }
    return EIDSP_OK;
}

void initialize_LED() {

    SIM->SCGC5 |= SIM_SCGC5_PORTB_MASK;
    PORTB->PCR[RED_LED] &= ~PORT_PCR_MUX_MASK;
    PORTB->PCR[RED_LED] |= PORT_PCR_MUX(1);

    GPIOB->PDDR |= MASK(RED_LED);

    GPIOB->PSOR |= MASK(RED_LED);
}

void configure_SysTick() {
    SysTick->LOAD = (20971520u/1000u)-1 ; //configure for every milli sec restart.
    SysTick->CTRL |= SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk
|SysTick_CTRL_TICKINT_Msk;
}

void SysTick_Handler(void) {
    counter++;
}

unsigned long millis(void) {
    return (unsigned long) counter;
}

#endif

```

C. Challenges and ways to address them

1. Undefined references to functions

This problem arises when using a board or device that is not supported by the Edge Impulse Platform. When using unsupported hardware, some functions have to be defined by the user. When these functions are not defined properly and the appropriate file not put in its correct location, an error (showing like Fig A below) occurs.

```
in function `ei::ei_matrix::ei_matrix(unsigned long, unsigned long, float*)'  
numpy_types.h:104: undefined reference to `ei_malloc(unsigned int, unsigned int)'
```

Fig A: Undefined reference error

To counter this error, ensure the file with the user defined functions is at its appropriate location and includes the appropriate header file. For more information, you can use the Edge Impulse Forum topic⁵ created which as links to example files and structures, and videos to watch.

2. Data Collection using the Edge Impulse CLI

For supported hardware, the Edge Impulse CLI can be installed and used to collect data for the machine learning model. The CLI also comes with predefined datasets that can be used to create a model. Detailed steps to use the Edge Impulse CLI can be found on the Edge Impulse Platform Website⁶.

⁵ <https://forum.edgeimpulse.com/t/deployment-of-project-as-a-c-library-for-frdm-k64f/4161>

⁶ <https://docs.edgeimpulse.com/docs/edge-impulse-cli/cli-installation>