MANCHESTER
1824
The University of Manchester

Deep Learning Neural Network for Human Activity Recognition Optimised for Implementation on Microcontroller Units

Third Year Individual Project

April 2021

Wenqiang Lai

10255317

Supervisor: Dr Alex Casson

Department of Electrical and Electronic Engineering

Contents

Abstract 1
1. Introduction 2
1.1. Motivation 2
1.2. Aim & Objectives 3
1.3. Report Structure 3
2. Literature Review 4
2.1. Machine Learning for Human Activity Recognition 4
2.2. Deep Learning Methods 5
2.2.1. Convolutional Neural Network: Evolution of Architectures 5
2.2.2. Recurrent Neural Networks: Long Short-Term Memory 7
2.3. Edge Machine Learning 9
2.3.1. Overview of AI accelerators 10
2.3.2. Optimisation for Deployment on Edge 11
2.3.3. Frameworks for Model Development 13
2.4. Related Works 14
3. Methods 15
3.1. Dataset Selection 15
3.2. Dataset Description & Pre-Processing 16
3.3. Proposed Model Architectures 18
3.4. Model Training, Conversion and Optimisation 20
3.5. Model Deployment & Measurement setup 21
3.5.1. Classification Performance 22
3.5.2. Latency and Energy Consumption 24
3.5.3. Memory Usage 26

4. Result and Discussion 27
4.1. Model Evaluation on PC 28
4.1.1. Classification Performance of Proposed Models 28
4.1.2. Effect of TensorFlow Lite Optimisation 30
4.2. Model Evaluation on Device 31
5. Discussion and Conclusion 33
5.1. General Discussion 33
5.2. Conclusion 33
5.3. Limitations and Future Works 34
6. References 34
7. Appendices 38
7.1. Appendix 1 – Code 38
7.2. Appendix 2 – Covid19 statement 38

Total word count: 9310

Abstract

With advances in chip technology and edge computing, deep learning methods for Human Activity Recognition have been deployed on various devices, outperforming conventional machine learning methods. Microcontroller units are ideal candidates for deep learning model deployment, as they usually contain one or more sensors, making them ideal for the deep learning model to perform classification. This on-device classification guarantees the privacy of users. However, neural network-based methods are computationally heavy, for which their integration into microcontroller units have not yet been evaluated comprehensively. With the recent improvements in neural network optimisation techniques, there is an increased demand to leverage these techniques to implement deep learning models on microcontroller units. In this project, a set of models based on convolutional neural network architecture were trained and assessed in terms of accuracy, latency, memory usage and power consumption. Models were trained using TensorFlow 2 library, while TensorFlow Lite library was used for model conversion, optimisation and deployment on the Arduino Nano 33 BLE board. From the measurement, the quantised models were up to 14.6x faster and 3.9x more memory efficient compared to non-quantised models. Such optimisation results ensured the feasible implementation of deep learning models on low-power microcontroller units for fast on-device inference.

1. Introduction

1.1. Motivation

Connected embedded devices, collectively referred to as the Internet of Things (IoT), are pervasive in people's daily lives. The omnipresence of microcontroller units (MCUs) with embedded sensors such as accelerometers and gyroscopes has created an ideal environment for the mass collection of raw data [1]. Currently, MCU sensor data is transmitted to a cloud server for processing, which then enables numerous applications in various domains, from healthcare to smart homes [2]. However, this transmission often requires a stable internet connection, which adds latency to the transmission's overall execution. One solution to this issue is performing the task at the data's source, also known as edge computing. Edge computing reduces transmission latency and saves network bandwidth, enhancing performance in areas with limited internet connection [3]. Moreover, edge computing prevents the unauthorised exploitation of user data [4].

MCUs have limited memory capacity, ranging from tens to a few hundred kilobytes, and poor computing capability which has made them inadequate for use as deployable targets for deep learning [3]. However, recent optimisation advancements have now made on-device implementation of deep learning models possible. A deep learning model implemented on edge computing MCUs might overcome the limitations of cloud-based transmission and carry further benefits such as rapid real-time inference and low-power usage. This would accelerate development of data-driven applications in various areas [1].

Deep learning methods are highly proficient at solving complex classification problems, and deep learning architectures such as Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) exploit the sequential, hierarchical nature of sensor data to perform classification tasks. Deep learning models therefore can extract abstract features automatically, without human intervention, and reduce time spent on data pre-processing [2]. Human Activity Recognition (HAR), a task that classifies a user's ongoing physical activity based on time-series sensor data, is a significant deep learning use case [5]. HAR is used in various critical applications such as chronic disease management, ambient assisted living and physical workload tracking [6],[7]. HAR applications have traditionally used conventional machine learning classifiers such as Support Vector Machine (SVM), Decision Trees and Naïve Bayes (NB). These classifiers are unable to process raw data and therefore require manual feature extraction, leading to reduced accuracy and problems with overfitting [7]. A recent work, however, suggests that deep learning models

perform better than conventional methods in classification task when trained with a large dataset [8]. As such, deep learning models implemented on MCUs have become the most promising solution for HAR issues. For real-world applicability, however, many aspects must be considered besides prediction accuracy, such as latency and power and memory consumption.

1.2. Aim & Objectives

Given recent advancements in optimising neural networks and an increased demand for applying on-device neural network-based solutions to real-life scenarios, this project aims to implement deep learning models for HAR on a MCU then evaluate the relationship between on-device performance and overhead. An evaluation of real-life implementation will then be provided.

The following objectives must be met to accomplish this research project's aim:

1) Conduct a literature review of related research focusing on methods used for HAR tasks.
2) Identify state-of-the-art deep learning methods for HAR, selecting a suitable model architecture.
3) Select an off-the-shelf AI accelerator MCU to be the target device, justifying its selection.
4) Select an open-source machine learning framework to train models and run inference on MCU.
5) Implement the selected model architecture with different configurations to obtain a set of trained models to be evaluated.
6) Develop an embedded software to feed accelerometer data to the model, run inference and return inference results.
7) Convert and optimise models to suit the MCU, using the selected framework.
8) Deploy and run the models on the selected MCU and record their performances in terms of accuracy, latency, power consumption and memory usage.
9) Discuss the overall performance and the practicability of on-device deep learning for HAR.

1.3. Report Structure

The contents of this report are arranged as follows. Section 2 is a literature review of popular deep learning methods with relevant background knowledge; information on the constraints and methods for deployment on MCUs will also be explored. Section 3 introduces the selected dataset, target device and proposed model architecture with different configurations; details on the model design, testing and deployment will be provided. Section 4 presents the performance results of all tested models analyses these findings. Section 5 gives a general discussion about the project and

draws conclusions concerning the overall practicability and limitations of on-device deep learning methods for HAR. The Conclusion contains recommendations for future study.

2. Literature Review

2.1. Machine Learning for Human Activity Recognition

Given the technological advancements and accompanying behavioural changes made over the last two decades, the amount of data in circulation has grown exponentially. Human intelligence is often incapable of understanding this data, so machine learning, a rapidly growing subset of the field of artificial intelligence (AI), may offer a way to make this data more accessible to human use. Machine learning refers to methods which allow machines to learn automatically from input data by identifying inherent statistical patterns and making evidence-based decisions on newly input data, all without human intervention.

Machine learning algorithms are divided into three paradigms: supervised learning, unsupervised learning and reinforcement learning. Supervised learning refers to a model trained with a labelled dataset consisting of pairs of input and desired output data [9]. Supervised learning methods have been successfully adopted for solving classification and regression tasks. In contrast, unsupervised learning uses an unlabelled training dataset, instead relying on grouping output data with common attributes into clusters. Finally, reinforcement learning algorithms rely on a closed-loop feedback scheme in which the algorithm's interactions with the environment produce qualitative feedback that provides either rewards or punishments. The ultimate goal of the feedback scheme is to maximise the reward, or minimise the punishment, received [9].

Human Activity Recognition (HAR) has been an active research field for the past two decades and is crucial to many prominent applications in various critical domains such as healthcare, human-centred computing and ambient assisted living [10]. HAR is used to classify people's ongoing physical activity using sensor data from an individual's environmental or body-worn sensors. As such, supervised machine learning algorithms are potentially ideally suited to HAR fields. While traditional machine learning algorithms used for classification tasks, such as Decision Trees (DT), Bayesian Network (BN), Support Vector Machine (SVM) and K-Nearest Neighbour (K-NN), have demonstrated relatively high accuracy in predicting human activities [10], they can hardly be implemented in a real world context because they learn from data described by shallow manual techniques [11]. In contrast, representation learning (or feature learning) methods may streamline

raw data transmission by extracting abstract features automatically [12].

2.2. Deep Learning Methods

Deep learning methods are forms of representation learning whose bioinspired hierarchical architecture enables the automatic extraction of complex and abstract features. Deep learning architectures consist of multiple stacked non-linear layers, each represented by numerous neurons with weighted connections, associated bias term and activation function. In gradient-based deep learning models, a backpropagation algorithm automatically tunes the weights and optimises the objective function to evaluate a model's performance. Thus, the objective function (often referred to as a "loss function") measures the error between predictions and desired outputs. In a stacked deep learning model, each layer is capable of transforming a previous layer's input into representation with a greater level of abstraction. This transformative process repeats throughout all intermediate, or hidden, layers until reaching the output layer, where the final outputs are represented in a human-understandable result [13].

Deep learning methods have clear advantages when compared to more conventional methods of extracting raw data, especially for HAR. Manual data collection tasks generally have lengthy durations, ranging from tens of seconds to a few minutes, which is reduced with deep learning methods. Meaningful predictions of ongoing activity for HAR require an analysis of sensor data using fixed-length time windows, which may be interrupted by manual data collection [10]. Deep learning methods, therefore, have been widely applied to HAR classification tasks. Typical neural network methods used for HAR include Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), both of which exploit spatial and temporal information from time series data.

2.2.1. Convolutional Neural Network: Evolution of Architectures

The first CNN architecture for pattern recognition, called Neocognitron, was developed by Kunihiko Fukushima in 1980 [14]. The main drawback to this architecture was its lack of a supervised learning algorithm (for example, an error backpropagation algorithm). In 1989, Yann LeCun proposed a series of new architectures, called LeNet, to improve these early CNN weaknesses. The latest version of LeNet, LeNet-5, has been trained with a backpropagation algorithm maintaining concepts similar to Neocognitron: local connectivity, shared weights and down-sampling. Figure 2.1 illustrates LeNet-5's architecture for digits recognition [15].

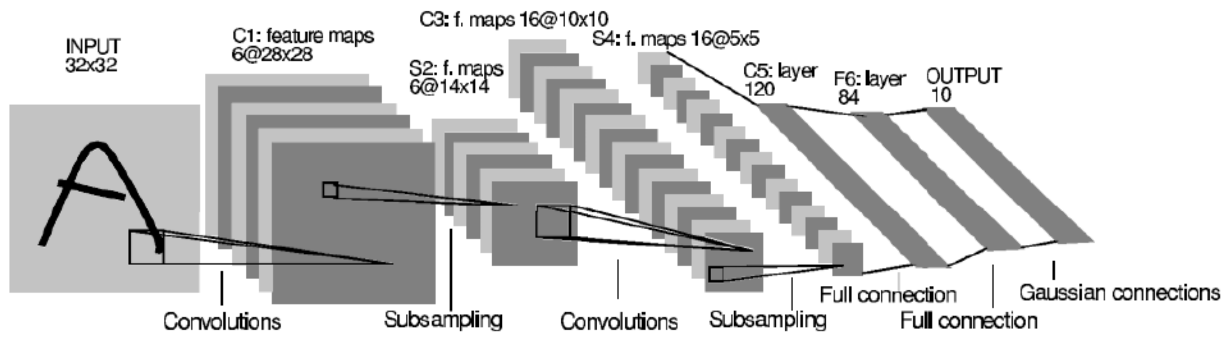

Figure 2.1: LeNet-5 for digits recognition [15]

LeNet-5 has two central building blocks, convolutional layers (C1 and C3) and down-sampling layers (S2 and S4), which are followed by two fully-connected layers (C5 and F6) and a final output layer.

In a convolutional layer, each unit is connected to a fixed-size local patch of a previous layer's output through a vector of weights called filter, comprising a set of trainable parameters. The collection of such units forms a feature map. One layer may contain multiple filters, each producing a feature map by computing dot product with input data. These feature maps then pass through a $tanh$ activation function, adding non-linearity to the output:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (1)$$

All units within a feature map share the same filter bank, and there may be multiple filter banks producing an equal number of feature maps within one layer.

In a down-sampling layer, also called a Max-Pooling layer, each unit covers a patch of the previous layer, through which several features are merged into a single feature by taking the local patch's maximum value. Advantages of Max-Pooling layers include reduced input size for the next layer, reduced computational cost and increased tolerance to small positional changes.

Actual classification is done in the last three layers. Each C5 neuron is connected to all the feature maps from S4, forming a vector of 120 units. F6 is fully connected to C5, which narrows the vector dimension to 84. The output layer uses a Euclidean RBF classifier to perform its final classification [15].

LeNet-5 has been more successful than other conventional methods in recognising patterns of small-scale images (32 x 32 pixels) [15]. However, it also suffered from various limitations,

including long training time and heavy computational overhead for large-scale image recognition tasks.

Consequently, CNN architectures were generally ignored by the computer vision community until 2012, when the ground-breaking CNN architecture AlexNet overcame these drawbacks and won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 [16]. AlexNet classified 1.2 million large scale images with a top-5 error rate of 15.3%, far lower than any competing methods.

AlexNet is a variant of LeNet that contains an increased number of layers and filters per layer, resulting in around 60 million trainable parameters (around 1000 times more than LeNet-5). Such a huge network was made possible with the implementation of several new features, such as dropout regularisation technique, ReLu activation functions and hardware accelerators. Dropout regularisation techniques are used in the last two fully-connected layers and reduce overfitting by randomly setting the output of a certain percentage of neurons to zero for each input batch so that these neurons will not participate in training. ReLu, an abbreviation of Rectified Linear Unit, is mathematically simpler than $tanh$ used in LeNet.

$$f(x) = \max(0, x) \tag{2}$$

(2) prevents gradients from approaching zero during backpropagation since its derivative is 1 even for large x. Finally, AlexNet is trained on two graphics processing units (GPUs) optimised to accelerate the calculation of 2D convolution. AlexNet initially used a normalisation layer, Local Response Normalisation (LRN), to aid with generalisation, but this layer has recently been replaced by a more effective layer named Batch Normalisation (BN). BN enhances generalisation and reduces training time by normalising an input batch's internal distribution [17].

The general structure and features of LeNet and AlexNet are used widely in computer research and applications. For multi-class classification problems, the softmax function is generally used in the output layer to map each class's output scores into a numerical vector, ranging from 0 to 1 and summing to 1, which represents its probabilities.

2.2.2. Recurrent Neural Networks: Long Short-Term Memory

Although the first RNN architecture was introduced by John Hopfield in 1982, it has gained widespread attention only in the last two decades thanks to increased computational power and advancements to RNN architecture. RNNs demonstrate outstanding performance in various tasks

with sequential input data, such as natural language processing, as their architecture allows them to exploit temporal correlations in input sequences [18].

Figure 2.2 illustrates a simple RNN and its unfolded workflow. For forward computations, current input x_t and past state s_{t-1} determine the current state s_t of neurons in the hidden layer, upon which output o_t is dependent (t indicates discrete time step). The matrices U, V and W contain parameters used for forward computation, which can be updated using backpropagation [12].

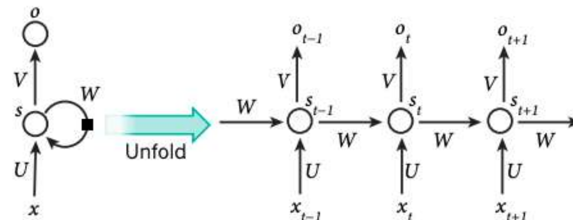

Figure 2.2: Architecture of a simple unidirectional RNN [12]

Training RNNs was initially challenging, for gradients in long sequences either explode or vanish during backpropagation [19]. In 1997, Sepp Hochreiter and Jürgen Schmidhuber overcame these issues with Long Short-Term Memory network (LSTM), a variant of RNN. LSTM introduced a special unit, the memory cell, which retained long-term memory. Figure 2.2 illustrates the structure of a typical memory cell, which uses three internal gates to control the flow of information [12]. The forget gate, f_t, controls what information taken from the previous step's memory cell time step, c_{t-1}, should be kept in the current memory cell, c_t. The input gate, i_t, controls how much of the new input data should flow into c_t. The output gate, o_t, controls how much information from c_t should flow into hidden state, h_t.

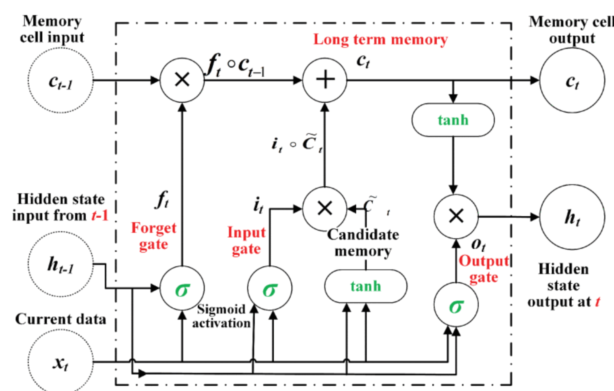

Figure 2.3: Memory cell structure [21]

As described by Figure 2.3, all three gates are multiplicative and attached to a sigmoid (σ) activation function with an output ranging from 0 to 1:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

The mathematical formulae for the three gates are as follows:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{4}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{5}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{6}$$

$[W_f, W_i, W_o]$ and $[b_f, b_i, b_o]$ are the weights and biases of the forget, input and output gates, respectively. Note that the candidate memory cell, $\tilde{C}_t$, which participates in updating the current memory cell's information, has its own associated weight and bias:

$$\tilde{C}_t = \tanh\,(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{7}$$

The output value of the candidate memory cell lies between -1 and 1, as its activation function is (1). Finally, (4), (5) and (7) may be used to update the memory cell:

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{C}_t \tag{8}$$

Note that new information received from the input gate and candidate memory cell is linearly accumulated in the current memory cell. Using (6) and (8), the hidden state of the current memory cell is calculated as follows:

$$h_t = o_t \circ \tanh\,(c_t) \tag{9}$$

As with CNNs, stacked-LSTMs use LSTM layers to extract features from raw data, while actual recognition tasks are performed by multiple fully-connected layers.

2.3. Edge Machine Learning

The proliferation of the IoT and increased capability of edge devices has created a new computing paradigm known as edge computing. In contrast to cloud computing, edge computing performs computations directly at data sources. Edge computing particularly improves applications requiring real-time responses and/or data privacy, as no data transmission is needed outside the

edge device [22]. HAR is an ideal task for edge computing, as a user's privacy may be undermined if the personal data required for recognition is transmitted to a third party. Moreover, deep learning methods are ideal candidates for HAR tasks, since they learn directly from raw data generated by embedded sensors. However, implementing deep learning models on edge devices is challenging, as these models are relatively large and complex for resource-constrained edge devices, especially MCUs. These problems can be addressed by optimising either the device's performance or neural network architecture.

2.3.1. Overview of AI accelerators

Different hardware manufacturers are currently producing AI accelerators to enhance deep learning-based tasks from the hardware perspective. These accelerators are based on GPUs, field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs), which allow more efficient computation than generic CPUs. In recent years, vendors have begun developing AI accelerators for edge implementation. For example, the Coral Dev Board by Google is a single-board computer with Tensor Processing Unit (TPU), an ASIC designed by Google itself [23]. These devices, however, are not yet popular, and their cost is relatively high when compared with typical embedded processors.

Current off-the-shelf chips are often leveraged to develop supporting software kernels for AI acceleration. For example, Arm Cortex-M series processors can now accelerate common neural networks by leveraging kernels that accelerate typical neural network layers (see details in section 2.3.2). Hardware manufacturers such as Arduino and Sparkfun Electronics have developed MCUs for deep learning-based tasks using Arm Cortex-M processors.

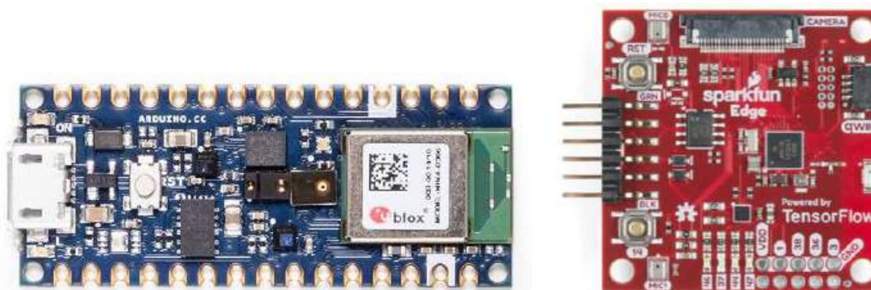

Figure 2.4: Arduino Nano 33 BLE Sense (left) and SparkFun Edge Development Board (right) [24][25]

Arduino Nano 33 BLE Sense is an Arduino product designed for AI acceleration. The board is based on the nRF52840 MCU, with a 32-bit Arm Cortex-M4F processor with 64MHz clock speed [24]. The board features a direct memory access module (DMA) and an inertial memory unit (IMU)

consisting of one 3D accelerometer, one 3D gyroscope and one 3D magnetometer. The board's size ($45mm \times 18mm$) and weight (5g) are ideal for unobtrusive applications. In addition to its IMU, the Arduino Nano has a variety of embedded sensors such as a microphone, pressure sensor and temperature sensor. These sensors provide abundant data used for a variety of neural network-based applications. Additionally, a power management unit (PMU) allows one to automatically switch between different operational modes to achieve the lowest power consumption.

The SparkFun Edge development board is a counterpart to the Arduino Nano, manufactured by SparkFun Electronics. The board is based on the Apollo3 Blue MCU, which also has an Arm Cortex-M4F processor. The nominal CPU clock frequency is 48MHz, which can be doubled in burst mode. SparkFun Egde features an ultra-low power MCU, consuming $6\mu A$ per MHz when running and only $1\mu A$ when asleep [25]. The board contains one 3D accelerometer, one microphone and a camera connector.

There are many other off-the-shelf MCUs that share similar features and constraints, such as the STM32F746, Adafruit EdgeBadge, Espressif ESP-32 etc. Comparison of these MCUs is beyond the scope of this project.

2.3.2. Optimisation for Deployment on Edge

Edge device optimisation is achieved by developing specific low-level computation kernels for generic processors to speed up common operations in neural networks. For example, Arm Cortex-M CPUs have a set of kernels, CMSIS-NN, designed specifically to enhance performance and reduce memory footprint when running neural networks [26]. The general structure of a CMSIS-NN neural network kernel shown in Figure 2.5:

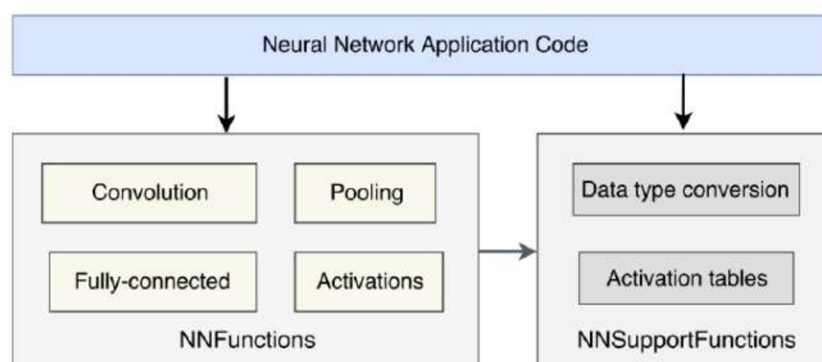

Figure 2.5: General structure of a neural network kernel in CMSIS-NN [26]

The kernel consists of NNFunctions and NNSupportFunctions. NNFunctions are designed to implement common neural network operations such as convolution and max pooling, while NNSupportFunctions complement NNFunctions with utilities such as data type conversions. According to [26], CMSIS-NN increases the throughput and energy efficiency of neural networks by 4.6x and 4.9x, respectively.

Neural network optimisation is currently an active research topic. Howard et al. [27] have recently proposed a parameter-efficient CNN architecture, MobileNet, for use in embedded applications. MobileNet utilises a new technique called depthwise separable convolution, which splits standard convolution into two distinct operations, depthwise convolution and pointwise convolution.

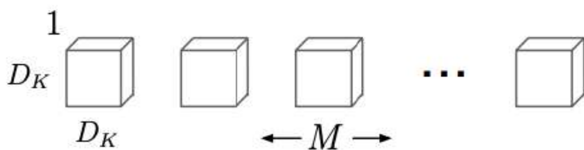

Figure 2.6: Depthwise convolutional filters [27]

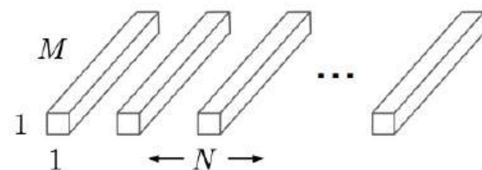

Figure 2.7: Pointwise convolutional filters [27]

In a depthwise convolution layer for M input channels, each input channel is convolved using a single filter sized $D_k \times D_k$, producing M feature maps. The output of depthwise convolutions is combined in the pointwise convolution layer by applying N filters sized $1 \times 1 \times M$, where N is the number of output channels. Such architecture has been shown to significantly reduce computation and model size with minimal impact on accuracy [27].

Another active research direction is the precision reduction of neural networks, also known as quantisation. Generally, deep-learning model parameters are trained using 32-bit floating-point data representation. Studies demonstrate that low-precision fixed-point data representation achieves similar performance to benchmark results [28]-[30]. Quantisation may be applied during training, known as quantisation-aware training, or afterwards, known as post-training quantisation. Quantisation-aware training requires a consistent procedure from design phase to implementation phase [28],[31], while post-training quantisation converts pre-trained floating-point models to fixed-point models.

Jacob *et al.* [31] have proposed a quantisation scheme, Integer-arithmetic-only inference quantisation, which allows performing inference with 8-bit integer input, output, weights and activations; only the bias vector is in 32-bit integer format. Figure 2.8 provides an illustration of

this scheme.

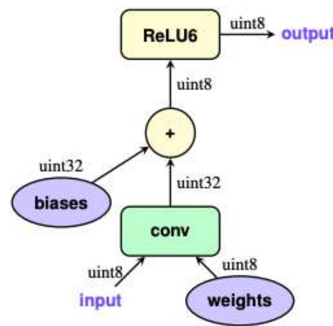

Figure 2.8: Integer-arithmetic-only quantisation [31]

2.3.3. Frameworks for Model Development

There are various open-source machine learning frameworks for the development of deep learning models, including Caffe, Caffe2, PyTorch, Keras and TensorFlow. Caffe2 has recently merged with PyTorch, while Keras has been integrated into TensorFlow 2.0. For edge implementation, TensorFLow and PyTorch offer lightweight solutions, namely TensorFlow Lite and PyTorch Mobile [32],[33]; the latter, however, does not support embedded systems without file systems.

TensorFlow was released by Google in 2015, aiming to facilitate the development process of large-scale machine learning models [34]. The libraries come in the form of Python libraries to provide high-level abstraction, but the kernels are written in C++ for better performance. TensorFlow supports a wide range of target devices, including generic processors, as well as edge devices. With the integration of Keras library into TensorFlow 2.0 released in 2019, the development process became even easier. TensorFlow Lite is the framework released by Google to provide support for model conversion and optimization for edge devices, e.g., conversion and optimization APIs [32]. For resource constrained MCUs, Google offered TensorFlow Lite for Microcontroller (TLFM), which is the state-of-the-art open-source framework for running inference on MCUs. TLFM uses a unique approach based on an interpreter that simulates and live neural network model. The interpreter-based approach increases portability and flexibility [35]. Other frameworks for inference on embedded devices include: the open-source Embedded Learning Library (ELL) by Microsoft [36], which is a cross-compiler that generate machine code to run on the device; STM32Cube.AI, which could convert and optmise the pre-trained model for STM32-series MCUs [37].

2.4. Related Works

There have been many research and works being conducted to evaluate the on-device performance of deep learning methods. This review of literature aimed to identify the state-of-the-art methods for HAR and aid the model architecture selection. Besides, previous works with similar research scope were also reviewed to help design the testing process.

[38]-[40] are early works conducted using classical machine learning algorithms, such as Decision tree, Naïve Bayes, K-Nearest Neighbors and Hidden Markov Models. Despite that high classification performance were achieved, these conventional algorithms suffered from a common drawback, as these algorithms only learn from data described by manually engineered features. Since the success of AlexNet in 2012, there have been a rise in the number of research using CNN architecture. Ronao and Cho [1] proposed a multi-layer CNN architecture with the attempt to apply convolutional kernels along the time axis. The proposed model showed the capability of exploiting the temporal correlation between time series data, achieving an overall accuracy of 94.79%. Ordonez and Roggen [41] proposed a classifier architecture consisting of Deep CNN (DCNN) and LSTM, achieving F1 scores of 0.93 and 0.958. F1 score is described later (section 3.5.1). This classifier worked by applying convolutional kernels to the input sensor signal image to extract abstract features and use LSTM to extract temporal dependencies in the feature maps. In a more recent work by Zebin et al. [5], a stacked CNN model was proposed, which obtained an accuracy of 96.4%. The effect of quantisation was evaluated in this work, which achieved 4x+ mode size reduction.

Zhang et al. [42] investigated the performance of various neural networks by deploying them on MCUs. This work evaluated the on-device performance by setting out three memory constraints. The model accuracy was measured under each of the constraint. The best performing model architecture was depthwise separable convolutional neural network (DS - CNN), which obtained 94.4% accuracy. Novac et al. [6] conducted a similar research on the relationship between the performance and memory usage of on-device deep learning model. This work compared the performance of supervised learning, unsupervised learning and semi-supervised learning algorithms. This work achieved an accuracy of 92.88% for supervised learning (CNN) and 84% for unsupervised learning (Self Organising Map). This work attempted to evaluate the models using battery run-time as a metric. A 19 hour batter run-time was reported for its best performing supervised model. A detailed review of other deep learning related works is available in [43]-[45].

3. Methods

In this section, the experimental method for on-device HAR performance evaluation is described in detail. Figure 3.1 is a flowchart illustrating the key experimental procedures. Python 3.8 and its built-in third-party libraries were used to pre-process the dataset, while TensorFlow 2.3.1, available as a Python library, was the framework used to train the model. The APIs from TensorFlow Lite were used for model conversion and optimisation.

The embedded device used for evaluation was Arduino board, which is highly compatible with TensorFlow Lite. TFLM, available as an Arduino library, was used for on-device inference. Arduino IDE and its built-in helper functions were used to develop the embedded software. Details of the experimental procedures are given in the following sub-sections. All code used in this project could be found in the GitHub repository link in Appendix A1.

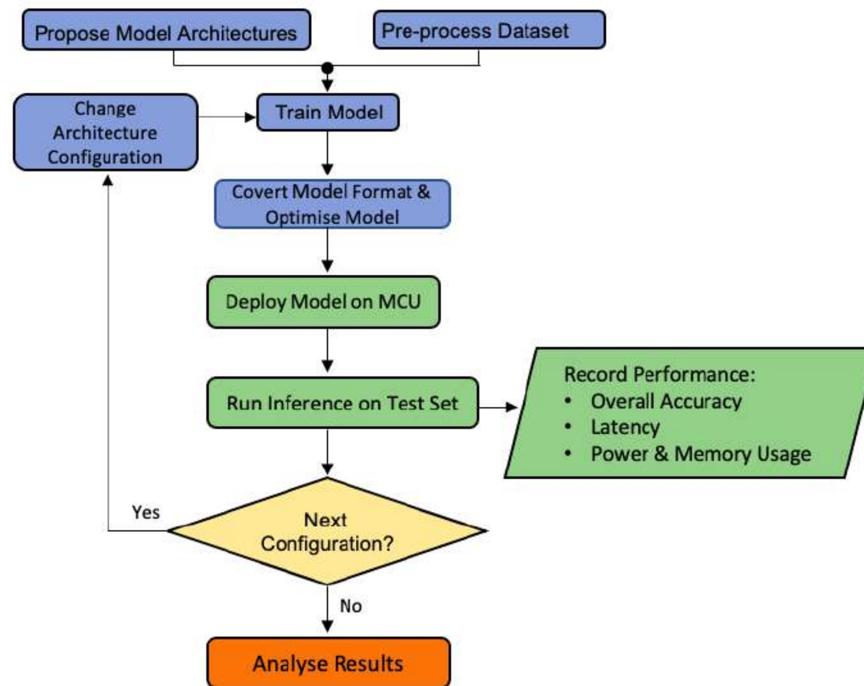

Figure 3.1: Experimental procedures for on-device performance evaluation

3.1. Dataset Selection

The dataset used to train a model determines its functionality and performance, and it is ideal to train a model using a dataset produced specifically for that model's requirements. Unfortunately, collecting and processing a large amount of raw data is usually costly, and due to this project constraints, a public dataset for HAR was selected for training and testing purposes. Several datasets, including Opportunity [46], Pamap2 [47], WISDM [48] and the dataset donated by

Anguita et al. (referred to as UCI-HAR dataset hereinafter) [49], were available on the UCI Machine Learning Repository.

The desired dataset had several important requirements. Since the goal of this project was to perform HAR on a body-worn device, the dataset had to originate from unobtrusive wearable devices with embedded sensors. The dataset also had to be balanced in terms of sensor quantity and activity types. A high number of sensors provide great amounts of data from which to learn, improving classification accuracy. However, some datasets have an excessive number of sensors for training a model meant to be implemented on devices with a limited amount of sensors. Therefore, the Opportunity dataset, which contains five sensors, was excluded. Moreover, datasets Pamap2 and WISDM dataset were also excluded due to having too many activities types, which could lead to inefficient classification [5]. Ultimately, the UCI-HAR dataset was selected for training and testing because it collected data from only two sensors, an accelerometer and a gyroscope, which was then classified into six basic daily activities.

3.2. Dataset Description & Pre-Processing

Data from a UCI-HAR dataset was collected from a group of 30 subjects wearing a smartphone (Samsung Galaxy S II) on their waist. The accelerometer and gyroscope, embedded in the smartphone, sample the triaxial total acceleration and triaxial angular velocity at a constant rate of 50 Hz. The body's estimated acceleration was obtained using a Butterworth low-pass filter with 0.3 Hz cut-off frequency, eliminating the low frequency gravitational component of total acceleration. Both total acceleration and estimated body acceleration were recorded in gravity unit 'g' (equivalent to $9.80665\ m/s^2$), while angular velocity was measured in rad/s. Each axis corresponds to one input channel, so the three triaxial measurements give nine channels. Time series data from each channel is segmented into fixed-width sliding-windows (128 samples/window), with a 50% overlap. The dataset classifies six activity types which are associated with numbers ranging from 1 to 6, as shown in Table 3.1.

Table 3.1: Activity types and their associated number

Activity type	Walking	Walking upstairs	Walking downstairs	Sitting	Standing	Lying
Activity number	1	2	3	4	5	6

The dataset contains 10,299 labelled activities and was split into a training set and a test set,

accounting for 70% and 30% respectively. A 20% samples in training set was hold as validation set, which were used during training. The test set was used for post-training model evaluation. Before starting the training process, the dataset has to be pre-processed by applying the following methods:

1) **Scaling:** All values in the dataset are scaled to lie within a small range of value, in order to accelerate computation and avoid training bias caused by large outliers in the dataset. Scaling is accomplished using the "MinMaxScaler" function from Python Library's scikit-learn class. This function acts as a scaler for each input channel, normalising all values into a predefined range according to the channel's minimum and maximum numbers. The selected range is [-1, 1], since the dataset contains negative values. Scaling is represented by the following function:

$$x_{scaled}\,(x) = \frac{(x - x_{min})(max - min)}{(x_{max} - x_{min})} + min \tag{10}$$

   Where x is the scaled value, x_{max} and x_{min} are a channel's maximum and minimum values and max and min represent the scaling range.

2) **Segmentation, Combination and Reshaping:** Raw data from the time series must be segmented into a fixed-width sliding window, enabling the CNN model to exploit the temporal correlation between samples. As mentioned, samples from each channel are segmented into windows of 128 samples, with sample windows from 9 channels combined together to create a matrix sized 128x9. Since 2D convolutional layers require 3D input sensors, input data is reshaped to 128x9x1. The "dstack" and "reshape" functions in Numpy library were used to combine and reshape the samples.

3) **One-Hot Encoding for Activity Labels:** the activity labels provided in the UCI-HAR dataset consist of integer numbers between 1 and 6, each represents an activity as shown in Table 3.1. These activity labels are categorical data which should be converted to One-Hot encoded labels. Since there are 6 categories, One-Hot encoding will create a binary label vectors of size 6, and the correct activity label is represented by 1 (Figure 3.2). Since index of arrays starts from 0, and the activity labels should be subtracted by one before encoding. One-Hot encoding could be done by using 'to_categorical' function from Keras library.

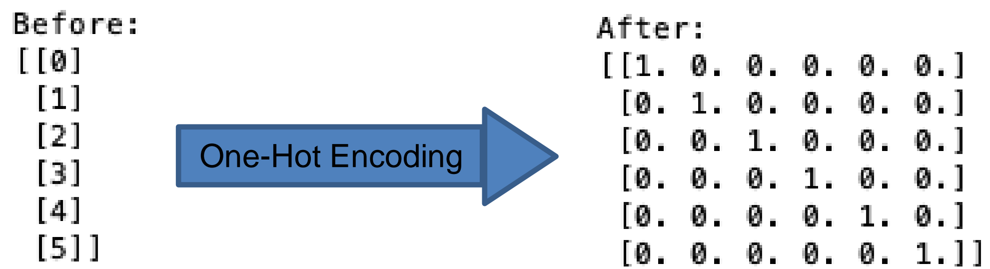

Figure 3.2: One-Hot encoding for activity labels

3.3. Proposed Model Architectures

As discussed in the Introduction, deep learning neural networks outperform traditional methods of HAR classification when dealing with large amounts of data. Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) architectures have often been used to develop HAR classification models. CNN architecture has been selected due to its high performance in previous cases [50] and its ability to handle small positional changes. Data segmentation allows CNN to exploit temporal correlations between data within one window of activity [5].

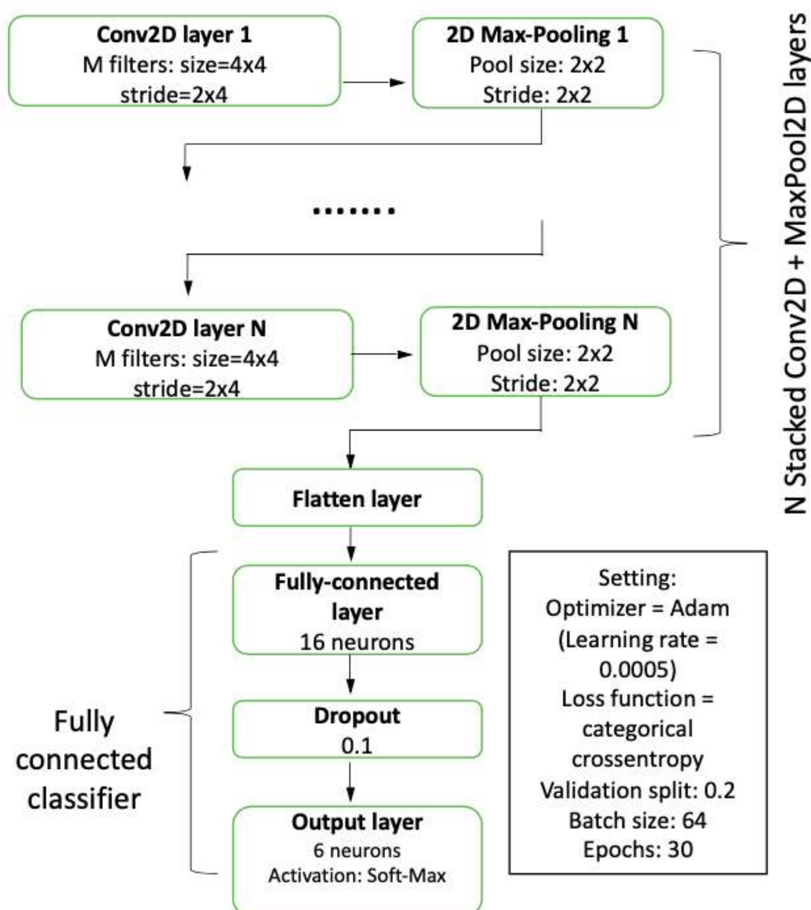

Figure 3.3: Stacked-CNN architecture and specifications

A stacked-CNN model consisting of stacked 2-D convolutional (Conv2D), 2-D max-pooling (MaxPool2D), flatten, dropout and fully-connected layers has been proposed. The general model architecture and setting specifications used in this project are illustrated in Figure 3.3. Conv2D is used instead of the 1-D convolution neural network (Conv1D), the usual selection for sensor signals, because TensorFlow Lite, which deploys models on MCUs, only supports TensorFlow 2 operational subsets; these subsets do not include Conv1D. The combination of sample windows from 9 channels are figured as an image with the dimensions 1x128x9. Each Conv2D layer is followed by a MaxPool2D to reduce the model complexity and increase model robustness. The output feature map of the last max-pooling layer feeds into the flatten layer, where feature maps are flattened into a 1-D tensor for further export. A dropout layer with dropping rate of 0.1 is inserted between the flatten and output layers to improve model generalisation.

The training setting will contribute to the performance variation of the model; however, the model size will not be affected. Therefore, some parameters in the setting were set to constant values to facilitate the training process. The optimiser algorithm chosen for the model was Adam, which needs minimal computational overhead [51]. The categorical cross-entropy loss function was selected and used for backpropagation, as it is ideal for HAR tasks, multi-class problems with a single output label. The activation function used for feature extracting-layers is ReLu, selected for its computational simplicity, while 'softmax' activation fucntion is used in the output layer to produce a vector containing the probabilities of each class. Learning rate is set to 0.0005 to prevent training loss from diverging sooner than intended.

Batch size refers to the number of samples fed into the model during each iteration, while number of epochs signifies the number of times the entire training dataset passes through the model. The number of batches is set to 64, the number of epochs is set 30.

The performance and computational budget of a CNN model is associated with its parameters, including number of layers, number of filters per layer, filter size and stride. To ease the evaluation of the relationship between performance and overhead, several architecture configurations are proposed by varying the number of stacked CNN layers N and the number of filters per layer M, while keeping filter size and stride constant. The details of these configurations are summarised in Table 3.2.

Table 3.2: CNN architecture configurations

Model Constants	Conv2D : filter size = 4 x 4; stride = 2 x 4 MaxPool2D: filter size = 2 x 2; stride = 2 x 2							
Configuration Name	**L1F16**	**L1F32**	**L2F16**	**L2F32**	**L3F16**	**L3F32**	**L4F16**	**L4F32**
No. of Layer (N)	1	1	2	2	3	3	4	4
No. of Filter per Layer (M)	16	32	16	32	16	32	16	32

3.4. Model Training, Conversion and Optimisation

The proposed model configurations have been trained on a standard personal computer with an Intel Dual Core i5 CPU (1.8 GHZ) and 4 GB memory, using TensorFlow library version 2.3.1. The sequential model and the Conv2D, MaxPool2D, Dropout, Flatten and Dense (Fully-Connected) layers are taken from the Keras library. After the building and training process, the TensorFlow models are saved as a protocol buffer with filename extension '.pb'. For edge implementation, the model must be converted to TensorFlow Lite, which employs the FlatBuffer format. FlatBuffer is more memory efficient, so a size reduction is expected after this conversion. The conversion is performed using the converter API provided by TensorFlow Lite. The converted model (hereafter referred to as the TFlite model) is then stored with the extension '.tflite' with an approximate 4x size reduction. Figure 3.4 illustrates the workflow for exporting a TensorFlow Lite model.

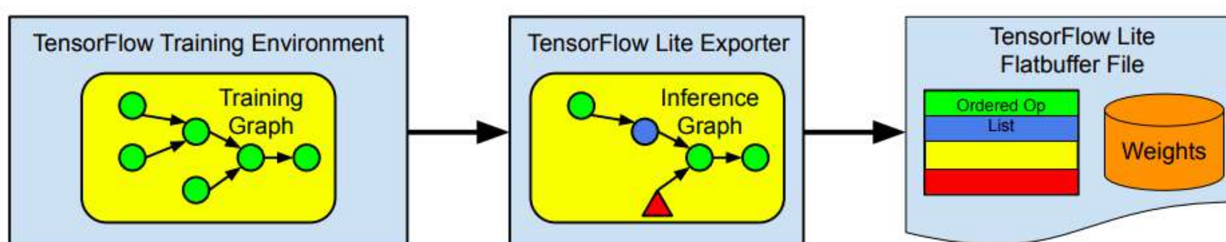

Figure 3.4: Workflow for exporting TensorFlow Lite model [35]

Before deploying the model on the target device, optimisation techniques, also known as post-training quantisation, may be used to reduce model size and latency. The model uses 32-bit floating point data representation as a default, but such high precision is often unnecessary for inference. Therefore, quantisation techniques are used to convert the model's weights and activations to fixed-point data representations (8-bit or 16-bit integers), which will facilitate computations and reduce memory footprint. Several quantisation schemes are provided by TensorFlow Lite that are suited to different scenarios: dynamic range quantisation, full integer

quantisation and float16 quantisation.

Table 3.3: Quantisation scheme specifications [32]

Scheme	Expected optimisation	Recommended for
Dynamic range	4x smaller; 2x-3x faster	Generic CPUs
Full integer	4x smaller; 3x+ faster	Generic CPUs and MCUs
Float16	2x smaller; GPU acceleration	Generic CPUs and GPUs

Dynamic range quantisation scheme act by reducing the precision of weights, from single-precision floating-point (float32) to 8-bit integer (int8). However, the weights are converted back to floating-point at inference, and the outputs are also stored using floating-point representation. Thus, dynamic range-quantised models are expected to be 4x more memory efficient, and 2x-3x faster. In contrast, full integer quantisation converts all tensors into 8-bit integer, including activations. Therefore, it is recommended for devices using int8 data representation, e.g., MCUs. Full integer should give a greater latency reduction due to its simplified computation. Lastly, float16 is the quantisation scheme recommended for GPUs, which use 16-bit data representation. The size reduction effect is halved compared to 8-bit quantisation, but float16 enables GPU acceleration. The quantisation was applied the TFlite model using the provided API. For Full integer quantisation, a representative dataset was required to calibrate the variable tensors, e.g., activations, input and output tensors. Therefore, 100 samples from the training set were selected to be the representative dataset. Note that by applying quantisation, insignificant or small accuracy loss is expected [32].

After being quantised, the TFlite model was converted into a C source file containing a char array using the 'xxd' command [32]. This C source file was then incorporated in the embedded software developed for the target device to handle the input sensor data and output inference outcomes.

3.5. Model Deployment & Measurement setup

The target device selected for model deployment is the Arduino Nano 33 BLE (hereafter referred to as Arduino Board) described in Section 2.3.1. The Arduino Board was selected for two reasons: first, Arduino Nano hosts a MCU based on the Cortex-M4 chip supported by AI acceleration kernels, which accelerate typical computations in neural networks; second, the Arduino platform provides many useful built-in functions that can be used for performance evaluation, reducing software development time.

TensorFlow Lite library version 2.1.0-Alpha is used to implement the on-device inference. An embedded software is developed under Arduino IDE using the library in order to provide an interface for the TFlite model. The software is designed to: 1) incorporate the C-source file containing the model, which is then used to instantiate an interpreter object; 2) feed data into the interpreter, run inference and return the results to the PC for further analysis.

The most intuitive way to evaluate the performance of an on-device deep learning model is to test it, using data collected from the device's embedded sensors. Unfortunately, if the sensors embedded in the MCU differ from those used to collect the training dataset, the newly collected data has a different pattern and thus cannot evaluate the model's performance. As such, an alternative approach has been conceived: rather than collecting raw data from the environment, test data from the UCI-HAR dataset is streamed directly into Arduino Board via USB, easing the measurement process. The models were measured and evaluated for four aspects: classification performance, latency, energy and memory usage.

3.5.1. Classification Performance

The classification performance of models in Table 3.2 are measured on both PC and Arduino Board, for comparison. TensorFlow provides an API to evaluate the model's overall accuracy on a given test set. The test set, consisting of 2,947 sample windows, was used for accuracy measurement. If the dataset has an unbalanced sample distribution – for example, if there are too many samples for one class and too few for others – accuracy might be compromised. It is therefore worth evaluating the model's performance using additional metrics.

Confusion matrix is a tool used to evaluate classifier models. Given the model's six activity classes, a 6x6 matrix is utilised (Figure 3.5) in which rows represent actual class (desired class) and columns represent predicted class. Every cell in the matrix is labelled; to introduce cell definitions it is necessary to reduce the 6x6 matrix into six 2x2 confusion matrices to enable a binary classification (True or False) problem. For example, Figure 3.6 shows a 2x2 confusion matrix for the class labelled 'Walking': True Positive (TP) and False Positive (FP) correspond to the number of correct and incorrect predictions of 'Walking' class; True Negative (TN) and False Negative (FN) correspond the number of correct and incorrect predictions of the remaining classes.

Confusion Matrix

Desired Class \ Predicted Class	Walking	Walking Up	Walking Down	Sitting	Standing	Lying
Walk	TP	FN FP	FN FP	FN FP	FN FP	FN FP
Walk Up	FN FP	TP	FN FP	FN FP	FN FP	FN FP
Walk Down	FN FP	FN FP	TP	FN FP	FN FP	FN FP
Sitting	FN FP	FN FP	FN FP	TP	FN FP	FN FP
Standing	FN FP	FN FP	FN FP	FN FP	TP	FN FP
Lying	FN FP	FN FP	FN FP	FN FP	FN FP	TP

Figure 3.5: Confusion matrix for performance evaluation

	Prediction: Walking	Prediction: No Walking
Actual: Walking	TP	FN
Actual: No Walking	FP	TN

Figure 3.6: Confusion matrix for class 'Walking'

With the aid of the confusion matrix it is possible to calculate key metrics for performance evaluation. These metrics include overall accuracy, precision, recall (True Positive rate) and F1 score:

1) **Overall accuracy.** Overall accuracy is the most intuitive metric for performance evaluation, which is also used during training. It is calculated by dividing the total number of correct predictions (TP + TN) to the total number of data entries (TP + TN + FP + FN):

$$Overall\ accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{11}$$

2) **Precision.** Precision indicates how much positive predictions are correct. It is calculated by dividing the number of correct positive prediction (TP) to the total number of positive prediction (TP + FP):

$$Precision = \frac{TP}{TP+FP} \tag{12}$$

3) **Recall (True Positive Rate).** Recall is concerned with how many correct positive predictions have been made, when it is actually positive. It is computed by dividing correct positive prediction (TP) to the total number of actual positive data entries (TP + FN):

$$Recall(True\ Positive\ Rate) = \frac{TP}{TP+FN} \tag{13}$$

4) **F1 score**. F1 score acts as the harmonic mean of precision and recall, and it ranges from 0 to 1. It is commonly used to measure the robustness of the model, as it analyses the results in a comprehensive manner:

$$F1\ score = \frac{2 \times Precision \times Recall}{Precision + Recall} = \frac{2TP}{2TP+FP+FN} \tag{14}$$

Overall accuracy and F1 score were calculated for each of the trained models. The confusion matrix was plotted using the "confusion_matrix" function from scikit-learn Python library.

3.5.2. Latency and Energy Consumption

Latency refers to time elapsed while processing one data sample and receiving an inference result. The Arduino built-in function "millis()" is used to evaluate latency. "Millis()" returns time passed since a program's execution in milliseconds [52]. For input data entry i, time t_{start_i} is recorded as soon as input data is fed into the interpreter, and t_{end_i} is recorded when the interpreter returns an output. Average latency in milliseconds can be computed using the following equation:

$$t_{latency} = \frac{\sum_{i=0}^{n-1} t_{end_i} - t_{start_i}}{n} \tag{15}$$

This approach is easy to implement and avoids the need for an external timer, which may introduce error due to signal transmission time. Since Arduino is single-threaded, this approach should give a reliable estimation of the execution time taken by the interpreter to run an inference.

Power consumption for running inference is estimated using the product specification nRF52840 MCU hosted by Arduino Board, and the reason is given in Appendix A2. The MCU has a PMU that automatically switches the device between different modes depending on the demand of any given moment. A constant voltage V of 3.3V is supplied to the MCU, while current consumption varies depending on the MCU's execution mode. The power consumption for each case can be calculated using the power equation (16).

$$P = V \times I \tag{16}$$

where I could be I_{system_On}, I_{system_Idle} and I_{system_Off}, the current consumption in different modes. The specifications for this project and power consumption calculated from (16) are summarised in Table 3.4.

Table 3.4: Electrical specifications of nRF52840 in three modes[53]

MODE	Description	Current Cons. (μA)	Power Cons. (mW)
System On	CPU running inference	6300	20.790
System Idle	CPU idle, DMA running	1202.35	3.968
System Off	CPU off, RAM retained	1.86	0.006

Note that the startup time for the CPU to wake up from Idle ($3\mu s$) or Off ($16.5\ \mu s$) modes are ignored, as they are insignificant to the calculation. Calculated power consumptions are used to provide an estimation of energy consumption. Energy is consumed in mWh during different phases of a complete activity cycle.

$$E_{inf} = P_{system_On} \times \frac{t_{latency}}{3600} \tag{17}$$

$$E_{idle} = P_{system_Idle} \times \frac{t_{sampling}}{3600} \tag{18}$$

$$E_{off} = P_{system_Off} \times \frac{t_{sleep}}{3600} \tag{19}$$

where E_{inf}, E_{idle} and E_{off} signify energy consumed during inference phase, idle phase and sleep phase, respectively. The length of the inference phase depends on the computation speed of the target device and the model's complexity. The Idle phase is the period of time taken before running inference, during which a full sample window is collected in "System Idle" mode, with the CPU awakened when data is ready. According to [49], the sampling rate of UCI-HAR dataset is 50 Hz and the sampling time $t_{sampling}$ for a 128-sample window is 2.56s. The sampling rate should be kept constant, as variations would change the collected signal pattern. Assuming a window overlap of δ (ranging from 0 to 1), $t_{sampling}$ can be expressed with the following equation (20):

$$t_{sampling} = 2.56 \times (1 - \delta) \tag{20}$$

Sleep phase is the period of time between each activity recognition. Sleep phase is used to reduce

energy consumption, since continuously running inference is unnecessary in real life scenarios. Taking t_{HAR} as the total time needed to perform one complete activity recognition, t_{sleep} is

$$t_{sleep} = t_{HAR} - t_{latency} - t_{sampling} \tag{21}$$

where $t_{latency}$ is calculated from (15) and t_{sleep} is specific to application settings. The overall energy consumption in mWh for a complete activity recognition cycle is the sum of the results of (17)-(19):

$$E_{HAR} = E_{inf} + E_{idle} + E_{off} \tag{22}$$

In a real-life setting, ongoing physical activity usually lasts for at least tens of seconds; therefore, a time interval may be inserted between each recognition of activity, during which the device may be turned off to extend battery life.

Table 3.5: Application settings for different scenarios:

Setting	inference per min(ipm)	$t_{HAR}(s)$	Overlap (%)	$t_{sampling}(s)$
S1	30	2	50	1.28
S2	10	6	50	1.28
S3	1	60	0	2.56

Three HAR application settings are proposed in Table 3.5. S1 assumes cases where continuous activity recognition is required to ensure prompt and accurate inference. S2 makes a trade-off between promptness and energy efficiency by inserting a few seconds of interval between inferences; this should provide satisfactory results in most cases. S3 concerns scenarios in which distant activity recognition is needed to determine user status, e.g., determining whether a user has been in one position for a certain period of time. Given a standard coin battery capacity $E_{batt} = 100\ mWh$ and using information from Tables 3.6 and 3.7, as well as results calculated from (15) and (22), expected battery run-time is calculated as follows:

$$t_{battery} = \frac{E_{battery}}{(ipm \times 60) \times E_{HAR}} \tag{23}$$

3.5.3. Memory Usage

TFlite models do not rely on dynamic memory allocation while running inference [35]. Rather, a fixed RAM area called the Tensor Arena is allocated beforehand to store input and output tensors, and any intermediate arrays. The overall memory usage of a TFlite model is the sum of the Tensor

Arena's memory and that of the model itself (ROM):

$$Overall\ Memory\ Usage = Tensor\ Arena\ Size(RAM)\ + TFlite\ Model\ Size(ROM) \quad (24)$$

The size of a model could be directly observed on a PC. However, the size of the Tensor Arena depends on both the TFlite model and the target device's architecture, which makes providing a quick and accurate estimation of memory needed for the Tensor Arena impossible. This project, therefore, sets the Tensor Arena size to 100KB, which is big enough to accommodate all evaluated models, and only takes the TFlite model size into consideration. This approach ensures that the evaluation of smaller-sized models, which need less of the Tensor Arena's memory, is not biased.

4. Result and Discussion

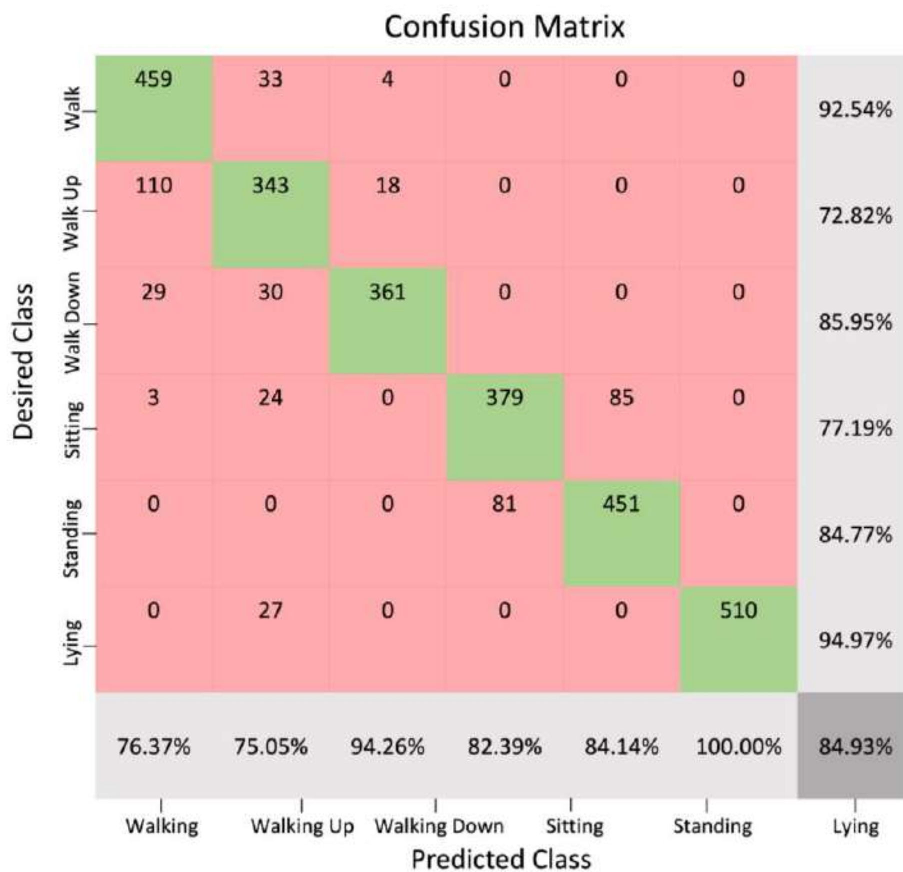

Figure 4.1: Confusion matrix for performance measurement of model L1F16

The results obtained during the implementation of models proposed in Table 3.2 of Section 3 are given in this section. The models were evaluated both on PC and on Arduino board for different aspects during different stage of implementation.

4.1. Model Evaluation on PC

4.1.1. Classification Performance of Proposed Models

After the building and training process, a set of TensorFlow models were obtained using the proposed architecture configurations in Table 3.2 of Section 3, which were measured using two metrics: overall accuracy and F1 score. Confusion matrices were plotted for each of the proposed models.

Figure 4.1 shows the 6x6 confusion matrix for model L1F16. The overall accuracy of the model was shown at the right bottom cell and it was calculated using (11). The precision and recall of each class were calculated using (12) and (13), and they were shown at the lowest row and right-most column respectively. Then, the class-wise F1 score was computed using (14), and an average F1 score of the model was recorded in Table 4.1, which summarises the measurement results and the size of TensorFlow models.

Table 4.1: Performance and memory usage of proposed models

Model	Accuracy (%)	F1 score	Model Size (KB)
L1F16	84.93	0.849	319
L1F32	86.73	0.865	509
L2F16	90.46	0.904	224
L2F32	92.26	0.923	395
L3F16	90.19	0.901	283
L3F32	91.18	0.911	581
L4F16	85.65	0.856	358
L4F32	89.79	0.900	797

As shown in Table 4.1, there was no big discrepancies between overall accuracy and the F1 score of each model, meaning that the models were robust, and the dataset was balanced. Thus, the overall accuracy could be reliably used as the metric of performance evaluation in the remaining part of project. For models with same number of layers, the model with a greater number of filters had better performance and bigger size. For example, L2F32 outperformed L2F16 by 1.8% in accuracy at the cost of an increased model size of 171KB. Generally, the models with two-layer configuration performed better and used less memory, as Figure 4.2 suggested.

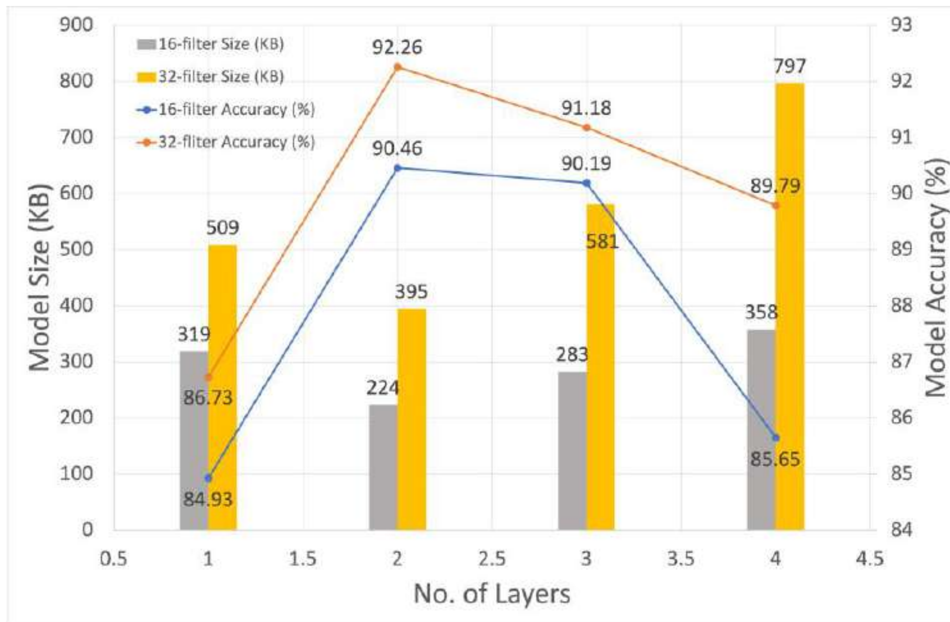

Figure 4.2: No. of Layer vs Model Size vs Model Accuracy

To evaluate further the relationship between model size and performance of two-layer models, three additional models, L2F8, L2F64 and L2F128 were trained and tested in addition to L2F16 and L2F32.

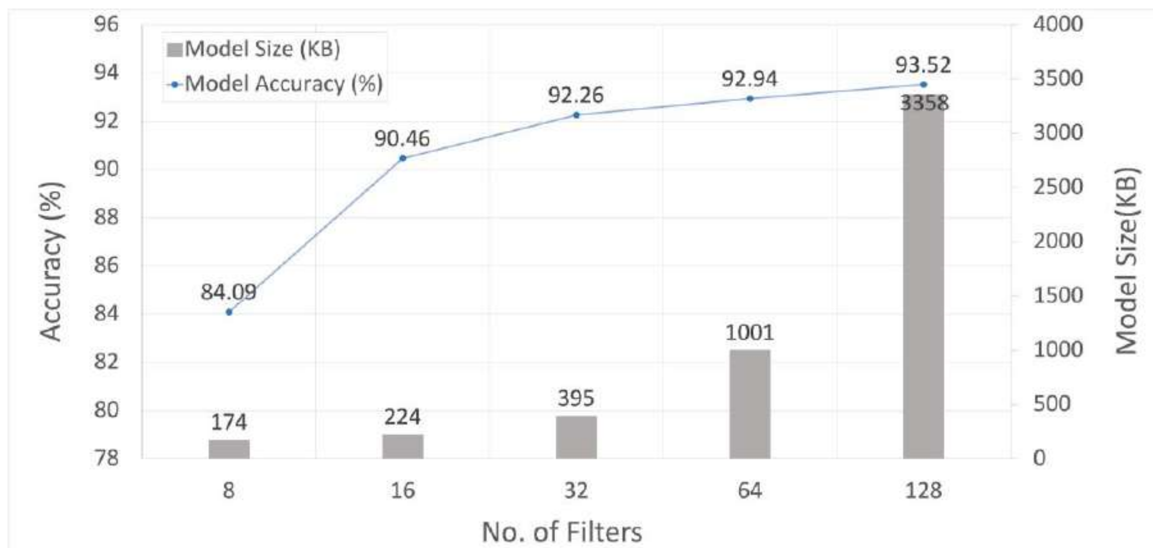

Figure 4.3: Effect of increasing number of filters per layer on model size

As shown in Figure 4.3, the increased complexity of a model enhanced model accuracy. However, as filter numbers grew exponentially, the upward trend of accuracy decreased: from 32 to 64, filters accuracy increased by 1.26% and model size increased by 606 KB, indicating that 100 KB would bring about 0.2% of accuracy growth. However, from 64 to 128 filters, 100 KB of model size growth only brought about a 0.025% accuracy increase. Since the two-layer models tested

exhibited promising performance and there was a positive correlation between their performance and size, they were brought to the next stage of implementation for further evaluation.

4.1.2. Effect of TensorFlow Lite Optimisation

TensorFlow Lite provides model optimization via conversion and quantisation. The two-layer models were converted to the '.tflite' format using the converter API, provided by TensorFlow Lite. These TFlite models were then further optimised through quantisation techniques. The effects of the conversion and quantisation were measured by testing the converted models. Model size and accuracy are provided in Tables 4.2 and 4.3, respectively.

Table 4.2: Optimisation effects on the model size

TFlite Model	Size Before conversion (KB)	No Quantisation Size (KB)	Float16 Size (KB)	Full integer Size (KB)	Dynamic range Size (KB)
L2F8	174	13	10	8	7
L2F16	224	29	18	13	12
L2F32	395	86	47	27	26
L2F64	1001	297	155	83	78
L2F128	3358	1075	554	288	279

The conversion to the '.tflite' format showed significant improvement for model size, especially for smaller models. The size of L2F8 went from 174 KB to 13 KB, more than a 13x reduction. Less size reduction was observed for more complex models like L2F64 and L2F128, which were reduced by around 3x in size. The TFlite models were further compressed with quantisation techniques: TFlite models were 1.3x to 1.9x smaller with float16 quantisation, 1.6x to 3.7x smaller with full integer quantisation and 1.9x to 3.9x smaller with dynamic range quantisation. Notably, TFlite model size was positively correlated to the effect of size reduction when applying quantisation.

Table 4.3: Optimisation effects on the model accuracy

TFlite Model	Accuracy Before conversion (%)	No Quantisation Accuracy (%)	Float16 Accuracy (%)	Full integer Accuracy (%)	Dynamic range Accuracy (%)
L2F8	84.09	84.09	84.09	82.97	83.85
L2F16	90.46	90.46	90.46	90.26	90.3
L2F32	92.26	92.26	92.26	92.26	92.3
L2F64	92.94	92.94	92.94	92.53	92.84
L2F128	93.52	93.52	93.52	93.45	93.59

According to the results in Table 4.3, model accuracy was unchanged after conversion to TFlite model and application of float16 quantisation, while insignificant accuracy loss was observed with full integer and dynamic range quantisation. After being quantised with a full integer scheme, L2F8, L2F16, L2F64 and L2F128 lost 1.12%, 0.2%, 0.41% and 0.07% accuracy, respectively. Dynamic range quantisation caused accuracy losses of 0.24%, 0.16% and 0.1% to L2F8, L2F16 and L2F64, respectively. Intriguingly, L2F32 and L2F128 gained small accuracy improvements (0.04% and 0.07%, respectively) after dynamic range quantisation.

4.2. Model Evaluation on Device

The TFlite models from the previous section, both quantised and non-quantised, were deployed on Arduino Board for an on-device performance evaluation. Three key aspects of on-device implementation were evaluated: accuracy, inference latency and energy consumption.

In terms of accuracy, the PC measurement results from the previous section should have remained valid for any target device, assuming the correct implementation. After repeated accuracy measurements on Arduino Board, the same inference results were returned. In contrast, a model's latency is dependent on the target device due to different computational speeds. Latency was measured for TFlite models before and after quantisation. Only full integer quantisation was supported for implementation on MCUs, as other quantisation schemes would cause the board to crash. Results of latency measurements are shown in Table 4.4.

Table 4.4: Latency comparison between non-quantised and full integer-quantised models

TFlite Model	No Quantisation Latency (ms)	Full integer Latency (ms)
L2F8	82	14
L2F16	204	22
L2F32	569	46
L2F64	1782	122
L2F128	N/A	393

Only L2F8 had an average latency below 100 milliseconds, while the latency of the remaining models far exceeded an acceptable range for real-time HAR applications. The effect of a full integer quantisation on latency was significant, with the models' average latency reduced by 5.9x to 14.6x+. It was not possible to evaluate the latency reduction effect of L2F128, which was too large to fit in the board without quantisation. High latency would invoke high energy consumption,

which is not ideal for embedded devices. Therefore, only full integer-quantised models were evaluated for energy consumption. Using (17), the estimated energy consumption of quantised models was computed:

Table 4.5: Estimation of energy consumed by full integer-quantised models to run one inference

TFlite Model	E_{inf} (μWh)
L2F8	0.081
L2F16	0.127
L2F32	0.266
L2F64	0.705
L2F128	2.270

E_{inf} is a critical indicator used to determine whether a specific neural network could be reasonably implemented on embedded devices. Due to the low-power characteristic, embedded devices exclude any power intensive application. Considering the three application settings proposed in Table 3.5 of Section 3, the expected battery lifespan $t_{battery}$ could be calculated using (23).

Table 4.6: Summary of estimated energy and expected battery run-time under different settings

TFlite Model	$t_{battery}$ in S1 (h)	$t_{battery}$ in S2 (h)	$t_{battery}$ in S3 (h)
L2F8	37	111	556
L2F16	36	108	547
L2F32	33	99	523
L2F64	26	78	460
L2F128	15	45	321

According to the results in Table 4.6, even L2F128, the most computationally intensive model, would be able to run continuously on board for at least 15 hours in S1 setting, which implies that the run-time of the recognition system would cover the most part of the day, without the need to recharge or replace the battery. For many real-life applications, such as elderly monitoring, the S3 setting is the most reasonable choice. The best performing model, L2F32, was estimated to have around three weeks of run-time.

5. Discussion and Conclusion

5.1. General Discussion

The proposed models were successfully implemented and measured on the target device, and the relationship between model performance and computational budget was evaluated. The state-of-the-art optimisation techniques for embedded devices were applied to the models, and the effects were assessed. These techniques exhibited promising effects on model speedup and size reduction at the cost of minimal accuracy loss. These techniques would enable the deployment of more complex and accurate network on low-power devices for fast and reliable on-device inference. Moreover, the lightest model tested in this project was only 7 KB in size, and its accuracy was 84.09%. This implies that the lightweight deep learning methods could be implemented on devices with an extremely low computational resource.

Another contribution of this project is the evaluation of energy consumption at different quantisation levels and neural network complexity was measured, using which an evaluation of battery lifespan was done to approach real-life settings. The aim of this battery lifespan evaluation was to offer guidance for future works, which might make use of the above evaluation procedures as reference. The application settings could vary greatly depending on the real-world requirements, and it was impossible to provide an exhaustive list of settings. Three representative settings were proposed and evaluated. Results from the S1 setting demonstrated that complex neural networks are now able to meet real-life requirements. On the other hand, results from the S3 setting proved the practicability of neural network-based HAR solutions.

5.2. Conclusion

In this project, a review of literature was conducted to aid the selection of software tools and hardware platform for on-device implementation. A set of CNN models of different complexities were proposed, trained and tested on PC using TensorFlow. These models achieved performance similar to previous works, with be best performing model obtained an overall accuracy of 93.52%. The models were then converted to a memory efficient format to be integrated into the embedded software developed using TensorFlow Lite. The converted models showed a size reduction of 3x to 13x, and then were deployed on Arduino board. Then the models were further optimised and measured. Overall, quantised models achieved latency reduction for up to 14.6x, and size reduction for up to 3.9x. The overall performance of the tested models was discussed in

the previous section, concluding that deep learning methods for HAR on MCU are highly practicable.

5.3. Limitations and Future Works

This project was subject to several limitations. Firstly, the models were not trained using self-collected data. As such, even if they displayed high accuracy on the chosen dataset, it is not possible to perform human activity recognition using the on-device models, as the data collected by the embedded sensor are strange for the model. This limited the possibility of testing the whole HAR system in real-life settings. Secondly, TensorFLow Lite is still in phase of development, and LSTM operators were not fully supported.

For future works, it is recommended to work on self-collected dataset so that the overall system could be evaluated in a comprehensive manner. In addition, with the advances in unsupervised learning algorithms, it might be worth to investigate what could be done using unsupervised learning.

6. References

[1] C. A. Ronao and S. B. Cho, “Human activity recognition with smartphone sensors using deep learning neural networks,” Expert Syst. Appl., vol. 59, pp. 235–244, 2016.

[2] J. Manjarres, P. Narvaez, K. Gasser, W. Percybrooks and M. Pardo, “Physical Workload Tracking Using Human Activity Recognition with Wearable Devices,” Sensors (Basel), 2019.

[3] L. Lai and N. Suda, "Enabling Deep Learning at the LoT Edge," 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Diego, CA, 2018, pp. 1-6

[4] M. Mukherjee, R. Matam, C. X. Mavromoustakis, H. Jiang, G. Mastorakis and M. Guo, "Intelligent Edge Computing: Security and Privacy Challenges," in IEEE Communications Magazine, vol. 58, no. 9, pp. 26-31, September 2020.

[5] T. Zebin, P. J. Scully, N. Peek, A. J. Casson, and K. B. Ozanyan, “Design and Implementation of a Convolutional Neural Network on an Edge Computing Smartphone for Human Activity Recognition,” IEEE Access, vol. 7, pp. 133509–133520, 2019.

[6] P. -E. Novac, A. Castagnetti, A. Russo, B. Miramond, A. Pegatoquet and F. Verdier, "Toward unsupervised Human Activity Recognition on Microcontroller Units," 2020 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, 2020, pp. 542-550

[7] N. Oukrich “Daily Human Activity Recognition in Smart Home based on Feature Selection, Neural Network and Load Signature of Appliances,” 2019.

[8] V. Menger, F. Scheepers and M. Spruit, “Comparing Deep Learning and Classical Machine Learning Approaches for Predicting Inpatient Violence Incidents from Clinical Text,” Appl. Sci. 2018, 8, 981.

[9] Z. Ghahramani, “Unsupervised Learning,” In: Bousquet O., von Luxburg U., Rätsch G. (eds) Advanced Lectures on Machine Learning. ML 2003. Lecture Notes in Computer Science, 2004, pp. 72-112.
[10] D. Lara and M. A. Labrador, “A survey on human activity recognition using wearable sensors,” IEEE Commun. Surv. Tutorials, vol. 15, no. 3, pp. 1192–1209, 2013.
[11] M. A. R. Ahad, J. K. Tan, H. S. Kim and S. Ishikawa, "Human activity recognition: Various paradigms," ICCAS, pp. 1896-1901, 2008.
[12] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” Nature, vol. 521, no. 7553, pp. 436–444, 2015.
[13] Y. Bengio, “Learning deep architectures for AI,” Found. Trends Mach. Learn., vol. 2, no. 1. 2009.
[14] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” Biol. Cybernetics 36, pp. 193–202, 1980.
[15] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proc. IEEE, vol. 86, no. 11, pp. 2278–2323, 1998.
[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” Adv Neural Inf Process Syst, vol. 25, pp.1097-1105, 2012.
[17] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," ICML, 2015.
[18] J. Mao, W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille, “Deep captioning with multimodal recurrent neural networks (m-rnn),” arXiv, 2014
[19] Y. Bengio, P. Simard and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” IEEE Trans. Neural Networks 5, pp. 157–166, 1994
[20] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural Comput. vol. 9, pp.1735–1780, 1997.
[21] T. Zebin, M. Sperrin, N. Peek, and A. J. Casson, “Human activity recognition from inertial sensor time-series using batch normalized deep LSTM recurrent networks,” Proc. Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. EMBS, pp. 1–4, 2018.
[22] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, “Edge computing: Vision and challenges,” IEEE internet of things journal, vol.3, no.5, pp. 647-646, 2016.
[23] “Edge TPU.” [Online]. Available: https://cloud.google.com/edge-tpu/
[24] “Arduino Nano 33 BLE Sense.” [Online]. Available: https://store.arduino.cc/usa/nano-33-ble-sense
[25] “Apollo3 Blue Datasheet.” [Online]. Available: https://cdn.sparkfun.com/assets/c/1/b/7/6/Apollo3_Blue_MCU_Data_Sheet_v0_10_0.pdf
[26] L. Lai, N. Suda and V. Chandra, “Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,” arXiv, 2018.(26)
[27] A. G. Howard et al., “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” arXiv , 2017.
[28] D. Lin, S. Talathi and S. Annapureddy, "Fixed point quantization of deep convolutional networks", ICML, pp. 2849-2858, 2016.
[29] R. Banner, Y. Nahshan, E. Hoffer and D. Soudry, “Post-training 4-bit quantization of convolution networks for rapid-deployment,” arXiv, 2018.

[30] S. Gupta, A. Agrawal, K. Gopalakrishnan and P. Narayanan, “Deep learning with limited numerical precision,” ICML, 2015. PMLR.

[31] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference.” Proc. IEEE CVPR, pp. 2704-2713, 2018.

[32] “Tensorflow Lite.”[Online]. Available: https://www.tensorflow.org/lite/guide

[33] PyTorch Mobile, 2021 [online] Available: https://pytorch.org/mobile/home/

[34] M. Abadi et al., “TensorFlow: Large-scale machine learning on heterogeneous distributed systems,” CoRR, vol. abs/1603.04467, pp. 1–19, 2016

[35] R. David et al., “Tensorflow lite micro: Embedded machine learning on tinyml systems,” arXiv, 2020.

[36] “Embedded Learning Library.” [Online]. Available https://microsoft.github.io/ELL/

[37] “STM32 solutions for Artificial Neural Network.” [Online]. Available: https://www.st.com/content/st_com/en/ecosystems/stm32-ann.html

[38] L. Bao, and S. S. Intille, “Activity recognition from user-annotated acceleration data,” ICPCA, 2004.

[39] U. Maurer, A. Smailagic, D. P. Siewiorek and M. Deisher, “Activity recognition and monitoring using multiple sensors on different body positions,” IWWIBSN (BSN'06). IEEE, 2006.

[40] C. Zhu and W. Sheng, “Human daily activity recognition in robot-assisted living using multi-sensor fusion,” ICRA, pp. 2154-2159, IEEE, 2009.

[41] F. J. Ordóñez and D. Roggen, “Deep convolutional and LSTM recurrent neural networks for multimodal wearable activity recognition,” Sensors (Switzerland), vol. 16, no. 1, 2016.

[42] Y. Zhang, N. Suda, L. Lai and V. Chandra, “Hello edge: Keyword spotting on microcontrollers,”, 2017.

[43] K. Chen, D. Zhang, L. Yao, B. Guo, Z. Yu and Y. Liu, “Deep learning for sensor-based human activity recognition: overview, challenges and opportunities,” arXiv, 2020.

[44] H. F. Nweke, Y. W. Teh, M. A. Al-Garadi and U. R. Alo, “Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges”. Expert Syst. Appl., v.105, pp. 233-261, 2018.

[45] J. Chen and X. Ran, “Deep Learning With Edge Computing: A Review,” Proc. IEEE vol.107, no.5, pp.1655-1674, 2019

[46] R. Chavarriaga et al., "The Opportunity challenge: A benchmark database for on-body sensor-based activity recognition," *Pattern Recognit. Lett.*, vol. 34, no. 15, pp. 2033-2042, 2009.

[47] A. Reiss and D. Stricker, "Introducing a new benchmarked dataset for activity monitoring," *Proc. 16th Int. Symp. Wearable Comput.*, pp. 108-109, 2012.

[48] G. M. Weiss, K. Yoneda and T. Hayajneh, "Smartphone and Smartwatch-Based Biometrics Using Activities of Daily Living," in *IEEE Access*, vol. 7, pp. 133190-133202, 2019.

[49] D. Anguita, A. Ghio, L. Oneto, X. Parra and J. L. Reyes-Ortiz, “A public domain dataset for human activity recognition using smartphones,” In Esann, vol. 3, p. 3.

[50] N. Y. Hammerla, S. Halloran, and T. Plötz, “Deep, convolutional, and recurrent models for human activity recognition using wearables,” IJCAI Int. Jt. Conf. Artif. Intell., vol. 2016-Janua, pp. 1533–1540, 2016.

[51] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," ICLR, 2015
[52] "Language Reference." [Online]. Available: https://www.arduino.cc/reference/en/
[53] "nRF52840 Product Specification." [Online]. Available: https://content.arduino.cc/assets/Nano_BLE_MCU-nRF52840_PS_v1.1.pdf

7. Appendices

7.1. Appendix 1 – Code

All code used during this project can be found in the following GitHub repository:

https://github.com/laiwenq/CNN_on_Arduino_Nano

The repository contains: 1) Python code used to pre-process dataset, build, train and test model on PC; 2) Scripts used to communicate with Arduino board via USB; 3) Embedded software developed for Arduino board to run inference, return inference result and latency.

7.2. Appendix 2 – Covid19 statement

Ideally, a laboratory power analyser should be used to measure the electrical quantities required for the energy evaluation. Unfortunately, due to the ongoing circumstance and export control, this was not possible. Therefore, an alternative approach was to look at the product specification. This document provided electrical specification for every possible scenario, which could be used in the estimation. This approach of might not be very accurate, but it guarantees that the estimation will not diverge too much from the true value.