Edge-Optimized Machine Learning Models for Real-Time Personalized Health Monitoring on Wearables

by

ATHAR NOOR MOHAMMAD RAFEE
20101396
ANTU DUTTA
20101282
AFSAN HAQUE
20301145
ASIF RAHMAN
20101287
ADITTA BARUA
20101023

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

Department of Computer Science and Engineering
School of Data and Sciences
Brac University
January 2024

© 2024. Brac University
All rights reserved.

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

ATHAR NOOR MOHAMMAD RAFEE

20101396

ANTU DUTTA

20101282

AFSAN HAQUE

20301145

ASIF RAHMAN

20101287

ADITTA BARUA

20101023

Approval

The thesis titled "Edge-Optimized Machine Learning Models for Real-Time Personalized Health Monitoring on Wearables" submitted by

1. ATHAR NOOR MOHAMMAD RAFEE (20101396)
2. ANTU DUTTA (20101282)
3. AFSAN HAQUE (20301145)
4. ASIF RAHMAN (20101287)
5. ADITTA BARUA (20101023)

of Fall, 2023 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science and Engineering on January 18, 2024.

Examining Committee:

Supervisor:
(Member)

Dr. Jannatun Noor

Assistant Professor
Department of Computer Science and Engineering
School of Data and Sciences
Brac University

Program Coordinator:
(Member)

Dr. Golam Rabiul Alam

Professor
Department of Computer Science and Engineering
School of Data and Sciences
Brac University

Head of Department:
(Chair)

____

Dr. Sadia Hamid Kazi

Associate Professor
Department of Computer Science and Engineering
School of Data and Sciences
Brac University

Ethics Statement

This note is a disclaimer to indicate that the research paper in question has not violated the preservation of human rights, welfare or dignity. The honesty and integrity of the research participants are incorporated in the work, which also includes highlighted citations from various reliable sources. No discrimination or disparity has been made in the data collection, and the results have not been manipulated for any prejudiced reasons. Rest assured, we hope our work reflects our respect for all intellectual property and is useful in the long term welfare of humanity.

Abstract

Personalized health monitoring, including Human Activity Recognition (HAR) and Fall Detection, is crucial for healthcare. Traditionally, most research in this field has relied on wearable sensors to collect data. The collected data is then typically sent to high-powered devices or servers for processing and analysis. However, there are some challenges with this approach. The reliance on high-powered devices can lead to delays in data processing and might not be suitable for real-time health monitoring. Additionally, the continuous transmission of data can raise privacy concerns and consume significant energy, which is not ideal for wearable devices that are often battery-powered, hence this study. Arduino UNO, based on the ATmega328P with 2 KiB SRAM, and ESP32 AI Thinker, with a dual-core Xtensa LX6 microprocessor, 320 KiB memory, and 3 MiB Flash Memory, are cost-effective and power-efficient, ideal for edge computing. In our research, we utilized the UCI HAPT and UMAFall Detection datasets for Human Activity Recognition and Fall Detection to optimize machine learning models for deployment on Arduino UNO and ESP32. On HAPT dataset, we achieved an impressive accuracy of up to 94% with a precision of 87% while on UMAFall Detection dataset, we achieved an accuracy of 81% with a precision of 77%. Notably, our trained Logistic Regression model for HAPT dataset clocked an average execution time of just 462 microseconds on ESP32 and 17634 micro seconds on Arduino UNO. Similarly, for UMAFall Detection dataset, our trained Decision Tree model clocked an average execution time of just 37 microseconds on ESP32 and 121 micro seconds on Arduino UNO.

Furthermore, we significantly optimized resource usage for both HAPT and UMAFall datasets using our trained Decision Tree model, with memory usage minimized to 0.508 KB and 0.509 KB and flash size managed efficiently to a minimum of 6.606 KB and 26.518 KB on Arduino UNO, leaving plenty of resources for developers to add other programs on top of the ML model. It significantly outdid recent studies in terms of highly resource-constrained MCUs and compute resource usage efficiency.

Keywords: Edge AI, TinyML, Embedded system, Micro-controller, Resource-constraint.

Dedication

We wholeheartedly dedicate this thesis to the unwavering pillars of our lives, our beloved parents, whose boundless support and encouragement have illuminated our academic journey, inspiring us to reach new heights. Our sincere appreciation goes out to our devoted supervisor, whose counsel and insight have been crucial to the development of this research. Finally, we dedicate this thesis to all the people who have contributed to the advancement of Human Activity Detection (HAR) and Fall Detection (FD), and to all the potential beneficiaries of this research, who may benefit from improved safety and well-being.

Acknowledgement

We owe a great debt of gratitude to Allah, the Almighty, for his bountiful help and direction as we worked on this thesis. For steady guidance, unwavering support and insightful advice of our Supervisor, Jannatun Noor, Assistant Professor (Ph.D.) we are extremely grateful. In addition, we appreciate BRAC University's support of our efforts to advance our education and undertake this study. To the Computer Science and Engineering Department, we appreciate the provision of the necessary facilities and support. Our parents and siblings have always been there to cheer us on as we pursue our education. We couldn't have gotten through the challenges we did without their unconditional support and affection.

Table of Contents

Declaration **i**

Approval **ii**

Ethics Statement **iv**

Abstract **v**

Dedication **vi**

Acknowledgment **vii**

Table of Contents **viii**

List of Figures **x**

List of Tables **xi**

Nomenclature **xii**

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Limitations . . . 3
- 1.3 Our Contribution . . . 4
- 1.4 Thesis Organization . . . 5

2 Related Works **6**

3 Background **13**
- 3.1 Logistic Regression . . . 13
  - 3.1.1 One-vs-Rest for Multi-Class Classification . . . 14
- 3.2 Decision Tree . . . 15
- 3.3 Neural Network . . . 16
- 3.4 Embedded System . . . 17
- 3.5 Hardware Acceleration . . . 18

4 Dataset **20**
- 4.1 HAPT Dataset . . . 20
  - 4.1.1 HAPT Training Dataset Balancing . . . 22
  - 4.1.2 Feature Reduction . . . 24
- 4.2 UMAFall Dataset . . . 26

5 **Research Methodology** 28
5.1 Dataset Selection and Preprocessing . . . 30
5.2 Model Selection . . . 30
5.3 Testing . . . 30
5.4 Model Performance Comparison . . . 30
5.5 Optimization And Model Conversation . . . 31
5.6 Profiling Converted Model on Emulated Raspberry Pi . . . 31
5.7 Optimized Model Deployment on MCUs . . . 32
5.8 Result Recording and Comparison . . . 33

6 **Experimental Evaluation** 34
6.1 Experimental Setup . . . 34
6.2 UCI HAPT . . . 35
6.2.1 Simulating inside Dockerized QEMU Pi Environment . . . 36
6.2.2 Deployment on ESP32 and Initial Observation . . . 37
6.2.3 Deployment on Arduino UNO and Initial Observation . . . 38
6.2.4 Optimization of Decision Tree . . . 39
6.2.5 Optimization of Logistic Regression . . . 41
6.3 UMAFall Detection . . . 42

7 **Discussion** 44
7.1 Limitations . . . 44
7.1.1 HAPT Dataset . . . 44
7.1.2 UMAFall Dataset . . . 45
7.2 Future Work . . . 45
7.2.1 HAPT Dataset . . . 45
7.2.2 UMAFall Dataset . . . 46
7.3 Comparative Analysis . . . 46
7.3.1 Used Edge Devices . . . 46
7.3.2 Comparative Study . . . 46

8 **Conclusion** 49

References 49

List of Figures

1.1 Tipical-Edge Network 2
1.2 On Device Data Processing 3

4.1 Bar Diagram of HAPT Training Dataset 21
4.2 Principal Component Analysis 22
4.3 Bar Diagram of Scaled HAPT Dataset After Scaling 23
4.4 Correlation of the Feature-Reduced HAPT Dataset for Decision Tree 25
4.5 Correlation of the Feature-Reduced HAPT Dataset for Logistic Regression 26
4.6 Histogram of UMAFall Dataset 27

5.1 Overview Of Research Methodology 29
5.2 Model Conversion, Testing and Debug 31
5.3 Steps to Run the Model on Emulated Raspberry Pi 32
5.4 Steps to Cross Compile and Deploy 33

List of Tables

4.1 Before of Data Duplication 24
4.2 After Data Duplication 24

6.1 Vanilla Trained ML Models and Neural Networks 35
6.2 Performance on Raspberry Pi 1 36
6.3 Performance on Raspberry Pi 2 B 37
6.4 Initial performance on ESP32 38
6.5 Initial performance on Arduino UNO 38
6.6 Best Performing Parameter for Decision Tree in Scaled HAPT Dataset 39
6.7 Performance of Scaled Decision Tree on Different Boards 40
6.8 Optimized Decision Tree Performance on Arduino UNO & ESP32 . . 40
6.9 Impact of L1 Regularization on Logistic Regression 41
6.10 Impact of Feature Reduction on Logistic Regression 42
6.11 Trained Models on UMA Fall Dataset 42
6.12 Best Performing Parameter for Decision Tree in UMAFall Dataset . . 43
6.13 Decision Tree Performance on ESP32 and Arduino UNO 43

7.1 Comparison of Raspberry Pi 1 A, ESP32 AI Thinker and Arduino UNO 46
7.2 Comparative Study of HAPT Dataset 47
7.3 Comparative Study of UMAFall Dataset 48

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

DT Decision Tree - A Machine Learning Model

elf Executable and Linkable Format - Used mainly in Unix Like Operating Systems

FD Fall Detection

HAR Human Activity Recognition

LR Logistic Regression - A Machine Learning Model

MCU Microcontroller Unit - A small computer on a single integrated circuit used to control other devices

ML Machine Learning

NN Neural Network

Chapter 1

Introduction

There is a rising interest in utilizing wearable devices and machine learning advancements to provide personalized health monitoring in real-time. Wearables, such as fitness trackers and smartwatches, have proliferated, offering a plethora of physiological data that can be used for health monitoring. However, the development of precise and effective machine learning models for health monitoring is significantly hampered by the constrained processing capabilities and power of edge devices. This study develops edge-optimized ML models for real-time health monitoring in order to close the gap between wearable technology, machine learning, and customized healthcare. The outcomes and conclusions of this study have implications for enhancing healthcare outcomes and giving people more control over how they actively manage their well-being.

1.1 Motivation

The potential of wearables to revolutionize healthcare is becoming more and more apparent, especially in the areas of continuous vital sign and activity tracking. Wearables are useful devices, but their limited processing power, memory, and storage make it difficult for them to process data efficiently and in real time. Privacy issues are brought up by cloud connectivity, and edge network latency and response time are crucial.

It is advantageous to deploy machine learning models for FD and HAR on edge devices such as ESP32 and Arduino UNO. Because these devices process data in real time, they can react quickly to situations like falls. They also use less power when operating, which prolongs the battery life for ongoing monitoring. By reducing reliance on cloud connectivity, on-device processing addresses privacy concerns. These small devices have reduced power consumption thanks to optimized machine learning algorithms, which makes them perfect for wearable applications.

We cannot use Raspberry Pi or similar power hungry board as it produces heat quickly and cannot be used as wearables because they are placed on crucial parts of the human body. Thus, this study emphasizes how edge computing affects wearable health monitoring devices. In order to overcome the challenges related to cloud-based processing, we hope to minimize latency, improve privacy, and minimize power consumption by enabling on-device data processing shown in Figure 1.2. With the

potential to drastically cut expenses and power consumption, wearable edge computing will become a more practical and useful option for real-time health monitoring.

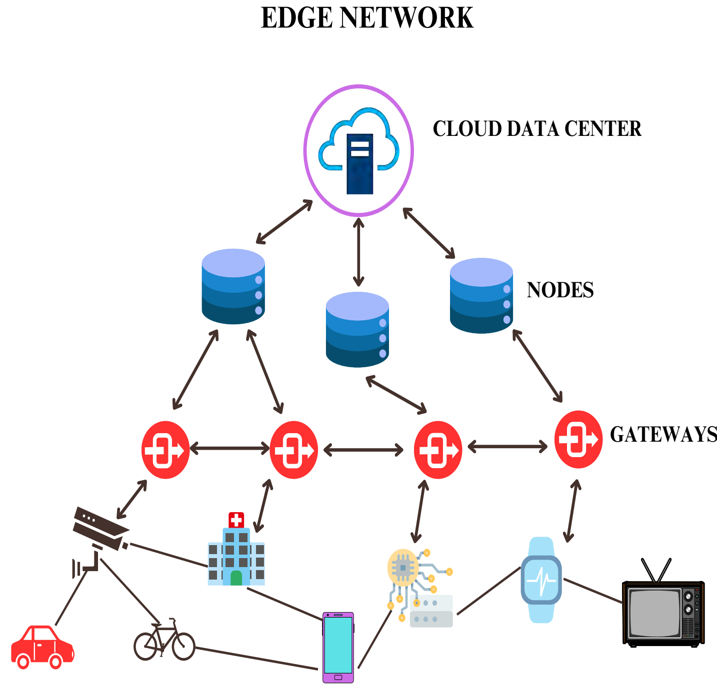

Figure 1.1: Tipical-Edge Network

As of now, these problems have been solved by various methods. Some existing solution includes collecting data with wearable sensor, the data will be sent to cloud or fog server. After processing and inferencing the data, the prediction is then sent back to the user, requiring a few network RRTs. Other solutions include placing various wifi capable sensors within rooms or zones. Fall and various HAR data will be collected by these sensors and then it will send it to the designated PC or edge server within the local network. But the problem here is that those sensors cannot detect outside of its zone and external noise can cause latency and data corruption. After data has been collected by sensor, it will be sent to the edge server where the data will be processed and ADL will be inferred. However, nowadays, there are on device MCU-based solutions. Here NN based quantized and optimized models are the common examples, for instance, TFlite [28], Pytorch [25].

The goal of the study is to minimize the drawbacks of wearables and enable real-time, personalized health monitoring like HAR, FD by optimizing machine learning models for edge devices. Arduino UNO and ESP32 can be used as they do not produce heat quickly like Raspberry Pi and these are also cheap, affordable, and power efficient. In order to create wearable ADL (*Activity of Daily Living*) technol-

ogy that is both practical and efficient, the study aims to achieve a balance between computational power and hardware ease of use.

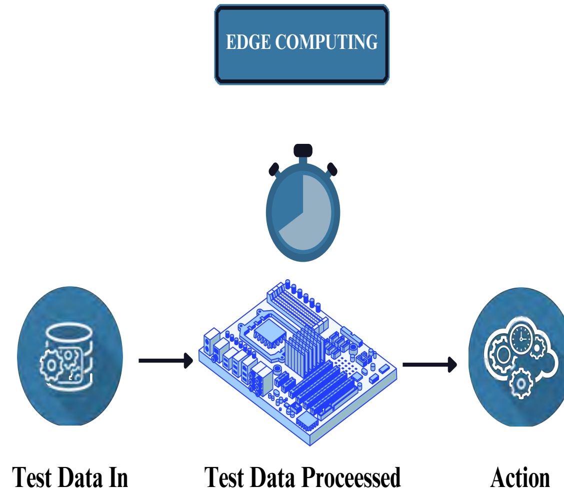

Figure 1.2: On Device Data Processing

1.2 Limitations

For real-time personalized HAR and FD using wearables, leveraging the power of edge devices can be a potent tool. However, this strategy has some drawbacks that must be taken into account:

- **Processing power limitations:** Wearables and other edge devices have limited processing capacity, which can restrict the complexity of machine learning models that can be used. This limitation may result in longer inference times or less accurate predictions.
- **Limited memory:** Edge devices also have memory restrictions, making it challenging to store and process large datasets locally. This limitation may restrict the range of analysis that can be performed on the device.
- **Data privacy issues:** Personal health information must be handled with care due to its sensitivity. Storing this data on an edge device can increase the risk of data breaches or unauthorized access to sensitive information.
- **Lack of connectivity:** Edge devices may not always have a reliable internet connection, which can limit their ability to update machine learning models or receive real-time recommendations from nearby edge servers if it follows any distributed architecture.
- **Model interpretability:** To transform a normally trained model on large datasets to an optimized format that is manageable by edge devices, it is essential to understand the overall architecture of the model. While edge-optimized

machine learning models may offer real-time predictions, their interpretability may be compromised due to the need for complexity reduction, memory optimization, and resource limitations.

- **Accuracy limitations:** Although edge-optimized machine learning models provide real-time capabilities, their processing power on highly resource-constrained devices is very limited, typically around 4 GHz in a standard x86 system, often results in lower accuracy compared to standard devices.

In spite of these limitations, the development and deployment of edge-optimized machine learning models hold great promise, and addressing these challenges can lead to significant advancements in the field of healthcare technology.

1.3 Our Contribution

In the domain of real-time personalized health monitoring on wearables, our research makes significant strides towards edge-optimized machine learning models, addressing several critical aspects that enhance efficiency and user experience. Our primary contributions can be summarized as follows:

- **Optimized for Processing Power:**
  - Designed and implemented machine learning models to reduce processing power for real-time HAR and FD monitoring on wearables.
  - Leveraged advanced algorithms and model architectures tailored for edge devices.
  - Ensured computational load minimization without compromising accuracy.
- **Reduction of Resource Consumption:**
  - Exhibited marked reduction in resource consumption for deployment on resource-constrained devices.
  - Carefully optimized model parameters and feature selection.
  - Achieved a balance between computational efficiency and predictive accuracy.
- **Memory Efficiency:**
  - Implemented techniques to significantly reduce the memory footprint of machine learning models.
  - Enhanced feasibility of deployment on wearables and integration into devices with constrained memory and storage capacities.
- **Localized Execution on Microcontrollers:**
  - Enabled the execution of models on microcontrollers.
  - Brought real-time personalized ADL and FD closer to the user.

  - Offloaded computation to the edge device itself, reducing reliance on external servers for a more responsive experience.

- **Enhanced Data Privacy:**
  - Operated entirely within the local environment of the wearable device.
  - Ensured sensitive user data is not exposed and remains secure with no transmission to external networks.

- **Uninterrupted Operation with No Connectivity Dependency:**
  - Ran models locally on the edge/wearable device itself, eliminating concerns related to connectivity issues.

In summary, our research not only advances the field of real-time personalized health monitoring on wearables but also addresses practical challenges by optimizing processing power, reducing resource consumption, improving memory efficiency, ensuring data privacy, and offering seamless operation independent of external network connectivity. These contributions collectively pave the way for a more efficient and user-centric approach to wearable health monitoring.

1.4 Thesis Organization

The study is divided into eight chapters to fully address the research questions. Chapter 1 introduces the study, explaining its significance and our objectives. We also discuss any limitations and our contribution to this topic. Chapter 2 reviews past studies, highlighting the areas covered previously and the gaps our study addresses. In Chapter 3, we explore essential background and theories for understanding our study. Chapter 4 talks about the data we used, its source, and the preparation process for analysis. Chapter 5 explains our study methods, including the design of the study, collected data, and how we analyzed it. Chapter 6 presents the experiments we conducted and the results. Chapter 7 interprets our results, compares them to existing studies, addresses the study limitations, and suggests ideas for future research. Finally, Chapter 8 concludes the study by summarizing key findings, contributions, and suggesting areas for future study.

Chapter 2

Related Works

Wearable technology has become popular as a tool for tracking one's fitness and health over the past ten years. Wearables give users access to real-time information on their physical activities and physiological responses thanks to sensors that can detect everything from heart rate to sleep quality. The interest in using this data for more sophisticated health monitoring, disease detection, and personalized recommendations has increased as the usage of wearables has become more popular. Smartwatches and fitness trackers are examples of wearable technology that collect data from sensors on the user's body, including heart rate, blood pressure, and activity level. The gathered data is sent to a nearby-edge computing device or a machine learning model running on the device. The model is intended to examine the data in real-time and spot any trends or anomalies that might point to the existence of a health issue.

Machine learning models that are edge-optimized are created to operate well on the constrained memory and processing resources of wearable technology or edge computing devices. This makes it possible to process health data in real-time without using a lot of computational power. Recent studies have been quite active on the topic of HAR and FD. The following are some important works in this field:

Several promising approaches exploit RISC-V microcontrollers for accurate and efficient on-device activity recognition in resource-constrained settings. Among them, Daghero et al.'s study proposes a collection of optimized 1D CNNs using hyper-parameter optimization, sub-byte quantization, and mixed-precision quantization for general-purpose microcontrollers. Their work also introduces adaptive inference, dynamically adjusting the neural network complexity based on the input, enabling a broader range of operating modes with efficient memory usage. They achieve real-time HAR with under 16ms latency on an ultra-low-power RISC-V MCU, surpassing previous deep learning methods in terms of memory footprint and performance [29]. This study showcase the potential of RISC-V microcontrollers for accurate and resource-efficient activity recognition on wearables and embedded devices, paving the way for new advancements in on-device intelligence for real-time applications.

Classification of anomalous gait using Machine Learning techniques and embedded sensors by Sa et al. suggests a way to classify various types of unusual walking pat-

terns using a wearable gadget with built-in sensors like accelerometer, gyroscope, and machine learning methods. The writers established a dataset comprising four different and equally distributed groups of unusual gaits and employed machine learning algorithms, including Principal Component Analysis, Support Vector Machines, and a Feedforward Neural Network. They attained accuracies of 94% and 96%, respectively. Additionally, the paper compares the computational efficiency of the models they used [26].

A unique approach to wearable sensor-based human activity identification (HAR) is presented in the study HiHAR: A Hierarchical Hybrid Deep Learning Architecture for Wearable Sensor-Based Human Activity identification. The paper tackles the problem of feature extraction from sensor signals, which in temporal and geographical contexts hold valuable information. Two robust deep neural network architectures—a convolutional neural network (CNN) and a bidirectional long short-term memory network (BiLSTM) with two stages, local and global—are the foundation of the authors' proposed hierarchical deep learning-based HAR model (HiHAR). In comparison to existing cutting-edge HAR models, the suggested hybrid model performs competitively, averaging 97.98% and 96.16% accuracy on two public datasets (UCI HAPT and MobiAct scenario), respectively. As there are 4,484,905 parameters and 1,186,235 FLOPs which is really high, it is relatively intensive for microcontroller [24].

A hybrid deep residual model for identifying transitional activities using wearable sensor data is presented in the paper A Hybrid Deep Residual Network for Efficient Transitional Activity Recognition Based on Wearable Sensors by Mekruksavanich et al. The authors contend that although a large number of learning-based methods for identifying human behavior have been developed, they frequently concentrate primarily on basic human activities and ignore transitional activities because of their brief duration and rarity. Postural transitions, however, are essential to the implementation of a HAR system and should not be disregarded. The suggested approach combines a Bidirectional Gated Recurrent Unit (BiGRU) with hybrid Squeeze-and-Excitation (SE) residual blocks to improve the ResNet model and hierarchically extract deep spatio-temporal information, as well as effectively discriminate transitional activities. The suggested hybrid technique achieved 98.03% and 98.92% classification accuracies for the HAPT and MobiAct v2.0 datasets, respectively, according to the results of experiments conducted on two available benchmark datasets (HAPT and MobiAct v2.0). Furthermore, the results demonstrate that, in terms of overall accuracy, the suggested strategy outperforms the cutting-edge techniques like HiHAR. The results show that transitional activity recognition can be enhanced more effectively with the SE module. This work advances the field of Human Activity Recognition (HAR) and its potential applications in home automation systems, geriatric Fall Detection, sports performance, medical rehabilitation, and misbehavior identification, among other disciplines [34].

An innovative method for Human Activity Recognition (HAR) on ultra-low power devices, such as smartwatches, is presented in the publication Two-stage Human Activity Recognition on Microcontrollers with Decision Trees and CNNs by Daghero et al. This work is the continuation of the author's previous work on MCUs. The

authors address the high energy consumption problem associated with Deep Learning (DL), which is a serious barrier for battery-operated and resource-constrained devices, by proposing a hierarchical architecture that integrates traditional Machine Learning (ML) models with DL. Two distinct subtasks are served by the architecture's cascading decision tree (DT) and one-dimensional convolutional neural network (1D CNN). While the CNN handles more complicated actions, the DT simply categorizes the simplest ones. Using studies on a cutting-edge dataset and a single-core RISC-V MCU, the authors show that this technique can save up to 67.7% of the energy used in comparison to a standalone DL architecture at iso-accuracy. Additionally, the two-stage approach lowers the overall memory usage or adds a minor memory burden (up to 200 B) [30].

AUTO-HAR: An adaptive human activity recognition framework using an automated CNN architecture design by Ismail et al. presents the AUTO-HAR framework, which uses an automated Convolutional Neural Network (CNN) architectural design, is a significant improvement in Human Activity Recognition (HAR). Conventional techniques for creating CNN structures can be laborious and error-prone. Neural Architecture Search (NAS), which enables the autonomous design and optimization of network topologies, helps AUTO-HAR overcome these problems by getting around the constraints imposed by human experience and traditional thought patterns. To enhance the NAS procedure, the framework uses evolutionary techniques, more especially Genetic techniques (GA). These algorithms can perform well on black-box optimization tasks without requiring gradient knowledge or explicit mathematical formulations. By introducing a new encoding schema structure and a large search space with a variety of operations, AUTO-HAR makes it possible to find the best designs for HAR tasks efficiently. Because AUTO-HAR does not impose restrictions on the maximum length of the task architecture, its search space also permits a respectable level of depth. We have evaluated the framework's performance on three datasets: Opportunity, DAPHNET, and UCI-HAR. The outcomes show that AUTO-HAR is capable of effectively identifying human actions with 98.5% ($\pm$1.1), 98.3%, and 98.7% average accuracy, respectively. The automated CNN architectural design used in this framework's HAR approach shows potential for the creation of more accurate and efficient recognition systems across a range of applications [40].

The use of deep learning (DL) algorithms in the field of human activity recognition (HAR) is examined in the study A Study on the Application of TensorFlow Compression Techniques to Human Activity Recognition by Contoli et al. The authors examine how convolutional neural networks (CNNs), long short-term memory (LSTM) networks, and a hybrid CNN and LSTM can be compressed using TensorFlow Lite basic conversion, dynamic, and complete integer quantization approaches. Two use case scenarios are examined in the study: standalone compression mode and cascade compression. The viability of implementing deep networks on an ESP32 device is discussed in the study, along with the effects of TensorFlow compression techniques on inference latency, classification accuracy, and energy consumption. It was not able to do the performance characterization in the cascade instance. Dynamic quantization is advised in the standalone scenario due to its little accuracy loss. Both dynamic and full integer quantization offer significant energy savings

over the uncompressed versions in terms of power efficiency: 31% to 37% for CNN networks and up to 45% for LSTM networks. Both dynamic and complete integer quantization perform comparably about inference latency [36].

Moving forward with FD, the study titled XAI-Fall: Explainable AI for Fall Detection on Wearable Devices Using Sequence Models and XAI Techniques presents a novel approach to FD, which is crucial for the safety of older people. TConventional FD techniques, like installing sensors on a room's walls or utilizing a single triaxial accelerometer, have a number of disadvantages. They are either mounted on a wall that is unable to detect a person falling outside its range, or they rely on a single sensor, which is inadequate for detecting falls. The authors suggest a reliable technique for identifying falls utilizing three separate sensors positioned at five different points on the subject's body in order to get around these drawbacks. In order to train the models for FD, sensor readings are obtained using the UMAFall dataset. The authors added the XAI method known as LIME to the system in order to enhance the interpretability of the model. This method aids in the explanation of the model's results. The models achieved an accuracy of 93.5%, 93.5%, 97.2%, 94.6%, and 93.1% on each of the five sensor models, and 92.54% is the overall accuracy achieved by the majority voting classifier. To improve the model's interpretability, the authors incorporated the XAI technique called LIME into the system. This technique helps explain the model's outputs [32].

The study titled Fall Detection in the Elderly With Android Mobile IoT Devices Using Nodemcu And Accelerometer Sensors by Sudirman et al. focuses on the development of an IoT-based Fall Detection system using NodeMCU ESP8266 and an Accelerometer MPU6050. Because it can alert a family member or concerned party, whenever it notices a fall, the system is especially helpful for senior citizens, as it lowers the possibility that they won't receive emergency care right away. The MPU6050 sensor module, which has an accelerometer and gyroscope, is used by the Fall Detection device. The accelerometer gives information about the angular parameter, such as the three-axis data, and the gyroscope is used to determine orientation. The system compares the acceleration magnitude with a threshold value in order to identify a fall. The device notifies the concerned party via SMS if it detects a fall. To send the SMS, the NodeMCU is utilized as a microcontroller and Wi-Fi module to establish a connection with the web-based IFTTT (If This Then That) service. The system can monitor thousands of senior citizens, detect falls, and alert caregivers. It is made to be scalable. The requirements to enable large-scale system operations are revealed by the scalability tests [27].

Using a wearable sensor, a novel energy-efficient algorithm for categorizing various fall types is presented in the study An Energy-Efficient Algorithm for Classification of Fall Types Using a Wearable Sensor by Kwon et al. The study tackles the problem of promptly delivering medical care to minimize fall-related injuries. The authors suggest Temporal Signal Angle Measurement (TSAM), a cutting-edge approach that can distinguish between the five most typical fall types. The first wearable system to be designed had an inertial measurement unit sensor. Next, the various fall kinds at different sample frequencies were classified using the TSAM. Three distinct machine learning algorithms were compared with the results. Both the machine learning al-

gorithms and the TSAM's overall performance were comparable. Nevertheless, at frequencies between 10 and 20 Hz, the TSAM performed better than the machine learning methods. The accuracy of the TSAM varied from 93.3% to 91.8%12 as the sample frequency decreased from 200 to 10Hz. For the same frequency range, the sensitivity and specificity range from 93.3% to 91.8% and 98.3% to 97.9%, respectively. The researchers came to the conclusion that their algorithm can be used to categorize various fall types at low sample rates using energy-efficient wearable devices. By giving the required information, this technology helps speed up medical assistance in emergency circumstances brought on by falls [16].

A creative method for FD with wearable technology is presented in the paper Fall detection on a wearable microcontroller using machine learning algorithms by Oden et al. The Bosnai machine learning algorithm is suggested by the authors as a way for embedded-edge devices to identify falls. The Arduino-based prototype can be incorporated into materials for belts, clothing, and other accessories. On the gadget, the FD is done offline. The researchers trained a tree-based machine learning model using information from publicly available datasets of movement and fall occurrences. They assessed various iterations of preprocessed parameter combinations as learning algorithm input characteristics. The microcontroller receives the learnt model and uses it to identify sensor data offline in real time. Due to the microcontroller's severe memory and processing capability limitations, learning features are restricted to a small set of features. Preprocessing the raw acceleration data and choosing the appropriate features for training and inference is crucial because of this. The authors discovered that using absolute acceleration and variance as characteristics, along with a sampling rate of 20 Hz and a recording window of 3s1, yields the best results (about 94.2% accuracy). The strongest system against outside interference is this one [18].

Anita Ramachandran et al.'s paper, Machine Learning-Based Techniques for Fall Detection in Geriatric Healthcare Systems, describes a machine learning and Internet of Things (IoT)-based FD system for senior healthcare. The system is a component of a larger class of systems known as Ambient Assisted Living Systems (AALS), which has seen a lot of recent academic activity. The suggested method makes use of the subject's biological and physiological profile as well as a variety of wearable sensor node parameter data. The subject's fall risk category is established based on their profile. The wearable sensor node readings from public datasets for FD were used by the authors in their machine-learning experiments. The subject's risk categorization was then fed back into the algorithms for retraining, and the analysis's outcomes were shown. The studies aimed to determine how a subject's risk classification affected the precision of FD. The system is a component of an all-encompassing, still-under-development geriatric healthcare system that includes cloud-hosted application servers, coordinator nodes, wearable sensor nodes, and an indoor localization framework. The goal of this research is to enhance the safety and quality of life for the aged population, making a valuable contribution to the expanding field of geriatric healthcare systems [14].

An automatic Fall Detection System (FDS) named CareFall is presented in the publication CareFall: Automatic Fall Detection through Wearable Devices and AI

Methods by Garcia et al. The system is intended to tackle the increasing problem of falls in the elderly population, which is a major worldwide public health issue. CareFall makes use of artificial intelligence (AI) techniques and wearable technology, particularly smartwatches. For FD, it takes into account the time signals from the accelerometer and gyroscope that were taken from the smartwatch. The research investigates threshold-based and machine-learning-based methods for feature extraction and categorization. The machine learning-based strategy, which integrates accelerometer and gyroscope data, performs better than the threshold-based approach in terms of accuracy, sensitivity, and specificity, according to experimental results on two public datasets. This study aids in the development of clever and practical strategies to lessen the detrimental effects of falls on the elderly population [43].

Artificial Intelligence (AI)-based algorithms for the recognition of Activities of Daily Living (ADL) are covered in the study Inertial Data-Based AI Approaches for ADL and Fall Recognition by Martins et al. The authors point out that data from wearable sensors, especially those at the lower trunk, can be processed to achieve ADL recognition. They do point out that these algorithms only identify a small number of ADL, seldom concentrating on transitional activities and failing to address falls. In order to improve the resilience of several ADL recognition methods and get over these restrictions, the authors combined nine public and private datasets. In order to compare several ADL Machine Learning (ML)-based classifiers, they created an AI-based framework. The investigation indicated that the best performance was obtained using the K-NN classifier with the first 85 features rated by Relief-F (98.22% accuracy). However, the Ensemble Learning classifier with the first 65 features ranked by Principal Component Analysis (PCA) provided 96.53% total accuracy while keeping a reduced classification time per window (0.039 ms), demonstrating a better potential for its implementation in real-time applications in the future. Although the results of testing Deep Learning algorithms were not as good as those of the previous procedure, they did show potential (overall accuracy of 92.55% for Bidirectional Long Short-Term Memory (LSTM) Neural Network), suggesting that they might be a viable option in the future. Compared to previous analyses, this paper addresses a greater number of ADL, identifying 20 classes, including falls (16 ADL and four types of falls) and transitional activities (such as sit-to-stand, stand-to-stand, lying-to-stand, pick objects from the ground, and bend and turn) [33].

With an emphasis on senior fall prevention, the study FallNeXt: A Deep Residual Model based on Multi-Branch Aggregation for Sensor-based Fall Detection by Mekruksavanich et al. introduces a unique deep learning network for fall detection. The authors draw attention to the growing demand for FD and prevention technology brought on by an aging population. Adults and the elderly are particularly vulnerable to falls' serious health risks and injuries. Early detection and intervention can stop additional damage. FallNeXt, the suggested remedy, is a sensor-based system made to protect people's privacy while improving FD capabilities. The system improves FD capability by utilizing a deep residual network that uses multi-branch aggregation. Three benchmark datasets for sensor-based FD—UpFall, SisFall, and UMAFall were used to assess FallNeXt's efficacy. Based on the experimental results, FallNeXt performed better on these datasets than benchmark deep learning models,

with the greatest overall accuracy and F1-score of 96.16% and 98.12%, respectively. The FallNeXt model's prediction capability is one of its main advantages since it makes it appropriate for real-world applications. This study makes a substantial contribution to the advancement of technologies meant to enhance senior citizens' quality of life [35].

Chapter 3

Background

The creation and refinement of machine learning models specifically designed for wearables' real-time personalized health (HAR and FD) monitoring forms the basis of this study. The work focuses on compilation and optimization of ML models to train and implement these models on microcontrollers with limited resources, namely the Arduino UNO and ESP32. The main goal is to demonstrate the real-world application of edge-optimized models, with a focus on Fall Detection (FD) and Human Activity Recognition (HAR). It is critical to grasp the fundamentals of Neural Networks, Embedded Systems, Hardware Acceleration, Decision Tree algorithms, and Logistic Regression in order to comprehend what we're doing. By illustrating how these models can be effectively deployed on microcontrollers and overcoming the difficulties brought on by resource constraints in the context of real-time health monitoring on wearable devices, this work advances the field.

3.1 Logistic Regression

When a dependent variable is binary or dichotomous—that is, only conceivable values, such as Yes/No, True/False, Success/Failure, etc.—we employ the statistical technique of logistic regression [2]. It is an effective statistical method applied in many different domains, such as the social sciences, marketing, finance, and medicine. It facilitates the modeling of the link between one or more independent factors (such as patient demographics, economic indicators, and advertising techniques) and a binary dependent variable (such as the existence or absence of a disease, customer retention).

Based on the independent variables, logistic regression calculates the likelihood of a particular event occurring (such as the likelihood of contracting a disease). This crucial component makes the likelihood linear on a log scale by converting it into a logit score. This makes it possible for us to estimate the correlation between the logit score and the independent variables using linear regression techniques. The model determines the coefficients that best fit the data using maximum likelihood estimation, showing how each independent variable affects the logit score and, in turn, the event probability [3].

The logistic regression equation models the probability of a binary outcome and is commonly used in statistics and machine learning. The logistic regression equation

is expressed as follows:

$$P(Y=1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_k X_k)}} \tag{3.1}$$

Here:
$P(Y = 1)$ is the probability of the event Y occurring (where Y is the dependent variable with two possible outcomes, often coded as 0 and 1).
e is the base of the natural logarithm.
β_0 is the intercept term.

$$\beta_1, \beta_2, \ldots, \beta_k \text{ are the coefficients associated with the independent variables}$$
$$X_1, X_2, \ldots, X_k, \text{ respectively.}$$

The logistic function, $\frac{1}{1+e^{-z}}$, transforms the linear combination of the coefficients and independent variables into a value between 0 and 1, representing probabilities [9].

3.1.1 One-vs-Rest for Multi-Class Classification

Multi-class classification is not supported by the Logistic Regression model by default. Since they were created for binary classification, algorithms like the Perceptron, Logistic Regression, and Support Vector Machines are not naturally capable of handling classification jobs involving more than two classes. Dividing the multi-class classification dataset into several binary classification datasets and fitting a binary classification model on each is one method of applying binary classification algorithms for multi-classification problems.

The One-vs-Rest and One-vs-One tactics are two distinct instances of this method. Binary classification methods are used in the One-vs-Rest (OvR) also referred to as One-vs-All (OvA) strategies for multi-class classification. The multi-class dataset is divided into several binary classification issues. On the contrary, One-vs-One (OvO) is an additional multi-class classification technique [7].

For multi-class classification issues, the One-vs-Rest (sometimes called One-vs-All or OvA) technique is applied as follows:

- **Multi-Class Classification Problem:** This type of classification involves more than two classes. For instance, categorizing fruits as banana, orange, or apple.
- **Strategy One-vs-Rest (OvR):** The OvR approach divides the multi-class classification issue into several binary classification issues. The next step in each binary classification problem is to differentiate between one class and the others. For example, the issues with the fruit classification example would be:

  Issue 1: Banana vs [Apple, Orange]
  Issue 2: Orange vs [Apple, Banana]
  Issue 3: Apple vs [Orange, Banana]

The above discussed method is the default multiclass classification approach in SK Learn Logistic Regression model [54].

This method is a heuristic method, which means it is a workable solution that may not be ideal or flawless but is nonetheless adequate to accomplish a task right now. Its primary drawback is that, in datasets with many classes, it may be computationally costly to train a single classifier for each class [22].

This method's primary drawback is the sheer number of models that must be made. The creation of n models is required for a multi-class problem with n classes, which could cause the process to lag. That is, nevertheless, highly helpful when dealing with datasets that have few classes, and we wish to apply a model such as SVM or Logistic Regression.

We would train K distinct binary Logistic Regression models using the One-vs-Rest (OvR) technique in a multi-class classification issue with K classes. We treat class k as the positive class and all other classes as the negative class for each model k (where k is from 1 to K). In 3.1, for one-vs-all logistic regression in a multi-class scenario, we train k models (where k is the number of classes). The generalized equation for each model is the same as binary logistic regression:

$$P(Y = i) = \frac{1}{1 + e^{-(\beta_0^{(i)} + \beta_1^{(i)} X_1 + \beta_2^{(i)} X_2 + \ldots + \beta_n^{(i)} X_n)}} \tag{3.2}$$

where:

$P(Y = i)$ is the probability that the instance belongs to class i.

$\beta_0^{(i)}, \beta_1^{(i)}, \ldots, \beta_n^{(i)}$ are the coefficients for class i.

During prediction, compute the probability for each class using the corresponding set of coefficients, and the class with the highest probability is the predicted class [7].

3.2 Decision Tree

The decision tree is a non-parametric supervised learning technique used in both regression and classification applications. Its tree structure is hierarchical, with internal, leaf, branch, and root nodes [1]. Decision trees are strong and adaptable instruments utilized in many domains, including finance, healthcare, data mining, machine learning, and more.

A decision tree is a model that resembles a tree and asks a series of questions to simulate the decision-making process. A question concerning a feature (attribute) of the data is represented by each internal node in the tree, and the potential solutions are represented by each branch. The ultimate classifications or predictions are represented by the leaf nodes (terminal nodes) at the end of the branches.

Numerous algorithms have been devised to build decision trees, the most prominent ones being. ID3 (Iterative Dichotomiser 3) which was developed by Ross Quinlan, aims to maximize information gain while creating trees. ID3 was extended with

C4.5, which brought additional features such as the ability to handle missing data and handle both continuous and categorical characteristics. Leo Breiman created the CART (Classification and Regression Trees) algorithm, a flexible method that may be used for both classification and regression applications. It uses mean squared error for regression and the Gini impurity for classification. Scalable Parallelizable Induction of Decision Trees, or SPRINT, is a decision tree algorithm that was created to overcome the memory and scalability issues of previous techniques such as SLIQ. The Quest Supervised Learning method (SLIQ) is a decision tree method that is optimized for handling huge datasets with efficiency. Its forerunner, the SPRINT algorithm, substantially enhances memory consumption and scalability [10].

Each node in a decision tree makes a series of binary decisions as part of the decision-making process. Based on the feature tests, it moves through the tree from the root to a leaf given an input instance until it reaches a final conclusion. A splitting criterion, usually geared at maximizing information gain or decreasing impurity, determines the decision at each node. Whereas impurity reflects the disorder present in a dataset, information gain quantifies the decrease in uncertainty regarding the output labels following a split.

The mathematical equation for decision trees is used to calculate the entropy of a dataset. Entropy is a measure of the amount of uncertainty in a dataset. The equation calculates the entropy of a dataset by summing over all the data points in the dataset and calculating the difference between the actual label and the predicted label for each data point. The equation is :

$$\sum_{i=1}^{m} [-y_i log_2(\hat{y}_i) - (1 - y_i) log_2(1 - \hat{y}_i)] \tag{3.3}$$

where y_i is the actual label of the i-th data point, $\hat{y}_i$ is the predicted label of the i-th data point, and m is the total number of data points in the dataset [20]. The equation is used to calculate the entropy of the dataset, which is then used to construct the decision tree.

Decision trees are a great option for this study because of their high interpretability, which makes it simple to comprehend and describe the model's decision-making process. Decision trees do not require intricate mathematical computations because they work by branching and comparing values. Because of this, they are ideal for edge devices with constrained processing capabilities, such as wearables. Moreover, decision trees are simple to understand and offer a clear picture of the reasoning behind a given choice. In health monitoring applications, where comprehending the logic underlying a forecast might be just as significant as the prediction itself, this transparency is essential.

3.3 Neural Network

An extensive definition of a neural network is given by Goodfellow et al. in their book Deep Learning. As per the book, a neural network is a computational model

that draws inspiration from the functioning of biological neural networks found in the human brain. The artificial neuron, which is intended to resemble a biological neuron in activity, is the fundamental unit of a neural network [11].

The different architectures of neural networks—Feedforward Neural Network (FNN), Radial Basis Function Neural Network, Kohonen Self-Organizing Neural Network, Multilayer Perceptron (MLP), Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), and Modular Neural Network—are each intended to handle particular types of tasks [17].

A neural network or NN is made up of several layers. Raw data is received by the input layer. Compute using weighted connections between nodes in hidden layers. Output Layer: Produces the data that has been processed. Both the Forward Propagation and Backpropagation (Training) layers carry information. The first hidden layer's nodes are activated by input data. Every node determines its output by computing the weighted sum of its inputs and applying an activation function. Until the output layer is reached, the procedure is repeated using this output as the input for the subsequent layer. An error signal is computed if the output deviates from the intended result. The weights of the connections between nodes are then modified to reduce further mistakes as a result of this error being backpropagated throughout the network. An equation for a neural network is:

$$\hat{y} = f(Wx + b) \tag{3.4}$$

where $\hat{y}$ is the predicted output, x is the input, W is the weight matrix, b is the bias vector, and f is the activation function. The activation function is used to introduce non-linearity into the neural network and is typically a sigmoid or ReLU function. The weight matrix and bias vector are learned during the training process and are used to transform the input into the output [21].

The main reason for not using neural networks in this study is their computational complexity. Multiplication of multidimensional vectors/matrices is a step that neural networks, and deep learning models in particular, need a lot of computing time and effort to do. They are therefore inappropriate for real-time applications due to the potential for increased latency. Furthermore, implementing large neural network models is challenging due to wearable devices' resource limitations, which include low processor and memory speeds. Therefore, less computationally expensive alternative machine learning models are favored to achieve faster execution and real-time answers on wearables. The objective of optimizing machine learning models for edge devices is in line with this strategy.

3.4 Embedded System

Embedded system is a small computer system that is integrated into a larger device or machine to perform a specific function [4]. An embedded system is a type of specialized computer system that is integrated into a larger machine or product to carry out a particular set of tasks. Embedded systems, in contrast to general-

purpose computers, are designed primarily for non-user interaction and usually have constrained memory, processing, and storage capacities.

In the world of devices, embedded systems are often overlooked, but they are just as powerful as their PC cousins. Embedded systems are pervasive in our everyday lives, controlling everything from the simple thermostats in our homes to the complex avionics in modern aircraft [5]. They are made for specialized jobs like operating a washing machine or sensing a phone touch, where they must respond quickly despite having little memory and brainpower. These real-time experts balance energy consumption, particularly in battery-operated devices, with the precision and dependability of medical equipment on small budgets.

A microcontroller or microprocessor, memory (RAM and ROM), input/output interfaces, and a variety of peripherals are often found in embedded system architectures. Program instructions and data are stored in memory, and the microcontroller serves as the processing unit. Peripherals offer extra functionality like timers and communication interfaces, while input/output interfaces facilitate communication with external devices. This architecture is made to be small, effective, and customized to meet the unique needs of embedded system architecture.Wearables that provide real-time, individualized health monitoring depend on embedded systems. These specialized systems, created for particular tasks, are excellent at responding instantly, which is essential for quickly analyzing health data. Embedded systems are well-suited to the limited resources and compact architecture of battery-operated wearables, guaranteeing peak performance and effective edge-optimized machine learning model implementation.

The development of edge-optimized machine learning models for wearable device-based real-time personalized health monitoring is the focus of this study. The platform of choice for implementing these models is an embedded system based on microcontrollers. The selection process is based on the characteristics of the microcontroller, which include its small size, low resource consumption, high power efficiency, and affordability. Because of these qualities, microcontrollers are the perfect choice for wearable technology, where small size and extended operation without regular recharging are critical requirements. By reducing dependency on external servers for processing, the embedded system method further improves power efficiency and increases user convenience and operating lifespans. The study essentially investigates creative ways to overcome the intrinsic constraints of microcontroller-based embedded systems and maximize their potential to progress the field of wearable real-time individualized health monitoring.

3.5 Hardware Acceleration

Hardware acceleration is the process of using specialized hardware to carry out particular tasks faster than CPU-based general-purpose software. This can lead to enhanced efficiency, decreased power usage, and quicker execution of specific tasks, like machine learning algorithms, graphics processing, and cryptography. Hardware acceleration is commonly used in applications where speed and efficiency are critical, such as in gaming, scientific computing, and artificial intelligence [23]. In machine

learning, hardware acceleration refers to the use of specialized hardware to accelerate the execution of machine learning algorithms [13]. Examples of this type of hardware include dedicated AI accelerators and GPUs (Graphics Processing Units).

Hardware acceleration is used to increase task-specific performance beyond what a CPU's general-purpose software can accomplish. This approach not only reduces power consumption but also significantly improves task performance by utilizing specialized hardware such as AI accelerators or GPUs. This is especially helpful in applications where efficiency and speed are crucial, like in artificial intelligence, scientific computing, gaming, and machine learning for tasks like inference and training. Hardware acceleration ensures quicker function execution and maximizes energy consumption, both of which improve system performance overall.

The goal of this study is to create machine learning models that are optimized for edge devices without using hardware acceleration. When compared to existing approaches that employ expensive and parallel hardware-dependent matrix multiplication, our strategy leverages edge optimization to enable wearables for real-time individualized health monitoring. By avoiding the constraints imposed by hardware acceleration, our suggested models aim to offer effective and user-friendly solutions that work with a variety of boards. In the context of wearable health monitoring, this study tackles the need for scalable and affordable machine-learning algorithms that guarantee greater accessibility and enhanced performance.

Chapter 4

Dataset

We have decided to use the Updated version of *Human Activity Recognition Using Smartphones Data Set* [8] called HAPT and *UMAFall: Fall Detection Dataset* [12] for our research. The HAPT dataset (Human Activities and Postural Transitions) is a great option for investigating how resource-constrained edge devices can process and classify a wide range of sensor data efficiently using edge-optimized models. Furthermore, we can look into how FD systems on edge devices with constrained resources can be made more accurate and efficient thanks to the FD specific UMAFall dataset. We chose these datasets in order to address issues related to real-time processing on edge devices and to provide insights into the design and optimization of machine learning models for edge deployment.

4.1 HAPT Dataset

In the HAPT dataset recordings of 30 participants wearing smartphone strapped to their waist that have internal sensors were used to create the database. Sorting activities into one of the 12 done activities is the goal. The updated version disregarded the preprocessed signal that is present in the earlier version rather gives access to sensors with raw inertial signals. Moreover, postural transitions were not present in the previous datasets. But the newer version has it by updating the activity levels. For using the dataset independently, the database is split into two pieces.

- Inertial sensor data:
  -*All participant-involved trials' raw triaxial signals from the accelerometer and gyroscope.*
  -*The initials of each activity that was performed.*

- Window activity records. Each one is made up of:
  -*A 561-feature vector containing time- and frequency-dependent variables.*
  -*A description regarding the associated action.*
  -*A unique identifier for the experimenter's subject.*

For the trial, a total of 30 individuals between the ages of 19 and 48 took part. Six main tasks were done by them that included (lying, sitting, standing) that are static

postures and (walking downstairs, walking upstairs, walking). During the test, each volunteer had a smartphone around their waists. The integrated accelerometer and gyroscope of the devices helped to record 3-axial angular velocity that has a constant rate of 50Hz [42]. For this study, when using this dataset, we kept the original train and test split provided by the authors of the dataset [42]. Besides, in the later section 4.1.1 we balanced the training dataset and left the test dataset unchanged for comparison with other studies. The below Figure 4.1 shows the initial class distribution of labels against their samples for the training dataset.

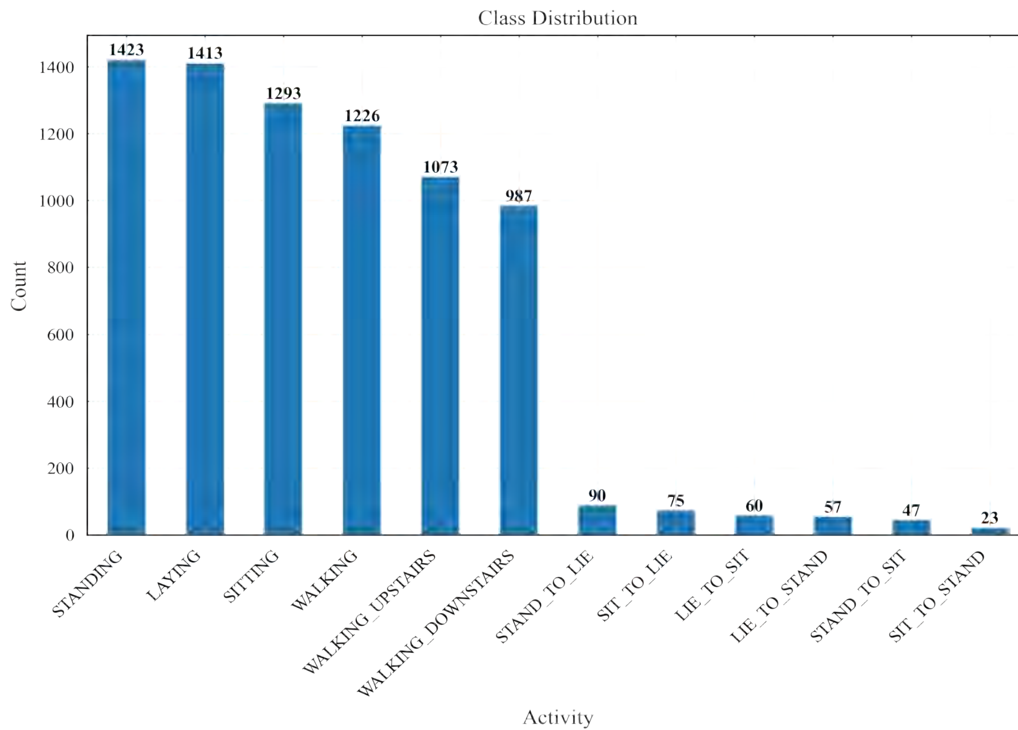

Figure 4.1: Bar Diagram of HAPT Training Dataset

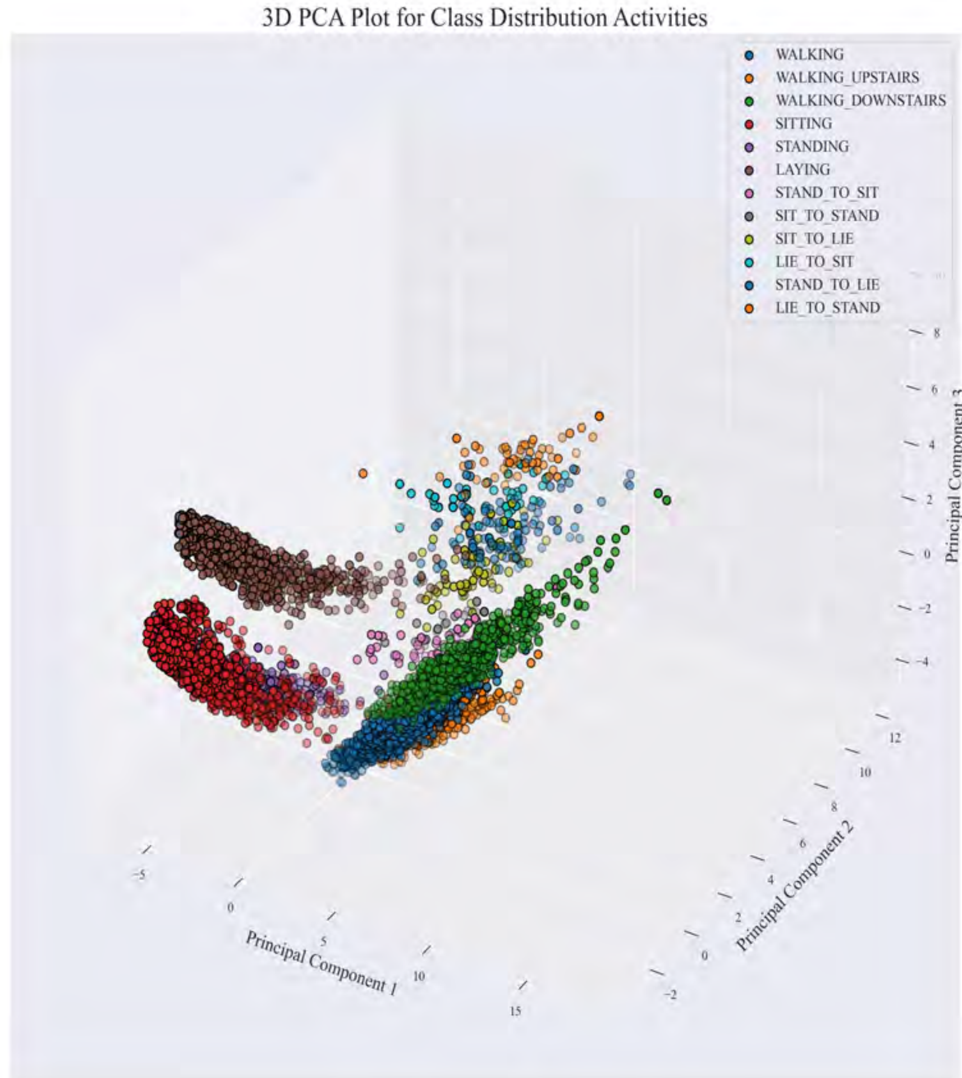

Figure 4.2: Principal Component Analysis

From the above PCA (Figure 4.2) of the dataset, it can be observed that the dataset is mostly linearly separable. On linearly separable datasets, linear models and perceptrons can provide competitive performance with reduced computational demands.

4.1.1 HAPT Training Dataset Balancing

From the Figure 4.1 it is clear that the training dataset is not entirely balanced as some activities have very low sample counts. This can negatively affect the model performance and make certain models bias towards the first six labels [31].

To address this issue, we have scaled sample associated with label 7 to 12 which are *stand_to_lie*, *sit_to_lie*, *lie_to_sit*, *lie_to_stand*, *stand_to_sit*, and *st_to_stand*. These activities were chosen for targeted oversampling due to their initially lower representation in the dataset. The decision to focus on these activities was driven by the goal of rectifying the imbalance in label frequencies to create a more balanced and representative dataset. In the below Figure 4.3, we can see the scaled features and their corresponding sample count.

The selected features consist of distinct human movements and transitions. By strategically duplicating rows associated with these specific features based on calculated oversampling factors, the scaling process ensures a more even distribution of these activities in the dataset. The oversampling factor for each label is calculated as follows:

- If the current count of a label is less than the minimum desired count:

$$factor = \lceil \frac{min_count}{label_count} \rceil \tag{4.1}$$

- If the current count of a label is more than the maximum desired count:

$$factor = \lfloor \frac{max_count}{label_count} \rfloor \tag{4.2}$$

The number of rows to be duplicated for each label is determined by the oversampling factor:

$$Number\ of\ duplicates = factor \times Number\ of\ existing\ rows\ for\ the\ label \tag{4.3}$$

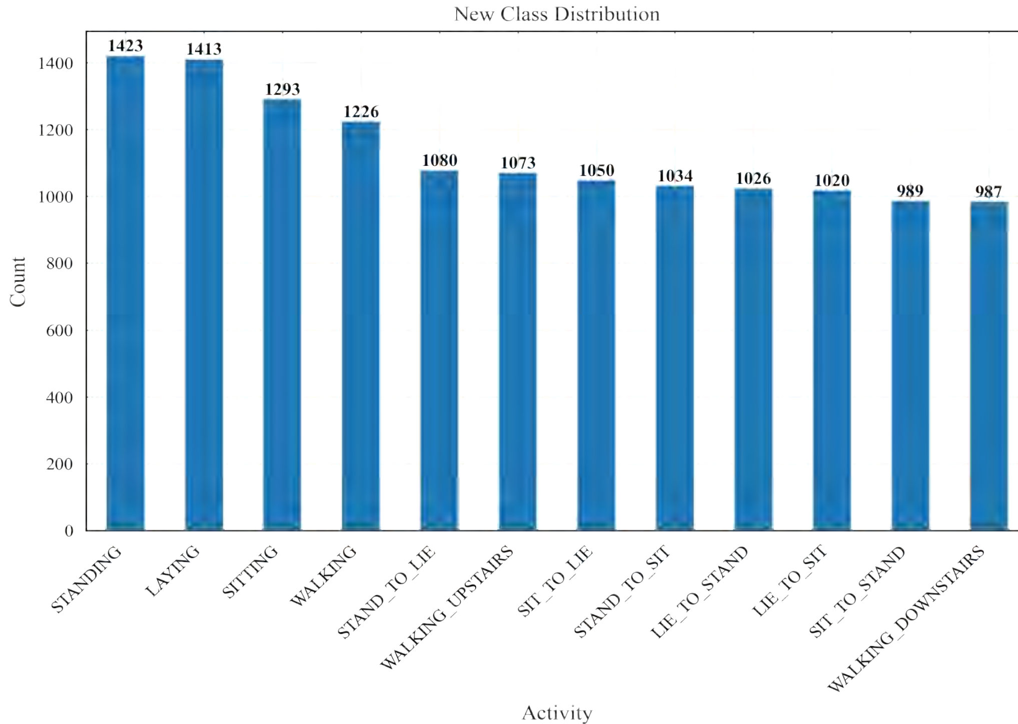

Figure 4.3: Bar Diagram of Scaled HAPT Dataset After Scaling

This approach helped us to make our dataset suitable for training. Besides, this strategy improved the accuracy of Decision Tree and LSTM which is given in the below Table 4.1 and 4.2.

Model	**Accuracy**	**Precision (macro)**	**F1 Score (macro)**
Decision Tree	81%	73%	73%
LSTM (TensorFlow)	83%	71%	63%

Table 4.1: Before of Data Duplication

Model	**Accuracy**	**Precision (macro)**	**F1 Score (macro)**
Decision Tree	85%	72%	72%
LSTM (TensorFlow)	$86\% \pm 1$	73%	72%

Table 4.2: After Data Duplication

The details about the models and experimental setup are discussed in Chapter 6.

4.1.2 Feature Reduction

The HAPT dataset has a total of 561 features of which all are floating point numbers. For most modern edge devices, 561 features are not a huge amount of data to process. However, as we are targeting highly resource constraint devices, we had to reduce the number of features individually for targeted ML models. Effect of feature reduction on the ML models can be seen in Chapter 6 section 6.2.4 and 6.2.5.

Feature Reduction For Decision Tree

At first to get an idea about the features and their impact on the output, we did L1 and L2 regularization. This helped us to reduce 151 features. After a number of experiments, we employed the Recursive Feature Elimination (RFE) [53] technique in conjunction with a Decision Tree Classifier, which we found to provide the best results. The Decision Tree was configured with specific hyperparameters, which is discussed in Chapter 6 section 6.2.4 in conjunction with RFE to select the top 80 features considered most valuable for the predictive task. The iterative process involved training the Decision Tree and eliminating the least important features until the desired number was achieved. The resulting reduced feature set was then applied to both the training and testing datasets. The top 5 features (0 based indexing) from the selected 80 are 0, 58, 57, 159, and 55. The below Figure 4.4 shows the correlation between reduced features and Labels for Decision Tree.

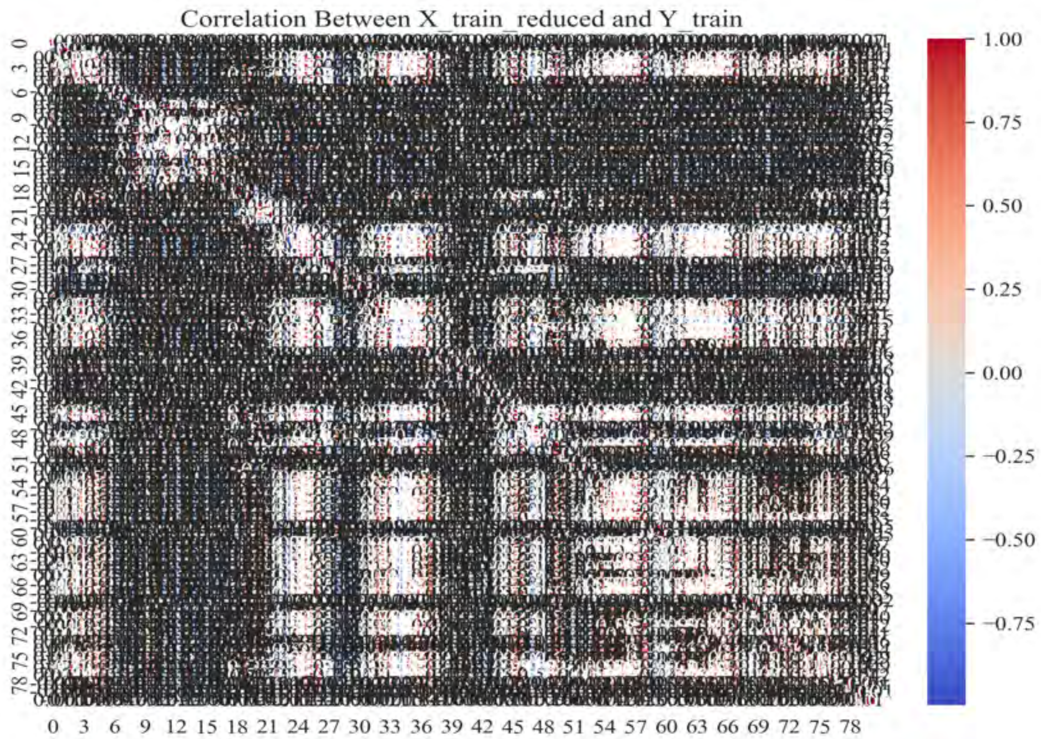

Figure 4.4: Correlation of the Feature-Reduced HAPT Dataset for Decision Tree

Feature Reduction For Logistic Regression

Similarly, here also we reduced the number of features for Logistic Regression to 80. Here we changed the *max_iter* parameter to 300 to speed up the process. The top five features (zero-based indexing) from the selected 80 are 56, 433, 166, 70, and 69. The below Figure 4.5 shows the correlation between reduced features and labels for Logistic Regression Model.

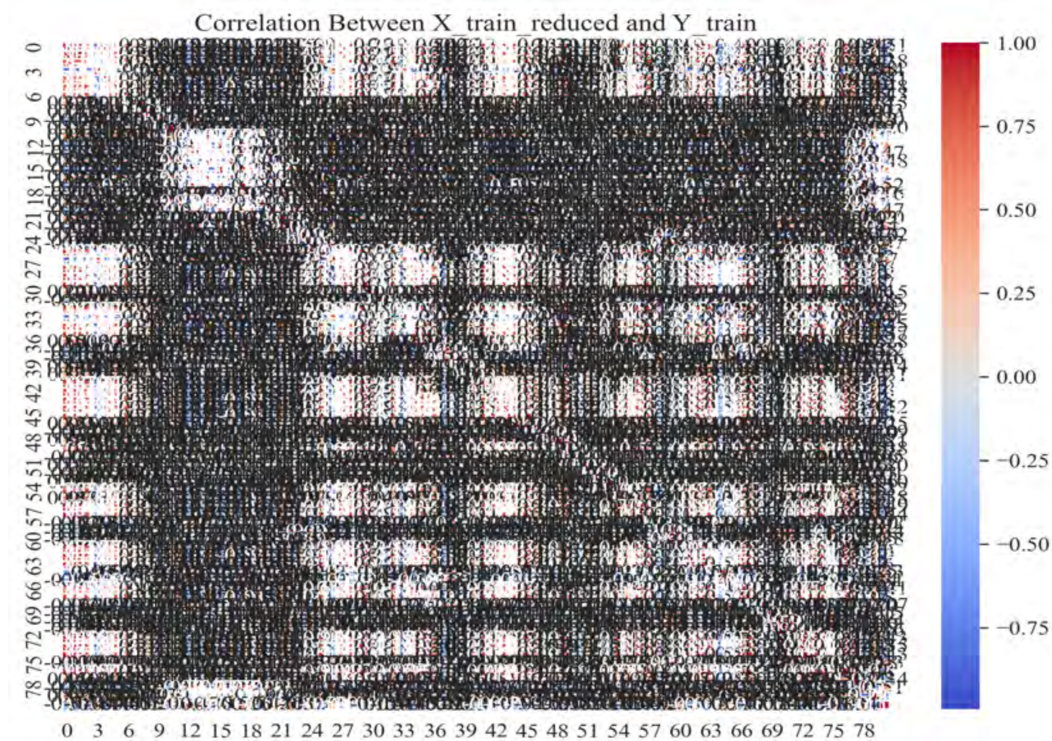

Figure 4.5: Correlation of the Feature-Reduced HAPT Dataset for Logistic Regression

4.2 UMAFall Dataset

The second dataset we used for FD is UMAFall dataset. UMAFall dataset includes data from 19 participants who were wearing sensors on five different body locations while conducting different Activities of Daily Living (ADLs) and simulating falls. In contrast to other datasets that only use one or two sensors, UMAFall provides more information to investigate how sensor location affects the accuracy of FD. Because of this, it is especially helpful for evaluating and contrasting various FD algorithms, which will ultimately improve safety for the elderly and other vulnerable people who are more likely to fall.

It is significant to note that risk categorization is not a specific focus of our investigation. Rather, we focus on the effectiveness, precision, and instantaneous performance of edge-optimized models in applications that are relevant to people. In preparing the UMAFall dataset for analysis, a Python implementation was employed to distill pertinent information from the raw CSV files. The dataset encompasses sensor data capturing diverse activities, including backward falls, forward falls, lateral falls, and non-fall activities. Each activity was assigned a corresponding label through a predefined mapping, facilitating the classification task. The implemented method selectively processed wrist sensor data (identified as sensor ID 3) because the wrist sensor data gave more accuracy [15] and extracted acceleration values along the X, Y, and Z axes.

Throughout the data extraction process, our main attention was devoted to han-

dling irregularities and errors to ensure dataset integrity. Extraneous lines with irrelevant information were skipped, and robust exception handling was implemented to address potential parsing errors. The resulting dataset was structured to include acceleration values and their corresponding activity labels.

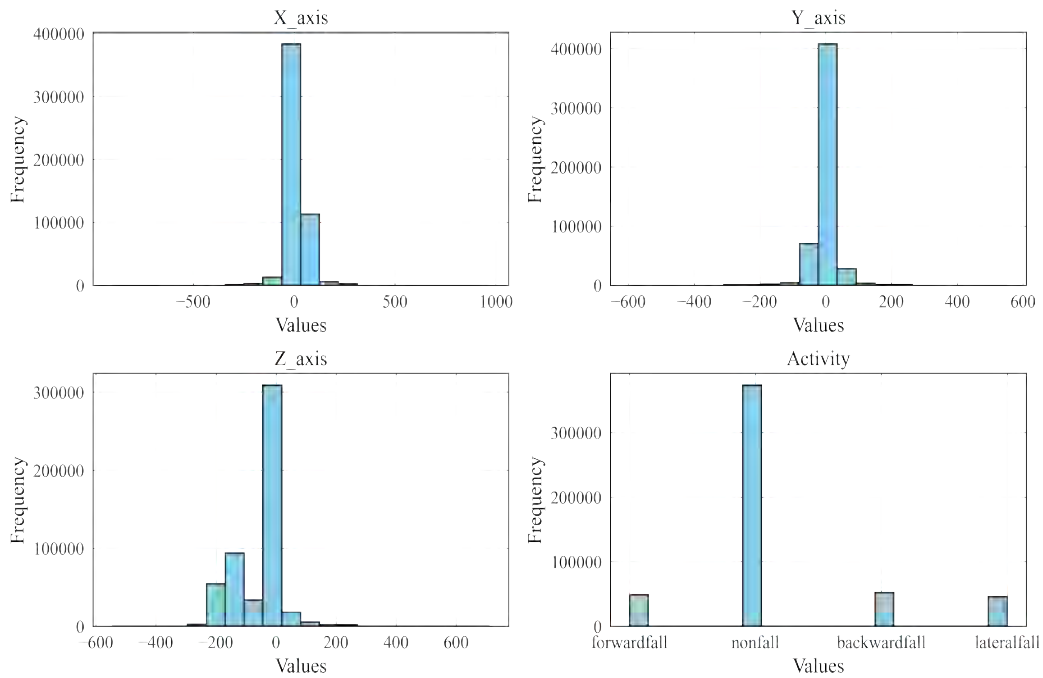

Figure 4.6: Histogram of UMAFall Dataset

The processed dataset was then transformed into a NumPy array, which provides a versatile framework for subsequent manipulation and analysis. To assess machine learning model performance, the dataset was stratifiedly split into training and testing subsets. The training subset, having 80% of the data, served as the basis for model training, while the remaining 20% formed an independent test set for model evaluation.

Chapter 5

Research Methodology

Our research methodology encompasses a systematic approach that involves taking a dataset, processing it through various stages, optimizing it, and ultimately deploying it to the target Microcontroller Unit (MCU) or Edge device. The below Flow Chart 5.1 shows the high-level overview of the entire process. This process includes training and optimizing the model, and ultimately converting the trained model into a CPP header file format. This format is compatible and deployable with the firmware of Microcontroller Units (MCUs).

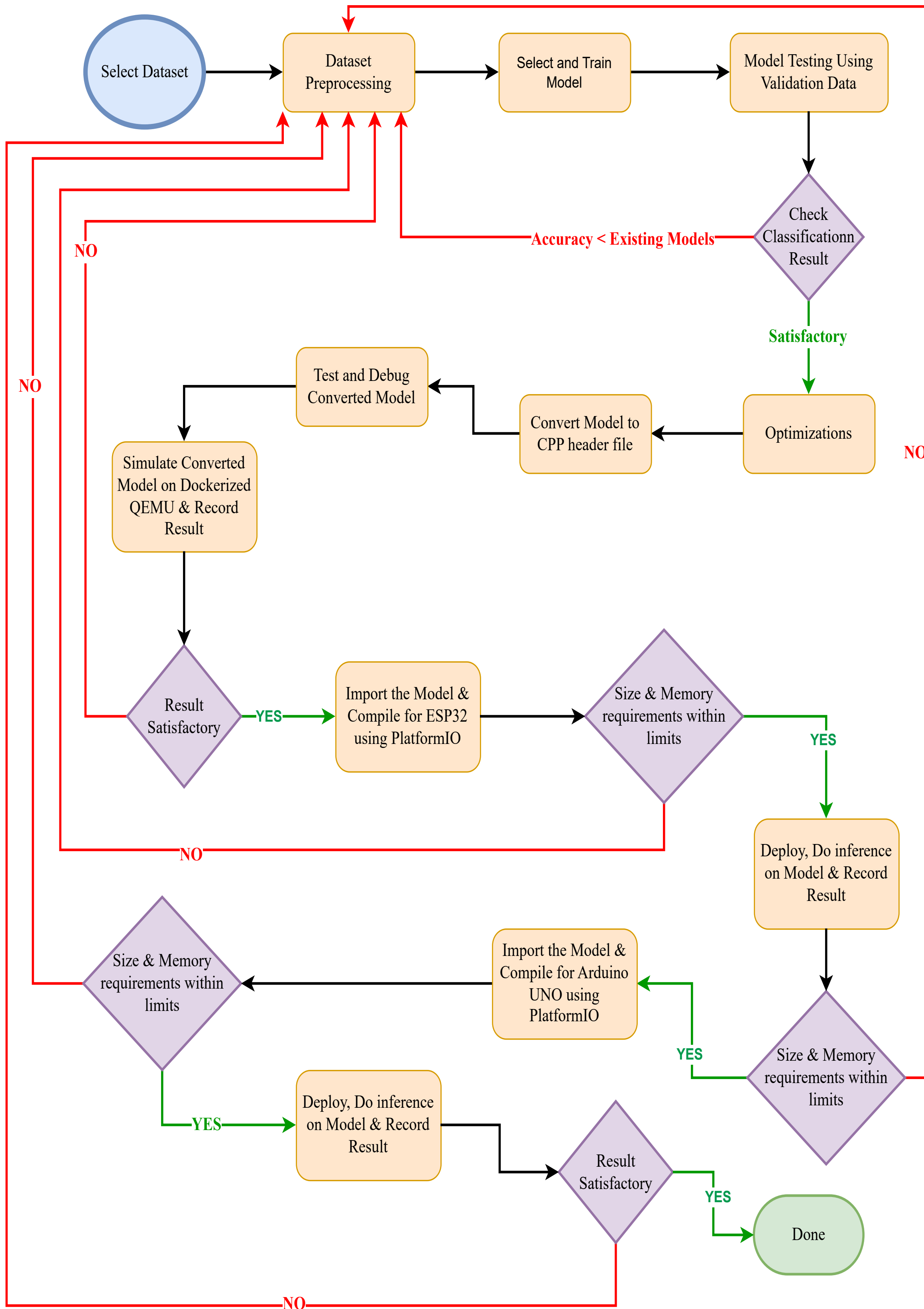

Figure 5.1: Overview Of Research Methodology

5.1 Dataset Selection and Preprocessing

The journey towards developing a successful machine learning model begins with the careful selection of the proper datasets. Once these datasets have been chosen, the dataset must undergo a process of preprocessing to ensure that the best possible results can be obtained. This involves the use of various techniques such as feature engineering, normalization, and data profiling. Furthermore, depending on the dataset and the task at hand, we may have to perform Adversarial training, Data Augmentation etc. This can help some machine learning algorithms perform better, reduce complexity by ensuring that all characteristics in a dataset are on the same scale.

5.2 Model Selection

The process of selecting the ideal machine learning model for a particular task is known as model selection. It involves fitting and evaluating a range of models on a given dataset and selecting the one that performs the best. However, model performance is not the only factor to consider. Other factors such as complexity, maintainability, and available resources also play a role in the selection process. Model selection is applicable to models of various types as well as to models of the same type with various hyperparameter configurations. The models are chosen based on a thorough evaluation of various metrics, including precision, recall, F1-score, support, accuracy, macro average, and weighted average. These metrics guide our decision to ensure the selected model aligns with the desired performance criteria. However, the most important factor to consider is to check if the model complexity is low enough to be deployable in MCU.

5.3 Testing

Testing is a crucial aspect in our methodology as it determines the credibility and runtime behavior of the selected model. By assessing a fully trained model on a distinct testing set, we can glean significant insights into the model's capacity to generalize to new data and make precise predictions.

Our testing set is meticulously curated to be separate from both the training and validation sets, yet it follows the same probability distribution as the training set. This approach ensures an unbiased and fair evaluation of the model's performance.

5.4 Model Performance Comparison

The phase of model comparison, which follows the critical stage of model testing, is integral to the development of robust and reliable machine learning models. This phase allows us to understand the strengths and weaknesses of the trained model by comparing its performance against other models that have undergone similar or more advanced training processes.

In the context of our research methodology, this comparison is not just a one-time process. It's an iterative cycle that feeds back into the 'Data Preprocessing' and 'Model Testing Using Validation Data' stages. This iterative approach ensures that our model is continually refined and improved, enhancing its performance metrics.

5.5 Optimization And Model Conversation

At this stage, the trained model is optimized and then converted to the CPP format for further processes depending on the edge device of interest. Conversion may or may not be simple, which vastly depends on the model that we have trained and tested. To achieve this conversion, we can use libraries such as micromlgen that can automatically generate CPP header file based on Scikit Learn [37] models. However, not all models are supported. In such cases, manual conversion to an equivalent C/CPP code is necessary.

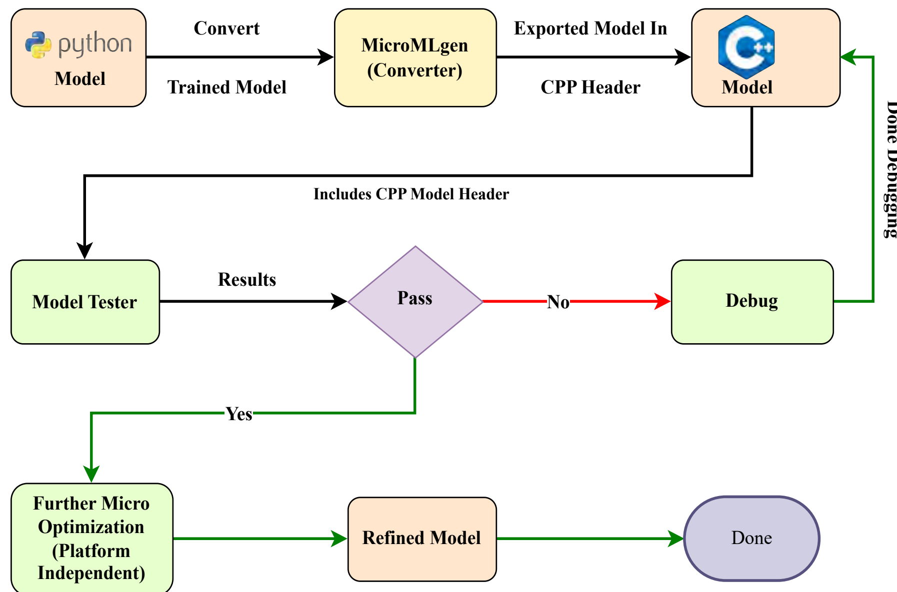

Figure 5.2: Model Conversion, Testing and Debug

After Model conversion, it is necessary to test the model and validate its input and output as well as other metrics. It is common that generated codes are not always correct and contain bugs or even performance issues. In such cases, using a debug tool such as GDB [49] and a profiling tool is imperative. A generic Python Model to CPP conversion, testing, and Debug process is illustrated using Figure 5.2.

5.6 Profiling Converted Model on Emulated Raspberry Pi

Running and Profiling the model on Emulated Raspberry Pis provides insight into the model performance in a resource constraint environment. The below Figure 5.3 illustrates the emulation process.

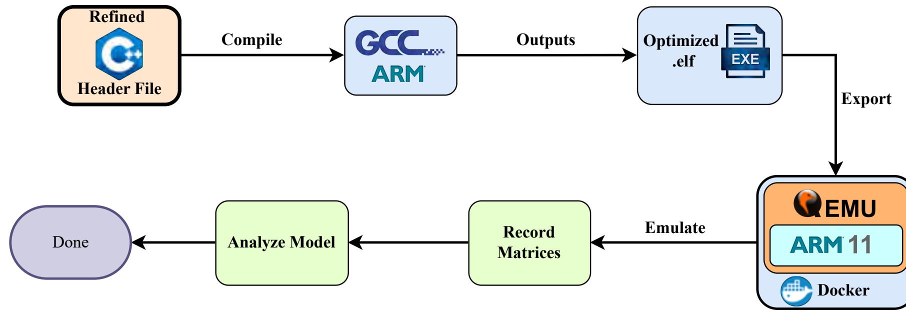

Figure 5.3: Steps to Run the Model on Emulated Raspberry Pi

Although a Pi is not as resource constraint as a generic MCU, but it is helpful to run models on them for a few reasons:

- Raspberry Pi supports Linux Operating, and it comes with gnu core utilities.
- This makes it easy to use tools such as *GDB* [49], *perf* [51], and *time* [50] which makes performance measurements relatively easy where on an MCU one needs to have debugging hardware to be able to measure, and the process is also quite complicated.
- The metrics that need to be collected are memory consumption by the ML models, execution time, max execution time, samples for which it takes max time to execute, samples for which is takes max memory and binary or executable size [19]. For the initial phase, collecting these metrics from Pi is fairly straightforward and does not require any complicated steps.

Before moving forward to deploying to a physical MCU, running and testing models on Raspberry Pi helps set the expectations and further helps to curate selected models even further.

5.7 Optimized Model Deployment on MCUs

Going forward, the CPP model needs to be cross-compiled for the target MCU board. If the compilation process is successful, then flash the generated executable or *elf* to the MCU.

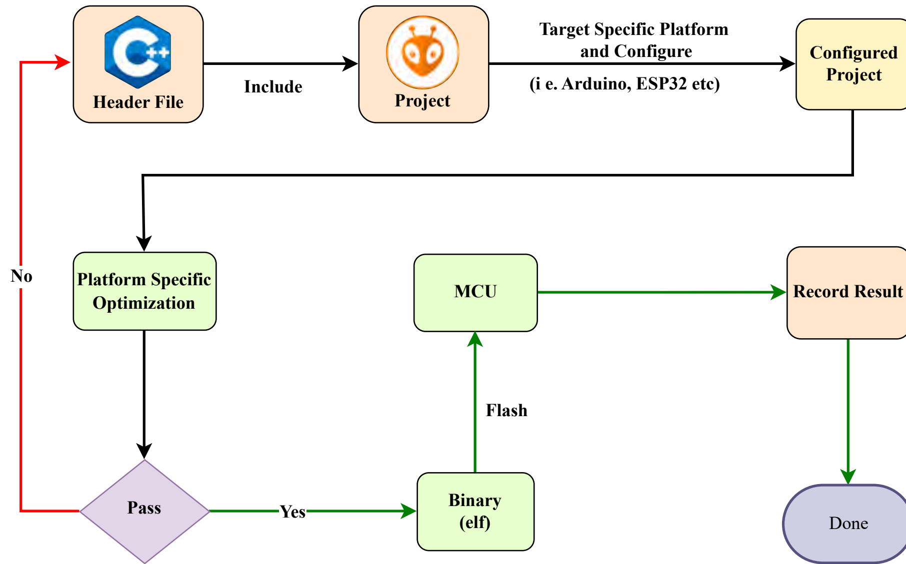

Figure 5.4: Steps to Cross Compile and Deploy

The above Figure 5.4 shows a visual illustration of the steps to deploy to a target board, compile using the appropriate tool chain, flash, and then finally record results. As we deployed to multiple MCUs, we followed the hierarchy of more capable board to less capable board. This was to track the progress and pinpoint the exact issue, if any.

5.8 Result Recording and Comparison

After deploying the model to the MCU, it is necessary to record its performance metrics. These include execution time, memory usage, and model accuracy. Based on performance metrics, we can then infer the effectiveness of the applied optimizations. However, in case the results are not satisfactory, it is imperative that we trace back to a previous stage and improve further up on this iteration.

Chapter 6

Experimental Evaluation

This chapter presents a detailed examination of our research's experimental component, which is centered around the development and optimization of Machine Learning (ML) models tailored for resource-constrained Microcontroller Units (MCUs). The entire life cycle of an ML model optimization, including dataset selection to finally deploying the model in the target board is discussed in Chapter 5. The subsequent sections will provide an in-depth look at our experimental setup, the metrics employed for evaluation, and the results garnered, thereby underscoring the practical applicability and success of our methodology.

6.1 Experimental Setup

Our personal computer, equipped with a *Ryzen 9–5900X* CPU, *RTX 3070 (8 GB)* GPU, and *32GB* of memory, served as the backbone of our computational infrastructure. Furthermore, the operating system was Pop!_ OS 22.04 LTS with Linux kernel 6.6.6 that had LLVM toolchain preinstalled. We opted for Python version 3.9 over 3.11 due to its maturity and stability, and utilized the Scikit Learn [37] library and TensorFlow framework [38] for our machine learning tasks as discussed in Chapter 5 in details. To generate *C++* header file for trained models, we used the micromlgen version 1.1.28 library [39] and used the generated code as our starting point. To emulate Raspberry Pis and validate our hypothesis, we used a dockerized QEMU environment called docker-qemu-rpi-os [41] and it allowed us to easily spin up isolated containers for Raspberry Pis. To easily create firmware, manage dependencies, and target different frameworks and architectures, we used PlatformIO core version 6.1.12a6 with Clion which includes PlatformIO [46] plugin essential for integrating PlatformIO with Clion [44]. Finally, the MCU units that we used for this study were the Arduino UNO and ESP32 AI Thinker. Arduino UNO is a robust open-source microcontroller board based on the ATmega328P microcontroller. It operates on an 8-bit AVR RISC-based architecture, with a clock speed of 16 MHz, and includes 32KiB of Flash memory, 2KiB of SRAM, and 1KiB of EEPROM. On the other hand, The ESP32 AI Thinker is a comprehensive wireless MCU module known for its advanced capabilities. It runs on a 32-bit, dual-core Tensilica Xtensa LX6 microprocessor, boasting clock speeds up to 240MHz and featuring 320KiB(DRAM) [45] of SRAM.

For our research, we used UCI HAPT dataset and UMAFall Detection dataset,

discussed in Chapter 4. For simplicity and ease of reading, we are going to discuss the experimental results in their own separate sections. Besides, for calculation memory consumption and flash size, we first calculated them in bytes and then converted them to KB by dividing by a 1000.

6.2 UCI HAPT

Our analysis revealed linear relationships within the data as we learned from Chapter 4, prompting us to employ a Logistic Regression model with default parameters using the SK Learn Python library. On top of that, we trained a list of models including Neural Networks to check and validate the performance of those on this dataset without any scaling or feature engineering. The below Table 6.1 shows the vanilla models and their performance on this dataset.

Model	Accuracy	Precision (macro)	F1 Score (macro)
Logistic Regression	95%	88%	87%
SVM	95%	90%	88%
KNN	90%	83%	79%
Decision Tree	81%	73%	73%
XGBoost	92%	83%	83%
MLP (TensorFlow)	92%	86%	84%
LSTM (TensorFlow)	83%	71%	63%
RNN (TensorFlow)	57%	36%	29%
CNN (TensorFlow)	94%	87%	79%

Table 6.1: Vanilla Trained ML Models and Neural Networks

As per previous assumption, the logistic regression model yielded high accuracy with good precision and F1 score. SVM was very closely comparable with logistic regression and has better precision and f1 score but is computationally more intensive and expensive. CNN, on the other hand, is trained in TensorFlow format and achieved 94% accuracy with 87% precision and 79% F1 Score. We could push it further by increasing the number of epochs and layers, but doing so would make it more computationally intensive and would require more memory to allocate for tensors. Despite achieving good accuracy, moving forward we will only continue with the ML models as this is the focus of the study.

Based on the approach and metrics discussed in Chapter 5, as a result, we selected the Logistic Regression, SVM, Decision Tree, and XGBoost models for further analysis. These models were chosen due to their simplicity and efficiency in terms of architecture, time, and space complexity. We decided to exclude the KNN model from our selection. Despite its decent performance, KNN suffers from high compu-

tational cost as the size of the dataset increases. This is due to the fact that KNN needs to compute the distance of a given test point to all training points, which can be computationally expensive for large datasets.

The selected models were then converted to C++ header files using the MicroML Gen tool. To verify the correctness of the converted models, we created a C++ project with CMake using Ninja and ran the codes in CLion. We encountered some bugs during this process.

One of the issues identified was with the label encoding in the generated C++ header file. In our code and dataset, there are 12 labels ranging from 1 to 12. However, in the generated code, the labels start from 0 to 11. This discrepancy needed to be addressed to ensure the correct functioning of the models.

Furthermore, we found opportunities for optimization in certain areas of the code. After debugging and micro optimizing the code, the converted models achieved the same level of accuracy and efficiency as the original models.

6.2.1 Simulating inside Dockerized QEMU Pi Environment

Moving ahead, we cross-compiled the models for Raspberry Pi1 & Pi2 and ran them using the *docker-qemu-rpi-os* project. For optimization reasons, we used the *-O3* or Optimization level 3 flag while compiling. The below Table 6.2 and 6.3 contains the collected results from Raspberry Pi 1 and Raspberry Pi 2B. The metrics were used to shorten the selected model list even further.

Model	**Accuracy**	**Precision (macro)**	**Execution Time AVG (µs)**	**Max Execution Time (µs)**	**Memory Used (MB)**	**Binary Size (KB)**
Logistic Regression	95%	92%	196.453	525.5	1.968	84
Decision Tree	81%	73%	13.677	100.4	1.896	11
XGBoost	92%	83%	209.200	636	1.844	11
SVM	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory

Table 6.2: Performance on Raspberry Pi 1

Model	Accuracy	Precision (macro)	Execution Time AVG (µs)	Max Execution Time (µs)	Memory Used (MB)	Binary Size (KB)
Logistic Regression	95%	92%	177.056	405.6	1.908	84
Decision Tree	81%	73%	0.98279	64.3	1.820	11
XGBoost	92%	83%	159.026	557.7	1.820	11
SVM	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory

Table 6.3: Performance on Raspberry Pi 2 B

The SVM model was excluded from our selection due to its high memory usage. During the compilation phase, the SVM model ran out of memory even when Raspberry Pi 2 B was used, which includes 1GB of memory. This is a common issue with SVMs as they require a significant amount of memory, especially for large datasets with more features. Given our aim to deploy the models on resource-constrained devices, SVMs were deemed unsuitable for our use case.

We also decided to avoid the XGBoost model despite its good performance in terms of accuracy and precision. XGBoost uses gradient boosting, a technique that builds many Decision Trees and combines them to make a final prediction. While this approach is powerful, it also requires a significant amount of computational resources. Given our end goal of deploying the models on resource-constrained Microcontroller Units (MCUs), using XGBoost was not feasible simply because of its high number of floating point arithmetic operations. Furthermore, XGBoost models are more complex and harder to interpret compared to simpler models like Decision Trees and Logistic Regression.

Our final selection focused on the Decision Tree and Logistic Regression models. These models were chosen for several reasons. Firstly, both models showed good performance in terms of accuracy and precision. While Decision Tree's performance was lower than XGBoost, the difference was not significant enough to outweigh the benefits of its simplicity and efficiency. Secondly, Decision Trees and Logistic Regression models are computationally efficient. They require less memory and have faster execution times compared to more complex models like XGBoost and SVM.

6.2.2 Deployment on ESP32 and Initial Observation

Using Clion we created two PlatformIO (*Decision Tree* and *Logistic Regression*) projects choosing ESP32 AI Thinker board with the Arduino framework, which is a wrapper around underlying ESP-IDF. Imported the CPP header file in the project

and did the necessary adjustments and configurations. Finally, compiled and flashed it on ESP32 using the PlatformIO toolchain. Below Table 6.4 contains the results.

Model	**Accuracy**	**Precision (macro)**	**Execution Time AVG (μs)**	**Memory Used (KB)**	**Flash Size (KB)**
Logistic Regression	95%	92%	10325	23.7 (7.81%)	400.60 (13%)
Decision Tree	81%	73%	5	23.7 (7.81%)	285.381 (9%)

Table 6.4: Initial performance on ESP32

It is evident from Table 6.4 that Decision Tree is both faster and consumes less storage. The only downside is the accuracy where Logistic Regression shines. Despite achieving good accuracy, Logistic Regression is orders of magnitudes slower than Decision Tree. The primary reason for Logistic Regression being slower is the fact that it involves floating point arithmetic, and MCUs particularly are not fast doing it due to its resource constraints nature. Decision Tree, on the other hand, does not have any floating point calculations. Contains only branching (*if else*) statements which in assembly just register comparison and jumping to the appropriate labels, which is the reason behind its execution speed and less resource consumption.

6.2.3 Deployment on Arduino UNO and Initial Observation

Deploying to Arduino UNO is very similar to the steps mentioned in the previous section 6.2.2. The difference here is that instead of targeting ESP32, here we are targeting Arduino UNO. The below Table 6.5 contains the collected metrics for Arduino UNO.

Model	**Accuracy**	**Precision (macro)**	**Execution Time AVG (μs)**	**Memory Used (KB)**	**Flash Size (KB)**
Logistic Regression	Compilation Failed, Executable Size Exceeded max specified Size				
Decision Tree	81%	73%	84.578	2.432 (121.5%)	14.132 (43.8%)

Table 6.5: Initial performance on Arduino UNO

The Table 6.5 provides an insightful comparison of the execution of two machine learning models, Logistic Regression and Decision Tree, on the Arduino UNO plat-

form. Interestingly, the Logistic Regression model could not be compiled for deployment on the Arduino UNO. This suggests that the converted CPP header file contains too many instructions for Arduino UNO to hold indicating a need for model optimization.

In contrast, the Decision Tree model was successfully deployed and demonstrated promising results. It achieved an accuracy rate of 81% and a macro precision of 73%, indicating a good predictive performance on the test dataset. However, the memory usage of the Decision Tree model was 121.5% of the available memory, which is a concern as it exceeds the Arduino UNO's default capacity. During the inference, we observed no issues with the Arduino UNO. The average execution time was quite low at 84.578 microseconds, suggesting a rapid response time. However, this could potentially lead to issues in a real-world deployment scenario and demands memory optimization. The flash size used was less than half of the available size, which is a positive aspect.

6.2.4 Optimization of Decision Tree

We aimed to enhance the performance of the Decision Tree model, which initially had an accuracy of 81%, significantly lower than the 95% accuracy of the Logistic Regression model.

As discussed in Chapter 4, we scaled features with low frequency and balanced the dataset. This preprocessing step led to a significant improvement in the performance of both the Decision Tree and LSTM models. The Decision Tree model's accuracy increased from 81% to 85% after scaling and parameter fine tuning. The below Table 6.6 contains the parameters that performed best in our experiments.

max_depth	**min_samples_leaf**	**min_samples_split**	**max_leaf_nodes**
20	10	10	600

Table 6.6: Best Performing Parameter for Decision Tree in Scaled HAPT Dataset

The LSTM model also saw an improvement, with accuracy increasing from 82% to 87%. However, our focus remained on non-neural network models, and thus, we did not further consider the LSTM model.

Following previously mentioned steps, we converted the models to CPP header files and deployed them on Raspberry PI, ESP32, and Arduino UNO. The results were consistent across all platforms, with the accuracy now at 85%. However, the memory issue on Arduino UNO persisted, as we had not yet optimized for memory usage. The below Table 6.7 shows the behaviour of the scaled model on Raspberry Pi 1, ESP32 and Arduino UNO.

Board	Accuracy	Precision (macro)	Execution Time AVG (µs)	Memory Used (KB)	Flash Size (KB)
Raspberry Pi 1	85%	70%	13.256	2.412	15
ESP32	85%	70%	37	23.720	280.597
Arduino UNO	85%	70%	110.98	2.432	8.538

Table 6.7: Performance of Scaled Decision Tree on Different Boards

To address the memory issue and further optimize the model, we turned our attention to feature reduction. UCI HAPT dataset has a total of 561 features, a substantial number for low-powered, resource-constrained MCUs. Moreover, not all features were correlated, and can be reduced, which we discussed in Chapter 4. In short, we applied L1 and L2 regularization techniques to determine the number of features we could exclude from the dataset while maintaining the same level of accuracy and precision for Decision Tree. Our analysis revealed that a significant number of features could be excluded. We then used Feature Elimination with Cross-Validation (REF) from sklearn for and fine-tuned specifically for the Decision Tree model. This trial and error process led us to reduce the number of features from 561 to 80, with little to no difference in the precision and accuracy of the model.

Board	Accuracy	Precision (macro)	Execution Time AVG (µs)	Memory Used (KB)	Flash Size (KB)
ESP32	85%	70%	38	21.8	278.641
Arduino UNO	85%	70%	114.50	0.508	6.606

Table 6.8: Optimized Decision Tree Performance on Arduino UNO & ESP32

Table 6.8 presents the performance metrics of the optimized Decision Tree model on two different MCU boards: ESP32 and Arduino UNO. The model achieved an accuracy of 85% and a macro precision of 70% on both boards, proving the effectiveness of our feature reduction and model optimization strategies. This is a significant improvement from the initial accuracy of 81%, bringing it closer to the 95% accuracy of the Logistic Regression model. The average execution time was 38 microseconds on ESP32 and 114.50 microseconds on Arduino UNO. Despite the higher execution time on Arduino UNO, it is still within an acceptable range for real-time applications.

In terms of memory usage, the optimized model used 21.8 KB on ESP32 and only 0.508 KB on Arduino UNO. This is a significant reduction from the initial memory usage of 121.5% on Arduino UNO, demonstrating the effectiveness of our memory optimization strategies. The flash size used was 278.641 KB on ESP32 and 6.606 KB on Arduino UNO, both of which are within the capacity of the respective boards. This leaves plenty of options for the developer to add extra functionalities on top of the HAR.

6.2.5 Optimization of Logistic Regression

Moving forward, we turned attention to the Logistic Regression model, which initially had an impressive accuracy of 95% and a precision score of 92%. However, we faced challenges in deploying this model on Arduino UNO due to its flash size exceeding the maximum allowed limit and high memory usage. This necessitated optimization for both flash size and memory.

We followed a similar process as with the Decision Tree model discussed in the previous section 6.2.4, applying L1 and L2 regularization and Recursive Feature Elimination (RFE) to reduce the number of features. This approach was based on our previous findings that feature reduction could effectively reduce both memory and flash size.

Before reducing the features, we applied L1 regularization to the Logistic Regression model and observed its impact. The results were promising: the execution time for ESP32 halved, as shown in the Table 6.9 below:

Model	Accuracy	Precision (macro)	Execution Time AVG (µs)	Memory Used (KB)	Flash Size (KB)
Logistic Regression	95%	92%	10325	23.7	400.60
Logistic Regression (Regularized)	94%	92%	5162.5175	23.7	333.645

Table 6.9: Impact of L1 Regularization on Logistic Regression

Encouraged by these results, we proceeded with feature reduction, reducing the number of features from 561 to 80. We also applied L1 regularization to make the computation faster by setting the weights of less important features to zero. However, to maintain original accuracy, we did parameter fine-tuning on Logistic Regression After these optimizations, we were able to deploy the Logistic Regression model on both Arduino UNO and ESP32. The performance metrics of the optimized model on these platforms are as follows:

Board	Accuracy	Precision (macro)	Execution Time AVG (µs)	Memory Used (KB)	Flash Size (KB)
ESP32	94%	87%	462	21.808	294.813
Arduino UNO	94%	87%	17634	0.509	16.64

Table 6.10: Impact of Feature Reduction on Logistic Regression

As evident from the Tables 6.10, the optimized Logistic Regression model achieved excellent results, with an accuracy of 94% and a precision score of 87% on both platforms. The execution time was significantly reduced, and the memory and flash sizes were within the limits of the respective boards.

6.3 UMAFall Detection

In addition to the UCI HAPT dataset, we applied our optimization strategies to another dataset, the UMAFall Detection dataset. For this dataset, we focused solely on optimizing the Decision Tree model as we learned in the previous section 6.2 that DT is both faster and extremely memory efficient.

Following the methodology of a previous study [15] on this dataset, we used only samples from sensor id 3 for our analysis. This sensor, located on the wrist, was found to yield the best results in the previous study. We did not consider any risk factors in this process. The below Table 6.11 shows different models' performance on wrist sensor data.

Model	Accuracy	Precision (macro)	F1 Score (macro)
KNN	82%	71%	67%
Decision Tree	82%	76%	66%
MLP (TensorFlow)	72%	23%	21%

Table 6.11: Trained Models on UMA Fall Dataset

As detailed in the previous section 6.2.4, we optimized the Decision Tree model using the same approach. The below Table 6.12 contains the best performing parameters for the model.

max_depth	min_samples_leaf	min_samples_split	max_leaf_nodes
15	20	20	900

Table 6.12: Best Performing Parameter for Decision Tree in UMAFall Dataset

After trail and error, we then deployed the model on ESP32 and Arduino UNO. Table 6.13 contains the final results we obtained from the experiments.

Board	Accuracy	Precision (macro)	Execution Time AVG (µs)	Memory Used (KB)	Flash Size (KB)
ESP32	81%	77%	37	21.504	296.685
Arduino UNO	81%	77%	121	0.2	26.518

Table 6.13: Decision Tree Performance on ESP32 and Arduino UNO

These results demonstrate that our optimization strategies are effective across different datasets and hardware platforms. The optimized Decision Tree model delivers high accuracy and precision, making it a viable solution for edge computing applications.

Chapter 7

Discussion

In this study, we demonstrated how to optimize ML models in Human Activity Recognition (HAR) and Fall Detection for resource constraint MCUs. For HAR our optimized Logistic Regression model running on MCUs achieved 94% accuracy and Decision Tree which achieved 85% accuracy. Although Logistic Regression is more accurate, but orders of magnitude more time-consuming than Decision Tree. Similarly, for FD we achieved 81% accuracy with consuming less 10% of the total memory on ESP32 and Arduino UNO. It's not an overstatement to say that these microcontrollers could potentially sustain their operations for years on a single battery, depending on the specific use case and power management strategies employed. We found The Decision Tree model to be both memory and time efficient; however, we only worked with two datasets, so we cannot certainly say that this will be the case when working with other datasets. More studies should be conducted on optimizing Random Forest since they tend to provide better accuracy compared to Decision Tree [6]. In the below sections 7.1.1 and 7.1.2 we discuss some of the shortcomings and future scope in this field.

7.1 Limitations

Our study, while promising, has several limitations that should be acknowledged. These limitations, detailed in the following subsections 7.1.1 and 7.1.2 provide valuable insights into the challenges faced during our study and highlight areas for potential improvement.

7.1.1 HAPT Dataset

- **Decision Tree Accuracy**: Although we improved the accuracy of the Decision Tree model to 85%, it still falls short of the desired performance. Better feature engineering methods should be researched to further improve this accuracy.

- **Unexplored Models**: We did not explore the Random Forest model due to its increased memory and flash size requirements. However, Random Forest often performs better than Decision Tree in terms of accuracy. Future research should explore better data engineering, feature extraction, and parameter/hyperparameter fine-tuning for this model.

- **Logistic Regression Performance**: The performance of the Logistic Regression model is satisfactory but could be improved by further feature reduction and floating-point arithmetic optimizations.
- **Unoptimized Models**: We did not optimize other machine learning models like SVM and XGBoost, which had good accuracy. These models could potentially yield better results with optimization.
- **Branch Miss Ratio**: The performance of the Decision Tree model can be further optimized by reducing the branch miss ratio, which involves reducing the number of if-else conditions.

7.1.2 UMAFall Dataset

- **Unexplored Models**: We only optimized the Decision Tree model and did not explore other algorithms. Other models could potentially yield better results with optimization.
- **Risk Factor**: We did not consider the Risk factor, which was suggested by another study [15]. Incorporating this factor could potentially improve the model's performance.

7.2 Future Work

Despite these limitations, our research opens up several exciting avenues for future work. The lessons learned from our study can guide future research efforts and help overcome the challenges identified. The potential areas for future work, discussed in the following sections, represent promising directions for advancing the field of edge computing with machine learning.

7.2.1 HAPT Dataset

- **Feature Engineering**: Future research should explore better feature engineering methods to improve the accuracy of the Decision Tree model.
- **Random Forest**: Future research should explore the optimization of the Random Forest model, which often performs better than the Decision Tree model.
- **Logistic Regression Optimization**: Further feature reduction and floating-point arithmetic optimizations could improve the performance of the Logistic Regression model.
- **Other Models**: Future research should explore and optimize other machine learning models like SVM and XGBoost in terms of memory, storage, and floating point calculations.
- **Branch Miss Ratio**: Future research should explore ways to reduce the branch miss ratio in the Decision Tree model considering low-powered MCUs with very limited cache/sram size.

7.2.2 UMAFall Dataset

- **Other Models**: Future research should explore and optimize other machine learning models for the UMAFall dataset.
- **Risk Factor**: Future research should consider the Risk factor suggested by the original study.
- **Improved Accuracy**: Since the Decision Tree model has good accuracy on this dataset, future research could explore using RandomForest or XGBoost or other algorithms to further improve the accuracy.

It is clear that while our study has its limitations, it provides a solid foundation for future research in this area. The strategies and insights gained from this study could guide the development of more efficient and effective machine learning solutions for edge computing. Future work could explore other optimization strategies, machine learning models, and hardware platforms and floating point arithmetic units to further advance this field.

7.3 Comparative Analysis

This section presents a comparative study of the topics at hand. The objective is to analyze and understand the similarities and differences between the two subjects under consideration. This comparative study is divided into two main sections, HAR and FD.

7.3.1 Used Edge Devices

The below table shows the key differences between the Edge devices that we used.

Specification	**Raspberry Pi 1 A** [52]	**ESP32 AI Thinker** [47]	**Arduino UNO** [48]
Compute	700 MHz	240 MHz	16 MHz
RAM	256 MiB	320 KiB	2 KiB
Power consumption	200 mW	240 mW	95 mW
Price	24.95 USD	23 USD	24 USD

Table 7.1: Comparison of Raspberry Pi 1 A, ESP32 AI Thinker and Arduino UNO

7.3.2 Comparative Study

In Table 7.2 and Table 7.3, we mainly focused on 4 metrics: Model, Compute, Accuracy, and F1 Score. The machine learning model and the hardware (Board) used are crucial for assessing computational efficiency. Performance is typically evaluated using Accuracy, which measures the proportion of correct predictions, and F1 score, which balances precision and recall. These metrics provide a comprehensive view of a model's performance, especially in cases of imbalanced data.

HAR or Human Activity Recognition

We have compiled metrics from previous works on Human Activity Recognition (HAR) on HAPT dataset from multiple sources in the below Table 7.2.

Author	Model	Board (compute)	Accuracy	F1 score
Daghero et al.,2022 [29]	CNN	Single-core PULPissimo	86%	Not specified
Mekruk-savanich et al.,2022 [34]	ResNet, ResNetSE, ResNetSE & SERes-NetBi-GRU	Google Colab Pro+ platform	97.87%, 97.89%, 97.43% & 98.03%	93.89%, 93.81%, 93.33% & 94.09%
Daghero et al.,2022 [30]	DT & CNN	Single-core RISC-V MCU	87%	Above 85%
Thu et al.,2021 [24]	HiHAR	Not specified	97.98%	Not specified
This study	**LR & DT**	**ESP32 (240 MHz) & Arduino UNO (16 MHz)**	**94% & 85%**	**87% & 70%**

Table 7.2: Comparative Study of HAPT Dataset

We utilized Logistic Regression (LR) and Decision Tree (DT) models for Human Activity Recognition (HAR), achieving accuracies of 94% and 85%, and F1 scores of 87% and 70%, respectively. While comparing with other studies, it is important to note that each study may have used different hardware platforms. However, Our models provide a balance between good accuracy, F1 score, precision, computational, and memory efficiency, which is crucial for real-time HAR applications.

FD or Fall Detection

Similar to subsection 7.3.2, we have compiled metrics from previous works on Fall Detection on UMAFall dataset from multiple sources in the below Table 7.3.

Author	Model	Board (compute)	Accuracy	F1 score
Garcia et al.,2023 [43]	SVM	Not specified	95.5%	Not specified
Mekruksavanich et al.,2022 [35]	FallNeXt model	Google Colab Pro+ platform	99.12%	98.87%
Ramachandran et al.,2018 [15]	KNN, Naive Bayes, SVM & ANN	Not specified	82.2%, 57.26%, 63.27% & 67.1%	Not specified
This study	**DT**	**ESP32 (240 MHz) & Arduino UNO (16 MHz)**	**81%**	**66%**

Table 7.3: Comparative Study of UMAFall Dataset

For FD, we employed a Decision Tree (DT) model, achieving an accuracy of 81% and an F1 score of 66% on the UMAFall dataset. It is important to note that direct comparisons with other studies may not be entirely fair due to the use of different hardware platforms. However, our DT model offers several advantages that make it stand out. Firstly, our model is highly compute, power, and cost efficient demonstrated by its successful implementation on both ESP32 (240 MHz) and Arduino UNO (16 MHz) where the execution times are 37 and 121 micro seconds respectively. This hardware compatibility is a significant advantage in real-world applications where latency, compactness, and resource efficiency are of interest. Lastly, the DT model is known for its simplicity and interpretability. Unlike more complex models such as SVM or Neural Networks, DTs are easier to understand, visualize, and fine-tune, making them more accessible for further improvement.

Chapter 8

Conclusion

In conclusion, our research represents a significant stride in the field of edge computing with machine learning. We have demonstrated that it is not only feasible but also efficient to deploy complex machine learning models on low-powered devices such as ESP32 and Arduino UNO.

Our optimization strategies have proven effective in maintaining high accuracy and precision while significantly reducing memory consumption, flash size and increasing power efficiency. This achievement opens up the possibility of integrating additional programs and functionalities alongside the deployed machine learning model, thereby expanding the scope of applications.

Moreover, our research underscores the importance of careful feature selection, model optimization, and hardware consideration in deploying machine learning models on microcontroller units. The strategies and insights gained from our work can guide future research in this area, contributing to the advancement of edge computing applications. The ultimate goal is to push the boundaries of what's possible in edge computing with machine learning, paving the way for real-time, on-device machine learning applications that are efficient, effective, and accessible.

References

[1] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[2] S. Menard, *Applied Logistic Regression Analysis*. SAGE Publications, 2001.

[3] C.-Y. J. Peng, K. L. Lee, and G. M. Ingersoll, "An introduction to logistic regression analysis and reporting," *The Journal of Educational Research*, vol. 96, no. 1, pp. 3–14, 2002. DOI: 10.1080/00220670209598786. [Online]. Available: https://doi.org/10.1080/00220670209598786.

[4] B. Florentz and M. Huhn, "Embedded systems architecture: Evaluation and analysis," in *International Conference on Quality of Software Architectures*, Springer, 2006, pp. 145–162.

[5] F. Vahid and T. Ghodoussi, *An introduction to embedded systems: A programmer's perspective*, 2nd ed. Springer, 2007.

[6] J. Ali, R. Khan, N. Ahmad, and I. Maqsood, "Random forests and decision trees," Sep. 2012. [Online]. Available: https://www.researchgate.net/publication/259235118_Random_Forests_and_Decision_Trees.

[7] K. Murphy, *Machine Learning: A Probabilistic Perspective* (Adaptive Computation and Machine Learning series). MIT Press, 2012, ISBN: 9780262018029. [Online]. Available: https://books.google.com.bd/books?id=psuMEAAAQBAJ.

[8] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, *Human activity recognition using smartphones*, 2013. arXiv: 1401.8212 [cs.HC].

[9] D. W. Hosmer, S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression*, 3rd ed. Wiley, 2013.

[10] A. Priyam, Abhijeet, R. Gupta, A. Rathee, and S. Srivastava, "Comparative analysis of decision tree classification algorithms," *International Journal of Current Engineering and Technology*, vol. 3, no. 2, pp. 334–337, 2013. [Online]. Available: https://inpressco.com/comparative-analysis-of-decision-tree-classification-algorithms/.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, urlhttp://www.deeplearningbook.org.

[12] E. Casilari, J. A. Santoyo-Ramón, and J. M. Cano-García, "Umafall: A multisensor dataset for the research on automatic fall detection," *Procedia Computer Science*, vol. 110, pp. 32–39, 2017, 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.

2017.06.110. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050917312899.

[13] R. Zhao, W. Luk, X. Niu, H. Shi, and H. Wang, "Hardware acceleration for machine learning," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017, pp. 645–650. DOI: 10.1109/ISVLSI.2017.127.

[14] "Machine learning-based techniques for fall detection in geriatric healthcare systems," in *2018 9th International Conference on Information Technology in Medicine and Education (ITME)*, IEEE, 2018. DOI: 10.1109/ITME.2018.00059. [Online]. Available: https://ieeexplore.ieee.org/document/8589291/.

[15] A. Ramachandran, A. R., P. Pahwa, and A. K.R., "Machine learning-based techniques for fall detection in geriatric healthcare systems," in *2018 9th International Conference on Information Technology in Medicine and Education (ITME)*, 2018, pp. 232–237. DOI: 10.1109/ITME.2018.00059.

[16] S. B. Kwon, J.-H. Park, C. Kwon, H. J. Kong, J. Y. Hwang, and H. C. Kim, "An energy-efficient algorithm for classification of fall types using a wearable sensor," *IEEE Access*, vol. 7, pp. 31 321–31 329, 2019. DOI: 10.1109/ACCESS.2019.2902718.

[17] S. J. Mohaiminul Islam Guorong Chen, "An overview of neural network," *American Journal of Neural Networks and Applications*, vol. 5, no. 1, pp. 7–11, 2019. DOI: 10.11648/j.ajnna.20190501.12. [Online]. Available: https://doi.org/10.11648/j.ajnna.20190501.12.

[18] L. Oden and T. Witt, "Fall-detection on a wearable micro controller using machine learning algorithms," in *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2020, pp. 296–301. DOI: 10.1109/SMARTCOMP50058.2020.00067.

[19] D. Preuveneers, I. Tsingenopoulos, and W. Joosen, "Resource usage and performance trade-offs for machine learning models in smart environments," *Sensors*, vol. 20, no. 4, 2020, ISSN: 1424-8220. DOI: 10.3390/s20041176. [Online]. Available: https://www.mdpi.com/1424-8220/20/4/1176.

[20] M. Azad, I. Chikalov, S. Hussain, M. Moshkov, and B. Zielosko, "Decision rules derived from optimal decision trees with hypotheses," *Entropy*, vol. 23, no. 12, 2021, ISSN: 1099-4300. DOI: 10.3390/e23121641. [Online]. Available: https://www.mdpi.com/1099-4300/23/12/1641.

[21] J. Berner, P. Grohs, G. Kutyniok, and P. Petersen, "The modern mathematics of deep learning," *CoRR*, vol. abs/2105.04026, 2021. arXiv: 2105.04026. [Online]. Available: https://arxiv.org/abs/2105.04026.

[22] J. Brownlee, "One-vs-rest and one-vs-one for multi-class classification," 2021, Accessed: 2024-01-21. [Online]. Available: https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/.

[23] N. Gupta, "Chapter one - introduction to hardware accelerator systems for artificial intelligence and machine learning," in *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*, ser. Advances in Computers, S. Kim and G. C. Deka, Eds., vol. 122, Elsevier, 2021, pp. 1–21. DOI: https://doi.org/10.1016/bs.adcom.2020.07.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245820300541.

[24] N. T. Hoai Thu and D. S. Han, "Hihar: A hierarchical hybrid deep learning architecture for wearable sensor-based human activity recognition," *IEEE Access*, vol. 9, pp. 145 271–145 281, 2021. DOI: 10.1109/ACCESS.2021.3122298.

[25] *Pytorch*, Accessed: 2024-01-11, 2021. [Online]. Available: https://pytorch.org/.

[26] T. R. D. Sa and C. M. S. Figueiredo, "Classification of anomalous gait using machine learning techniques and embedded sensors," *arXiv:2110.06139*, 2021. [Online]. Available: https://arxiv.org/abs/2110.06139.

[27] S. Sudirman, Z. Zainuddin, and A. Suyuti, "Fall detection in the elderly with android mobile iot devices using nodemcu and accelerometer sensors," 2021.

[28] *Tensorflow lite*, Accessed: 2024-01-11, 2021. [Online]. Available: https://www.tensorflow.org/lite.

[29] F. Daghero, A. Burrello, C. Xie, *et al.*, "Human activity recognition on microcontrollers with quantized and adaptive deep neural networks," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 4, Aug. 2022, ISSN: 1539-9087. DOI: 10.1145/3542819. [Online]. Available: https://doi.org/10.1145/3542819.

[30] F. Daghero, D. Jahier Pagliari, and M. Poncino, "Two-stage human activity recognition on microcontrollers with decision trees and cnns," *arXiv preprint arXiv:2206.07652*, 2022. [Online]. Available: https://arxiv.org/abs/2206.07652.

[31] R. M. C. Lucas B.V. de Amorim George D.C. Cavalcanti, "The choice of scaling technique matters for classification performance," 2022. DOI: 10.48550/arXiv.2212.12343. arXiv: 2212.12343 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2212.12343.

[32] H. Mankodiya, D. Jadav, R. Gupta, *et al.*, "Xai-fall: Explainable ai for fall detection on wearable devices using sequence models and xai techniques," *Mathematics*, vol. 10, no. 12, p. 1990, 2022. DOI: 10.3390/math10121990. [Online]. Available: https://www.mdpi.com/2227-7390/10/12/1990.

[33] L. M. Martins, N. F. Ribeiro, F. Soares, and C. P. Santos, "Inertial data-based ai approaches for adl and fall recognition," *Sensors*, vol. 22, no. 11, p. 4028, 2022. DOI: 10.3390/s22114028. [Online]. Available: https://doi.org/10.3390/s22114028.

[34] S. Mekruksavanich, N. Hnoohom, and A. Jitpattanakul, "A hybrid deep residual network for efficient transitional activity recognition based on wearable sensors," *Applied Sciences*, vol. 12, no. 10, p. 4988, 2022. DOI: 10.3390/app12104988. [Online]. Available: https://doi.org/10.3390/app12104988.

[35] S. Mekruksavanich and A. Jitpattanakul, "Fallnext: A deep residual model based on multi-branch aggregation for sensor-based fall detection," *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, vol. 16, no. 4, pp. 352–364, Sep. 2022. DOI: 10.37936/ecti-cit.2022164.248156. [Online]. Available: https://ph01.tci-thaijo.org/index.php/ecticit/article/view/248156.

[36] C. Contoli and E. Lattanzi, "A study on the application of tensorflow compression techniques to human activity recognition," *IEEE Access*, vol. 11, pp. 48 046–48 058, 2023. DOI: 10.1109/ACCESS.2023.3276438.

[37] S.-l. developers, *Release highlights for scikit-learn 1.3*, https://scikit-learn.org/stable/auto_examples/release_highlights/plot_release_highlights_1_3_0.html, Accessed: 2024-01-07, 2023.

[38] T. developers, *Tensorflow 2.13.0*, https://github.com/tensorflow/tensorflow/releases/tag/v2.13.0, Accessed: 2024-01-07, 2023.

[39] EloquentArduino, *Micromlgen: A library to port sklearn decision trees and random forests to c*, https://github.com/eloquentarduino/micromlgen, Accessed: 2024-01-07, 2023.

[40] W. N. Ismail, H. A. Alsalamah, M. M. Hassan, and E. Mohamed, "Auto-har: An adaptive human activity recognition framework using an automated cnn architecture design," *Heliyon*, vol. 9, no. 2, e13636, 2023, ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2023.e13636. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405844023008435.

[41] C. Perate, *Docker qemu rpi os*, https://github.com/carlosperate/docker-qemu-rpi-os, 2023.

[42] U. M. L. Repository, *Smartphone-based recognition of human activities and postural transitions data set*, 2023. [Online]. Available: https://archive.ics.uci.edu/dataset/341/smartphone+based+recognition+of+human+activities+and+postural+transitions.

[43] J. C. Ruiz-Garcia, R. Tolosana, R. Vera-Rodriguez, and C. Moro, "Carefall: Automatic fall detection through wearable devices and ai methods," *arXiv preprint arXiv:2307.05275*, 2023. [Online]. Available: https://arxiv.org/abs/2307.05275.

[44] JetBrains, *Clion: A cross-platform ide for c and c++ by jetbrains*, Accessed on: January 16, 2024, 2024. [Online]. Available: https://www.jetbrains.com/clion/.

[45] PlatformIO, *Espressif 32 — platformio latest documentation*, Accessed: 2024-01-05, 2024. [Online]. Available: https://docs.platformio.org/en/latest/platforms/espressif32.html#boards.

[46] PlatformIO, *Platformio: Professional collaborative platform for embedded development*, Accessed on: January 16, 2024, 2024. [Online]. Available: https://platformio.org/.

[47] E. Systems, *Esp-eye development board overview*, Jan. 2024. [Online]. Available: https://www.espressif.com/en/products/devkits/esp-eye/overview.

[48] Arduino, *Arduino uno rev3*, https://docs.arduino.cc/hardware/uno-rev3/, Accessed: 2024-01-06.

[49] *Gdb user manual*, Accessed: 2024-01-09. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_toc.html.

[50] *Gnu time*, Accessed: 2024-01-09. [Online]. Available: https://www.gnu.org/software/time/.

[51] *Perf wiki*, Accessed: 2024-01-09. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page.

[52] *Raspberry pi 1 model a+*, https://www.raspberrypi.com/products/raspberry-pi-1-model-a-plus/, Accessed: 2024-01-06.

[53] *sklearn.feature_selection.RFE {–} scikit-learn 1.4.0 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html, Accessed: January 22, 2024.

[54] *sklearn.linear_model.LogisticRegression {–} scikit-learn 1.4.0 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html, Accessed: January 22, 2024.