PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL SCHILD REUSCH

DEPLOYING ARTIFICIAL INTELLIGENCE IN RESOURCE-CONSTRAINED DEVICES FOR HUMAN ACTIVITY RECOGNITION

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

DEPLOYING ARTIFICIAL INTELLIGENCE IN RESOURCE-CONSTRAINED DEVICES FOR HUMAN ACTIVITY RECOGNITION

RAFAEL SCHILD REUSCH

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2023

Ficha Catalográfica

R445a Reusch, Rafael Schild

Aplicação De Inteligência Artificial Em Dispositivos Com Recursos Limitados Para Reconhecimento De Atividades Humanas / Rafael Schild Reusch. – 2023.

85 p.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fernando Gehm Moraes.

1. Inteligencia Artificial. 2. Reconhecimento de Atividades Humanas. 3. Baixo Consumo. I. Moraes, Fernando Gehm. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS com os dados fornecidos pelo(a) autor(a).

Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

RAFAEL SCHILD REUSCH

DEPLOYING ARTIFICIAL INTELLIGENCE IN RESOURCE-CONSTRAINED DEVICES FOR HUMAN ACTIVITY RECOGNITION

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on August 17, 2023.

COMMITTEE MEMBERS:

Prof. Dr. Everton Carara (PPGCC/UFSM)

Prof. Dr.César Augusto Missio Marcon (PPGCC/PUCRS)

Prof. Dr. Fernando Gehm Moraes (PPGCC/PUCRS - Advisor)

APLICAÇÃO DE INTELIGÊNCIA ARTIFICIAL EM DISPOSITIVOS COM RECURSOS LIMITADOS PARA RECONHECIMENTO DE ATIVIDADES HUMANAS

RESUMO

A inteligência artificial (IA) tem se mostrado eficaz na resolução de tarefas complexas, como o reconhecimento de atividades humanas e de fala. No entanto, a introdução de modelos de IA orientados para a precisão trouxe novos desafios em relação à sua aplicabilidade em sistemas com recursos limitados. No reconhecimento de atividades humanas (RAH), as abordagens estado-da-arte frequentemente dependem de redes complexas com múltiplas camada, do tipo LSTM. A literatura sugere que as redes LSTM são adequadas para tratar dados de séries temporais, um aspecto crucial do RAH. A maioria dos trabalhos na literatura se concentra principalmente em alcançar a maior precisão possível, com apenas alguns considerando o custo computacional associado à execução da fase de inferência. No RAH, dispositivos de Internet das Coisas (IoT) de baixo consumo, como matrizes de sensores vestíveis, são frequentemente usados como dispositivos de coleta de dados. No entanto, observamos um esforço limitado para implantar a tecnologia de IA diretamente nesses dispositivos. Em vez disso, a abordagem predominante envolve o uso de arquiteturas de computação em borda ou em nuvem, em que o papel do dispositivo final é coletar e enviar dados para dispositivos em borda/nuvem. Assistentes de voz populares, como o Alexa da Amazon e o Assistente do Google, comumente empregam essa abordagem arquitetural. No entanto, em aplicações do mundo real, especialmente em setores como saúde, depender exclusivamente de dispositivos em borda/nuvem muitas vezes não é viável, pois esses dispositivos nem sempre estão disponíveis ou acessíveis. Este trabalho tem como objetivo fornecer uma Rede Neural Convolucional ajustada para uso em sistemas embarcados com recursos limitados, utilizando técnicas de otimização e modelagem eficiente de redes neurais. Após ajustar o modelo de CNN no framework PyTorch, apresentamos um modelo equivalente em C. Utilizamos técnicas de otimização, como representação inteira para evitar unidades de ponto flutuante (FPUs), quantização de parâmetros e compressão de conjunto de dados. Os resultados mostram que, em comparação com a CNN de referência, o modelo otimizado reduziu o tamanho do modelo de CNN em 2,34 vezes e melhorou a precisão de 74% para 85,2%. Este modelo otimizado pode ser executado em dispositivos com recursos limitados sem exigir FPUs e grandes memórias para armazenar os parâmetros da CNN.

Palavras-Chave: Inteligência Artificial, Aprendizado de Máquina, Reconhecimento de Atividades Humanas, Sistemas Embarcados, Dispositivos com Restrições de Recursos.

DEPLOYING ARTIFICIAL INTELLIGENCE IN RESOURCE-CONSTRAINED DEVICES FOR HUMAN ACTIVITY RECOGNITION

ABSTRACT

Artificial intelligence (AI) has proven highly effective in solving complex tasks such as human activity and speech recognition. However, introducing accuracy-driven AI models has brought new challenges regarding their applicability in resource-constrained systems. In Human Activity Recognition (HAR), current state-of-the-art approaches often rely on complex multi-layer LSTM networks. The literature suggests that LSTM networks are well-suited for handling temporal-series data, a crucial aspect of HAR. Most existing works in the literature focus primarily on achieving the highest possible accuracy, with only a few considering the overall computational cost associated with running the inference phase. In HAR, low-power IoT devices, such as wearable sensor arrays, are frequently used as data-gathering devices. However, we observed a limited effort to deploy AI technology directly on these devices. Instead, the prevailing approach involves using edge or cloud computing architectures, where the end device's role is to collect and send data to edge/cloud devices. Popular voice assistants like Amazon's Alexa and Google Assistant commonly employ this architectural approach. However, in real-life applications, especially in industries like healthcare, relying solely on edge/cloud devices is often not feasible, as these devices may not always be available or accessible. This work aims to provide a Convolutional Neural Network tuned for resource-scarce embedded systems, using optimization and efficient neural network modeling techniques. After tuning the CNN model in the Pytorch framework, we present an equivalent C model. We employ optimization techniques such as integer representation to avoid floating point units (FPUs), parameter quantization, and dataset compression. The results show that compared to the baseline reference CNN, the optimized model reduced the CNN model 2.34 times and improved accuracy from 74.9% to 85.2%. Such an optimized model may run on resource-constrained devices without requiring FPUs and large memories to store the CNN parameter.

Keywords: Artificial Intelligence, Machine Learning, Human Activity Recognition, Embedded Systems, Constrained Devices.

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

LIST OF ACRONYMS

ADL – Activities of Daily Living

AI – Artificial Intelligence

ANN – Artificial Neural Network

APM – Application Performance Management

ARM – Advanced RISC Machine

AR – Activity Recognition

CAM – Class Action Mapping

CCTV – Closed Circuit Television

CNN – Convolutional Neural Network

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

DNN – Deep Neural Network

DVR – Digital Video Recorder

FC – Fully Connected

RELU – Rectified Linear Unit

GPU – Graphical Processing Unit

GRU – Gated Recurrent Unit

HAR – Human Activity Recognition

ICD-10 – International Classification of Disease-10

ICD-9 – International Classification of Disease-9

IOT – Internet of Things

KNN – K-Nearest Neural Network

LSTM – Long Short-Term Memory

MLP – Multi-Layer Perceptron

ML – Machine Learning

OVP – Open Virtual Platform

PCA – Principal Component Analysis

RF – Random Forest

RNN – Recurrent Neural Network

RTL – Register Transfer Level

SVM – Support Vector Machine

CONTENTS

1. INTRODUCTION

With the ever-increasing elderly population, falls were accepted as an illness for older adults in the International Classification of Disease-9 (ICD-9) and ICD-10 [Zeng et al., 2014]. Monteiro et al. [2021] mention that in Brazil, more than 30% of death along people older than 60 are due to a hard fall. Much is done to prevent emergencies in elderly homes, from bathroom hold bars to 24/7 nurse care. In a real-world environment, it's impossible to prevent all emergencies, thus the need to detect these situations in real time and act accordingly.

Human Activity Recognition (HAR) is a research area that has received significant attention in recent years, thanks in part to the proliferation of mobile phones and IoT sensors equipped with complex sensor arrays such as accelerometers and magnetometers. Embedded devices have been identified as crucial components in HAR. Data gathering for HAR can be conducted through wearable sensors like smartwatches and smart bracelets. With the ubiquity of smartphones, even within the elderly demographic, recent studies have used these devices to record and categorize human activity [Zeng et al., 2014].

Artificial intelligence (AI) has effectively solved various tasks in industries, especially areas with complex correlations, such as speech recognition and medical diagnosis. Recent studies proposed different neural network models applied in HAR tasks, including complex architectures that presented low real-life applicability due to the computational costs demanded [Park et al., 2023]. This is an issue in various AI tasks. More complex tasks such as speech recognition for virtual assistants are mainly cloud-based [Janak et al., 2021], where local devices only function is to work as input data for the neural network located off-site in a dense server dedicated to AI.

Speech recognition is similar to HAR since both contain high temporal correlation, where past processed data affect the current output. LSTM neural networks have been widely used in these tasks and currently represent the state-of-the-art accuracy in speech recognition and HAR. LSTM contains complex dependencies that require resources scarce in most embedded systems. Recent approaches using battery-powered devices suggest using Edge Computing concepts to transfer the task of processing the inference to devices with more power availability, using embedded devices only for data gathering [Zeng et al., 2014]. This proposal neglects the situation where edge devices are unavailable, consequently creating gaps in the processed data array, increasing the chance of the system not detecting emergencies in real time.

Recent advances in Convolutional Neural Networks (CNN) extended the usability of AI powered-devices. Person detection can be found inside various camera surveillance systems (CCTV). Even though running the inference algorithm is commonly done on Digital

Video Recorders (DVR), which are high-powered devices, modern cameras use embedded AI for better response times [Intelbras, 2020].

1.1 Motivation

The motivation for this work arises from two primary sources: (*i*) the Author's prior experience with embedded devices and their inherent limitations; (*ii*) the challenges posed by current state-of-the-art approaches for HAR.

The Author's background in embedded systems, particularly in the healthcare sector, revealed that many systems fall short in terms of real-world applicability due to the lack of robust and scalable architectures. AI implementations on resource-constrained systems often resolve this issue by offloading the processing step to edge or cloud devices, but this solution imposes severe limitations on system robustness, especially in critical fields such as healthcare.

The prevailing focus in HAR research is accuracy. While most studies use embedded systems for data recording, many fail to assess resource requirements and the suitability of different embedded devices for AI deployment.

1.2 Objectives

The strategic objective of this work is to provide a Convolutional Neural Network tuned for use in resource-scarce embedded systems, using optimization and efficient neural network modeling techniques. The specific objectives of this work consist of:

1. Optimize the accuracy of a reference 1D CNN Model for HAR.
2. Develop a C model for the reference CNN. The C language is a common choice for programming embedded systems due to the high degree of control over system resources, the small memory footprint, and the fact that C compilers create compact and efficient code.
3. Optimize the CNN C model for power-constrained embedded devices, targeting processors not requiring floating point units.
4. Optimize the CNN C model in terms of memory footprint.

1.3 Methodology

The Pytorch framework, described in Section 4.1, is used as the modeling Neural Network tool for the present work. This work adopts as a baseline model a 1D CNN. This reference model, described in Section 4.2, is modified to achieve the strategic objective of this work. Using the C language to model the CNNs, we evaluate the computational cost of the inference phase using a RISC-V processor in a virtual platform (OVP). The profile of the execution of the CNN model guides the optimizations, such as the adoption of integer representation, quantization, and parameter compression.

1.4 Document organization

This work is organized as follows:

- Chapter 2 describes basic concepts related to AI and HAR, required to follow this work.
- Chapter 3 presents the state-of-the-art of machine learing in HAR tasks, positioning this work w.r.t. the literature.
- Chapter 4 presents the CNN reference model and optimizations made using the Pytorch framework. This chapter meets the first specific goal.
- Chapter 5 presents the C model and the evaluation of using integer representation. This chapter meets the second and third specific goals.
- Chapter 6 presents the method adopted to compress the CNN parameters, weights and bias, to reduce the CNN model size. This chapter meets the fourth specific goal.
- Chapter 7 concludes this work, pointing out directions for future work.

2. BASIC CONCEPTS

Activity recognition is a research field that can directly impact areas such as senior care and fitness monitoring. It is one of the most contemporary research topics, which is not surprising considering the growth of the senior population worldwide. Artificial intelligence is a promising way to deal with the activity recognition complexity.

This chapter introduces concepts related to Human Activity Recognition (HAR) and methods adopted to treat this research field. Section 2.1 defines HAR, presenting treatment approaches. Section 2.2 introduces Machine Learning (ML) techniques used in HAR. Section 2.3 presents Convolutional Neural Networks with their phases and layers.

2.1 Human Activity Recognition (HAR)

Human Activity Recognition is the process in which data is analyzed and processed to determine the person's activities, such as walking, running, sitting, and showering. Historically, sensor data of human activity was scarce, and retrieval was expensive and complex. Currently, where smartphones and wearable devices are affordable and filled with different sensors such as accelerometers and gyroscopes, data can be easily obtained and recorded [Mekruksavanich and Jitpattanakul, 2021].

There are two approaches to solve the HAR problem:

- **Vision-based:** usage of devices such as cameras or other image-sensing equipment;
- **Sensor-based:** most common approach, processing data from sensors such as accelerometer, gyroscope, and magnetometer.

The activities to be recognized, known as Activities of Daily Living, first defined by [Katz et al., 1970], can be used in eldercare and healthcare when combined with IoT devices to detect and prevent an emergency like a fall detection. HAR also includes medical diagnosis and smart home automation. Activity data can be obtained from wearable devices such as smartwatches, other devices type such as smartphones, or specific IoT sensors. Shi et al. [2020] show that combining these devices can improve accuracy.

2.2 Machine Learning for HAR

HAR is essentially a time series classification problem in which data should be processed in time frames, resembling signal processing problems. The classification involves

predicting the movement of a person based on raw sensor data. In traditional AI approaches, it requires knowledge of signal processing to correctly extract features from data to use it as input to a machine learning model.

HAR can also be treated as a pattern recognition problem using ML approaches. Examples of ML approaches used for HAR include decision trees, support vector machines (SVM), and Markov models. Nowadays, three main types of supervised machine learning techniques are used to implement pattern recognition solutions (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8372231/):

- Multi-Layer Perceptrons (**MLP**) - is a class of a feedforward artificial neural network (ANN). MLP models are the most basic deep neural network, which is composed of a series of fully connected layers.
- Convolutional Neural Networks (**CNN**) - in CNN models, one or multiple convolution layers extract simple features from input by executing convolution operations. CNNs are detailed at Section 2.3.
- Recurrent Neural Networks (**RNN**) - use sequential data feeding. RNNs have been developed to address the time-series problem of sequential input data. Examples of RNN include Long short-term memory (LSTM), LSTM Bidirectional, and Gated recurrent units (GRUs). LSTM are detailed in the next paragraphs.

These techniques can be implemented using a large number of layers, which is called DNN approach. Also, according to Hayat et al. [Hayat et al., 2022], the following ML methods applied to HAR, besides MLP:

- Decision Trees. A “random forest” (RF) is a collection of decision trees aggregating partial results into one final result. RF has a similar concept to Multi-Class SVM (Support Vector Machines), using a sequence of decision trees to give a multi-class result. RF requires larger labeled training data to increase the prediction precision than other traditional approaches.
- k-NN (k-nearest neighbors). It is a supervised learning method used for classification and regression where the neural network searches for the most similar data points to classify. The distance between points is commonly calculated using the Euclidean distance formula. The nearest points are considered to be in the same class.
- SVM. This method is applied to binary problems. Studies have shown good accuracy to create Multi-Class SVMs. This method divides the Multi-Class SVM into multiple small binary problems.
- LSTM. Has advantages when used in time-series problems due to their ability to store past information. This storage of past information enables LSTM to capture temporal

information from sequential data inputs. This feature allowed great accuracy compared to other approaches [Hayat et al., 2022]. Figure 2.1 shows a LSTM unit. It contains a cell with input, output, and forget gates. This cell is responsible for storing and reading values over time, working as shared memory.

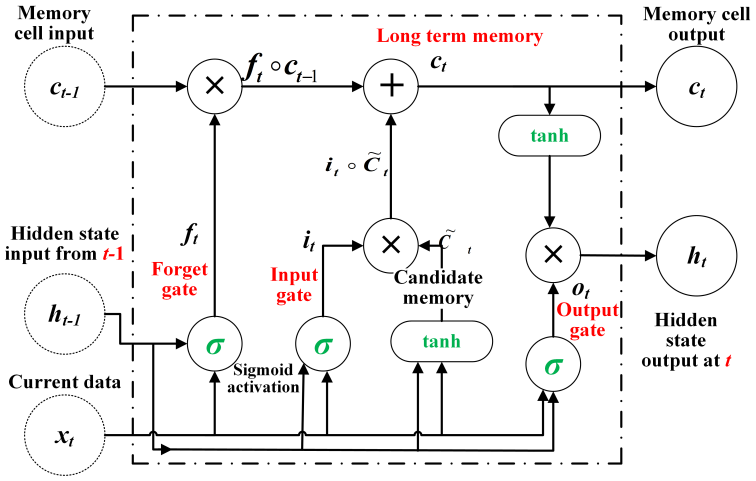


Figure 2.1 – LSTM unit [Zebin et al., 2018].

Table 2.1 [Hayat et al., 2022] compares the accuracy of the previously mentioned methods. The complexity and overhead of the LSTM implementations, which is composed by the processing time and memory footprint, lead to hybrid approaches mixing CNNs with LSTMs [Xu et al., 2019]. The goal is to combine the CNN feature extraction properties with the time series processing ability of LSTM.

Table 2.1 – Accuracy of ML methdos applied to HAR [Hayat et al., 2022].

Activity	RF (%)	k-NN (%)	SVM (%)	ANN (%)	LSTM (%)
Walking	85.60	86.31	89.47	93.78	95.08
Going Upstairs	76.02	78.88	82.84	86.50	92.78
Going Downstairs	76.54	75.42	80.10	86.55	94.75
Sitting	87.62	93.81	93.92	95.57	96.85
Standing	87.22	90.54	91.54	94.88	95.94
Laying	83.10	87.34	89.50	90.10	91.77
Overall Performance	**82.68**	**85.38**	**87.90**	**91.23**	**94.53**

Recent advances in AI adopt CNN-only approach to various problems, seeking the overhead reduction during the inference phase while maintaining a state-of-the-art accuracy. CNNs have shown promising results in applications such as image processing in battery-powered devices. In a sensor-based HAR scenario, battery-powered devices are the only option, and only a few studies evaluate how a low-power CNN-only approach would perform in this scenario.

2.3 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a feed-forward neural network. One of the CNNs main goals is to reduce data pre-processing. CNN specializes in processing data with a grid-like topology, such as images. This characteristic leads to the adoption of CNNs by the industry, especially in the health and image-processing areas. Sarvamangala and Kulkarni [2021] presented in his work that CNNs are efficient for the analysis of medical images.

Figure 2.2 shows an example of a CNN, with its phases and operations. According to Figure 2.2, an image classification comprises two phases:

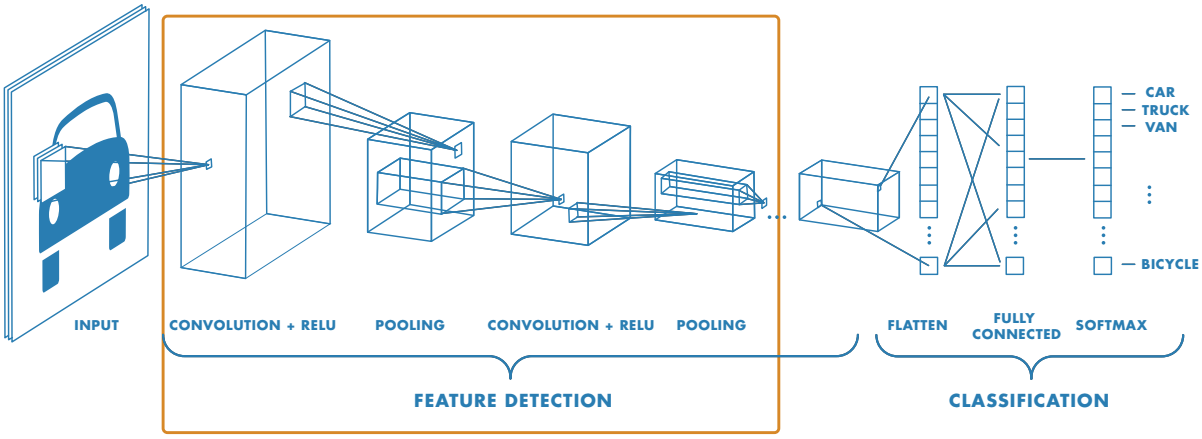


Figure 2.2 – Convolutional Neural Network example [Mathworks, 2021].

- **Feature Detection**: Convolution, ReLu and Pooling layers extract key points of the input data, using a set of filters.
- **Classification**: Flatten, Fully-Connected and Softmax Layers classifies the input data using the extracted features. This phase returns the result of the classification.

The most common used layers in a typical CNN implementation includes:

- **Convolution Layer**. This is the first CNN layer. The convolution layer performs a mathematical operation of convolution between the input data and a set of filters. By moving and performing the operation along the input data, a set of features is obtained, called Feature Map. Figure 2.3 shows the first convolution in a 8x8 image, using a 3x3 filter.
- **Activation Layer**. Activation functions introduce non-linearity to the model which allows it to learn complex functional mappings between the inputs and response variables. In most CNN Implementations, the Rectified Linear Unit (ReLu) is used in the

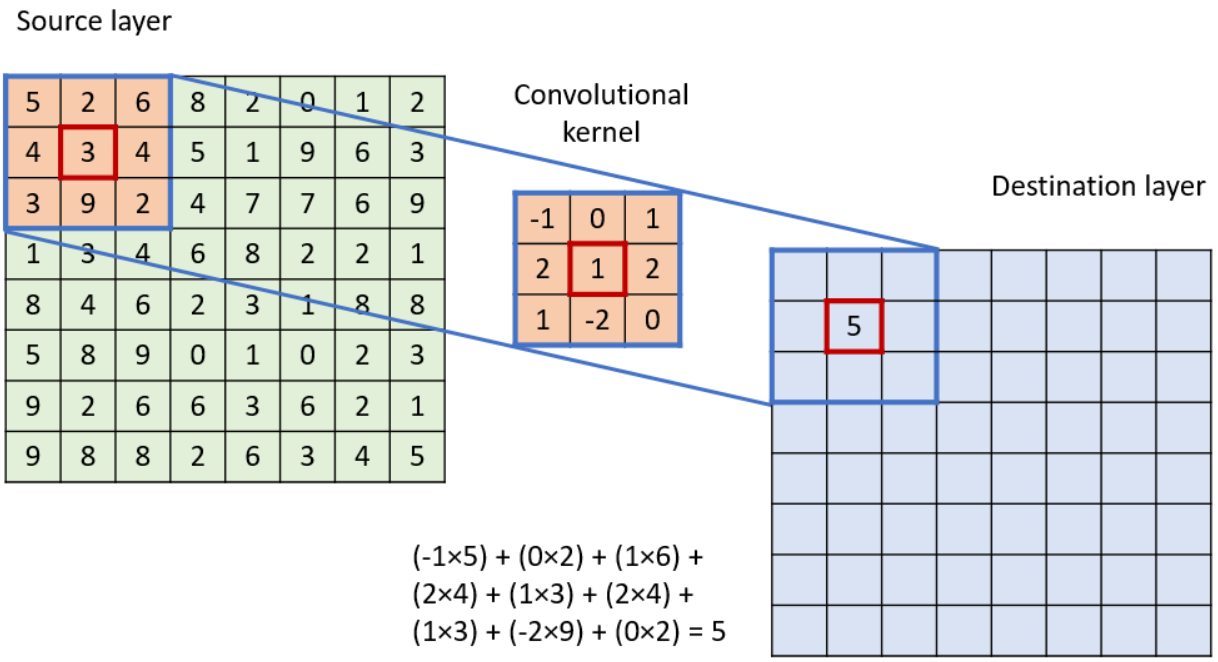


Figure 2.3 – Illustration of a convolution process [Podareanu et al., 2019].

Activation Layer. RelU function is a piecewise linear function that outputs the input directly if is positive, i.e. > 0, otherwise, it will output zero.

- **Pooling Layer**. Its goal is to reduce the size of the feature map, reducing the computational cost of the subsequent layers. There is various pooling methods such as: (*i*) Max Pooling, (*ii*) Average Pooling, (*iii*) Sum Pooling.

- **Fully Connected Layer (FC)**. This layer has two hidden neuron layers, where each neuron is connected to every other neuron in the other hidden layer, where each connection has its respective weight. When needed, a Flatten layer is used before the FC to convert a two dimensional array into a single dimension, as required by the input dimension of the FC.

3. RELATED WORK

Table 3.1 summarizes the related work, where each row color corresponds to an ML method: LSTM in red, CNN in green, and hybrid approaches of CNN and LSTM in blue. The last row presents our work compared to the literature.

Table 3.1 – Related works on ML applied for HAR.

Work	Model	Modeling	Goals
LSTM Networks Using Smartphone Data for Sensor-Based Human Activity Recognition in Smart Homes [Mekruksavanich and Jitpattanakul, 2021]	LSTM	TensorFlow	Compare different LSTM approaches
Human activity recognition from inertial sensor time-series using batch normalized deep LSTM recurrent networks [Zebin et al., 2018]	LSTM	Keras Python	LSTM accuracy for temporal correlation analysis
Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors [Zeng et al., 2014]	CNN	not defined by the Authors	Novel CNN approach to achieve state-of-the-art accuracy in HAR problem
A Fully Onchip Binarized Convolutional Neural Network FPGA Implementation with Accurate Inference [Shimoda et al., 2017]	Binarized CNN	Pytorch	State-of-the-art accuracy in a FPGA using low-power and low-area techniques
Human Activity Recognition Using Cascaded Dual Attention CNN and Bi-Directional GRU Framework [Ullah and Munir, 2023]	CNN + GRU	TensorFlow	Human activity recognition in video streams
Inception inspired CNN-GRU hybrid network for human activity recognition [Dua et al., 2023]	CNN + GRU	TensorFlow	Develop a HAR model that is reasonably accurate and less complex so that it can be later deployed in embedded devices
Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition [Ordóñez and Roggen, 2016]	CNN + LSTM	Lasagne	Improve accuracy of LSTM approach using CNN as Feature Extractor
Towards effective detection of elderly falls with CNN-LSTM neural networks [García et al., 2022]	CNN + LSTM	not defined by the Authors	Accurate fall detection using wearable sensors in the elderly population
MultiCNN-FilterLSTM: Recognition of human activity based on sensor-based resources in IoT applications Park et al. [2023]	CNN + LSTM	Pytorch	Support IoT systems that require a resource-efficient model
A novel vision-based fall detection scheme using keypoints of human skeleton with long short-term memory network Inturi et al. [2023]	CNN + LSTM	not defined by the Authors	Solution for fall detection using vision-based approaches
This work	1D CNN	Pytorch and C models	Integer representation with reduced model size targing processors with limited resources

Section 3.1 overviews feature extraction methods for activity recognition. Section 3.2 and Section 3.3 present LSTM and CNN approaches, respectively. Hybrid approaches are described in Section 3.4. Section 3.5 presents optimization techniques in neural networks. Finally, Section 3.6 presents the related work final remarks and how this work fills the gaps observed in the literature.

3.1 Feature Extraction for Activity Recognition

Zeng et al. [2014] classifies Activity Recognition (AR) as a classification problem, where inputs are frames with time-series data, and the output is an activity class. Many features such as mean, standard deviation, and correlation coefficients are widely used [Figo et al., 2010]. More complex hand-crafted features include Fourier and wavelet transforms, and more recent proposals also include time-delay embeddings. According to the authors, this more recent method adopts a nonlinear time series analysis, with good results, in periodic activities such as cycling or walking [Frank et al., 2010]. However, the approach lacks performance in non-periodic activities. Figure 3.1 shows an example of an activity recognition workflow.

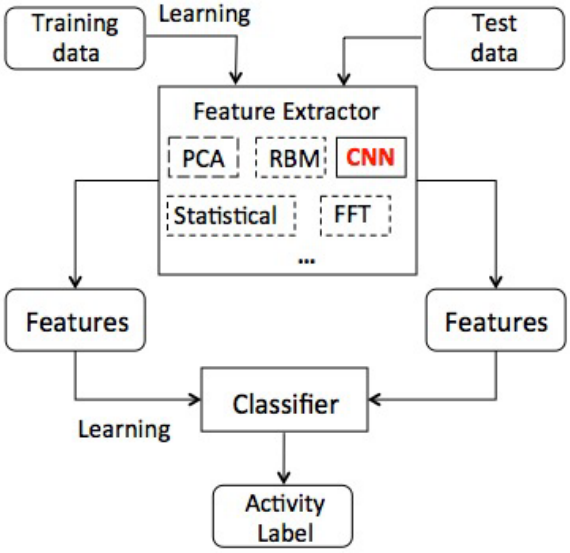


Figure 3.1 – Overview of an activity recognition workflow [Zeng et al., 2014].

Other approaches, such as principal component analysis (PCA), are also commonly used. While other methods have fixed feature extraction techniques, studies have shown that PCA can adapt its feature extraction to the dataset, improving performance.

CNNs have been proposed to extract more meaningful features from raw input data. CNN does not require a hand-crafted method or statistical analysis to retrieve key features from the input dataset. Furthermore, CNN requires much less training data to accurately adapt its feature extraction based on the input data [O'Shea and Nash, 2015].

Traditional statistical feature extraction methods can be considered single-depth, where the output nodes represent the correlation between the predefined methods. Zeng et al. [2014] defines PCA as a single-depth method. Although this approach can find complex correlations in the dataset, the output nodes are still one layer after the features. DNN has a depth of n layers, allowing the model hidden layers to find complex dependencies inside the input data. Figure 3.2 compares statistical, PCA, and DNN topology.

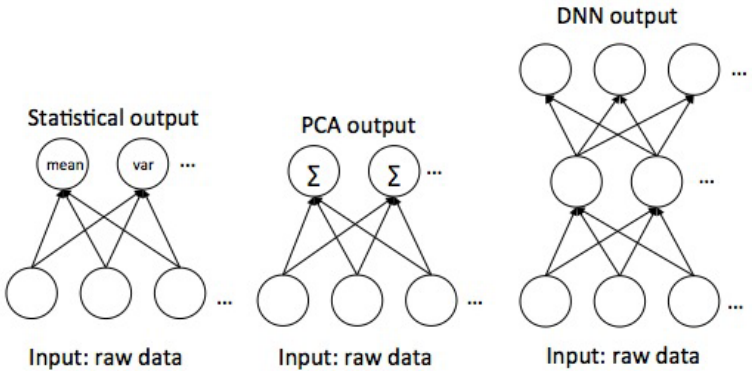


Figure 3.2 – (a) Statistical feature computation, (b) PCA model, (c) DNN model [Zeng et al., 2014].

3.2 LSTM Approaches

Table 2.1 (page 19) presented the accuracy superiority of LSTM over other ML methods for HAR. Standard Recurrent Neural Networks (RNNs) suffer from short-term memory due to a vanishing gradient problem that arises when working with longer data sequences [Zebin et al., 2018]. More advanced versions of RNNs, such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU), can preserve information from earlier sequence parts and carry it forward. LSTM contains “memory cells”, allowing predictions to be made based on prior information [Mekruksavanich and Jitpattanakul, 2021].

Mekruksavanich and Jitpattanakul [2021] compared LSTM architectures in activities of daily living recognition: (*i*) Vanilla LSTM; (*ii*) 2-stacked LSTM; (*ii*) 3-stacked LSTM. The original LSTM model (Vanilla LSTM) consists of an individual LSTM layer followed by a classifier with of a Dropout Layer, a Fully Connected Layer, and a SoftMax layer. The stacked LSTM architectures contain the same classifier layers but include more LSTM layers, taking advantage of the temporal feature extraction of each layer. Figure 3.3 shows these architectures.

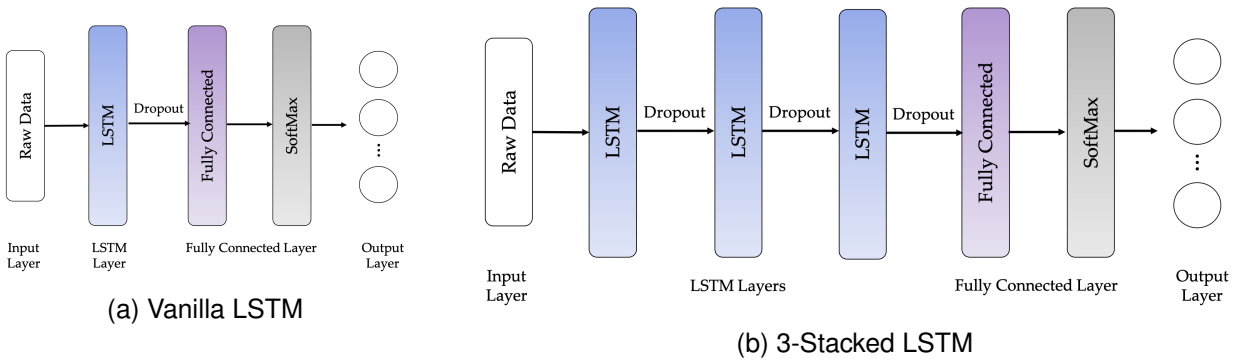


Figure 3.3 – Vanilla and 3-Stacked LSTM Network architectures [Mekruksavanich and Jitpattanakul, 2021].

Mekruksavanich and Jitpattanakul [2021] proposes a sampling rate of 2.56 seconds with a fixed-width sliding window. Each window overlaps each other by 50%. Figure 3.4 presents this process. These values are empirical, designed to detect repetitive rate activities, such as walking. More work is done reshaping the input to allow a 3D input as required for the first LSTM layer. Mekruksavanich and Jitpattanakul [2021] also proposes a 4-layer CNN-LSTM, later discussed in Section 3.4, presenting the best performance of all evaluated networks with the UCI-HAR dataset. The model achieved 99.39% accuracy, improving 2.24% from the baseline LSTM.

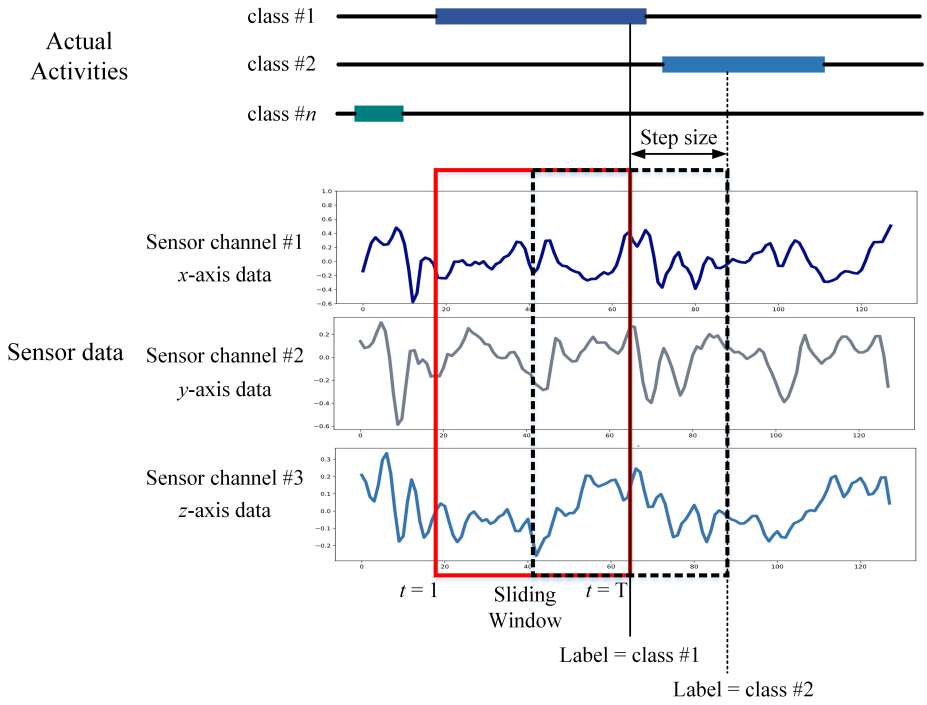


Figure 3.4 – Sliding-Window method [Mekruksavanich and Jitpattanakul, 2021].

Zebin et al. [2018] proposed a stacked LSTM approach, with 2 LSTM layers added before the classifier. Differently from Mekruksavanich and Jitpattanakul [2021], this architecture uses Batch Normalization inside the classifier to reduce training epochs needed to increase the accuracy, as demonstrated by Mekruksavanich and Jitpattanakul [2021]. This work uses a waist-worn sensor with two sensors: (*i*) accelerometer; (*ii*) gyroscope. Each sensor contains three axes. Sensors data are recorded at a 50 Hz sampling frequency. The raw data from the dataset was reshaped to allow a 3-D structure, as required for the LSTM Layer 1 input shape. In this work, 128 1D samples of each sensor axis are grouped. Figure 3.5 shows the proposed LSTM architecture. This work achieved a 92% average recognition accuracy for 6 daily-life activities. The authors highlighted the reduction in training epochs and added robustness due to the Batch Normalization and Dropout layers.

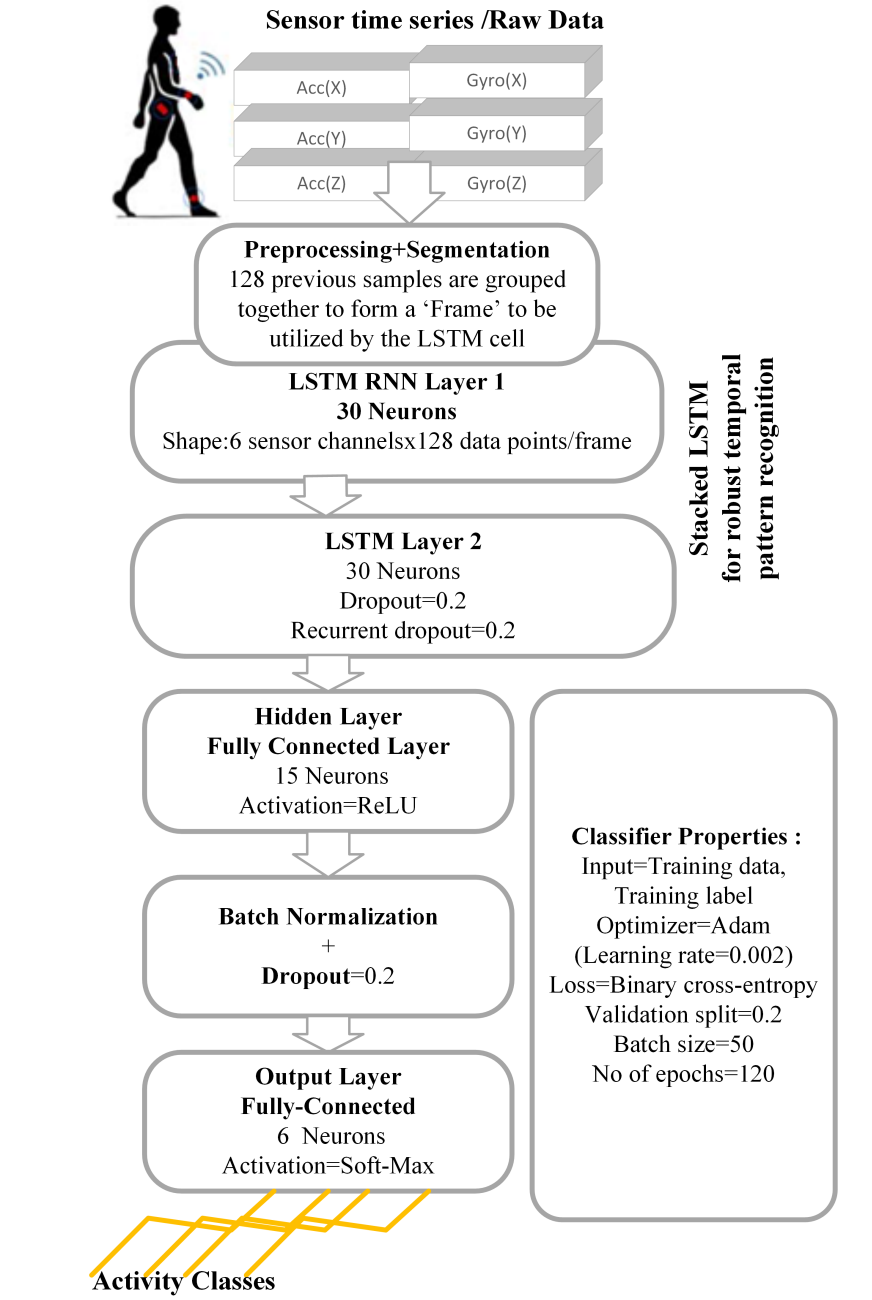


Figure 3.5 – Zebin et al. [2018] LSTM network architecture.

3.3 CNN Approaches

The feature extraction is a key component of CNN approaches to the HAR problem. Zeng et al. [2014] proposed in his work a traditional CNN approach using one Convolutional Layer, a Max-Pooling layer, and two fully connected layers (hidden layers). Figure 3.6 shows the proposed architecture.

Similarly to Mekruksavanich and Jitpattanakul [2021], Zeng et al. [2014] use a sliding window of 64 samples with a certain percentage of overlap to extract input data for the

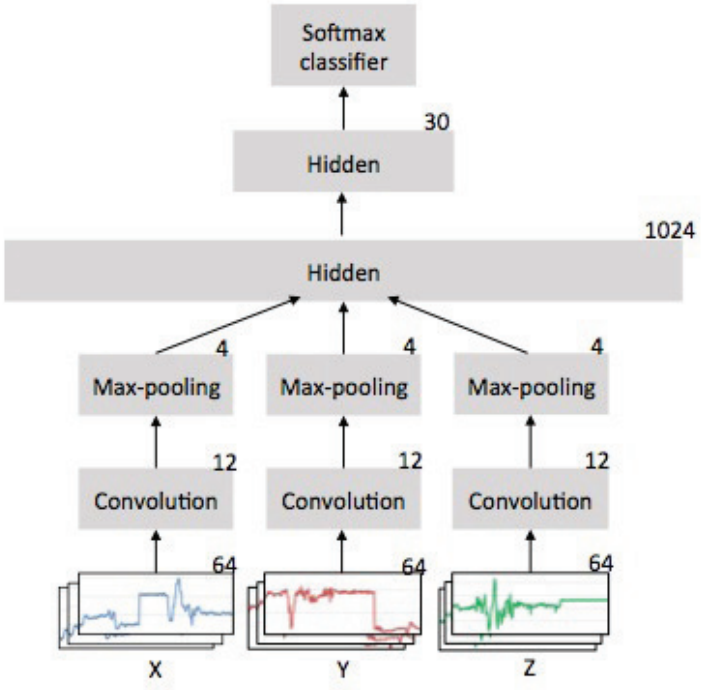


Figure 3.6 – Zeng et al. [2014] proposed CNN Network architecture.

CNN. Differently from other works, the authors use only one sensor: an accelerometer with 3 axes. Input data is shaped into a two-dimensional array with 64 samples of each of the three axes. Zeng et al. [2014] compared its proposal accuracy to other methods like a PCA-variant and a statistical method. Figure 3.7 presents this comparison. This work showed the superiority of the CNN approach over other traditional methods. The authors highlighted that their results are experimental, and more experiments with larger datasets, such as MobiAct [Vavoulas et al., 2016], are needed to study the robustness of the proposed architecture.

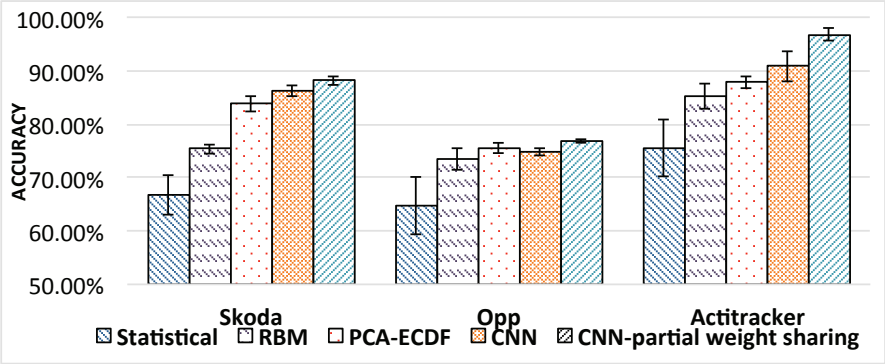


Figure 3.7 – Accuracy comparison between different methods [Zeng et al., 2014].

Shi et al. [2020] propose a similar architecture, with three 1D convolutional layers with 32 output channels, 64 and 128 channels, respectively. The classifier contains a pooling layer, a fully connected layer, and a SoftMax layer. Figure 3.8 shows the proposed architecture. The author's goal is to detect pre-impact movement using the MobiAct dataset, with a sensor worn on the waist, with a sampling rate of 200 Hz. In the training and validation

phases, Shi et al. [2020] used data of falls and frequent ADLs, such as walking, jumping, and jogging. This work does not use all MobiAct classes because there are activities unrelated to falls, such as elevator riding. This work achieved an accuracy of 95% using a combination of CNN and Class Activation Mapping (CAM). Unlike other articles that used private datasets, this study used the MobiAct, a publicly available dataset upon request.

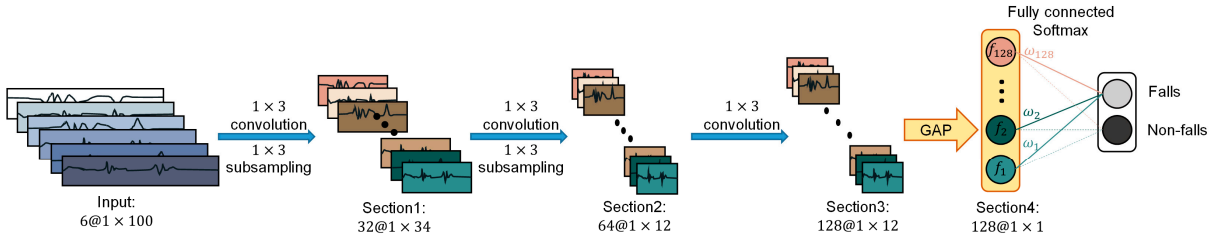


Figure 3.8 – Shi et al. [2020] proposed CNN Network architecture.

Hayat Ullah and Arslan Munir [Ullah and Munir, 2023] propose a dual attentional convolutional neural network (DA-CNN) architecture that leverages a unified channel–spatial attention mechanism to extract HAR features in video frames. The dual channel–spatial attention layers and the CNN layers learn to be more selective in the spatial receptive fields that have objects within the feature maps. The extracted features are then forwarded to a stacked bidirectional gated recurrent unit (Bi-GRU) for long-term temporal modeling and recognition of HAR features.

Figure 3.9 shows the proposed architecture. It contains the modules: (i) CNN architecture; (ii) Dual channel; (iii) Spatial attention module; (iv) bidirectional GRU network. The CNN module utilizes a dual-attention mechanism to extract salient CNN features from video frames effectively. The bi-directional GRU network learns the activity representation for hidden sequential patterns, which is the temporal modeling of long-term human action sequences. This architecture is modeled using TensorFlow.

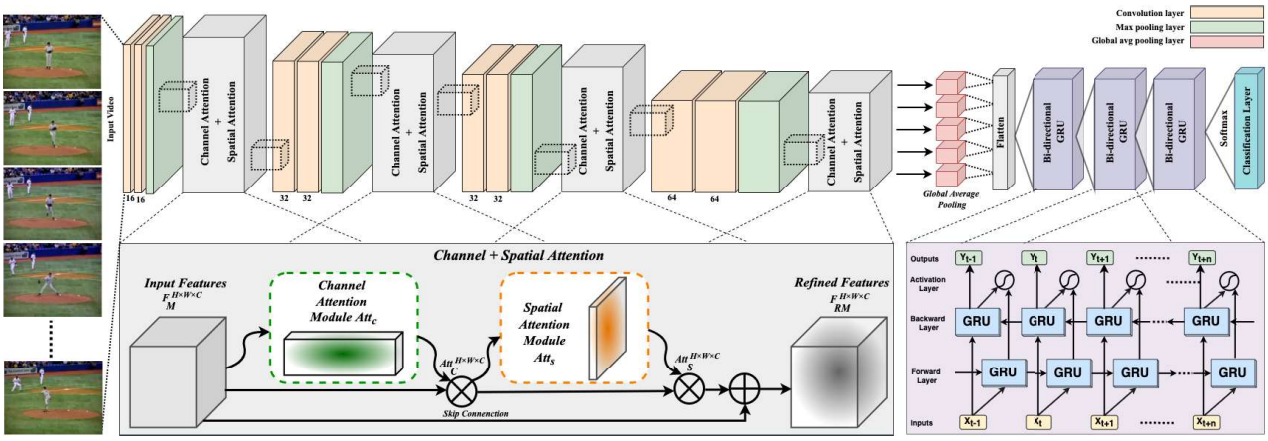


Figure 3.9 – Ullah and Munir [2023] DA-CNN+Bi-GRU architecture.

The experiments use YouTube action, UCF50, HMDB51, UCF101, and Kinetics-600 datasets. The results are compared with the state-of-the-art methods regarding accuracy and inference runtime. Results show that the DA-CNN+Bi-GRU framework has improved execution time up to 167 times in terms of frames per second compared to most of the contemporary action-recognition methods. Future work includes using the temporal attention mechanism together with spatial attention to improve human activity-recognition performance.

Nidhi Dua et al. [Dua et al., 2023] propose a HAR classifier called ICGNet, a hybrid of Convolutional Neural Network (CNN) and Gated Recurrent Unit (GRU). It uses multiple-sized convolutional filters simultaneously over the input, allowing to capture the information in the data at multiple scales, which helps to compute more abstract features for local data patches. The proposed classifier can capture the multivariate time series data's local features and long-term dependencies of the multivariate time series data. Figure 3.10 shows the complete classifier.

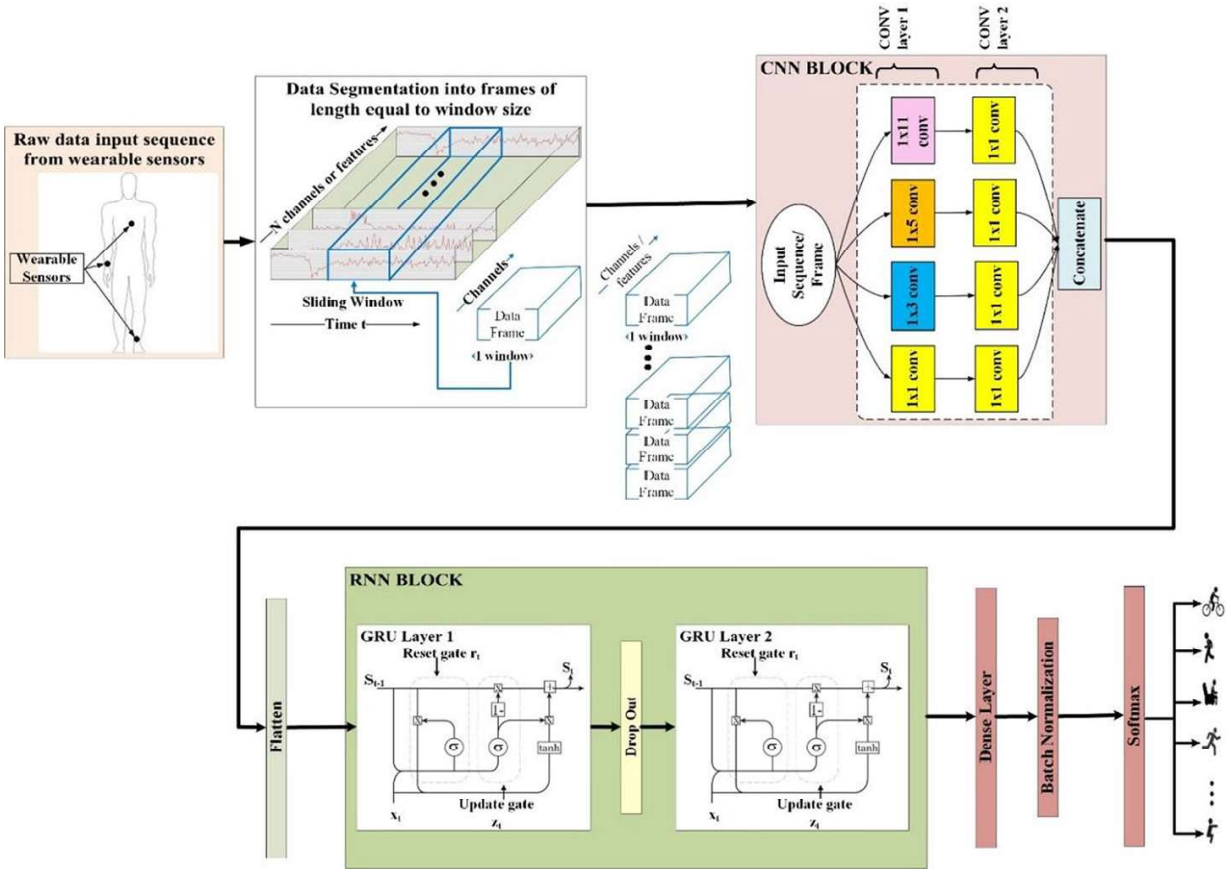


Figure 3.10 – Dua et al. [2023] ICGNet architecture.

The classifier is implemented using TensorFlow. The first block captures activity data through wearable body sensors. The captured human activity sequence data is in time-series format and is segmented using the sliding window technique. The segmented data frames are then forwarded to the CNN and GRU for feature extraction, and finally, the dense

classifier layer with SoftMax activation classifies the data. The overall accuracies achieved on two benchmark datasets, MHEALTH and PAMAP2, are 99.25% and 97.64%, respectively. The results indicate that the proposed network outperformed similar architectures proposed for HAR in the literature.

3.4 Hybrid CNN+LSTM Approaches

Implementations that combine CNN and LSTM are common in many areas, such as video classification and fall detection [Ng et al., 2015; García et al., 2022]. This hybrid approach combines the feature extraction of CNNs, with the temporal analysis of LSTMs.

García et al. [2022] uses this approach to detect elderly falls effectively. Their model includes two CNN layers and one LSTM layer. Figure 3.11 shows the proposed network. An accelerometer sensor worn in the waist area produces data. The raw data are processed using a sliding window of 1.28 seconds with 50% overlap to extract time frames from time-series raw sensor data. Figure 3.12 illustrates the process of extracting time-frames from raw sensor data. Using the UCI-FALL [Podareanu et al., 2019] dataset, García et al. [2022] concluded that combining CNN and LSTM improves accuracy compared to standalone models, achieving 95% in the test dataset. The authors highlighted that accuracy improvement could be obtained in the raw data processing phase.

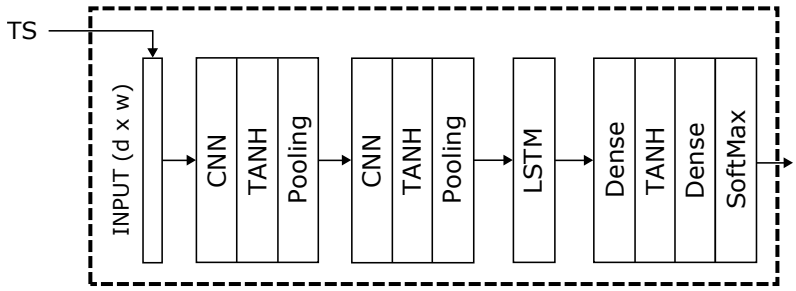


Figure 3.11 – García et al. [2022] proposed CNN+LSTM network

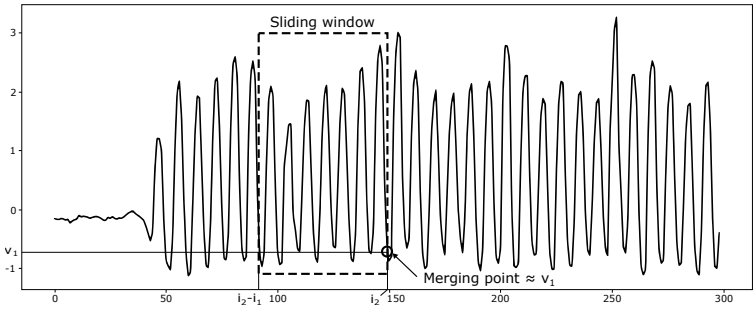


Figure 3.12 – García et al. [2022] sliding-window method.

Ordóñez and Roggen [2016] proposed a new Deep Neural Network framework designed for activity recognition using wearable sensors. This architecture, named DeepConvLSTM, combines convolutional and recurrent layers. Similarly to other implementations, this work uses the convolutional layer as the automated feature extractor, and the recurrent layer analyses the temporal dynamics of the feature maps created by the convolutional layer. Figure 3.13 presents the proposed network. Layers 2 to 5 are convolutional layers. Layers 6 and 7 are the dense layers with LSTM cells, thus composing the hybrid architecture. The authors highlight that the proposed framework improved the accuracy by 4% on average compared to a baseline CNN approach. The paper did not discuss power and memory overheads when comparing the hybrid approach with the standalone CNN approach.

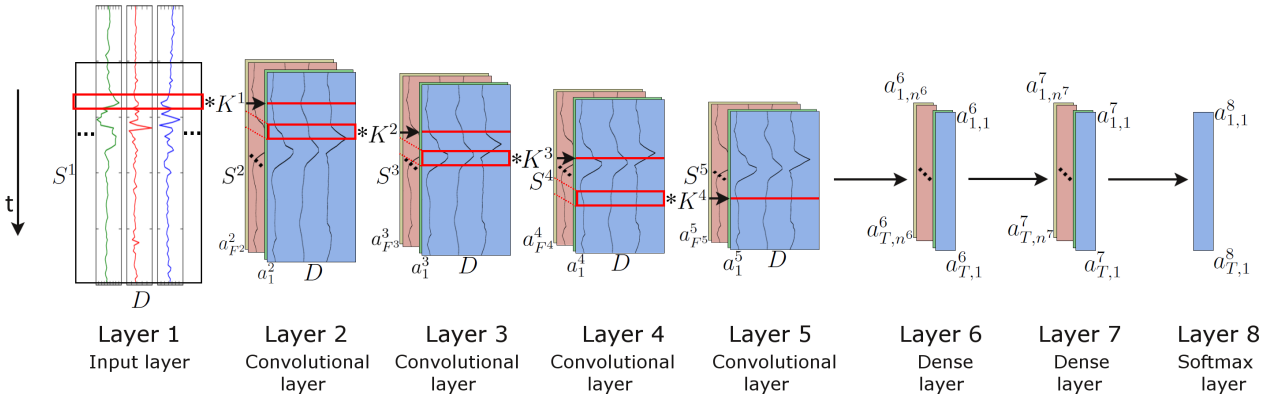


Figure 3.13 – Ordóñez and Roggen [2016] proposed hybrid network architecture.

Hyunseo Park et al. [Park et al., 2023] propose a deep learning-based HAR model called MultiCNN-FilterLSTM, which is a combination of a multi-head CNN with an LSTM through a residual connection that processes the feature vectors in a hierarchical order. Also, they propose a new LSTM cell, called filter-wise LSTM (FilterLSTM), allowing to learn the HAR model features based on hierarchical levels.

Figure 3.14 shows the proposed model. This model is implemented using Pytorch. A two-dimensional multi-head CNN extracts feature maps from the preprocessed accelerometer data. The multi-head CNN consists of multiple independent convolutional heads, where each head captures different levels of information from the data. Thus, each convolutional head in the proposed HAR model uses different filter sizes to extract feature maps in different hierarchical levels. The model also presents a fully-connected layer at the end of each head. The different filter sizes in each CNN head have different receptive fields, making the extracted from each CNN head have diverse hierarchical feature levels. Thus, the FilterLSTM hierarchically integrates the features in different levels to generate additional contexts that the multi-head CNN could not capture through the convolutional filters with the given sizes, learning the causal effects from the low-level to high-level features extracted from a multi-head CNN.

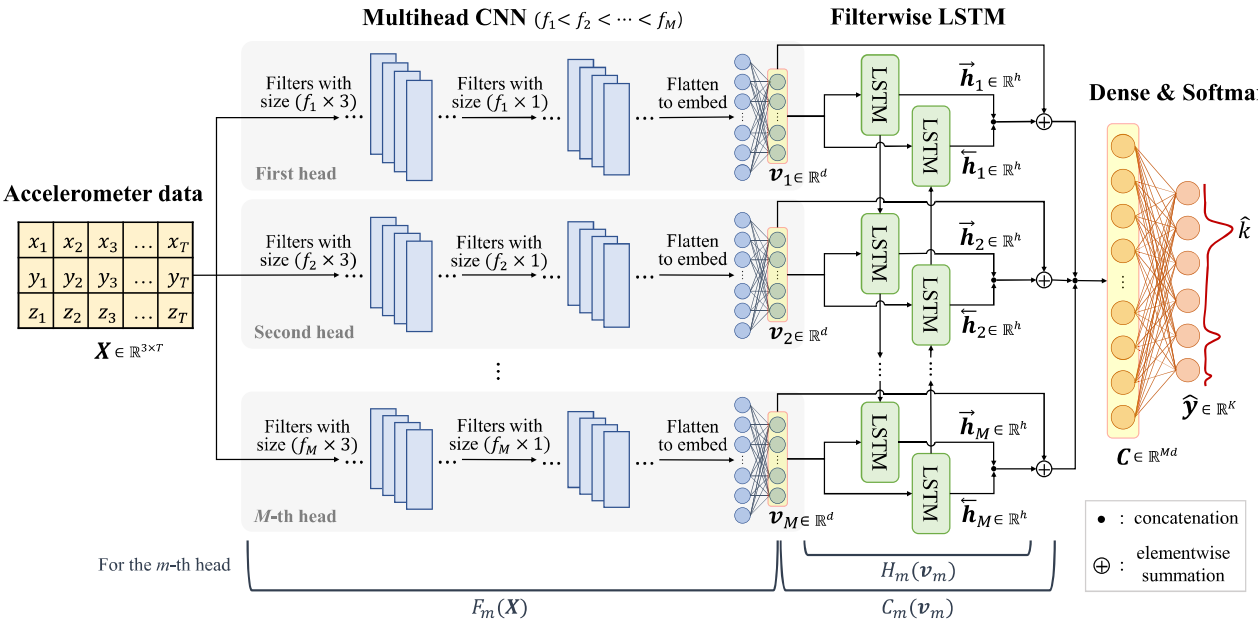


Figure 3.14 – Park et al. [2023] proposed hybrid MultiCNN-FilterLSTM network architecture.

The proposed HAR model was evaluated on two publicly available datasets, WISDM and UniMiB SHAR. The results show that the proposed HAR model achieved a classification accuracy of 94.59% and an F1 score of 92.50%, enhancing the classification accuracy by 2.3%–4.4%, while requiring 21%–70% fewer operations than the state-of-the-art models. Additionally, the proposed model is deployed to a Raspberry Pi 4 for further analysis in terms of deployment, achieving 4.2% improvement in classification performance with UniMiB SHAR while achieving approximately 34% and 38% reductions in inference time.

Anitha et al. [Inturi et al., 2023] present a new solution for fall detection using vision-based approaches. For this method, a set of key points of the subject is acquired by applying the AlphaPose pretrained network, which is the joint points. A CNN process the acquired key points, while an LSTM architecture preserves the long-term dependencies. Figure 3.15 shows the proposed CCN plus LSTM architecture. The CNN comprises a window of size 64 and an overlap of size 48. The input feature map has a dimension of 17×3.

The results were validated using the UP-FALL detection dataset and achieved commendable results compared to the state-of-the-art approaches, achieving an accuracy of 98.59%. The OpenPose network is used for key point detection to perform a comparison, and the results show that the AlphaPose network is more precise.

3.5 Optimization Techniques

Battery-powered embedded processors, such as those in smartwatches, have a small memory footprint (few kilobytes), low-clock frequencies, and low power availability.

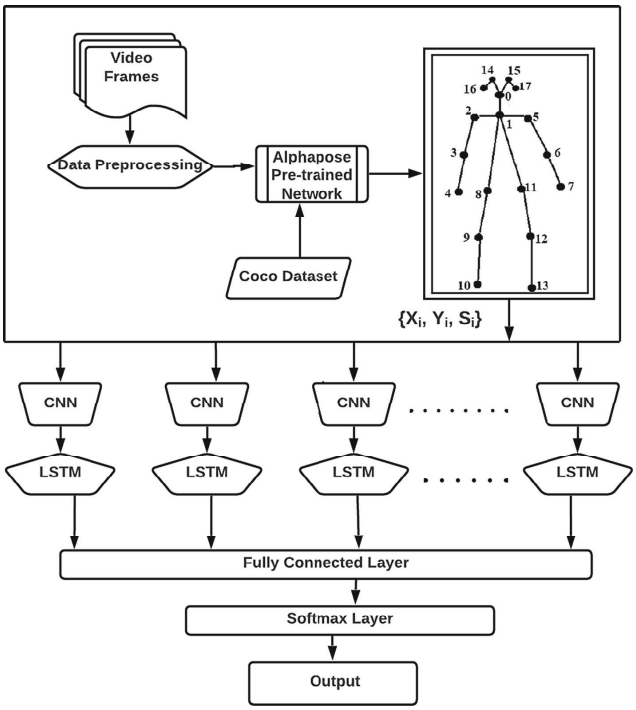


Figure 3.15 – Inturi et al. [2023] proposed hybrid CNN and LSTM network architecture.

Some techniques have been developed to reduce the power and memory footprint of recent neural network approaches. This section discusses some of the available optimization techniques.

Pruning. Pruning is a compression technique, first mentioned in LeCun et al. [1989]. The basis of the method is the fact that many parameters in the network are redundant or do not have any significant contribution to the output. Two pruning methods may be used:

- **Unstructured Pruning**: this method removes individual parameters. It is also known as **Weight Pruning**, meaning that individual weights could be set to zero, essentially deleting the connection of the respective nodes.
- **Structured Pruning**: This method considers parameters as groups, removing filters and channels, to reduce the overall computational costs related to these layers.

Gholami et al. [2021] presents a method to determine which filters can be removed from the CNN without significant loss in the original accuracy. VGGNet running CIFAR-10 inference reduced the computational cost by 30%, when using the proposed pruning technique.

Quantization. Quantization techniques are used when the target implementation platform has resource limitations, such as battery-powered processors. This technique reduces the cost of the numerical representation of weights in the Neural Network.

The first naive quantization technique replaces the floating point representation with a fixed point representation, using only integer values. This method enables the usage of simpler processors. The authors in [Gholami et al., 2021] reported a minimal accuracy loss.

Longguang Wang et al. [Wang et al., 2022] propose a quantization process using lookup operations and propose to learn lookup tables as quantizers. The method proposes using differentiable lookup tables and several training strategies for optimization. Results show that, compared to other methods such as QIL [Jung et al., 2019], the quantized networks using lookup tables achieve state-of-the-art performance on image classification, image super-resolution, and point cloud classification tasks.

Other quantization methods map an interval of values to a certain value, which may be uniform [Yang et al., 2020] or non-uniform [Yamamoto, 2021]. Figure 3.16 compares uniform quantization (left) and nonuniform quantization (right). Values in the continuous domain r are mapped to discrete lower precision values in the quantized domain Q [Gholami et al., 2021]. With this technique, floating-point values can be converted to 8-bit integers, reducing the memory area required to store the neural network parameters (weights and features).

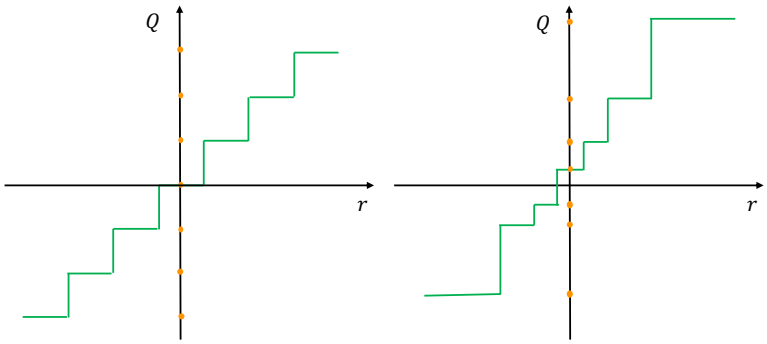


Figure 3.16 – Quantization techniques [Gholami et al., 2021].

Binarization. Binarization is a quantization method where values are limited to a 1-bit representation, drastically reducing memory requirements. This technique achieves significant acceleration even against dedicated hardware using 8-bit representation [Gholami et al., 2021]. The disadvantage of binarization is accuracy degradation. Recent proposals use a hybrid binarized approach, where only the first CNN layer uses binarized weights, achieving 1.2 times more efficiency than the conventional approach. Shimoda et al. [2017] in his work achieved 0.3 watts of power consumption while achieving 82.8% accuracy in the CIFAR-10 dataset.

3.6 Final Remarks

This chapter presented works related to ML frameworks and models, almost all using IoT sensors to capture data. We observe a lack of studies using IoT devices as the target device for ML inference. Optimization techniques were proposed, for example, by Gholami et al. [2021], focusing on minimizing accuracy loss without power and memory footprint analyses, key parameters for embedded systems. Mekruksavanich and Jitpattanakul [2021] mention the use of smartphones as a data capture source, but little effort is made to use the smartphone as the target for the inference phase.

Many of the proposed models [Zebin et al., 2018; Xu et al., 2019; Ordóñez and Roggen, 2016; García et al., 2022; Park et al., 2023; Inturi et al., 2023] consist of complex, stacked, LSTM structures, which require the use of resources that are scarce in embedded systems, such as volatile memory area. On the one hand, these works achieved state-of-the-art accuracy for many datasets. The same occurs for [Ullah and Munir, 2023] and [Dua et al., 2023], but using a combination of CNN and GRU. On the other hand, they do not apply to battery-powered devices that require an edge or cloud device to process this information. In practice, edge or cloud devices are not available everywhere and at all times, creating the risk of not detecting an emergency, such as a hard fall event.

Chapters 5 and 6 present how this work contributes to the research topic of optimizing an ML model for the inference phase, specifically for devices with limited resources. Our objective is to combine optimization techniques with efficient neural network modeling, evaluating not only accuracy but also the costs associated with the processor ISA and the memory requirements during the inference phase.

4. CNN REFERENCE AND OPTIMIZED MODELS

This chapter presents the CNN reference model [Chiang, 2022], and optimizations made to this model. Section 4.1 presents the Pytorch framework, the ML framework that the reference model uses. Section 4.2 shows the neural network architecture of the reference model. Section 4.3 presents optimizations made in the reference model to assess the impact of the CNN components on accuracy. Section 4.4 presents and discusses the obtained results. Section 4.5 concludes this chapter, summarizing the main contributions.

4.1 PyTorch Framework

PyTorch is a machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing. Paszke et al. [2019] introduced the Pytorch framework. It is an open-source Python library that performs tensor computation with GPU acceleration capability. The main goal of Pytorch is to accelerate the path from research prototyping to production deployment, while maintaining a performance comparable to the fastest current libraries for deep learning [Paszke et al., 2019]. Despite Pytorch being a Python framework, most of this library is written in C++ to improve performance. It is compatible with most commonly used GPU technologies such as CUDA and recently received support for Apple Silicon GPU.

One of the key features of the Pytorch framework is the ability to execute dataflow on the GPU asynchronously. Figure 4.1 presents a timeline execution of the first operation of a ResNet-50 model.

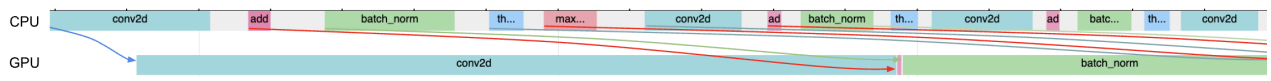


Figure 4.1 – Timeline of CPU and GPU computation of the first operands of ResNet-50. Colored areas on the CPU correspond to the work done to queue operators and to load on the GPU. Colored areas on GPU correspond to operators being computed. Gray areas are Python code executed by its interpreter [Paszke et al., 2019].

The CPU queues the tasks before sending them to the GPU, allowing Pytorch to achieve almost 100% GPU utilization. This asynchronous parallelism avoids GPU stalls, increasing the overall PyTorch performance.

4.2 CNN Reference Model

The CNN reference model contains three convolutional layers and two fully connected layers [Chiang, 2022]. It is a relatively simple architecture by today's standards. Its goal is not to achieve state-of-the-art accuracy, but to demonstrate the effectiveness of simple CNN models in HAR tasks. Figure 4.2 presents the network architecture. Code 4.1 presents the reference CNN modeled using the Pytorch framework.

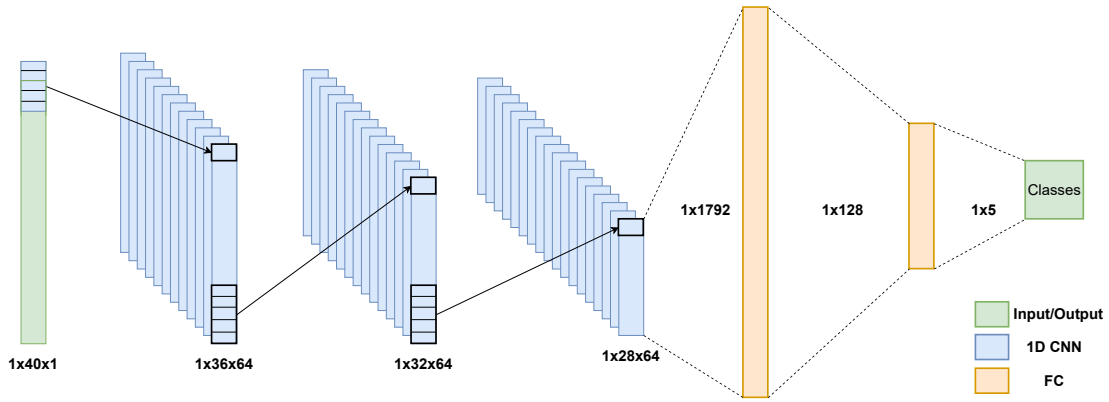


Figure 4.2 – 1D CNN Reference Model.

Code 4.1 – Pytorch network modeling.

```
    def __init__(self, input_size, num_classes):
        super().__init__()

        # Extract features, 1D conv layers
        self.features = nn.Sequential(
            nn.Conv1d(input_size, 64, 5),
            nn.ReLU(),
            nn.Dropout(),
            nn.Conv1d(64, 64, 5),
            nn.ReLU(),
            nn.Dropout(),
            nn.Conv1d(64, 64, 5),
            nn.ReLU(),
            )
        # Classify output, fully connected layers
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(1792, 128),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(128, num_classes),
        )
```

Line 6 to 13 of Code 4.1 present the Feature Extractor. Each line corresponds to a specific layer inside this network. Some layers, such as ReLu and Dropout, are hidden in Figure 4.2. Convolutional layers accept many parameters. In this implementation, the parameters used are: (*i*) input size; (*ii*) number of filters; (*iii*) kernel size. Line 17 to 21 model the classifier. Lines 18 and 21 declare the Fully Connected layers, called “Linear” by Pytorch. The parameters are: (*i*) input size; (*ii*) output size.

As discussed in Section 3.3, activation layers follow the convolutional layers. The reference model adopts ReLu as the activation function. The dropout layer reduces the probability of overfitting by adding noise to the ReLu output. This last layer is useful in models that use a low volume of sensors on each training iteration [Chiang, 2022].

This CNN uses a public dataset, similar to the MobiAct dataset, containing 20,000 sensor readings from 6 people, each performing five different actions. Appendix A presents a partial snippet fo the adopted dataset. Each sensor reading has 3 axes (roll, pitch, yaw) of 3 sensors (accelerometer, gyroscope, magnetometer). A total of 4 sensor positions are included (belt, arm, dumbbell, forearm). Magnetometers are often used to normalize the raw data of other sensors. This CNN model does not include it as an input feature, using the accelerometer and gyroscope inputs. The division between training and evaluation data is as follows: 80% for training and 20% for evaluation.

Unlike other implementations, the 1D CNN presented in this reference model can only process single timesteps, resulting in a 1x40 unidimensional input. Figure 4.3 shows the input shape of the reference model.

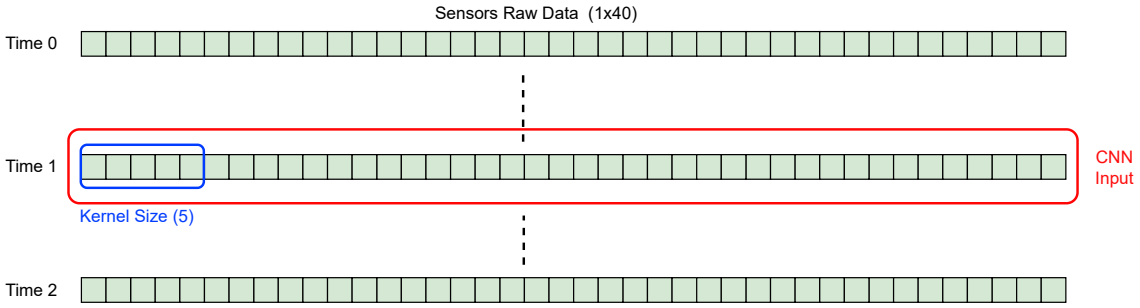


Figure 4.3 – Reference CNN input shape.

Although processing single timesteps do not allow precise temporal analysis, accelerometer and gyroscope sensors contain temporal characteristics embedded in raw data, thus allowing the model to detect human activities with 74% accuracy.

In this model, the training parameters are set in a class named *params* and later inserted into the training function. Code 4.2 presents this class.

The method *pytorch.train* is responsible for starting the training phase. It receives the model defined in Code 4.1, training parameters, and training and evaluation data from

Code 4.2 – Pytorch network parameters.

```
class params:
    def __init__(self):
        self.params = {
            "root": "data/pml-training.csv",
            "test": "data/pml-testing.csv",
            "resume": None,
            "input_dim": 1,
            "num_classes": 5,
            "workers": 0,
            "batch_size": 32,
            "epochs": 300,
            "lr": 0.003,
            "momentum": 0.9,
            "weight_decay": 1e-4,
            "split": 0.2,
            "use_cuda": torch.cuda.is_available(),
        }
```

the dataset. Other fields include the optimizer to be used and the training criterion to be respected. Code 4.3 presents the call of the training function.

Code 4.3 – Pytorch training function call.

```
    train(train_loader=train_loader,
          val_loader=val_loader,
          model=model,
          criterion=criterion,
          optimizer=optimizer,
          params=params,
          logger=logger,
          )
```

4.3 Optimization of the Reference Model

A simple 1D CNN model has benefits, such as reduced processing power requirements during training and evaluation, and reduced memory usage. The reference CNN does not provide complex optimization methods, such as pruning and quantization, neither it is optimized to achieve state-of-the-art accuracy. This model achieves 74% accuracy in randomly selected activities from the dataset.

Our optimization process aims to improve accuracy without substantially increasing the CNN complexity. The following subsections present the modifications done to the model and evaluated in the sequel.

4.3.1 Classifier Changes

The Fully Connected (FC) layer is present before the flatten layer. The FC has an important effect on accuracy and computational cost. The first layer of the FC nodes is a function of the number of output nodes from the previous layer. While changing the number of output nodes is possible, increasing it may improve the accuracy with the cost of increasing the number of weights. Initially, the number of output nodes was changed from 128 to 256 and later to 512.

4.3.2 Feature Extraction Changes

The reference model has 3 1D convolutional layers. Ordóñez and Roggen [2016] proposed a hybrid approach that included four CNN layers, taking advantage of its feature extraction capabilities. We modified the reference model using this knowledge to include a fourth convolutional layer. This modification results in a smaller output feature map at the end of the feature extraction phase, requiring adaptation on the FC layer to be compatible with the new feature map input.

4.3.3 Kernel size

The reference model adopts a kernel size equal to 1x5 (Figure 4.3). With the stride[1] defined at its default value (1), the convolution calculation generates 64 filters with this kernel. The selection of the kernel size is commonly determined during the training phase. We evaluated the impact on the accuracy using 1x3, 1x5, and 1x7 kernels.

4.3.4 Extended 1D Model

The major disadvantage of this 1D CNN model is the lack of temporal analysis. Human activities are highly related to previous and future movements. Although temporal

[1] Stride in convolutional neural networks is the step size by which the filter slides over an input image.

characteristics are embedded in sensor data, it is insufficient to detect complex human activities accurately. Recent approaches [Zeng et al., 2014; Xu et al., 2019; García et al., 2022] process data in time frames extracted from raw sensor data with a sliding-window method. Each work uses a different window width and overlap percentage. These approaches increase accuracy by reducing the effect of unintended data (noise or random human motions) on the readings.

To add temporal awareness to the reference model while keeping its reduced complexity, the convolutional layer was modified to allow multiple time steps to be included simultaneously in a modified input vector. Most HAR datasets use a time-per-row topology. Thus, a single row in the dataset contains the sensor data of this single time step, together with the activity label. A Python data repackager algorithm was written to convert the dataset into a timeframe-per-row topology. The data repackager joins a parameterizable number of time steps in a single row, adding a temporal correlation to the CNN without modifying its topology. This process is done for training and inference data at runtime, loaded to the GPU via PyTorch's data loader. Figure 4.4 presents how the 120-width array is created using three different time steps.

Using a single time step as input (40-with array), the resulting feature map size is 1x28 (3rd convolutional layer), combined with the number of filters (64), the flatten layer transforms these multiple arrays into a single 1x1792 vector (Figure 4.2). In the Extended 1D Model, the newly formed 120-width timeframe requires the input shape to be changed to accommodate the new array size. One of the benefits of using a 1D Convolution is that the input array can be increased without increasing the number of weights, reducing memory area, but that is not true for the classifier. Using the proposed model, the resulting feature map size is 1x108, containing roughly 3.8 times more information than the reference model, consequently improving accuracy. To accommodate the larger feature map, the new fully connected (FC) input size is 1x6912. Despite the increased number of weights in the FC layer, it requires fewer weights than other 1D CNN evaluated optimizations while achieving higher accuracy.

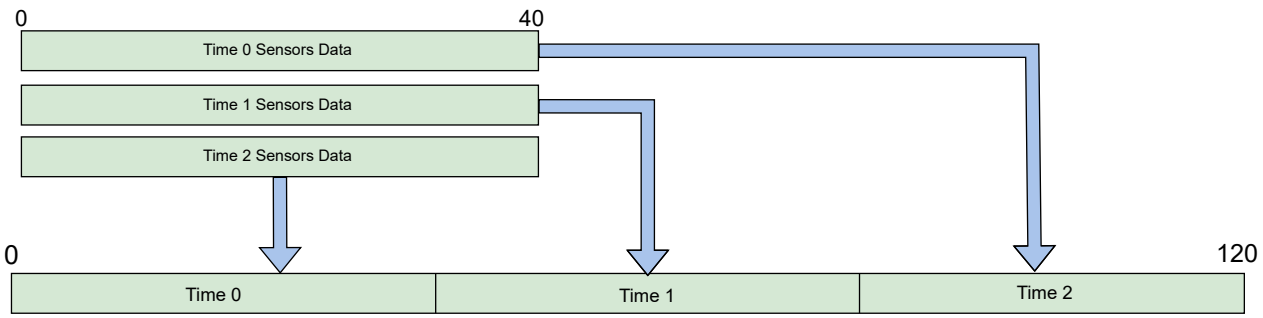


Figure 4.4 – Extended 1D Model.

4.3.5 Temporal-Extended 1D Model

This Section optimizes the Extended 1D Model, to provide better accuracy without increasing the weights array size, using a data preprocessing techniques.

The Extended 1D Model added temporal awareness to the reference model, thus significantly improving accuracy. The limitation of this proposal is that in the convolution layers, the same sensor cannot be processed in different timesteps simultaneously. Figure 4.5 illustrates this issue.

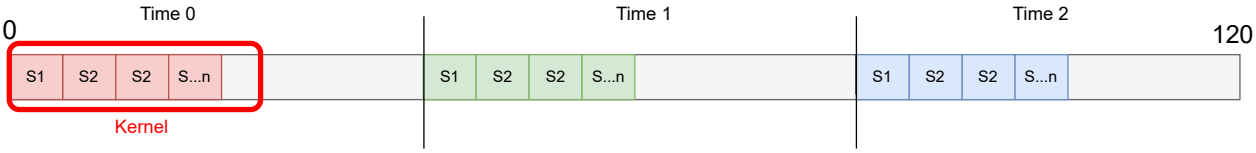


Figure 4.5 – Current Extended 1D Model.

A possible solution to this problem is to increase the kernel size, but as discussed in Section 4.4, the increase in the kernel size reduces the accuracy. The 2D model can use a square kernel, allowing input data to be arranged as a grid, solving the issue. Unfortunately, this approach is not easily modeled using the current dataset.

The *Temporal-Extended 1D Model* consists of a refactoring of the current data preprocessing algorithm used in the Extended 1D Model proposal, with the addition of an rearrange technique to group sensor data from different timesteps. Figure 4.6 illustrates the proposal.

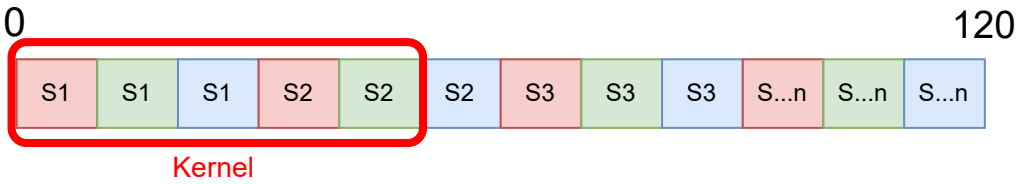


Figure 4.6 – Proposed Temporal-Extended 1D Model

This proposal allows the neural network to process different timesteps of the same sensor, allowing the network to identify mathematical trends in the input data during the feature extraction phase while maintaining the low complexity of the 1D CNN Model. This proposal effectively works by simulating a 2D network by actively preprocessing the input data in real time.

4.4 Results

The training and evaluation phases were performed using PyTorch using a CUDA-enabled GPU. This model is cross-compatible with the new Apple Silicon (arm64) architecture. The design phase was done in an ARM-powered MacBook Pro and the training and evaluation phase was done on a PC equipped with 16GB RAM, processor i7 7700k, and a GTX 1080.

Table 4.1 presents results using the optimizations made in the reference model, evaluating the number of parameters, the model size (i.e., memory footprint), and accuracy. The 2nd row presents results related to the baseline 1D model.

Table 4.1 – Reference and optimized models accuracy.

	Model	**Number of Params**	**Model Size (MB)**	**Accuracy**
2	Baseline 1D model	271,621	1.18	**74%**
3	FC - 256 output nodes	501,765	2.06	76%
4	FC - 512 output nodes	962,053	3.82	79%
5	Kernel size = 7	238,981	1.04	72%
6	Kernel size = 3	304,261	1.32	76%
7	Added Convolutional Layer	259,397	1.17	70%
8	Extended 1D Model - Accurate	926,981	4.03	**89%**
9	Extended 1D Model - Reduced	484,229	2.34	**85%**
10	Temporal Extended 1D Model	926,981	4.03	**89.5%**
	T. S. Jordan CNN Jordan [2016]	738,000	-	78%
	ResNet-18 Du et al. [2018]	11,000,000	-	94.8%

The 3rd and 4th rows evaluate the impact of increasing the fully connected (FC) layer. The accuracy increases by increasing the number of output nodes of the first fully connected layer at the cost of raising the number of weights in this layer. The number of weights in the fully connected layer doubled and quadrupled using 256 and 512 output nodes, respectively. The quadrupled memory area is hardly justified by just a 5% accuracy increase in a memory-constrained device.

The 5th and 6th rows evaluate the impact of the kernel size. Increasing the kernel size results in a smaller output feature map, consequently with less information and reduced accuracy. A smaller kernel produces larger feature maps and allows a better feature extraction phase, resulting in an accuracy increase at the cost of a larger output feature map

and increased computational cost. More study needs to be made to decide if the memory overhead increase justifies small increments in performance.

Increasing the convolutions, 7th row, while not changing the kernel size could result in a smaller feature map to be used in the classifier, negatively affecting the accuracy. It is not clear the reason explaining the accuracy reduction to 70%. One reason may be the unbalanced ratio between the feature extraction and the classification. It is necessary to further investigate the model to take advantage of this extra convolutional layer.

The 8th and 9th rows present the accuracy of the proposed extended 1D models. This result corroborates how relevant the time-series analysis is to HAR. The accuracy improved 15% when adding two extra time steps in the input shape. Even though the number of weights increased due to the change on the FC layer, it is roughly 5% smaller than the FC with 512 output nodes approach, with 10% higher accuracy.

The FC is responsible for more than 95% of the total number of weights in the model. We evaluated a reduced version of the proposed model by minimizing the number of weights in the FC layer. This version consists of changes in the classifier, focusing on reducing the number of parameters and model size while achieving similar accuracy. Compared to the "accurate" version, the number of parameters and model size was reduced by roughly 50%, with only a 4% accuracy reduction. Both extended 1D models increased the accuracy at the cost of a larger number of parameters and model size than the baseline model.

The 10th row presents the accuracy of the Temporal Extended 1D Model. Although this model replies the processing of more accurate 2D neural networks, this technique does not improve accuracy significantly in our model. It is unclear why no improvement was seen on this model. One possibility is that the Extended 1D model has the best possible accuracy given the input size, which means more timesteps would need to be added to increase accuracy significantly.

For comparison purposes, the network proposed in Jordan [2016], contains 5 2D layers (64x64x128, 32x32x128, 16x16x128, 8x8x64, 4x4x64) and 2 FC (256 and 6 classes). This network achieves an overall accuracy of 78% in all activities, achieving 94% in specific test sets, using 738,000 parameters. State-of-the-art CNNs, as ResNet-18 [Du et al., 2018], achieves 94.79% with 11M parameters.

4.5 Final Remarks

Using the PyTorch framework, this chapter introduced and optimized the CNN reference model for HAR. Although relatively simple, the reference model demonstrated the effectiveness of CNN models in HAR tasks. The accuracy of the model was improved through

various optimizations, such as modifying the classifier, exploring different kernel sizes, and adding convolutional layers.

One significant enhancement was the introduction of the Extended 1D Model, which added temporal awareness to the reference model. The model achieved higher accuracy by incorporating multiple time steps in the input data without significantly increasing its complexity. This approach proved effective in capturing temporal correlations embedded in sensor data. Additionally, the Temporal Extended 1D Model further refined the preprocessing algorithm, allowing the neural network to process different timesteps of the same sensor, resulting in improved accuracy.

Overall, the optimizations made to the reference model demonstrated the importance of fine-tuning architecture and parameters to achieve better accuracy in HAR tasks, *corresponding to the first contribution of this MSc work*. The results showed the trade-offs between model complexity, number of parameters, model size, and accuracy. The Extended 1D Model, in particular, showed the potential to incorporate temporal analysis while maintaining a relatively simple architecture.

The next chapters adopted as reference the **Extended 1D Model - Accurate**, naming it as **Ext1D**.

5. CNN C MODEL

This chapter presents the *second* contribution of this MSc work: converting the PyTorch reference model to a C model using integer representation, avoiding floating point representation. The C language is a common choice for programming embedded systems due to: (*i*) efficiency and control: C is a low-level language that provides a high degree of control over system resources; (*ii*) small memory footprint: embedded systems often have limited resources, and C compilers create compact and efficient code. Floating point units (FPUs) are costly cores in processors and should be avoided in energy- and area-constrained devices. Table 5.1 presents the silicon area required to implement an integer core versus an FPU core [Ness, 2018], demonstrating the cost induced by FPU cores.

Table 5.1 – Area consumption of the RISC-V Rocket SoC [Ness, 2018].

Module	Area (Equivalent Gates)	Area %
Integer Core	**43,729**	2.0
FPU Core	**73,582**	3.4
Caches (I&D)	1,972,361	92.4
Others	44,750	2.2

The FPU core can be completely removed from the processor with optimized integer-only software. The removal of the FPU core also has a major impact on energy consumption and thermal characteristic. Works such as [Yu et al., 2019] show that changing the floating point by integer representation can significantly reduce power dissipation.

This chapter is organized as follows:

- Section 5.1 presents the modifications made in the Pytorch Model to enable parameter export. Additionally, this section details the scripts to process the data exported from Pytorch.
- Section 5.2 presents the CNN *C model*.
- Section 5.3 details the method employed in the C Model to quantize the CNN for integer representation instead of floating point numbers.
- Section 5.4 evaluates the performance of the models considering accuracy, number of executed instructions (profiling), execution time, and energy consumption.
- Section 5.5 concludes this chapter.

5.1 Exportation and Conversion of Parameters

This step corresponds to the link between the Pytorch model and the model described in C language. This step has two functions: (*i*) generate files with the input features, weights, and bias in C header files; (*ii*) provide the reference accuracy.

Although Pytorch does not officially support the execution of the inference phase outside its environment, significant efforts have been made to extract values (weights and bias) from the Pytorch training algorithm. Juracy et al. [2021] suggest a method for extracting parameters after the training phase in TensorFlow. This work proposes a similar method to be used with Pytorch. Upon completing the training phase, we observed that the trained parameters were preserved within the model structure. This work proposes the method presented in Code 5.1 to read the parameters of the first layer and export them to two files.

Code 5.1 – Pytorch layer parameters extraction.

```
1     torch.set_printoptions(profile="full")
2     original_stdout = sys.stdout #
3     with open('0_bias.txt', 'w') as f:
4         sys.stdout = f
5         print(model.features[0].bias)
6         sys.stdout = original_stdout
7
8     with open('0_weight.txt', 'w') as f:
9         sys.stdout = f
10        print(model.features[0].weight)
11        sys.stdout = original_stdout
```

We observed that PyTorch uses a preprocessing algorithm in the dataset input. Therefore, extracting the dataset after processing it in the training phase is necessary, as presented in Code 5.2. Firstly, Lines 2 and 3 set the parameters to configure the print statements to write the data to a *txt* file instead of the terminal. Lines 5 to 9 present a loop that iterates through all the inference data, named `val_loader`, writing a combination of sensor data (steps) and labels (targets) to the output file. Pytorch's algorithm is designed to optimize the internal training and inference phase, making running inference difficult in a separate environment.

Codes 5.1 and 5.2 generate a file similar to the one presented in Code 5.3. PyTorch-generated files cannot be directly included in the C model. It would be necessary to parse them, incurring an overhead for the C model. Thus, a Python script, "ConvertTensor.py", reads the generated files and creates C header files, which are included directly in the C model. Code 5.4 presents an example of a header file created by the "ConvertTensor.py" script.

Code 5.2 – Pytorch normalized dataset extraction.

```
    if (os.path.isfile("inputOutTraining_evaluation.txt") == False):
        torch.set_printoptions(precision=16,sci_mode=False)
        original_stdout = sys.stdout #
        with open('inputOutTraining_evaluation.txt', 'w') as f:
          for iteration, (steps, targets, _) in enumerate(tqdm(val_loader)):
                sys.stdout=f # Change the standard output to the file we created.
                print(  steps )
                print(  str(targets.item()) + "," )
            print( "};" )
        sys.stdout = original_stdout
```

Code 5.3 – Output from Code 5.1.

```
tensor([-0.0317, -0.3036, -0.0525, -0.1189, -0.1458,  0.0667,  0.2298, -0.2369,
        -0.2612,  0.3024,  0.2583,  0.3375, -0.4156,  0.0210,  0.4200, -0.4175,
         0.3598,  0.0619,  0.3964, -0.1526, -0.4423,  0.2947, -0.2718, -0.0208,
         0.2301,  0.3514,  0.1893,  0.2788,  0.3751, -0.0169, -0.4218, -0.0517,
        -0.1678, -0.3444,  0.4381, -0.0477, -0.0077, -0.2968, -0.0628,  0.3663,
        -0.1707,  0.0942, -0.2148, -0.1124, -0.1517,  0.1301,  0.0936, -0.1396,
        -0.3354, -0.0221, -0.1553, -0.0620, -0.1392,  0.4240,  0.2299,  0.0107,
        -0.2744, -0.2092, -0.0731, -0.3572,  0.0859,  0.2750, -0.2670,  0.3482],
       requires_grad=True)
```

Code 5.4 – Output from ConvertTensor.py.

```
extern const float conv0_bias[64] = {
-0.0317,
-0.3036,
-0.0525,
-0.1189,
-0.1458,
....
0.3482 };
```

The model described in C language (*C model*) executes only the inference phase using the header files created by “ConvertTensor.py”. The goal is to have a code in C language that executes only the CNN model without preprocessing files or performing normalization computations. In this way, the evaluation of the *C model* corresponds only to the execution of the inference phase. The *C model* generates for each inference the tuple *{correct class, inferred class}*. An auxiliary program, “Evaluator.c”, reads the generated data, computes the accuracy, and generates the confusion matrix[1], as presented in Code 5.5.

[1] A confusion matrix is a table that is used to describe the performance of a classification model on dataset for which the true values are known. Informally, a confusion matrix shows where the model is getting confused, i.e., which classes the model predicts correctly and which classes the model predicts incorrectly

The report presented in Code 5.5 initially provides the number of samples evaluated in the inference phase. PyTorch's reference model considers the batch size to compute the accuracy value, which contains 32 samples. Thus, for this example, we obtain 123 sets ($\lceil \frac{3923}{32} \rceil$ = 123). As a result, the maximum number displayed in the confusion matrix is 123 because, out of the 123 evaluated sets, there was at least one correct prediction for a given class in one of the 32 samples. For example, in Code 5.5, the horizontal row for class 1 shows the detection of 29 sets (17 + 6 + 6) of a class different than 1, but in all sets, there was at least one correct detection in this class. Thus, the maximum accuracy is achieved when the diagonal line has the number of sets without any incorrect prediction. PyTorch computes the accuracy using the formula ($\frac{correct\ predictions}{correct\ pred.\ +\ wron\ pred.}$), resulting in this example in an accuracy of 88.5% ($\frac{615}{615+80}$).

Code 5.5 – Accuracy evaluation and confusion matrix generated after the inference phase for 3,923 samples, with a batch size equal to 32, using the floating point C model.

```
TOTAL SAMPLES: 3923
----------------------
Accuracy: 88.5
-----------------------
Correct predictions: 615
Wrong predictions: 80
----------------------
Confusion Matrix:
      0       1       2       3       4
0:    123     1       1       0       0
1:    17    123       6       0       6
2:    0       5     123       2       0
3:    0       0      33     123       2
4:    0       0       3       4     123
```

5.2 Description of CNN Layers

We implemented the *C model* without using standard libraries or function calls. The goal is to improve the performance of the *C model*. The training phase uses layers not required during inference, such as the Dropout Layer. Therefore, we did not include these layers in the *C model*. Subsequent subsections detail the implementation of the layers required to model the CNN in C language.

5.2.1 Convolution

Code 5.6 shows how to invoke a convolution in Pytorch. The complete model was previously presented in Code 4.1

Code 5.6 – Pytorch layers.

```
nn.Conv1d(input_size, 64, 5),
nn.ReLU()
nn.Dropout()
nn.Conv1d(64, 64, 5)
nn.ReLU()
nn.Dropout()
nn.Conv1d(64, 64, 5)
nn.ReLU()
```

We implemented two convolution models. The first model, presented in Code 5.7, receives a unidimensional vector corresponding to the raw data extracted from the dataset as input. This raw data input has a shape of 1×120. In Code 5.7, the outer loop (starting at line 1) iterates through the output filters, whose value is defined by `NUM_FILTERS` parameter. In our model, all convolution layers use `NUM_FILTERS` equal to 64. The code segment between Lines 4 and 9 searches for the weights and bias related to the currently processed filter (k). Then, the loop from lines 12 to 18 executes the kernel operation.

Note the MAC (multiply-accumulate) operation in line 16. Considering that the input vector has 120 entries, and the `KERNEL_SIZE` equal 5, this first layer executes 37,120 MAC operations ($64 \times 116 \times 5$).

The second and third layers use the second convolution model, detailed in Code 5.8. This model uses a bidimensional input shape of 64×116, and 64×112 in the second and third layers. The first dimension, 64, corresponds to the number of filters. The second parameter corresponds to the input features size minus ($kernelsize - 1$). The external loops (lines 1 and 2) iterate through the output filters (`filterToGenerate`) and output vectors (`inputOffset`). Line 3 initializes the counter used to generate one result (`conv3_featureMap [filterToGenerate][inputOffset]`). The two inner loops (lines 4 and 5) execute the kernel operation.

The number of MAC operations is equal to 2,293,760 ($64 \times 112 \times 64 \times 5$) and 2,211,840 ($64 \times 108 \times 64 \times 5$) for the second and third layers, respectively.

Code 5.7 – 1D convolution layer, using values from the dataset.

```
for (int k = 0; k < NUM_FILTERS; k++)
{
    // Load Current Weights
    for (i = 0; i < KERNEL_SIZE; i++)
    {
        conv0_currentKernel[i] = conv0_weights[i + (k * KERNEL_SIZE)];
    }
    // Load Current Bias
    conv0_current_bias = conv0_bias[k];

    // Perform Kernel operation
    for (i=0; i<=sizeof(input_vector)/sizeof(input_vector[0])-KERNEL_SIZE; i++)
    {
        float totalSum = 0;
        for (int j = 0; j < KERNEL_SIZE; j++)
             totalSum += input_vector[i+j] * conv0_currentKernel[j];  // MAC oper.
        conv0_featureMap[k][i] = totalSum + conv0_current_bias;
    }
}
```

Code 5.8 – 1D convolution layer, using a bidimensional input shape as input.

```
for (int filterToGenerate=0 ; filterToGenerate<NUM_FILTERS ; filterToGenerate++ ){
   for (int inputOffset = 0 ; inputOffset < CONV3_INPUT_SIZE-4 ; inputOffset++){
     conv3_totalSum = 0;
     for (int filterIn = 0 ; filterIn < NUM_FILTERS ; filterIn++){
       for (int kernelIndex = 0 ; kernelIndex < KERNEL_SIZE ; kernelIndex++){
        int weightIndex = kernelIndex + (filterIn * KERNEL_SIZE) +
↪  (filterToGenerate * NUM_FILTERS * KERNEL_SIZE) ;
        int indexIn = kernelIndex + inputOffset;
        conv3_totalSum += conv0_featureMap[filterIn][indexIn] *
↪  conv3_weights[weightIndex];
       }
    }
    conv3_totalSum += conv3_bias[filterToGenerate];
    conv3_featureMap[filterToGenerate][inputOffset] = conv3_totalSum;
  }
}
```

5.2.2 Activation Layer

The reference CNN model uses the rectified linear activation function (ReLU) as the function of the activation layer. The ReLU [Keras, 2023] is a piecewise linear function that outputs the input directly if it is positive. Otherwise, it outputs zero. It is a function used in many types of neural networks due to its simplicity of implementation. Code 5.9

presents the ReLU implementation. The outside loop iterates through all filters generated in the convolution, while the internal loop iterates through the feature map for each filter.

Code 5.9 – ReLu layer modeled in C language.

```
1 for (int i = 0; i < NUM_FILTERS; i++)
2     for (int j = 0; j < CONV0_INPUT_SIZE-4; j++)
3         if (conv0_featureMap[i][j] <= 0)
4             conv0_featureMap[i][j] = 0;
```

5.2.3 Fully Connected Layer

The reference model uses two Fully Connected (FC) layers with the same input shape. Code 5.10 presents the implementation of the FC layer. The external loop, starting at line 1, iterates according to the FC size (`FC1_OUTPUT_SIZE=128` and `FC2_OUTPUT_SIZE=5`). The inner loop iterates through the input vector, performing MAC operations. The input vector of the first LC layer (LC1) has 6,192 elements, corresponding to the flattened output of the third convolutional layer (112×64). Thus, LC1 executes 884,736 MAC operations (6192×128). The LC2 executes 640 MAC operations (128×5).

Code 5.10 – Fully Connected C implementation.

```
1 for (int outputIndex = 0; outputIndex < FC_OUTPUT_SIZE; outputIndex++){
2   totalValue = 0;
3   for (int i = 0; i < fc1_inputSize; i++)
4   {
5    totalValue += flatten1_vector[i] * fc1_weights[(fc1_inputSize*outputIndex)+i];
6   }
7   fc1_out_vector[outputIndex] = totalValue + fc1_bias[outputIndex];
8 }
```

5.2.4 Summary of the CNN Parameters and MACs Operations per Inference

Table 5.2 summarizes the CNN main parameters and the number of MAC operations per inference. Note that it is necessary to execute 5,428,096 MAC operations per inference.

Given the number of mathematical operations, evaluating the adoption of integer representation is important to reduce processor complexity and execution time. Using simpler processors also reduces the power dissipation of the device running the CNN model.

Table 5.2 – CNN main parameters and number of MAC operations per inference.

	Input Size	Output Size	# of filters	kernel size	MAC operations
CONV 1	120	116	64	5	37,120
CONV 2	116	112	64	5	2,293,760
CONV 3	112	108	64	5	2,211,840
FC 1	6,192	128	128	–	884,736
FC 2	128	5	5	–	640
				Total:	**5,428,096**

5.3 Quantization Process for 32-bit Integer

The Pytorch framework is compatible with multiple quantization methods. It is important to emphasize that all quantization methods in Pytorch are currently in the beta stage (https://pytorch.org/docs/stable/quantization.html).

In the Extended 1D Model, two quantization modes were evaluated:

1. Post-Training Dynamic Quantization - this mode changes the types of the weight from 32-bit floating point to 8-bit integers after the training. This quantization was applied to the Extended 1D Model presenting accuracy varying from 60% to 70%. A negative effect on performance was expected, but in our analysis, considering the variance in the accuracy during testing, we assume that the PyTorch Beta Quantization method is not entirely stable in its current form.

2. Quantization Aware Training for Static Quantization - this mode simulates the quantization effect during the training phase. The values in training still operate in a 32-bit floating point, but PyTorch performs mathematical rounds to simulate an 8-bit integer operation. We could not successfully apply this quantization to our model, as it needs significant changes in our current stable training phase.

We concluded that the quantization process should not be executed in the PyTorch framework. We propose to validate the CNN model described in C, first using floating point types. The equivalence between the PyTorch and floating point C models provides the gold model for further development using quantization and memory compression methods.

The exported weights and bias, as presented in Section 5.1, always use a floating point representation. A Python script, “Quantizator.py” converts all floating-point values to a fixed point by multiplying them by a constant, as illustrated in Figure 5.1. Despite simple to implement, the method requires the definition of the multiplication value to avoid overflow in the convolution or fully connected layers. We evaluated different values, observing that larger values, such as 10,000, reduce accuracy due to overflow in fully connected layers. Thus, empirically, we define 1,000 as the multiplier for weights and bias. This multiplier factor can

be easily modified, as it is an input for the “Quantizator.py”. The output of “Quantizator.py” is C-Header files used by the integer C implementation.

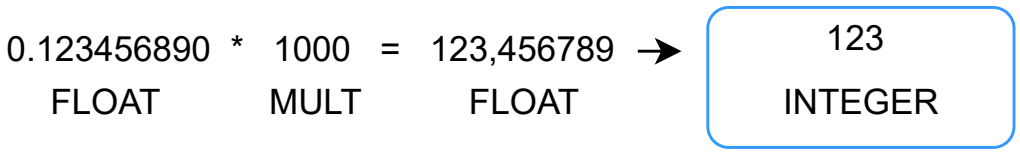


Figure 5.1 – Converting floating point numbers to integers.

Using a multiplication factor M for the IFMAP values, weights and bias requires attention when connecting the output of one layer to the next one. Given that the weights are multiplied by the input data, we have the values generated in the outputs multiplied by M^2. So, before starting a new layer, we must divide all results by M, as shown in Figure 5.2. The final results must be divided by M^2 to generate values equivalent to the floating point before quantization.

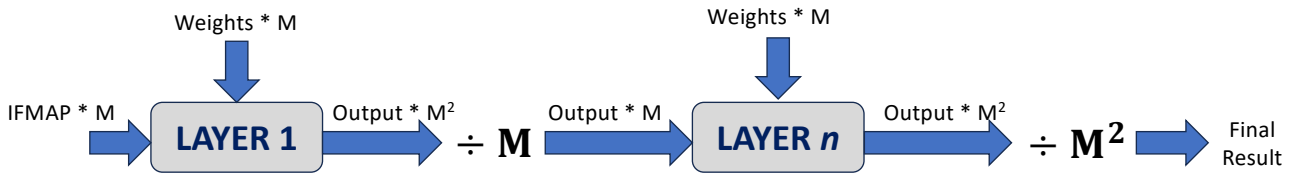


Figure 5.2 – Connection between layers using a multiplication factor.

5.3.1 CNN Integer C Model

The *C model* is the same with representation in floating point and integers, not requiring changes in the previously presented codes. Only two changes are needed in the code. The first is to change the data type from *float* to *int*. The second change is to include the header files with integers using the “Quantizator.py” script. Code 5.11 displays variables of the first layer in a CNN with floating point presentation and in a CNN with integer presentation.

5.4 Results

This Section evaluates four performance figures for the three models: accuracy, number of executed instructions (profiling), execution time, and energy consumed.

Code 5.11 – Changes in the C code for floating point and integer representations.

```
// Float
float conv0_currentKernel[KERNEL_SIZE];
float conv0_current_bias = 0;
float conv0_featureMap[NUM_FILTERS][CONV0_INPUT_SIZE-4];

// Integer
int INTconv0_featureMap[NUM_FILTERS][CONV0_INPUT_SIZE-4];
int INTconv0_currentKernel[KERNEL_SIZE];
int INTconv0_current_bias = 0;
```

5.4.1 Model Accuracy

Initially, to validate the correctness of the C model, we tested each layer individually using controllable inputs. This validation used Excel spreadsheets, comparing the results against Pytorch. Minimal differences in accuracy were observed due to the Python internal parameter normalization. After validating the method to compute the CNN outputs analytically, we implemented the script to extract the data from Pytorch (Section 5.1) and the C models (Section 5.2).

Table 5.3 and Code 5.12 present the model size and accuracy for the three models, using the testing dataset (3,923 samples). The reference model showed a slight increase in accuracy (0.9%). This occurs because the testing dataset used by PyTorch (20%) and C models may differ. The accuracy values in PyTorch correspond to an average of several executions of the inference phase. The Ext1D FP model increased the accuracy by 13.6% ($88.5 - 74.9$) due to the reduction in the wrong predictions, from 207 to 80. The Ext1D INT model slightly reduced accuracy, 1.3%, due to an increase in wrong predictions, from 80 to 90.

Table 5.3 – Model size (number of parameters) and accuracy comparison between Pytorch and C models, using the reference and optimized models (Ext1D).

Model	**Model Size**	**Accuracy (%)**
Pytorch - Reference	271,621	74.0
Pytorch - Ext1D	926,981	**89.0**
Reference Model (C)	271,621	74.9
Ext1D FP (C)	926,981	88.5
Ext1D INT (C)	926,981	87.2

Code 5.12 – Confusion matrices for the C models.

```
----------------------
Accuracy: 74.9  - Reference
----------------------
Correct predictions: 613
Wrong predictions:   207
----------------------
Confusion Matrix
     0    1    2    3    4
0:  123    4    6    2    0
1:   47  122   42    2    5
2:    0    9  123    6    3
3:    0    0   64  123    8
4:    0    2   10    7  122
```

```
----------------------
Accuracy: 88.5 - Ext1D FP
----------------------
Correct predictions: 615
Wrong predictions:    80
----------------------
Confusion Matrix
     0    1    2    3    4
0:  123    1    1    0    0
1:   17  123    6    0    6
2:    0    5  123    2    0
3:    0    0   33  123    2
4:    0    0    3    4  123
```

```
------------------
Accuracy: 87.2 - Ext1D INT
------------------
Correct predictions: 615
Wrong predictions:    90
------------------
Confusion Matrix
     0    1    2    3    4
 0:  123   20    9    3    0
 1:   35  122   55    3   10
 2:    0    7  123   21    6
 3:    0    0   41  123   18
 4:    0    0    6    5  123
```

This result shows the effectiveness of the quantization method presented in Section 5.3, which allows using this CNN model on processors without FPU cores.

5.4.2 Instruction Profiling

To evaluate the implementations made in the C language, we used the OVP simulator [Imperas, Software Ltd, 2023], with the RISCV32IMF processor (RISCV32I: Base 32-bit Integer Instruction Set, M: Integer Multiplication and Division, F: Single-Precision Floating-Point) [RISC-V Foundation, 2019]. We partitioned the instruction set according to Table 5.4, instrumenting the OVP simulator through callbacks that evaluate each instruction fetched from memory. The compilation parameters use the OVP cross compiler for RISCV32IMF, with O2 optimization.

Table 5.4 – Partition of the RISCV32IMF processor for instruction profiling.

Instruction Class	**Opcodes**
LOAD	`LB`, `LH`, `LW`, `LBU`, `LHU`, `FLW`
STORE	`SB`, `SH`, `SW`, `FSW`
IMM	`LUI`, `AUIPC`, `SLTI`, `SLTIU`, `XORI`, `ORI`, `ANDI`, `SLLI`, `SRLI`, `SRAI`
REG	`SLL`, `SLT`, `SLTU`, `XOR`, `SRL`, `SRA`, `OR`, `AND`
ADDSUB	`ADD`, `ADDI`, `SUB`
MULT	`MUL`, `MULH`, `MULHSU`, `MULHU`
DIV	`DIV`, `DIVU`, `REM`, `REMU`
BRANCH	`BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, `BGEU`
JUMP	`JAL`, `JALR`
FP_MUL	`FMUL.S`, `FMADD.S`, `FMSUB.S`, `FNMSUB.S`, `FNMADD.S`
FP_DIV	`FDIV.S`
FP_AddSub	`FADD.S`, `FSUB.S`
FP_OP	all other floating point operations

The RISCV32IMF processor has instructions that implement the MAC function, called “single precision fused multiply addition”: `FMADD.S`, `FMSUB.S`, `FNMSUB.S`, `FNMADD.S`. These instructions reduce the number of instructions when executing floating point models.

In addition to accounting for the number of executed instructions, instruction profiling is used to:

1. evaluate the number of memory accesses (LOAD/STORE), which implies energy consumption and processing time due to the memory access latency;
2. quantify the number of executed arithmetic operations, integer (ADDSUB, DIV, DIV) or floating point (FP_AddSub, FP_MUL, FP_DIV).

Table 5.5 presents the profiling for the reference, Ext1D FP and Ex1D INT models. This Table shows that:

1. Memory Access: increased by 3.7 times. Expected increase since the Ext1D model increases the model size (refer to Table 5.3) by 3.4 times. The increase in model size also implies an increase in the total number of executed instructions by 3.7 times.

2. Arithmetic Operations: the number of multiplications (FP and INT) is equal to that obtained in Table 5.2 (5,428,096), showing that the C implementation did not perform unnecessary multiplications.

3. The FP implementation does not require division operations, while INT does, given the need for divisions in the quantization adjustments between the layers.

4. Integer addition/subtraction operations. In the INT model, we observe an increase of 5,507,305 operations. This is due to the MAC instructions in the FP instruction set. This increase reflects the number of instructions executed in the INT model.

Table 5.5 – Instruction profiling of the CNN models using the RISC-V processor.

Instruction Class	**Reference Model**	**Ext1D FP**	**Ext1D INT**
LOAD	**2.945.844**	**10.871.684**	**10.893.314**
STORE	12.179	41.946	63.668
IMM	90	53	46
REG	0	0	0
ADDSUB	4.944.252	18.196.854	23.704.159
MULT	0	0	**5.428.096**
DIV	0	0	21.637
BRANCH	1.737.527	6.401.926	6.423.755
JUMP	242.376	887.496	887.496
FP_OP	**16.406**	**57.358**	0
FP_MUL	**1.470.336**	**5.428.096**	0
FP_DIV	0	0	0
FP_AddSub	**6.277**	**21.637**	0
Total	**11.375.287**	**41.907.050**	**47.422.171**

The summary of this table is: (1) an increase in the number of executed instructions to increase the accuracy, 13.6%, at the cost of 3.7 times more executed instructions; (2) a loss of only 1.3% in accuracy when using the Ext1D INT model.

5.4.3 Execution Time

We evaluated the three models executing 3,923 inferences on an Apple M2 processor (16GB of SRAM, 2.4 GHz), using a clang 14.0.3 compiler with O2 optimization. Table 5.6 presents the time to execute the 3,923 inferences and the time per inference. This is an average result over five executions of each model.

Table 5.6 – Execution time to execute the models (Apple M2 processor).

	Reference Model	**Ext1D FP**	**Ext1D INT**
Total time (sec)	7.885	28.568	4.175
Time per inference (ms)	2.010	7.282	1.064

The time per inference increased by 3.6 times due to the increased model size (Ext1D FP). The Ext1D INT model is 6.84 times faster than the FP model (the only modification in the C code is the type of the variables and the added division layers), and despite the increase in model size, the Ext1D INT model is 1.9 times faster than the reference model.

To illustrate the influence of processor architecture on execution time, Table 5.7 displays the execution time for an Intel i7 Quad-Core processor (16GB of SRAM, 2.8 GHz), using the clang version 11.0.0 compiler (clang-1100.0.33.8). We observe that this CISC architecture operates more efficiently with floating point (FP), with the ratio between FP and INT operations being 3.25 times instead of 6.84 times in the M2 processor (RISC ARM cores). We keep the M2 architecture for the other results due to its RISC architecture and more efficient integer operations handling.

Table 5.7 – Execution time to execute the models (Intel i7 processor).

	Reference Model	**Ext1D FP**	**Ext1D INT**
Total time (sec)	6.252	21.743	6,694
Time per inference (ms)	1.594	5.542	1.706

This result demonstrates the significant latency of floating-point operations. Although the integer representation executes 47.4 million instructions, compared to 11.4 million in the reference model (4.17 times more instructions), the integer representation is 1.9 times faster in the M2 processor and 7% slower in the i7 processor. Despite accounting for instructions on the RISC-V processor and execution time on the other processors, [Ness,

2018] shows that the latency is 2 to 8 times higher for floating-point instructions than for integer instructions, which is consistent with the obtained results.

5.4.4 Energy Consumption

The energy consumption evaluation should be considered cautiously, as the data are obtained from different sources. In [Zaruba and Benini, 2019], the average cost for executing integer operations is 20pJ@1.7GHz, at 22nm. BOOM-2w [Celio et al., 2015] and Shakti [Shakti, 2023], both with a RISCV-IMAFD ISA, the energy cost per floating point operation is 133pJ and 122 pJ, respectively. In [Ness, 2018], the FMAC operation consumes 16.519 pJ@100Mz, for a 55nm technology. Considering that the 55nm is two technological nodes above 22nm (0.7^2) and that the frequency is 17 times lower, we transpose the energy value to 1.7GHz at 22nm by computing $energy = 16.519 \times (0.7^2) * 17 = 137.6pJ$, a value close to that obtained in the literature. Thus, we use 20pJ for integer instructions and 131pJ (average of 133, 120, 137.6) for floating point instructions.

Table 5.8 presents the estimated energy for the three models. To estimate the energy we used Table 5.5 (instruction profiling). Integer instructions were multiplied by 20pJ, and floating point instructions by 131pJ. Consistently, the energy consumed by the FP model concerning the reference model increases by 3.7 times due to the model size. However, due to the lower energy cost of INT operations than the cost of FP operations, the energy consumed by the integer model is only 2.41 times higher.

Table 5.8 – Energy consumption to execute the models (20pJ for integer operations and 131pJ for floating point operations).

	Reference Model	**Ext1D FP**	**Ext1D INT**
Energy (pJ)	393	1,449	948

5.5 Final remarks

We may summarize the comparison of the **reference C model**, which uses floating point, with the **Ext1D INT** implementation as follows:

- Model Size: **3.41 times larger** (in terms of the number of parameters)
- Accuracy: **12.3% better**, reaching 87.2%
- Number of executed instructions (from the instruction profiling): 4.17 times higher

- Execution time: 47% faster (Apple M2 processor)
- Energy: 2.41 times higher (estimated from the instruction profiling and energy values from the literature)
- Instruction set: **integer**

The Ext1D INT successfully achieved its main goal: improve accuracy without using floating point representation. However, one challenge remains, the model size. The memory footprint is a critical parameter in embedded systems with limited resources. The next chapter presents a technique to reduce the memory footprint without sacrificing accuracy, keeping the use of integer representation. On the other hand, the reduction in the memory footprint will come at the cost of increased processing requirements, which will negatively affect instruction profiling, execution time, and energy.

6. MODEL OPTIMIZATION AIMING MEMORY USAGE REDUCTION

This chapter corresponds to the *third contribution* of this MSc work, which is the reduction of the CNN model to use it on devices with memory restrictions. The previous chapter presented the feasibility of representing data for a Convolutional Neural Network (CNN) using integer representation (32 bits), allowing the use of processors without a Floating Point Unit (FPU). This chapter aims to meet another design requirement: reducing the CNN model size and consequently decreasing the amount of memory required to model the CNN.

Reducing the number of bits to represent the data is a method many researchers use, where the extreme case is binary networks where weights and biases have 1 bit [Shimoda et al., 2017]. Considering that we are working on a processor without an FPU, an alternative would be an 8-bit representation (*char* type). We discarded this option due to the overflow occurring in arithmetic operations, which would negatively impact accuracy. An option is to train the CNN model using 8-bit representation precision, which was not done in this work.

Another option is the use of Lookup Tables (LUTs). The original values of the weights and biases are grouped into bins. A LUT with n bins contains the model values, replacing the original values with indexes pointing to a position in the LUT [Wang et al., 2022]. Consider, for instance, a CNN layer with 100,000 parameters, partitioning these values into 32 bins (LUT with 1024 bits). Each parameter previously represented by 32 bits is now represented by 5 bits. The original parameters require 3,200,000 bits, while index representation requires 501,024 bits, representing a memory requirement reduction of 6.38 times.

Initially, in this chapter, we do not consider memory reduction as a primary goal but rather assess the impact of quantization on the CNN model, and the selection of a bin value that does not negatively impact accuracy. Section 6.1 presents the quantization methods adopted in this work. Section 6.2 evaluates the quantization methods in terms of accuracy and selects a bin size to be used by the CNN model. Section 6.3 evaluates the impact of quantization on the number of executed instructions, execution time, and consumed energy. Section 6.4 corresponds to the main contribution of this chapter by presenting the method to achieve the goal of memory reduction. Section 6.5 concludes this chapter.

6.1 Quantization Model

A Look-Up Table (LUT) is a simple and efficient method for performing quantization. When performing quantization using LUTs, the input values are the indices of the LUT.

The corresponding quantized values are the entries stored in the LUT. This way, quantization becomes as simple as accessing an array element. In this work, we adopt two LUT quantization methods, fixed and variable:

- Fixed quantization employs bins of fixed size. To find the bin widths, the maximum and minimum values of the input values are calculated and then divided by the number of bins. Equations 6.1 and 6.2 show the computation of the lower and upper limits of each bin i. Once the intervals are defined, each parameter is assigned the index of the interval into which it falls. Fixed bin sizes provide a consistent and easy-to-interpret data representation. However, it may not be optimal for datasets with varying densities or when important features are masked by bin size selection.

$$\text{Lower Bin Limit}_i = \text{Minimum Value} + i \times \left(\frac{\text{Maximum Value} - \text{Minimum Value}}{\text{Number of Bins}} \right) \tag{6.1}$$

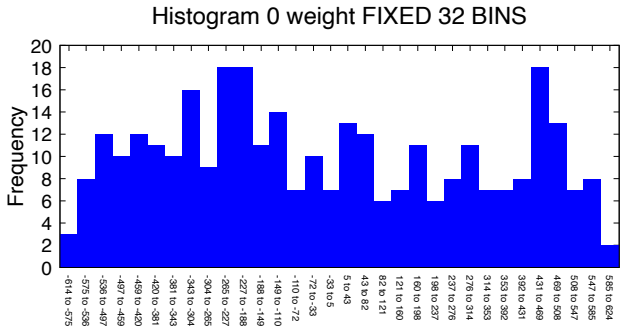
$$\text{Upper Bin Limit}_i = \text{Minimum Value} + (i+1) \times \left(\frac{\text{Maximum Value} - \text{Minimum Value}}{\text{Number of Bins}} \right) \tag{6.2}$$

- Variable quantization seeks to have an equal number of elements in all bins. Equation $n\_elem = \frac{\#parameters}{nbins}$ gives the number of elements in each bin. To execute a variable quantization, the input parameters are sorted, inserting n_elem parameters into each bin. This approach offers flexibility in capturing distribution details and is useful for parameters with varying densities or outliers. In a normal distribution, the tails (the extreme ends) have a lower density than the central region. Using variable bin sizes allows for allocating larger bins to the tails where data points are sparser, thus capturing the distribution shape more accurately in these regions. Another advantage of variable bin sizes is outlier detection, with smaller bins allocated around outlier regions, making them more distinguishable.

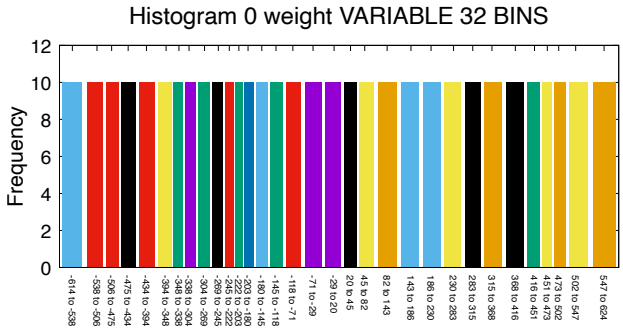
The value used in each bin is the central value of the bin. We evaluate in Section 6.2.2 the mean or median value of the bin and find no advantages in using these locations for the bin. Thus, we decided to use the central value of the bin.

To visualize the impact of quantization, we present histograms, considering 32 bins as an example. Histograms are graphical representations of the parameters distribution. They consist of a series of bars, each representing a range of values, and the height of the bar represents the frequency or number of values within that range. Figures 6.1, 6.2, 6.3, and 6.4 present distributions for weights and biases for the three convolutional layers and the two fully connected layers.

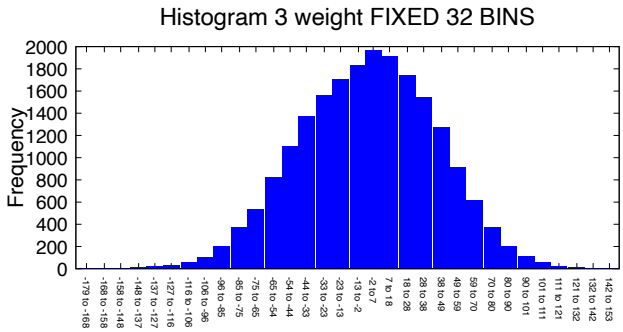
Fixed-width bin histograms capture the original parameter distribution; we note histograms with a clearly defined normal distribution (weights of the 2nd and 3rd convolutions –



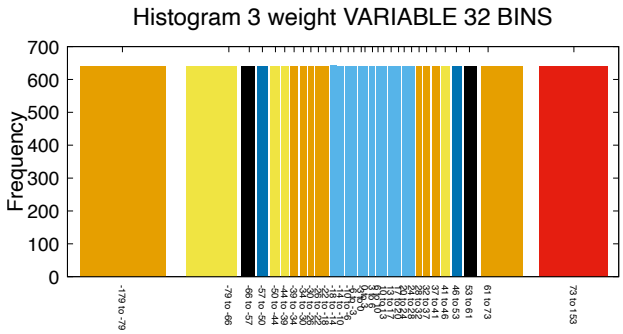
(a) **CONV 1 - 32 bins - fixed quantization.**



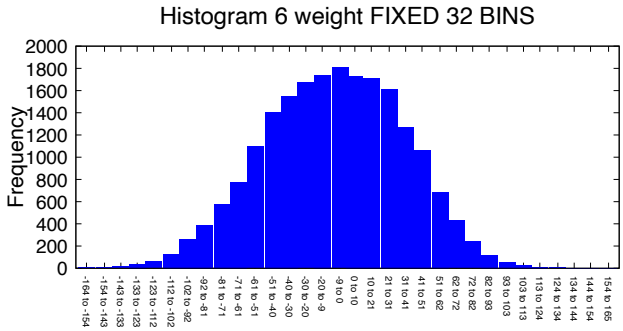
(b) **CONV 1 - 32 bins - variable quantization.**



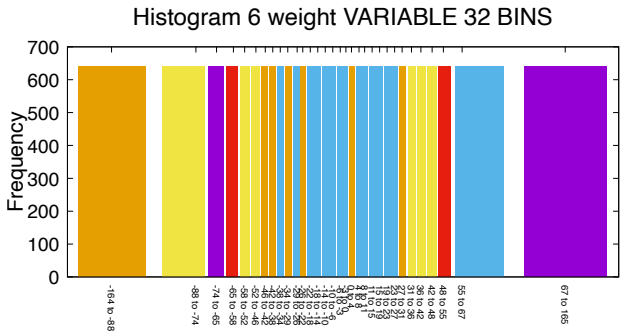
(c) **CONV 2 - 32 bins - fixed quantization.**



(d) **CONV 2 - 32 bins - variable quantization**



(e) **CONV 3 - 32 bins - fixed quantization.**

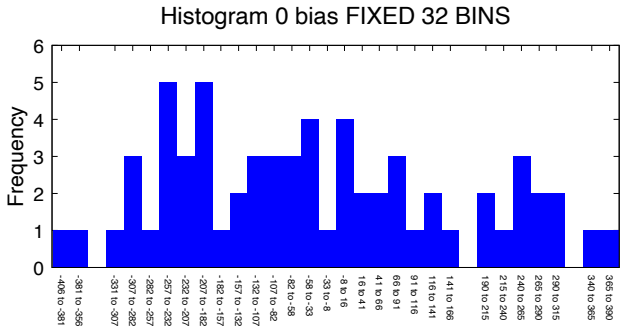


(e) **CONV 3 - 32 bins - variable quantization.**

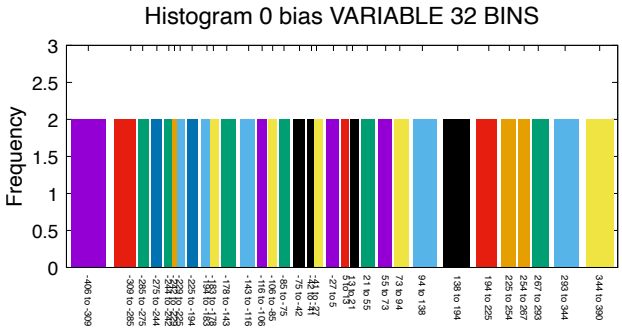
Figure 6.1 – Quantization of the **weights** for the convolutional layers - number of parameters: 320 – 20480 – 20480.

Figures 6.1(c) and 6.1(d)). However, clearly defined distributions are absent in other fixed-width histograms, such as the 1st fully connected layer weights, with about 350,000 parameters concentrated in a single bin (Figure 6.3(a)).

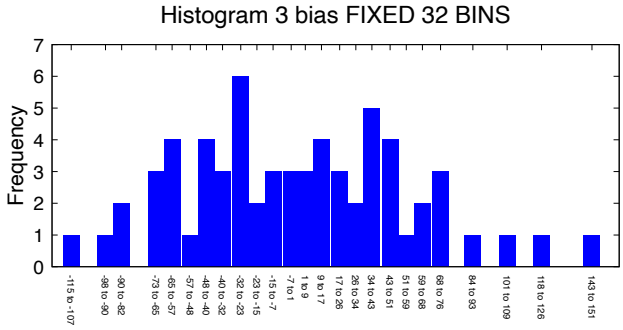
Observe the histograms with variable bin widths. The width of the bin is inversely proportional to the number of values in the original distribution. The tails, which have fewer parameters, present wider bins, whereas regions with more parameters have thinner bins. The most representative example of this behavior is observed in the weights of the 1st fully connected layer, concentrating most parameters in a small region, now equally distributed in the variable bin distribution (Figure 6.3(b)).



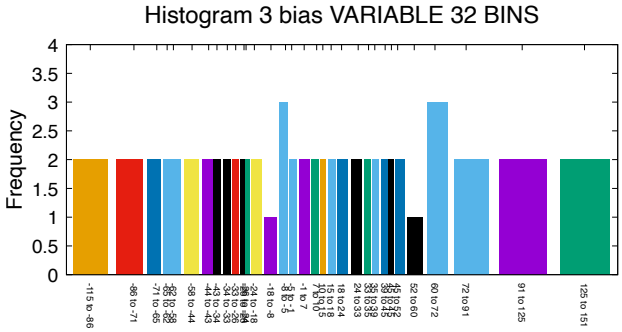
(a) **CONV 1 - 32 bins - fixed quantization.**



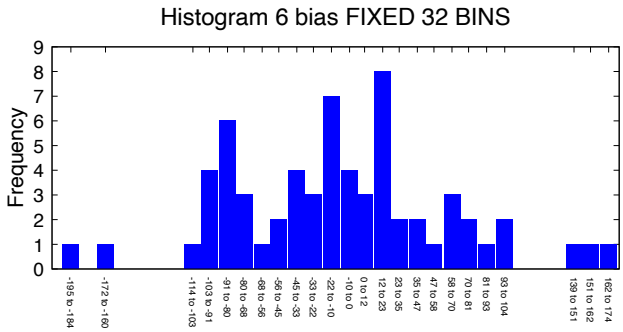
(b) **CONV 1 - 32 bins - variable quantization.**



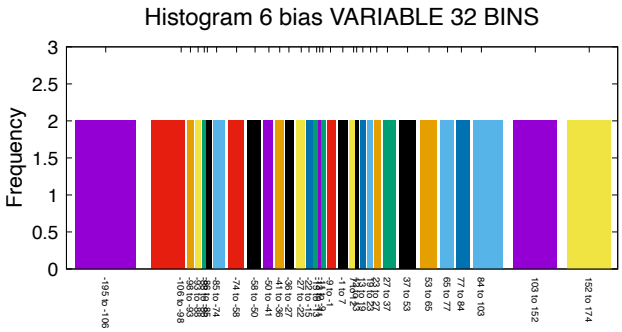
(c) **CONV 2 - 32 bins - fixed quantization.**



(d) **CONV 2 - 32 bins - variable quantization**



(e) **CONV 3 - 32 bins - fixed quantization.**

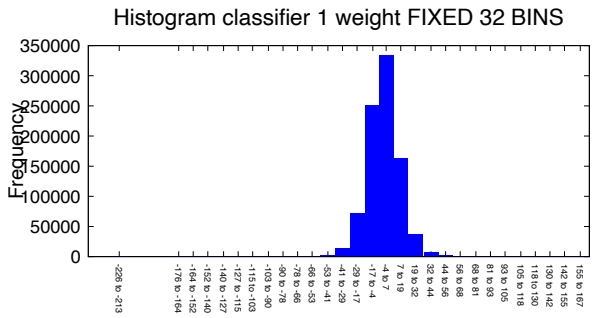


(e) **CONV 3 - 32 bins - variable quantization.**

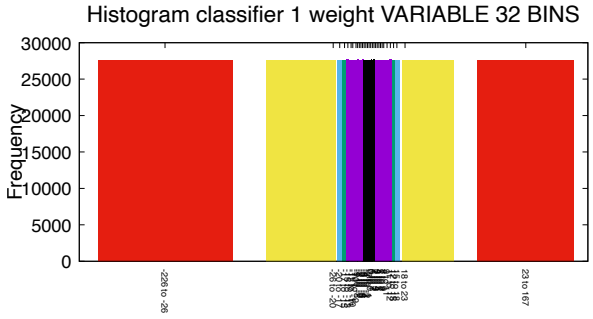
Figure 6.2 – Quantization of the **bias** for the convolutional layers - number of parameters: 64 – 64 – 64.

All these histograms are generated per layer. Another option studied is to use a single LUT for the entire CNN. The issue with this approach is the minimum and maximum ranges of each layer [Xu et al., 2021], as shown in Table 6.1. There would be a problem with elements per bin since quantization should occur between all layers' minimum and maximum values. There would be layers where all values fall into a single bin. Given that the memory consumption of LUTs is much smaller than the parameter vectors, quantizing the entire model in a single LUT was not considered.

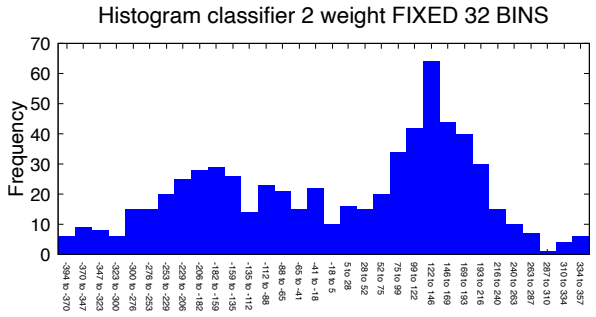
There is also the possibility of using a different number of bins per layer to maximize accuracy. This is a suggestion for future work and was not covered in this MSc due to time constraints.



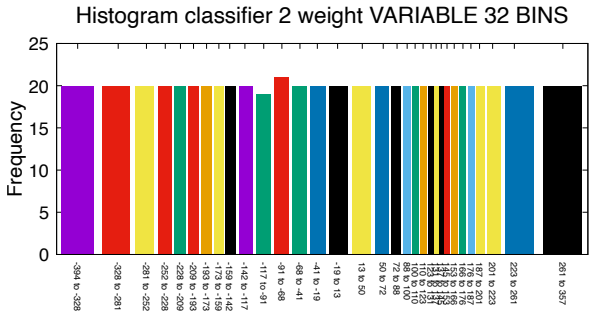
(a) **Fully connected 1 - 32 bins - fixed quantization.**



(b) **Fully connected 1 - 32 bins - variable quantization.**

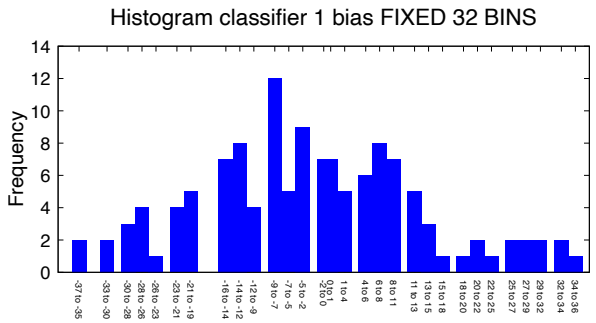


(c) **Fully connected 2 - 32 bins - fixed quantization.**

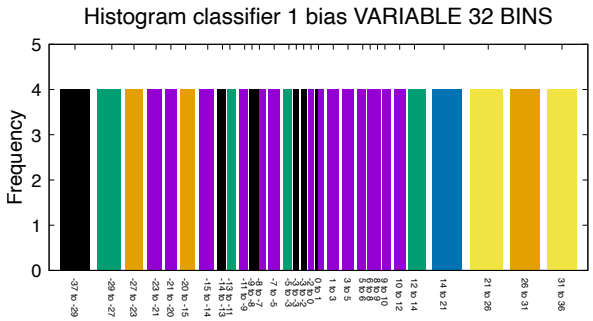


(d) **Fully connected 2 - 32 bins - variable quantization**

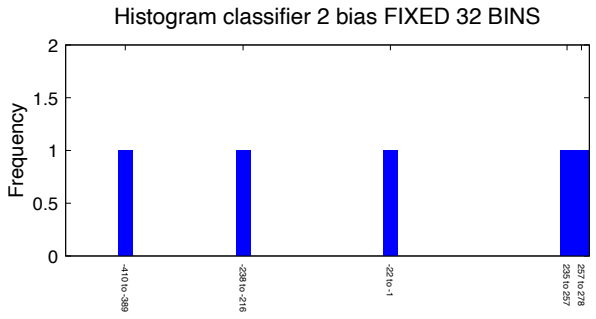
Figure 6.3 – Quantization of the **weights** for the fully connected layers - number of parameters: 884,736 – 640.



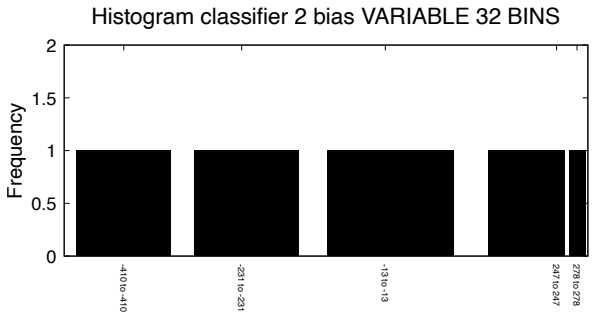
(a) **Fully connected 1 - 32 bins - fixed quantization.**



(b) **Fully connected 1 - 32 bins - variable quantization.**



(c) **Fully connected 2 - 32 bins - fixed quantization.**



(d) **Fully connected 2 - 32 bins - variable quantization.**

Figure 6.4 – Quantization of the **bias** for the fully connected layers - nb. parameters: 64 - 5. The number of parameters of the second fully connected layer is smaller than the number of bins.

Table 6.1 – Minimal and maximum values per layer (integer quantization.

Layer	**Minimal Value**	**Maximum Value**
CONV weight 0	-614	624
CONV weight 1	-179	153
CONV weitht 3	-164	165
CONV bias 0	-115	151
CONV bias 1	-195	174
CONV FC1 weight	-226	167
CONV FC2 weight	-394	357
CONV FC1 bias	-37	36
CONV FC2 bias	-410	278

6.2 Quantization Evaluation and Bin Selection

Table 6.2 displays the accuracy data, varying: (1) bin size, (2) data representation, and (3) the type of quantization. The quantization method uses one LUT per layer.

Table 6.2 – Accuracy per bin size, varying the quantization method. Reference values: 88.5% for floating point representation and 87.2% for integer representation.

Accuracy	Float representation %		Integer representation %	
Number of bins	**Fixed**	**Variable**	**Fixed**	**Variable**
256	89.20	87.50	87.4	87.0
128	88.20	88.40	86.9	86.0
64	88.10	87.10	87.1	86.7
32	88.20	86.70	86.5	86.7
16	88.00	86.90	86.0	**87.1**
8	47.30	85.80	38.9	85.5
4	28.50	75.70	28.2	75.7
2	20.00	47.90	22.3	41.6

Note that the loss of accuracy is minimal with LUTs containing up to 16 positions (bin size equal to 16). The most significant difference for floating-point representation is 1.6%, with a slight advantage for fixed quantization. The same behavior is observed for integer representation, with the most significant difference being 1.2%. This is a surprising result considering that there is a layer with 884,736 parameters, and it is possible to represent it with just 16 values, with minimal loss in accuracy. Another unexpected data is the equivalence between fixed and variable quantizations, given how the quantization is executed. The most significant difference observed in the floating-point representation was 1.73% between the quantization methods; for the integer representation, it was 1.26%. This result suggests

that the number of bins equal to 16, for this CNN model, is the value to be adopted for the remainder of this work.

It is also worth noting the limits these quantization methods have. For fixed quantization in this CNN, 16 bins is the minimum value. Quantizations with 8, 4, and 2 bins fail to generate an acceptable accuracy value. However, with variable quantization, we can have an aggressive quantization, with only eight bins, with little loss of accuracy (-2.7% for float and -1.7% for integer). Therefore, when adopting quantization methods, it is recommended to adopt variable quantization, especially if the designer adopts an aggressive strategy for value reduction.

6.2.1 *Model-wise* quantization

We also evaluated the use of a single LUT for the entire CNN. We call this model *model-wise* quantization. Table 6.3 displays the accuracy data, varying: (1) bin size, (2) data representation, and (3) the type of quantization, using a single LUT with all weights and bias of all layers.

Table 6.3 – Accuracy per bin size, varying the quantization method, using the *model-wise* quantization. Reference values: 88.5% for floating point representation and 87.2% for integer representation.

Accuracy	Float representation %		Integer representation %	
Number of bins	**Fixed**	**Variable**	**Fixed**	**Variable**
256	89.10	39.70	86.4	20.0
128	88.20	27.80	85.3	20.0
64	89.10	20.00	**86.7**	20.0
32	79.40	20.00	75.3	20.0
16	24.60	20.00	20.0	20.0
8	20.00	20.00	20.0	20.0
4	20.00	20.00	20.0	20.0
2	20.00	20.00	20.0	20.0

Observe that this CNN has five possible output classes, and hence, the accuracy equal to 20% presented in many data points in Table 6.3 corresponds to a random prediction. The *model-wise* quantization method yields mixed results. As expected, the accuracy was negatively affected due to the increase in the parameters represented in each LUT. As discussed in Section 6.1, quantization should occur between the minimum and maximum values. However, as Figures 6.1, 6.2, 6.3, and 6.4 illustrate, the values exhibit different distributions and values, leading to an unequal representation of all layers in the LUTs. In a feedforward neural network, a misparameterized layer can significantly impact accuracy [Goodfellow et al., 2016]. This issue can be mitigated by using a **fixed** bin size that does not

rely on the population size of each layer to define the boundaries. This approach prevents a layer with a larger population, such as the first fully connected layer, from significantly influencing the computation. It is important to note this effect when using LUTs containing up to 256 bins. Accuracy becomes unusable when using **variable** bin sizes, which is the case for both floating and fixed point representations.

Note that there are two highlighted values in Tables 6.2 and 6.3. These values indicate quantization options using integer representation, with an accuracy close to the reference accuracy (87.2%). For quantization using one LUT per layer, we can use 16 bins with variable quantization (87.1%), and for the *model-wise* method, we can use 64 bins with fixed quantization (86.7%). We decided to use one LUT per layer because this model does not rely on the values of the entire CNN. If another CNN model is adopted, it may be necessary to use larger LUTs, or it may not even be possible to achieve high accuracy values, as observed with variable quantization.

6.2.2 Bin Position Selection

The accuracy values presented so far use the central value of the bin. In Tables 6.4 and 6.5, we evaluate the impact of using the median and average values inside the bin using the *model-wise* quantization. The "central" values correspond to the accuracy obtained in Table 6.3.

Table 6.4 – Effect of the value selected inside the bin (central, median, mean) for **fixed** quantization.

Accuracy	Float representation, *model-wise* fixed quantization		
Number of bins	**Central**	**Median**	**Mean**
256	89.1	89.0	84.2
128	88.2	88.5	81.5
64	89.1	82.3	77.7
32	79.4	55.7	67.3
16	24.6	20.0	20.0
8	20.0	20.0	20.0
4	20.0	20.0	20.0
2	20.0	20.0	20.0

For fixed quantization, using mean or median values does not bring accuracy benefits. On the other hand, in variable quantization, we obtained a significant improvement for 256 bins. This improvement is explained by the larger number of elements in each bin, and using the mean or median better captures the behavior of the parameters distribution. An

Table 6.5 – Effect of the value selected inside the bin (central, median, mean) for **variable** quantization.

Accuracy	Float representation, *model-wise* variable quantization		
Number of bins	**Central**	**Median**	**Mean**
256	39.7	76.2	71.4
128	27.8	69.1	35.0
64	20.0	59.8	20.0
32	20.0	20.0	20.0
16	20.0	20.0	20.0
8	20.0	20.0	20.0
4	20.0	20.0	20.0
2	20.0	20.0	20.0

in-depth study to select the value position in each bin is a suggestion for future work.

In conclusion of this Section, for this CNN, we adopted in the remainder of this chapter: (*i*) integer representation; (*ii*) one LUT per layer; (*iii*) variable quantization; (*iv*) 16 bins per LUT; (*v*) the central bin value.

6.3 Quantization Profiling

Tables 6.6 and 6.7 present the instruction profile, execution time, and estimated energy, comparing the standard models to those using LUTs with 16 bins. The use of quantization implies a larger number of memory accesses (`LOAD` instructions), a larger number of executed instructions, and consequently, an increase in the time to execute the inference and energy consumption. Note that the increase in the number of executed instructions is roughly the same for the float (17,026,844) and integer representations (17,026.644).

To understand the reason for the increase in memory accesses and the number of executed instructions, we present in Code 6.1 the same convolution presented in Code 5.8, but now using LUTs. Observe lines 8 and 11 of Code 6.1. On these lines, we are first accessing the LUT index and then accessing the value in the LUT. Thus, we execute two memory accesses instead of just one and add `IMM`-type instructions to access the LUT indexes. Note that the number of `IMM` mode operations roughly corresponds to the number of `MULT` operations.

The results obtained in this section suggest that there is no benefit in using LUTs when running the CNN model in software, as the performance figures are worse compared to the standard model. Using quantization and LUTs is appropriate for **hardware** imple-

Table 6.6 – Instruction profiling of the CNN models using the RISC-V processor.

Instruction Class	**Ext1D FP**	**Ext1D 16bins FP**	**Ext1D INT**	**Ext1D 16bins INT**
LOAD	**10,871,684**	**16,249,344**	**10.893.314**	**16,270,975**
STORE	41,946	41,756	63.668	63,187
IMM	53	5,377,703	46	5,377,697
REG	0	0	0	0
ADDSUB	18,196,854	24,468,579	23.704.159	29,975,949
MULT	0	0	5,428,096	5,428,096
DIV	0	0	21.637	21,637
BRANCH	6,401,926	6,401,926	6.423.755	6.423,755
JUMP	887,496	887,496	887.496	887.496
FP_OP	57,358	57,358	0	0
FP_MUL	5,428,096	5,428,096	0	0
FP_DIV	0	0	0	0
FP_AddSub	21,637	21,637	0	0
Total	**41,907,050**	**58,933,895**	**47,422,171**	**64,448.792**

Table 6.7 – Execution time to execute the models (Apple M2 processor), and estimated energy (20pJ for integer operations and 131pJ for floating point operations).

	Ext1D FP	**Ext1D 16bins FP**	**Ext1D INT**	**Ext1D 16bins INT**
Total time (sec)	28.568	34.230	4.175	11.860
Time per inference (ms)	7.282	8.725	1.064	3,023
Energy (pJ)	1,449	1,790	948	1,289

Code 6.1 – 1D convolution layer code, using LUTs to access the model parameters.

```
for (int filterToGenerate=0 ; filterToGenerate<NUM_FILTERS ; filterToGenerate++ ){
    for (int inputOffset=0 ; inputOffset<CONV3_INPUT_SIZE-4 ; inputOffset++){
        INTconv3_totalSum = 0;
        for (int filterIn=0 ; filterIn<NUM_FILTERS; filterIn++){
            for (int kernelIndex = 0 ; kernelIndex < KERNEL_SIZE ; kernelIndex++){
                int weightIndex = kernelIndex + (filterIn * KERNEL_SIZE) + ( filterToGenerate * NUM_FILTERS *
↪  KERNEL_SIZE ) ;
                int indexIn = kernelIndex + (inputOffset);
                INTconv3_totalSum +=INTconv0_featureMap[filterIn][indexIn]  *
↪  conv3_weights_lut[conv3_weights_indices[weightIndex]];
            }
        }
        INTconv3_totalSum += conv3_bias_lut[conv3_bias_indices[filterToGenerate]];
        INTconv3_featureMap[filterToGenerate][inputOffset] = INTconv3_totalSum;
    }
}
```

mentation since the parameters can be represented with fewer bits, and the LUTs can be stored in registers, avoiding unnecessary memory accesses. However, employing a technique for packing indices into a single 32-bit word can significantly reduce the model size, as we demonstrate in the next section.

6.4 Compressed Model

The compressed model modifies the weight and bias files, which have already been quantized, by “packing” eight values into a single value. Code 6.2 shows an example of the compression method for the weights of the first fully connected layer. Note that the number of parameters has been divided by 8, and each vector position packs eight indices. For instance, the first value, `26144250` or `0x018EEDFA`, corresponds to the first eight parameters (10, 15, 13, 14, 14, 8, 1, 0). A script executes this data packaging for the compressed model.

Code 6.2 – Original parameter array and compressed array.

```
const char fc1[884,736] = {10,
15,
13,
14,
14,
8,
1,
0,
15,
5,
5,
13,
8,
7,
1,
7,
6,
4,
6,
2,
9,
7,
1,
14,
14,
11,
1,
15,
14,
...
4,
11,
8};
```

```
const int fc1_comp[110,592] = {26144250,
1903744351,
3782813254,
1041166782,
4016513328,
2493242926,
1960532140,
1951286549,
1232396650,
2139000964,
...
...
3521681353,
2336831745};
```

The compressed model requires more computational effort to execute the CNN. Code 6.3 describes a convolutional layer. In this code:

- The variable `realIndex` (`weightIndex` or `filterToGen`) – lines 9 and 19, corresponds to the address in the original dataset vector.
- Lines 11 and 21 compute the address in the compressed dataset (`realIndex >> 3`). As we are using 16-bin LUTs, it is necessary to use 4 bits to encode an index. Thus, each 32-bit word encodes eight 4-bit indices, as presented in Code 6.2.
- Lines 10 and 20 isolate the three least significant bits of the current address. Memory is only accessed when this value is zero, resulting in one memory access for every eight addresses of the compressed vector. *This procedure is responsible for reducing the number of `LOAD` operations*.
- Lines 15 and 25 access the LUT. The operation `weightDecompIndex*4` defines how many bits the read word should be shifted (0, 4, .. 28) to obtain the LUT index. Additional masking (`& 0x0F`) is necessary to ensure that the index corresponds to one of the 16 positions in the LUTs. Once the LUT index is obtained, the LUT is accessed.

Code 6.3 – 1D convolution layer code, using for the compressed model.

```
1  for (int filterToGen=0; filterToGen<NUM_FILTERS ; filterToGen++){
2     for (int inputOffset=0 ; inputOffset<CONV3_INPUT_SIZE-4 ; inputOffset++){
3        totalSum = 0;
4        for (int filterIn = 0 ; filterIn < NUM_FILTERS ; filterIn++){
5           for (int kernelIndex = 0 ; kernelIndex < KERNEL_SIZE ; kernelIndex++){
6              int weightIndex = kernelIndex + (filterIn * KERNEL_SIZE) + ( filterToGen * NUM_FILTERS *
   ↪  KERNEL_SIZE ) ;
7              int indexIn = kernelIndex + (inputOffset);
8
9              int realIndex = weightIndex;
10             weightDecompIndex = realIndex & 7;
11             weightCompressedIndex   = realIndex >> 3;
12             if (!weightDecompIndex)
13                 weightCompData = conv3_weights_indices_compressed[weightCompressedIndex];
14
15             totalSum +=  conv0_featMap[filterIn][indexIn] *
   ↪  (weights_lut[(weightCompData>>(weightDecompIndex*4)) & 0x0F]);
16          }
17       }
18
19       int realIndex = filterToGen;
20       biasDecompIndex = realIndex & 7;
21       biasCompressedIndex   = realIndex >> 3;
22       if (!biasDecopmIndex)
23           biasCompData = bias_indices_compressed[biasCompressedIndex];
24
25       totalSum += conv3_bias_lut[ (biasCompData>>(biasDecompIndex * 4)) & 0x0F ];
26
27       INTconv3_featureMap[filterToGen][inputOffset] = totalSum;
28    }
29 }
```

Table 6.8 presents the instruction profile, comparing the standard integer model, the model with 16-bin LUTs, and the compressed one. The increased complexity of the code increased `REG` – shift operations, `IMM` – logical operands such as AND, and `BRANCH` instructions. Comparing the number executed of instructions, the compressed model doubled the number of instructions compared to the standard model with integer representation.

Despite the increase in the number of instructions, an important gain must be highlighted, the number of memory accesses. The compressed model has almost the same memory accesses as the standard model.

Table 6.8 – Instruction profiling of for the compressed model.

Instruction Class	**Ext1D INT**	**Ext1D 16bins INT**	**Ext1D 16bins Compressed**
LOAD	**10.893.314**	**16,270,975**	**11,575,297**
STORE	63.668	63,187	50,637
IMM	46	5,377,697	**32,377,317**
REG	0	0	**5,405,573**
ADDSUB	23.704.159	29,975,949	30,853,867
MULT	5,428,096	5,428,096	5,428,096
DIV	21.637	21,637	9,084
BRANCH	6.423.755	6.423,755	**11.807.503**
JUMP	887.496	887.496	201
Total	**47,422,171**	**64,448.792**	**97.507.575**

Table 6.9 presents the execution time, and estimated energy, including the standard floating point model. Note the compressed model, besides the larger number of instructions, is still faster than the float model. On the other hand, the estimated energy presents the higher values among the evaluated models.

Table 6.9 – Execution time to execute the models (Apple M2 processor) and estimated energy.

	Ext1D Float	**Ext1D INT**	**Ext1D 16bins INT**	**Ext1D 16bins Compressed**
Total time (sec)	28.568	4.175	11.860	18.734
Time per inference (ms)	7.282	1.064	3.023	4.765
Energy (pJ)	1,449	948	1,289	1,950

Remember that the accuracy and profiling results reflect the CNN model without considering estimates. However, the inference time and energy results are estimates. The inference time is obtained with the Apple M2 processor and the energy from data collected from the literature. An important future work involves evaluating these metrics on a processor described at the RTL level, allowing accurate performance (number of clock cycles), power, and energy estimates. Another relevant future work is to assess hardware acceleration, such as the extensions available in the RISC-V ISA.

Table 6.10 summarizes the results and presents the most significant result of this chapter in its 2nd row, the memory requirements to store the model (number of parameters). Since the index vectors are reduced by a factor of 8, a reduction in memory usage of this proportion is expected. According to the table, the reduction is 7.99, given the need to store the LUTs. Note that the compressed model requires less memory than the reference model, which has an accuracy of 74.9%, compared to 87.1% for the proposed model. Thus, we have achieved the second goal for resource-constrained devices: reduce the memory footprint for storing the CNN model.

Table 6.10 – Summary of the results. Emphasis on the model size. The presented values in the table correspond to the number of parameters required by the CNN model.

	Reference 1D	Ext1D Float	Ext1D Int	Ext1D Float LUT 16	Ext1D Int LUT 16	**Compressed**
Model size (#param)	271,621	926,981	926, 981	927,061	927,061	**115,953**
Accuracy %	74.9	88.5	87.2	87.2	87.1	87.1
LOAD instructions (10^6)	2.95	10.87	10.89	16.25	16.27	11.58
Total inst. (10^6)	11.38	41.91	47.42	58.94	64.45	97. 51
Energy (pJ)	393	1,449	948	1,790	1,289	1,950
Time per inference (ms)	2.01	7.28	1.06	8.73	3.02	4.77

6.5 Final Remarks

We may summarize the comparison of the **Compressed** with the **Ext1D Int LUT 16** implementation as follows:

- Model Size: **7.99 times smaller**
- Accuracy: **87.1%** for both models
- `LOAD` instructions **29% reduction**
- Number of executed instructions (from the instruction profiling): 1.5 times higher
- Energy: 1.5 times higher (estimated from the instruction profiling and energy values from the literature)
- Execution time: 58% slower (Apple M2 processor)
- Instruction set: **integer**

Although the number of instructions increased 50%, the main goal of the **Compressed** model was achieved: reduce memory accesses with a smaller model size. Note

that the compression method has only 6% more `LOAD` instructions than the simpler, non-quantized Ext1D INT. Conversely, the reduction in memory footprint comes with the cost of increased total instructions, thus increasing execution time and estimated energy consumption.

7. CONCLUSION

The standard 1D CNN model used as reference in this research lacks temporal awareness in its implementation, a crucial aspect for accurately detecting patterns in sensor data, especially for HAR. Despite its simplicity and modest hardware requirements in terms of computation, the model has trade-offs. It only achieves a 74% accuracy level in the testing dataset, making it an inefficient model for practical applications.

The MSc's *first contribution* was optimizing the reference CNN model. This research improved the reference model, naming it "Extended 1D Model" – Ext1D. This model was designed to incorporate temporal awareness while preserving the inherent simplicity of the original model (three convolutional layers and two fully connected layers). This inclusion of temporal awareness resulted in a significantly improved accuracy, with the new model reaching an 89% accuracy level in the same dataset. Furthermore, we evaluated a variant of the Ext1D model, referred to as the "Reduced Model". This model was designed to decrease the number of parameters while maintaining accuracy. It reduced the model size by half, incurring only a modest 4% decrease in performance. We selected the original Ext1D, named "Accurate Model" for its accuracy, to evaluate the deployment of CNNs in resource-constrained devices.

This contribution was published in:

Assessment and Optimization of 1D CNN Model for Human Activity Recognition.
REUSCH, Rafael; JURACY, Leonardo R.; MORAES, Fernando Gehm.
In: SBESC, 2022. – https://ieeexplore.ieee.org/document/9964520

The MSc's *second contribution* was deploying a C model equivalent to the Pytorch model, using integer and floating point representations. The goal of the C model is to avoid FPUs and reduce the memory footprint. To enable the development of the C model, we introduced a novel approach for exporting Pytorch's parameters using parameterized scripts. This integer-only model incurred only a 1.3% accuracy loss compared to its floating-point counterpart. Although the number of instructions increased compared to the floating-point model, the execution time was more than eight times faster. The integer model also outperformed the reference model, which has a smaller input size and was 13.2% slower.

The *third contribution* of this MSc is the reduction of the CNN model to use it on devices with scarce resources. After validating the integer-only model, we reduced the model size by using quantization and LUTs. We observed that the CNN model could use 16-bin LUTs, causing only a 1.6% decrease in accuracy. We proposed the "Compressed Model" to effectively reduce the model size by packing the indexes of 8 parameters (weight or bias) into a single integer word. This model reduced the number of LOAD instructions by 29% and minimized the model size by almost eight times, keeping the accuracy of the integer model. However, this model increased the number of executed instructions, increasing execution time and energy consumption.

7.1 Future Work

This research paves the way for several future work. The following points highlight the key areas of interest for future research:

1. Pruning techniques. Pruning was initially considered during this research but had to be deferred due to time constraints. Future studies could delve into these techniques to enhance the efficiency of the proposed models.

2. Accuracy improvement using a 2D model with temporal awareness. Building upon the Extended 1D Model developed in this research, future studies could explore a 2D model with temporal awareness. This may provide further improvements to the accuracy of the model.

3. Public dataset. Future research could employ a publicly available dataset for model training and validation. This could improve the applicability and comparability of the results.

4. Impact of the compiler in instruction profiling. This work used O2 optimization to obtain results. More aggressive optimization techniques, such as O3, could be investigated in future research. Preliminary results suggest that such optimization can reduce the number of executed instructions by 30%, reducing execution time and energy consumption.

5. Representation of values within each bin. This research introduced a quantization method that uses a minimal LUT. Future work could focus on developing better ways to represent the values within each bin, potentially allowing for more aggressive quantization. We only evaluated central, mean, and average positions, keeping the central position. The preliminary results showed that selecting the correct values inside each box could increase accuracy.

6. Bin optimization per layer. Future work could explore the possibility of using a different number of bins per layer to maximize accuracy and reduce the model size.

7. Evaluate 8-bin LUTs. We observed that variable quantization maintained an accuracy of 85.5% using 8-bin LUTs. This aggressive quantization can potentially reduce the model size further (10x), but it is necessary to evaluate the impact in terms of executed instructions and execution time.

8. Evaluation at the Register-Transfer Level (RTL). Future research could involve evaluating the inference time and energy metrics on a processor described in RTL. This would allow for accurate performance estimates (number of clock cycles), power, and energy.

9. Assessment of Hardware Acceleration: Considering the extensions available in the RISC-V Instruction Set Architecture (ISA), future research could evaluate the impact of hardware acceleration on the model's performance.

10. Hardware Accelerator. Apply the proposed software implementation methods to a dedicated HAR IP, having energy consumption reduction as its main design goal.

REFERENCES

Celio, C., Patterson, D. A., and Asanović, K. (2015). The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. Technical report, EECS Department, University of California. Source: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html, May 2023.

Chiang, J. (2022). Simple 1D CNN approach to human-activity-recognition (HAR) in PyTorch. Source: https://github.com/jchiang2/Human-Activity-Recognition, May 2023.

Du, H., He, Y., and Jin, T. (2018). Transfer Learning for Human Activities Classification Using Micro-Doppler Spectrograms. In *IEEE International Conference on Computational Electromagnetics (ICCEM)*, pages 1–3. Source: https://doi.org/10.1109/COMPEM.2018.8496654.

Dua, N., Singh, S. N., Semwal, V. B., and Challa, S. K. (2023). Inception inspired CNN-GRU hybrid network for human activity recognition. *Multimedia Tools and Applications*, 82(4):5369–5403. Source: https://doi.org/10.1007/s11042-021-11885-x.

Figo, D., Diniz, P., Ferreira, D., and Cardoso, J. (2010). Preprocessing techniques for context recognition from accelerometer data. *Personal and Ubiquitous Computing*, 14:645–662. Source: https://doi.org/10.1007/s00779-010-0293-9.

Frank, J., Mannor, S., and Precup, D. (2010). Activity and Gait Recognition with Time-Delay Embeddings. In *Conference on Artificial Intelligence (AAAI)*, pages 1581–1586. Source: https://doi.org/10.1609/aaai.v24i1.7724.

García, E., Villar, M., Fáñez, M., Villar, J. R., de la Cal, E., and Cho, S.-B. (2022). Towards effective detection of elderly falls with CNN-LSTM neural networks. *Neurocomputing*, 500:231–240. Source: https://doi.org/10.1016/j.neucom.2021.06.102.

Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. (2021). A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR*, abs/2103.13630:1–33. Source: https://doi.org/10.48550/arXiv.2103.13630.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. 781p.

Hayat, A., Dias, M., Bhuyan, B. P., and Tomar, R. (2022). Human Activity Recognition for Elderly People Using Machine and Deep Learning Approaches. *Information*, 13(6):1–13. Source: https://doi.org/10.3390/info13060275.

Imperas, Software Ltd (2023). Open Virtual Platforms the source of Fast Processor Models & Platforms. Source: http://www.ovpworld.org/, May 2023.

Intelbras (2020). Conheca as Vantagens de utilizar a Inteligencia Artificial em Sistemas de CFTV IP. Source: https://backend.intelbras.com/sites/default/files/2020-08/e-book-linha-future.pdf, May 2023.

Inturi, A. R., Manikandan, V., and Garrapally, V. (2023). A novel vision-based fall detection scheme using keypoints of human skeleton with long short-term memory network. *Arabian Journal for Science and Engineering*, 48(2):1143–1155. Source: https://doi.org/10.1007/s13369-022-06684-x.

Janak, J., Tseng, T., Isaacs, A., and Schulzrinne, H. (2021). An Analysis of Amazon Echo's Network Behavior. In *IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. Source: https://doi.org/10.1109/GLOBECOM46510.2021.9685138.

Jordan, T. S. (2016). Using convolutional neural networks for human activity classification on micro-Doppler radar spectrograms. In *Sensors, and Command, Control, Communications, and Intelligence (C3I)*, pages 1–9. Source: https://doi.org/10.1117/12.2227947.

Jung, S., Son, C., Lee, S., Son, J., Han, J.-J., Kwak, Y., Hwang, S. J., and Choi, C. (2019). Learning to quantize deep networks by optimizing quantization intervals with task loss. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4350–4359. Source: https://doi.org/10.1109/CVPR.2019.00448.

Juracy, L. R., Moreira, M. T., de Morais Amory, A., Hampel, A. F., and Moraes, F. G. (2021). A High-Level Modeling Framework for Estimating Hardware Metrics of CNN Accelerators. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(11):4783–4795. Source: https://doi.org/10.1109/TCSI.2021.3104644.

Katz, S., Downs, T. D., Cash, H. R., and Grotz, R. C. (1970). Progress in Development of the Index of ADL1. *The Gerontologist*, 10(1_Part_1):20–30. Source: https://doi.org/10.1093/geront/10.1_part_1.20.

Keras (2023). Layer activation functions. Source: https://keras.io/api/layers/activations/, June 2023.

LeCun, Y., Denker, J. S., and Solla, S. A. (1989). Optimal Brain Damage. In *Advances in Neural Information Processing Systems (NIPS)*, pages 598–605. Source: https://proceedings.neurips.cc/paper_files/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf.

Mathworks (2021). DeepLearning using MATLAB. Source: https://www.mathworks.com/campaigns/offers/next/deep-learning-ebook.html, May 2023.

Mekruksavanich, S. and Jitpattanakul, A. (2021). LSTM Networks Using Smartphone Data for Sensor-Based Human Activity Recognition in Smart Homes. *Sensors*, 21(5):1–25. Source: https://doi.org/10.3390/s21051636.

Monteiro, Y. C., Vieira, M. A., Vitorino, P. V., Queiroz, S. J., Policena, G. M., and Souza, A. C. (2021). Trend of fall-related mortality among the elderly. *Revista da Escola de Enfermagem da USP*, 55:1–8. Source: https://doi.org/10.1590/1980-220x-reeusp-2020-0069.

Ness, T. V. (2018). Low Power Floating-Point Unit for RISC-V. Master's thesis, Norwegian University of Science and Technology. Source: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2564801/19493_FULLTEXT.pdf?sequence=1.

Ng, J., Hausknecht, M., Vijayanarasimhan, S., Vinyals, O., Monga, R., and Toderici, G. (2015). Beyond short snippets: Deep networks for video classification. In *Computer Vision and Pattern Recognition (CVPR)*, pages 4694–4702. Source: https://doi.org/10.1109/CVPR.2015.7299101.

Ordóñez, F. J. and Roggen, D. (2016). Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition. *Sensors*, 16(1):1–25. Source: https://doi.org/10.3390/s16010115.

O'Shea, K. and Nash, R. (2015). An Introduction to Convolutional Neural Networks. *CoRR*, abs/1511.08458:1–11. Source: http://arxiv.org/abs/1511.08458.

Park, H., Kim, N., Lee, G. H., and Choi, J. K. (2023). MultiCNN-FilterLSTM: Resource-efficient sensor-based human activity recognition in IoT applications. *Future Generation Computer Systems*, 139:196–209. Source: https://doi.org/10.1016/j.future.2022.09.024.

Paszke, A. et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR*, abs/1912.01703:1–12. Source: https://doi.org/10.48550/arXiv.1912.01703.

Podareanu, D., Codreanu, V., Aigner, S., Leeuwen, C., and Weinberg, V. (2019). Best Practice Guide - Deep Learning. Source: https://prace-ri.eu/wp-content/uploads/Best-Practice-Guide_Deep-Learning.pdf, May 2023.

RISC-V Foundation (2019). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213. Source: https://riscv.org/technical/specifications, May 2023.

Sarvamangala, D. R. and Kulkarni, R. V. (2021). Convolutional neural networks in medical image understanding: A survey. *Evolutionary Intelligence*, 15(1):1–22. Source: https://doi.org/10.1007/s12065-020-00540-3.

Shakti (2023). Open Source Processor Development Ecosystem. Source: https://shakti.org.in, May 2023.

Shi, J., Chen, D., and Wang, M. (2020). Pre-Impact Fall Detection with CNN-Based Class Activation Mapping Method. *Sensors*, 20(17):1–12. Source: https://doi.org/10.3390/s20174750.

Shimoda, M., Sato, S., and Nakahara, H. (2017). All binarized convolutional neural network and its implementation on an FPGA. In *International Conference on Field Programmable Technology (FPT)*, pages 291–294. Source: https://doi.org/10.1109/FPT.2017.8280163.

Ullah, H. and Munir, A. (2023). Human Activity Recognition Using Cascaded Dual Attention CNN and Bi-Directional GRU Framework. *Journal of Imaging*, 9(7):1–30. Source: https://doi.org/10.3390/jimaging9070130.

Vavoulas, G., Chatzaki, C., Malliotakis, T., Pediaditis, M., and Tsiknakis, M. (2016). The MobiAct Dataset: Recognition of Activities of Daily Living using Smartphones. In *International Conference on Information and Communication Technologies for Ageing Well and e-Health (ICT4AWE 2016)*, pages 143–151. Source: https://www.scitepress.org/papers/2016/57924/57924.pdf.

Wang, L., Dong, X., Wang, Y., Liu, L., An, W., and Guo, Y. (2022). Learnable Lookup Table for Neural Network Quantization. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12423–12433. Source: https://doi.org/10.1109/CVPR52688.2022.01210.

Xu, J., He, Z., and Zhang, Y. (2019). CNN-LSTM Combined Network for IoT Enabled Fall Detection Applications. *Journal of Physics: Conference Series*, 1267:1–6. Source: https://doi.org/10.1088/1742-6596/1267/1/012044.

Xu, S., Wang, Q., Wang, X., Wang, S., and Ye, T. T. (2021). Multiplication Through a Single Look-Up-Table (LUT) in CNN Inference Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1916–1928. Source: https://doi.org/10.1109/TCAD.2021.3095825.

Yamamoto, K. (2021). Learnable companding quantization for accurate low-bit neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5029–5038. Source: https://doi.org/10.1109/CVPR46437.2021.00499.

Yang, Z., Wang, Y., Han, K., Xu, C., Xu, C., Tao, D., and Xu, C. (2020). Searching for low-bit weights in quantized neural networks. In *Neural Information Processing Systems (NeurIPS)*, pages 4091–4102. Source: https://dl.acm.org/doi/abs/10.5555/3495724.3496068.

Yu, Y., Zhi, T., Zhou, X., Liu, S., Chen, Y., and Cheng, S. (2019). BSHIFT: A Low Cost Deep Neural Networks Accelerat. *International Journal of Parallel Programming*, 47:360–372. Source: https://doi.org/10.1007/s10766-018-00624-9.

Zaruba, F. and Benini, L. (2019). The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI

Technology. *IEEE Transactions Very Large Scale Integration Systems*, 27(11):2629–2640. Source: https://doi.org/10.1109/TVLSI.2019.2926114.

Zebin, T., Sperrin, M., Peek, N., and Casson, A. J. (2018). Human activity recognition from inertial sensor time-series using batch normalized deep LSTM recurrent networks. In *IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 1–4. Source: https://doi.org/10.1109/EMBC.2018.8513115.

Zeng, M., Nguyen, L. T., Yu, B., Mengshoel, O. J., Zhu, J., Wu, P., and Zhang, J. (2014). Convolutional Neural Networks for human activity recognition using mobile sensors. In *International Conference on Mobile Computing, Applications and Services (MobiCASE)*, pages 197–205. Source: https://doi.org/10.4108/icst.mobicase.2014.257786.

APPENDIX A – HAR DATASET

The following table consists of randomly selected data extracted from the training dataset used in the reference model. Each line represents a single timestep and columns represent features extracted from the sensors. The original training file is presented in CSV format. The reference model only uses 40 features of various sensors, and many more features, such as magnetometer sensors, are also available in the dataset.

id	user_name	raw_timestamp_part_1	cvtd_timestamp	roll_belt	pitch_belt	yaw_belt	total_accel_belt	gyros_belt_x	gyros_belt_y	gyros_belt_z	accel_belt_x	accel_belt_y	accel_belt_z	roll_arm	pitch_arm	yaw_arm	total_accel_arm	gyros_arm_x	gyros_arm_y	gyros_arm_z	accel_arm_x	accel_arm_y	accel_arm_z
1	pedro	1323095002	05/12/2011 14:23	123	27	-475	20	-5	-2	-46	-38	69	-179	407	-278	178	10	-165	48	-18	16	38	93
2	jeremy	1322673067	30/11/2011 17:11	102	487	-889	4	-6	-2	-7	-13	11	39	0	0	0	38	-117	85	-43	-290	215	-90
3	jeremy	1322673075	30/11/2011 17:11	87	182	-885	5	5	2	3	1	-1	49	0	0	0	44	21	-136	113	-341	245	-87
4	adelmo	1322832789	02/12/2011 13:33	125	-416	162	17	11	11	-16	46	45	-156	-109	55	-142	25	22	-51	92	-238	-57	6
5	eurico	1322489635	28/11/2011 14:13	135	333	-886	3	3	2	0	-8	4	27	761	276	102	29	-196	79	-54	-197	200	-30
6	jeremy	1322673149	30/11/2011 17:12	-592	159	-877	4	1	5	-13	-11	-16	38	0	0	0	14	2	5	-7	-26	130	-19
7	jeremy	1322673128	30/11/2011 17:12	12	444	-873	4	-6	0	0	-14	2	35	0	0	0	15	236	-101	89	99	79	-67
8	jeremy	1322673076	30/11/2011 17:11	43	415	-885	4	-18	-2	-3	-10	-2	42	0	0	0	22	-371	185	-69	-98	175	-78
9	carlitos	1323084240	05/12/2011 11:24	93	672	-937	4	1	0	-2	-15	1	32	-137	112	-167	34	3	-2	-2	-287	111	-122
10	charles	1322837822	02/12/2011 14:57	114	224	-131	18	14	11	-16	-25	63	-158	-824	-638	-753	32	26	-5	79	-301	-42	-80
11	carlitos	1323084277	05/12/2011 11:24	92	594	-927	3	5	0	0	-18	4	27	70	151	113	33	-8	2	0	-277	113	-112
12	jeremy	1322673101	30/11/2011 17:11	101	496	-878	5	-1	0	2	-22	8	40	0	0	0	30	257	-106	79	-192	204	-75
13	eurico	1322489661	28/11/2011 14:14	54	245	-886	3	-6	0	-13	-8	5	24	110	-317	-136	23	-344	108	-15	106	97	-168
14	jeremy	1322673043	30/11/2011 17:10	45	502	-879	5	-5	0	0	-14	2	49	0	0	0	37	-2	-2	-1	-277	157	-183
15	jeremy	1322673156	30/11/2011 17:12	534	-309	-803	4	24	0	5	8	19	28	0	0	0	23	366	-209	49	41	85	-204
16	eurico	1322489713	28/11/2011 14:15	165	347	-87	2	2	2	0	-12	5	20	881	558	123	33	2	-6	1	-236	163	-148
17	pedro	1323094971	05/12/2011 14:22	129	278	184	21	-5	0	-48	-47	69	-187	-147	-345	-551	3	-19	-24	13	2	-9	27
18	carlitos	1323084285	05/12/2011 11:24	92	531	-931	3	2	0	-5	-13	3	24	152	-264	-85	42	1	-37	-3	35	-65	-404
19	pedro	1323094999	05/12/2011 14:23	123	267	-268	19	-31	-5	-44	-48	72	-169	307	-27	-151	11	199	-12	-44	-7	57	89
20	eurico	1322489658	28/11/2011 14:14	14	32	-887	3	6	-2	2	-9	4	23	906	115	117	30	-47	13	-18	-223	166	-105

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br