TINYML-BASED CNNS INFERENCE AND ACCELERATION

TINYML INFERENCE ENABLEMENT AND ACCELERATION ON MICROCONTROLLERS: THE CASE OF HEALTHCARE

By BAILIAN SUN, BS

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University

MASTER OF APPLIED SCIENCE (2024)

Hamilton, Ontario, Canada (Electrical and Computer Engineering)

TITLE:	TinyML Inference Enablement and Acceleration on Microcontrollers: The Case of Healthcare
AUTHOR:	Bailian Sun BS (Electrical and Computer Engineering), Beijing University of Posts and Telecommunications, Beijing, China
SUPERVISOR:	Mohamed Hassan
NUMBER OF PAGES:	xv, 75

Lay Abstract

Having continuous Blood Pressure (BP) monitoring is a must to prevent cardiovascular diseases (CVDs). This thesis presents a new solution using small, efficient devices and advanced machine learning algorithms to realize real-time BP estimation. Similar to how traditional BP measurements are taken by the pulse rate, the small devices use the changes in blood volume as input, instantly inferring the BP. The thesis aims at addressing the challenges when incorporating large network capacities into tiny devices.

The contributions are as follows: First, this thesis explores a variety of optimization strategies to shrink the machine learning networks while achieving comparable accuracy. Those techniques are not tied to any specific framework, making them flexible and portable. Second, this thesis investigates several acceleration techniques from both software and hardware perspective. With the novel optimization strategies, the work demonstrates accurate and efficient BP monitoring.

Abstract

Controlling high blood pressure can eliminate more than half of the deaths caused by cardiovascular diseases (CVDs). Towards this target, continuous BP monitoring is a must. The existing Convolutional Neural Network (CNN) -based solutions rely on server-like infrastructure with huge computation and memory capabilities. This entails these solutions impractical with several security, privacy, reliability, and latency concerns. To address the challenges, an alternative solution has merged to conduct the machine learning algorithms into tiny devices. The unprecedented boom in tinyML development also drives the high relevance of optimizing network inference strategies on resource-constrained microcontrollers (MCUs)

The contributions of the thesis are: First, the thesis contributes to the general field of tinyML by proposing novel techniques that enable the fitting of five popular CNNs - AlexNet, LeNet, SqueezeNet, ResNet, and MobileNet - into extremely-constrained edge devices with limited computation, memory, and power budget. The proposed techniques use a combination of novel architecture modifications, pruning, and quantization methods. Second, utilizing this stepping stone, the thesis proposes a tinyML-based solution to enable accurate and continuous BP estimation using only photoplethysmogram (PPG) signals. Third, the thesis proposes several techniques to accelerate the CNNs inference process. From a hardware perspective, we discuss

architecture-aware accelerations with cache and multi-core specifications; from the software perspective, we develop application-aware optimizations with an existing real-time compatible C library to maximize the computation and intermediate buffer reuse. Those solutions only require the general MCU features thus demonstrating board generalization across various networks and devices.

We conduct an extensive evaluation using thousands of real Intensive Care Unit (ICU) patient data and several tiny edge devices and all the five aforementioned CNNs. Results show comparable accuracy to server-based solutions. The proposed acceleration strategies achieve up to 71% reduction in inference latency.

To my mother and father

Acknowledgements

I would like to sincerely thank all those who have contributed to the successful completion of this thesis.

First, I wish to thank my supervisor, Dr. Hassan, for the valuable guidance, continuous support, and insightful suggestions throughout my research, as well as for giving me the opportunity to pursue graduate studies. It has been an honour to learn from you.

I am deeply thankful to my parents for providing me with both financial and emotional support throughout my academic journey. I couldn't have completed this thesis without you.

Lastly, I extend my appreciation to my friends and Sisi for their care and companionship. Thanks so much for always being around.

Table of Contents

List of Figures

List of Tables

Notation and Abbreviations

BP	Blood Pressure
CVD	cardiovascular disease
CNN	Convolutional Neural Network
MCU	Microcontroller
PPG	Plethysmogram
ICU	Intensive Care Unit
IoT	Internet of Things
SVM	Support Vector Machines
RF	Random Forest
ML	Machine Learning
AAL	Ambient Assisted Living
SoC	Systems-on-Chip
AAMI	Association for the Advancement of Medical Instrumentation

BHS British Hypertension Society

ABP Arterial Blood pressure

SBP Systolic BP

DBP Diastolic BP

ECG Electrocardiography

PTT Pulse Transit Time

HR Heart Rate

RelU Linear Rectified Linear Unit

MAE Mean Absolute Error

Batch Normalization
BN

Conv Convolutional

FC Fully Connected

ResNet Residual Neural Network

FLOPs Floating Point Operations

ME Mean Error

SD Standard Deviation

CDF Cumulative Distribution Function

NN	Neural Network
TPU	Tensor Processing Unit
MMU	Matrix Multiply Unit
AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
LR	Linear Regression
PR	Polynomial Regression
DT	Decision Tree
AB	Ada Boost
DNN	Deep Neural Network
FDT	Fused Depthwise Tiling
K2C	Keras2C
SIMD	Single Instruction Multiple Data

Declaration of Academic Achievement

The following is a declaration that the research described in this thesis was completed by Bailian Sun under the supervision of Dr. Mohamed Hassan, ECE Department, McMaster University.

The work "The Case for tinyML in Healthcare: CNNs for Real-Time On-Edge Blood Pressure Estimation" is accepted at SAC 2023. Bailian Sun contributed to the design and development of the research.

The work "HW&SW Co-Optimizations For TinyML Inference Time Acceleration" is submitted to DSD 2024. Bailian Sun contributed to the study's design, evaluation and writing.

Chapter 1

Introduction

Over the past decade, ML has revolutionized a variety of industries and is evolving rapidly. To accommodate the increasing demand for data and computation, the training and execution of networks rely on powerful cloud computing and data centers. Motivated by the need to reduce energy and computational consumption and to prevent data privacy leakage issues [75], the miniaturization of tiny edge devices and the optimization of ML algorithms is developed as a new prospect of the Internet of Things (IoT) [19]. Healthcare is one of the fields that can benefit most from the tinyML field as it protects patients' sensitive information and reduces communication latency by processing data locally [79]. Previous works in healthcare monitoring predominantly utilize two main approaches. The first approach utilizes classic ML algorithms like Support Vector Machines (SVM) and Random Forest (RF), which fall short during complex tasks [36, 58, 3]. The second approach relies on a specific network and hardware device, which may not generalize well to other systems [24, 87, 42].

This thesis addresses two main challenges in the tinyML field. First, the CNN

model size far outpaces the capacity of the edge device. The thesis investigates CNNs model compression solutions including the novel architecture modifications and a combination of model pruning, and quantization techniques. Multiple CNNs are evaluated including LeNet, AlexNet, SqueezeNet, ResNet, and MobileNet are evaluated for BP estimation and achieve impressive performance using three edge devices. This topic is elaborated in the Chapter 2. The second challenge is the CNN inference latency reduction, focusing on methods that are compatible to the vast majority of the MCUs, including both architecture- and application-aware optimizations. They efficiently speed up the inference of LeNet, AlexNet, ResNet, and SqueezeNet on ESP32. This topic is described in the Chapter 3.

Chapter 2

The Case for tinyML in Healthcare: CNNs for Real-Time On-Edge Blood Pressure Estimation

2.1 Introduction

The rise of big data has propelled researchers in all domains to realize the tremendous potential of Machine Learning (ML). The healthcare sector is no exception. Researchers have proposed several ML-based solutions to improve different aspects of the sector including diagnosis, treatment recommendations, medical imaging, Ambient Assisted Living (AAL), administrative tasks, and personalized health monitoring

[68]. However, these solutions are based on conventional ML infrastructures that are resource-intensive, require powerful processors and large memory units with theoretically unlimited power budgets. This gave rise to cloud-based solutions which made great advancements in several applications, such as medical imaging [5] and drug discovery [27]. Unfortunately, cloud-based methods fall short when it comes to healthcare applications with safety critical and real-time requirements, such as health monitoring and AAL. The reliance on network connectivity raises latency, security and privacy issues. These concerns deem cloud-based solutions unreliable for such applications. An alternative to cloud computing is edge computing, where processing is carried out at close proximity to the sensors that collect data [25]. This mitigates the need for continuous availability of network connectivity. The computing elements in this architecture are called edge devices. However, the considered edge devices were mostly relatively powerful computing systems with high-performance multi-core CPUs alongside GPUs such as in Raspberry PI, Nvidia Jetson Nano, and Qualcomm Snapdragon Systems-on-Chip (SoCs). Recently, the field of tinyML [44] gained tremendous interest both from academia and industry. Unlike general edge computing, tinyML explores the deployment of ML models onto extremely-constrained low-power devices. Healthcare is identified as one of the highest growing industries with the advancement of tinyML [66].

This thesis focuses on investigating techniques that enable the deployment of tinyML-based solutions on personalized healthcare monitoring. In particular, We identify BP monitoring as one usecase which we believe will immensely benefit from

tinyML. BP disorders are one of the leading causes of deaths [29]. Therefore, continuous monitoring of BP is essential for early detection of such abnormalities. Unfortunately, traditional cuff-based devices prohibit continuous monitoring. To address this problem, several ML solutions have been recently proposed for BP estimation [88, 74, 70, 7, 47]. All these solutions depend on the availability of resource-intensive machines with powerful computation capabilities and large memory capacities and bandwidth. This makes them targeting server- or cloud-based infrastructure, while clearly being ill-suited for tinyML usecases. With requiring continuous, reliable operation, while mandating privacy, we argue that personalized healthcare monitoring (including BP estimation) cannot practically depend on these solutions for mass adoption. A tinyML-based solution avoids the inconvenience of cuff-based methods, while also avoiding the drawbacks of the server-based solutions.

This thesis contributes to both the healthcare as well as tinyML domains as follows. 1) The thesis contributes to the field of tinyML by proposing novel techniques that enable the fitting of popular CNNs into extremely-constrained edge devices with limited computation, memory, and power budget. Namely, the thesis successfully manages to fit the following five popular CNNs into tiny edge devices: LeNet, AlexNet, SqueezeNet, ResNet and MobileNet. This enables us to run inference completely on the edge without dependency on connectivity or cloud infrastructure. The proposed techniques use a combination of novel architecture modifications, pruning and quantization methods.

2) Moreover, the thesis presents a detailed comparative study of the five networks from the perspective of tinyML. This study covers several aspects of the edge devices, such as inference time, model size, operational memory size, and model accuracy. In

doing so, we explore the deployment of these CNNs into different tinyML devices including Raspberry Pi Pico, ESP32, and Arduino Nano BLE in addition to comparing against more powerful systems (Raspberry Pi and an Intel i7 multi-core machine) as a baseline. 3) Utilizing this stepping stone, the thesis proposes a tinyML-based solution to enable accurate and continuous BP estimation using only photoplethysmogram (PPG) signals. We conduct extensive evaluation using thousands of real ICU patient data and several tiny edge devices and all the five aforementioned CNNs. Results show that the proposed solutions offer comparable accuracy to server-based solutions and also meet the Association for the Advancement of Medical Instrumentation (AAMI) British Hypertension Society (BHS) standards.

2.2 Motivation

As aforementioned, one of the key goals of this work is to bring the powerful CNN-based inference to the extreme edge devices without compromising model accuracy. In this section, we discuss why this is a challenging problem that even using existing tinyML frameworks does not completely solve. In doing so, we run an extensive set of experiments over the five popular CNNs we focus on and we tabulate our findings in Table 2.1.

To investigate the potential of source-constrained devices, we run the five original CNNs with $96 \times 96 \times 1$ input size as a baseline on the ESP32. As a result, none of the networks can fit it. This is indicated by the all $\times$ in the Baseline column. With the optimization techniques provided by TFLite, we managed to fit three of the five networks: SqueezNet, 20-layer ResNet (ResNet20), and MobileNet with hyperparameter of 0.55 (MobileNet-0.55). However, only ResNet20 and SqueezeNet present

reasonable accuracy. This is the reason for having ✓✓ for them, while MobileNet gets only one ✓ for the TFLite column in Table2.1. After adopting the architecture adjustments we propose in this work, the five compressed models achieve results comparable to powerful edge devices.

In the next section, we discuss in detail how we managed to fit these networks in the tinyML devices while keeping comparable accuracy.

Table 2.1: Deployment results with different solutions

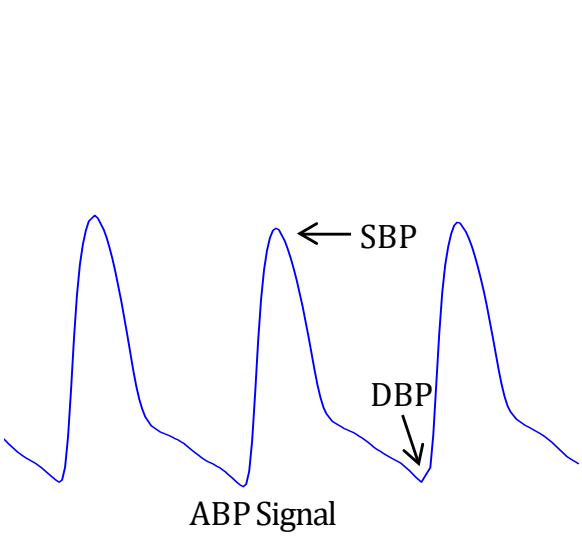
	Baseline	TFLite compression	Proposed
LeNet	×	××	✓✓
AlexNet	×	××	✓✓
SqueezeNet	×	✓✓	✓✓
ResNet20	×	✓✓	✓✓
MobileNet-0.55	×	✓×	✓✓

2.3 Background and State-of-the-Art

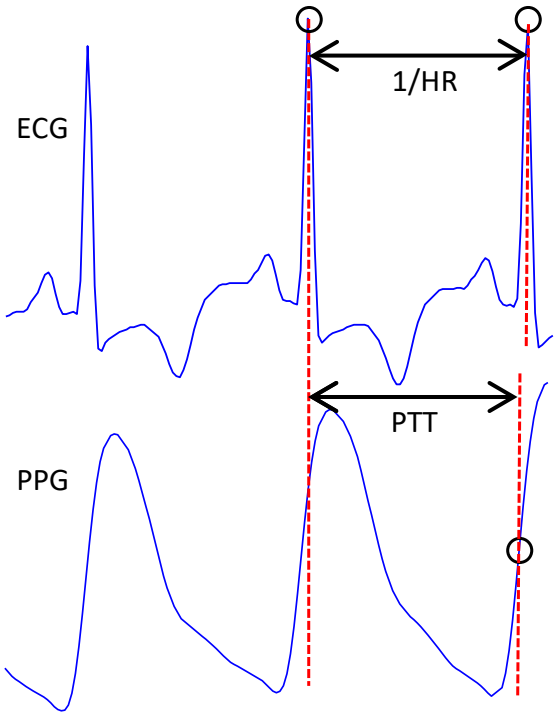
We cover the necessary background as well as state-of-the-art from both angles of the thesis: tinyML paradigm, and the BP estimation use-case.

2.3.1 Blood Pressure Basics

BP disorders cause the largest number of disabilities and deaths around the world [29]. Arterial Blood pressure (ABP) disorders can happen in either directions: hypertension, or hypotension. The two commonly reported ABP are the Systolic BP (SBP), and the Diastolic BP (DBP). SBP is the pressure in the artery when heart's ventricle contracts, while DBP represents the pressure in the artery when ventricle relaxes. This is illustrated in Figure 2.1a. Unfortunately, the traditional cuff-based



(a) SBP and DBP



(b) PTT, HR relation to ECG and PPG signals

Figure 2.1: Blood pressure basics figures

mercury sphygmomanometer is not handy and requires a specific setup, which entails it to be difficult for continuous monitoring and for regular patient use. On the other hand, there are several non-invasive cuffless sensors that measure other cardiovascular signals such as Electrocardiography (ECG) and photoplethysmography (PPG). In this work, we use PPG signals only to estimate the patient's SBP and DBP using CNNs. ECG is only used in data preprocessing along side PPG to determine Pulse Transit Time (PTT) and Heart Rate (HR). The relationship between PTT, ECG, and PPG is illustrated in Figure 2.1b.

Several classical ML-based [40, 53] as well as neural-network-based [18, 74, 70, 7] solutions are proposed to estimate BP; nonetheless, all these works conduct training

and inference in powerful devices that are not deployable in the tinyML paradigm. To our knowledge, tinyCare [3] is the only existing solution addressing the BP estimation problem with the tinyML constraints. Two main differences are worth highlighting between our proposal and tinyCare. 1) The solution in tinyCare was limited to classical machine learning techniques, while we investigate the deployment of CNNs. Our results show that using CNNs achieves better accuracy for estimating BP as we detail in Section 2.5. 2) tinyCare requires synchronously capturing both ECG and PPG since both signals are used in the inference process. This synchronization adds additional circuitry constraints on the used sensors and is a source of measurement errors. In contrast, our proposal relies solely on PPG signal to estimate BP eliminating the need not only for synchronization but also for real-time capturing of ECG.

2.3.2 tinyML paradigm

*tiny*ML is one of the fastest growing subfields of machine learning technologies [78, 84] with the goal of enabling ML techniques to run on extremely resource-limited MCUs at the network edge [64]. Several key industry players introduced frameworks to enable tinyML such as the TensorFlow Lite from Google [77], ELL from Microsoft [20], STM 32Cube.AI from STMicroelectronics [1], and CMSIS-NN from Arm [12]. These frameworks and potential applications of tinyML is covered by the survey in [66]. An example of academic frameworks is CMixNN [8], which offers an open-source library to enable low-precision neural networks facilitating the deployment on tinyML devices. These frameworks are orthogonal to the solution this thesis proposes since most of them focus mainly on quantization techniques to reduce model sizes. This thesis instead introduce a variety of tools to enable CNNs in tinyML constrained

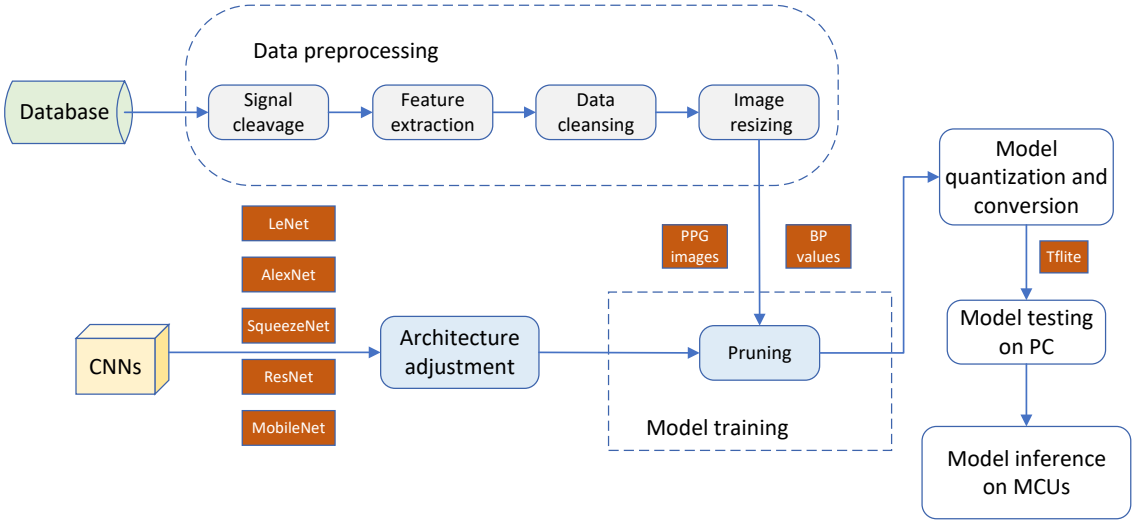


Figure 2.2: Block diagram of proposed methodology.

systems including network architecture modifications, tuning input sizes, and intermediate parameter reduction among others as we detail in Section 2.4. Indeed, we deploy the proposed usecase of BP estimation on top of one of these frameworks: TensorFlow Lite as an example, while the proposal itself is not restricted to a certain framework. For the application usecases, tinyML has been mainly explored for audio and vision domains [22, 81, 83, 10, 4], with some recent efforts exploring tinyML for other domains including autonomous vehicles [17], gaming [51], and smart agriculture [82]. Yet, healthcare domain seems to be lacking behind in terms of studies investigating the applicability of tinyML. An example of such study is tinyCare [3] that we have already discussed earlier.

2.4 Proposed Methodology

In this section, we discuss several novel techniques that enable us to deploy five commonly used CNNs on extreme-edge devices using real-time blood pressure prediction

as a usecase. Fig 2.2 illustrates the entire pipeline of training and inference on the tinyML devices. The approach can be summarized into the following main steps.

1) We use the PPG and BP data from the MIMIC database for training purpose. 2) We perform data preprocessing, including data cleansing and input resizing. 3) We deploy several techniques to enable the fitting of the CNNs on the resource-limited edge devices. This includes architecture modification, pruning, and quantization, as we will detail later in this section. 4) We train and test the models on the PC for exploration purposes. 5) Afterwards, we use the TFlite flow to convert the model and further optimize it to deploy on edge devices for inference.

2.4.1 Data Preprocessing

We use the MIMIC-IV database. MIMIC is a freely available database with various data sets for clinical trials [39]. It contains multiple physical indicators such as patient population distribution, clinical trial data, and medical records. Those advantages make it widely used in academia. To estimating BP, we only use the PPG signals as our training and testing dataset. The data is collected from 12,000 patients and recorded as 125 Hz. To increase the reliability and stability of the data, we preprocess the data as follows.

1. Considering that each signal in MIMIC covers a different length of time, we divide all the signals into 4-second windows, each segment consisting of 500 sampling points.

2. In addition to PPG, ECG and ABP signals are also used in the data processing stage. We use ECG alongside PPG to calculate the PTT, and the HR. PTT, HR, SBP, and DBP from the training dataset are utilized in outliers removal.

In particular, we consider anything outside the range of $80mmHg \leq SBP \leq 180mmHg$ and $60mmHg \leq DBP \leq 130mmHg$ to be an outlier. Then, HR computed from both ECG, and PPG is averaged and any data outside the range $54.4 \leq HR \leq 155.8$ is considered an outlier. Furthermore, we consider only data of PTT less than $400msec$ from the MIMIC dataset for training purposes. Readings beyond this limit can be caused by abnormal conditions, like moving the sensor abruptly, or after abnormally strenuous physical activity, or due to rare medical conditions. So we discard them. Abnormal values may happen due to measures. After removing those abnormal windows, 101404 fragments are preserved.

3. Considering the limited resources of edge devices, we scale the PPG signal image to the $96 \times 96 \times 1$ format instead of the $224 \times 224 \times 3$ format to limit the number of input channels in the CNNs. After the shuffle, we split them into 80%:10%:10% segments for training, validation, and testing, respectively. The final experimental results prove that this size can not only meet the needs of predicting blood pressure but also reduce the memory cost of the model. Finally, the distributions of SBP and DBP after cleansing are illustrated in Figure 2.3.

2.4.2 BP Estimation With CNNs

CNN is an excellent choice for regression tasks with enormous amounts of images. However, most CNNs exhibit high computational complexity, while having millions of parameters mandating a large memory footprint. This poses a significant challenge

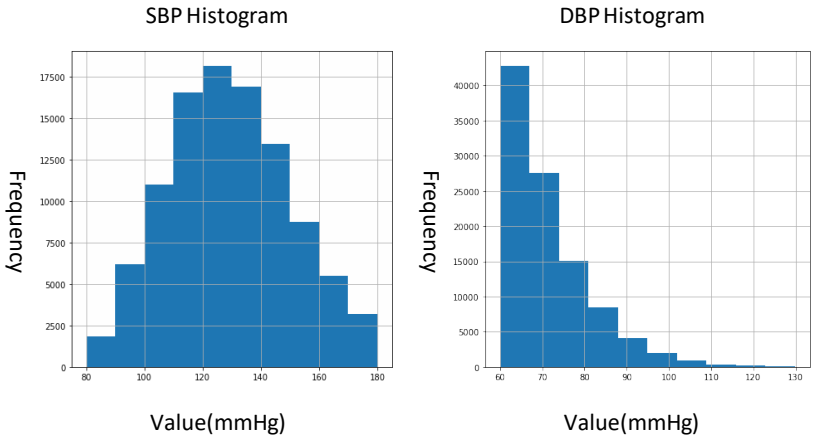


Figure 2.3: Distribution histograms after data cleansing.

(a) AlexNet (b) LeNet (c) SqueezeNet (d) ResNet (e) MobileNet

Figure 2.4: CNNs before and after architectural optimizations for BP measuring task.

on tinyML enablement, where the inference needs to be done exclusively on resource-limited edge devices. We target in this thesis five popular CNN architectures: LeNet, AlexNet, SuqeezeNet, ResNet and MobileNet. Figure 2.4 shows the CNN architectures before and after the proposed architecture modifications. The origin features and modifications are marked in blue and red, respectively. This section presents the architectural modifications conducted to the network structures to shrink down the model size. In the next section, we further reduce the models with compression techniques so that they eventually fit on MCUs.

The general model tuning involves four parts:

1. We scale down the input shape dimension to $96 \times 96 \times 1$ so that the width of all channels could be reduced while preserving the salient features of input images.

2. We replace any non-linear *tanh* activation function with the linear Rectified Linear Unit ($ReLU$) activation. This is because it is easier to implement in tiny devices and still provides a good accuracy with fast computation time.

3. The last fully connected layer has no activation equation and a single neuron to provide a one-dimensional output without restrictions on the range.

4. In the training phase, the networks are trained with the Adam optimizer. Initially, we set the learning rate to 0.005 for warming up and switch it to 0.0001 after the Mean Absolute Error (MAE) drops below 8 and 16 for DBP or SBP. A batch size of 32 is adapted.

LeNet

LeNet is a simple architecture improved by Yann LeCun in 1998 [45] (Figure 2.4b). It has been widely used in detecting handwritten and is regarded as a base for other CNNs. LeNet comprises seven layers, among which are three convolutional (Conv) layers, the first two Conv are followed by a pooling layer. The output flows into a series of two connected layers.

We propose several modifications to the original LeNet-5 model towards making it lightweight, while providing suitable accuracy for a critical task such as the BP estimation. Compared with handwritten characters, there are more features to be extracted from PPG signal images. 1) The first Conv layer is performed using 3×3 filters instead of the original 5×5 filters. 2) The third Conv layer amounts to a full connection because the original feature map size is 1×1. As we have a bigger dimension input, we replace it with a flatten layer. Moreover, theoretically, two fully connected (FC) layers are sufficient to solve most problems, given that there are

enough neurons [50][26]. 3) The filter count in the first FC is decreased to 32 as a compromise between model size, computation complexity and accuracy.

AlexNet

Another network we employ is modified from AlexNet architecture primarily designed by Alex Krizhevsky in 2012 [43] (Figure 2.4a). Except for the Pooling layers, there are eight layers in the architecture, including five Conv layers and three FC layers. The outputs of the Conv layers are connected with the FC layers through a flattening operation.

AlexNet has been widely used in the vision field, but its 240MB model size makes it ill-suited for tinyML targeting extremely resource-limited edge devices. Besides, too many neurons in the hidden layer can lead to overfitting, especially for noisy setups (such as our considered healthcare domain). For a Conv layer cl with N_{cl}^{ch} input channels, a filter size of $n_{cl} \times n_{cl}$ and m_{cl} filters, the number of parameters in such Conv layer will be $(N_{cl}^{ch} \times n_{cl} \times n_{cl} + 1) \times m_{cl}$. Similarly, for the FC layer fl, the number of parameters will be $(N_{fl}^{ch} + 1) \times m_{fl}$. Please note that while the expression for the FC layer seems smaller than that of the Conv layer, in fact, the number of the parameters of the FC dominates the AlexNet's model size. For example, in our case, the max $N_{cl}^{ch} = 384$ and $m_{cl} = 384$, while $N_{fl}^{ch} = 6400$ and $m_{fl} = 4096$. Besides, the first Conv layer in the network is sensitive to pruning and adjusting because it directly connects with the image [28][80]. If the first layer fails to capture high level features, the lost information cannot be covered no matter how strong the rest of the network is [23].

As a solution, we aim to reduce the number of nodes in the other layers for model

reduction. The number of neurons in the second, third, and fourth Conv layers has been reduced from 256, 384, and 384 to 100, 128, and 128, respectively. Additionally, we add a Batch Normalization (BN) layer after each of the first four Conv layers for regularization [69]. Furthermore, AlexNet requires more data due to the large amount of weights in the FC layers. To address this limitation, we change the number of neurons in the first two FC layers to 256 and the number of neurons in the last layer to 1. **All these architectural modifications enables us to achieve a 95× smaller model size compared to the baseline AlexNet baseline model.**

SqueezeNet

SqueezeNet is a smaller network released in 2016, which achieves AlexNet-level accuracy on ImageNet with 50× fewer parameters [35] (Figure 2.4c). The architecture employs two main strategies to achieve it: 1) removing the dense layers, and 2) introducing the *fire module* consisting of a 1×1 Conv layer followed by a mix of 1×1 and 3×3 Conv layers. This fire module limits the number of 3×3 filters and the input channels to a 3×3 layer so that the Conv parameters are reduced.

We implement SqueezeNet v1.1 for BP estimation. The model starts with a standalone Conv layer, feeding into eight Fire modules and ends with a FC layer. As aforementioned, SqueezeNet was successfully deployed using only available compression techniques and without requiring architectural modifications (except reducing input size as highlighted in Figure 2.4c).

ResNet

A residual neural network (ResNet) was introduced in 2015 by Kaiming He [30] (Figure 2.4d). Over the years, researchers tend to add more layers to the neural network to tackle complex tasks. As a result, the networks become hard to train, and performance gets saturated. ResNet solves the problem by introducing residual blocks and adding the direct connections to skip some extra connections during the training. ResNets have variable sizes with a certain number of neural network layers. We deploy the ResNet20 on CIFAR10, which has twenty stacked weight layers and nine shortcuts. It starts with a Conv layer followed by a BN layer and then feeds into a series of three residual blocks, ending up in a FC layer. Although ResNet20 is a deeper architecture with only 0.27 million parameters, it can only be executed on tinyML resource-constrained devices after deploying compression techniques as highlighted in Table 2.1.

MobileNet

MobileNet is a small model introduced by Google in 2017, aiming to fit on mobile and embedded vision applications [32]. We use the MobileNetv2 [67], whose core idea is the bottleneck residual block. There are three Conv layers in the block. The data start with a 1×1 expansion layer to expand the input channels and then go through a 3×3 depthwise Conv layer, finally flowing into a 1×1 projection layer for shrinking down the output channels. Similar to Resnet, MobileNetv2 adds residual blocks, using a direct connection to connect the expansion and projection layers.

MobileNet v2 offers a width multiplier as tunable hyperparameter to dynamically

adjust the size of a network. While some tradeoffs exist between accuracy and computational cost, small model performs better with fewer parameters. After empirical experiments, we set the parameter to 0.55 to reduce the number of model parameters while maintaining the model's performance. Another architecture modification we apply is that we add a dropout layer with a ratio of 50% before the FC layer to combat overfitting[76]. Along with the above changes in the network structure, we apply the MAE as a loss function since the regression focuses on the difference between predictions and ground truths.

As a result of all the proposed architectural modifications, we reach the number of model parameters for each CNN as depicted in Table 2.2.

2.4.3 Compression Techniques

By adjusting the structure of the model, we shrink down the number of parameters while maintaining its accuracy. Moreover, some compression techniques need to be applied to the model to integrate the ML algorithms in the MCUs. In this section, we carry out pruning, and quantization using the TFLite model optimization APIs.

Pruning. Model pruning is proposed to eliminate the less important connections. We implement the weight-pruning on the entire model. After quantization, the pruned models can be compressed within a small size using file compression algorithms (zip compression. e.g.). Zipped files are not supported by devices, but pruning yields

Table 2.2: Number of Parameters in CNNs.

	LeNet	AlexNet	SqueezeNet	ResNet20	0.55 MobileNet
Million Parameters	0.30	0.94	0.80	0.27	0.82

power-saving and speed up in inference phase.

Quantization. Model quantization can be divided roughly into two forms from high level: post-training quantization and quantization-aware training. In this study, we implement the post-training quantization strategy because it requires less training time and computational power with little degradation in model performance [37]. While some studies have proposed the methodology for 4-bit or lower quantization[9][55], they usually result in significant accuracy degradation and are not supported by most edge devices. We take the full integer quantization for all ops except the input and output tensor and compress the model size by approximately $4x$. Table 2.3 presents the gzipped model size compassion between the base model and the pruned and quantized models.

Table 2.3: Gzipped Model Size (bytes).

	LeNet	AlexNet	SqueezeNet	ResNet 20	0.55 MobileNet
Quantized and pruned	293054	952158	834514	281221	902296
Baseline	1904923	10230582	7394506	3066412	8958208

2.4.4 Model Conversion And Assessment

Model conversion is needed for inference and assessment on edge devices. Tflite provides a converter tool to transfer the saved file to a serialized Tflite model in FlatBuffer format for implementing machine learning Ops at the edge scenario like the mobile and IoT devices [37]. In addition, we use the interpreter to run the Tflite simultaneously in Python on the PC side. The generated Tflite needs to be transferred to a C array because ESP32, Arduino Nano BLE and Raspberry Pi Pico are programmed in C language. Finally, we compare the results collected from the

Table 2.4: Technical Specifications of Edge Devices.

Device	Processor	Cores	Width	Flash Memory	RAM
Raspberry Pi 3 Model B	ARM Cortex A-53	4	64 bit	16 GB	1 GB
ESP32 Wrover IE	Xtensa LX6	2	32 bit	4MB	4 MB
Raspberry Pi Pico	ARM Cortex M0+	2	32-bit	2 MB	264 KB
Arduino Nano 33 BLE	ARM Cortex M4	1	32 bit	1 MB	256 KB

edge devices with the ground truths to get the MAEs for model assessment. We discuss the evaluation results in the following section.

2.5 Evaluation

We deployed the full tinyML pipeline described in Section 2.4 to evaluate the effectiveness of the proposed techniques from different edge-based perspectives.

2.5.1 Experimental Setup

We implement the five CNNs we consider in this thesis including both the vanilla baseline version, as well as the version adopting our proposed techniques to suit tinyML requirements. This is prototyped in python using the Keras library, which runs on top of TensorFlow.

As described in Section 2.4, we use the MIMIC-IV dataset [39] for the training and testing phases. After preprocessing, our proposed BP estimation model only requires the PPG signals as the input. We apply an 80%:10%:10% split of the datatset for training, cross-validation, and testing, respectively. The training phase is conducted on a PC machine running an octa-core Intel i7 processors with 32 GB of RAM.

After the training phase, we convert the models to the TFLite format which are

Table 2.5: Results Summary. 'X' indicates that the architecture did not fit in the device

				Inference Time			
Architecture	**Model Size Reduction**	**Loss in Performance**	**Number of FLOPs** (Millions)	**Raspberry Pi**	**ESP32**	**Pico**	**Nano**
AlexNet	99%	8%	78	27 ms	7.67 s	5.02 s	X
MobileNet	92%	30%	38	12 ms	4.75 s	X	X
SqueezeNet	83%	15%	85	25 ms	7.30 s	X	X
ResNet20	72%	3%	732	186 ms	60.82 s	X	X
LeNet	-21%	-22%	13	8 ms	1.53 s	0.18 s	0.16 s

compatible with MCUs. We deploy these models on four edge devices with a wide range of processing and memory capabilities. The devices are Raspberry Pi, ESP32, Raspberry Pi Pico and Arduino Nano, and their technical specifications are described in Table 2.4. Each of these devices introduce a different set of constraints that affect the performance and inference times of the models. Raspberry Pi is not considered a tinyML device since compared to the other used extremely-constrained MCUs, it entertains a much more capable processing unit and memory capacity. That said, we include it in our evaluation for comparative purposes as a baseline of a higher-end edge device. As we will show later in the section, using our techniques, we manage to achieve comparable results to those of server-based solutions and to the solution running on Raspberry Pi using only the extremely-constrained lower-end MCUs. Since the ESP32, Pico, and Nano are programmed in C, the Tflite models must be further converted to a C array before deployment. This is done using the unix command, *xxd*[85].

After running inference on all the edge-devices, we compare the performance of the CNN architectures in terms of memory requirements, inference times, and model accuracy.

2.5.2 Tuning Model Size

Our proposed methodology to reduce model size includes reduction of input dimension, modifying the network architecture, and applying Tflite compression techniques. For all the networks, Table 2.5 shows accuracy suffered for the percentage of model size reduction. By just applying the compression techniques, we reduce the model sizes of SqueezeNet and ResNet by 83% and 72% respectively. Table 2.5 also shows these models fit in Raspberry Pi and ESP32.

MobileNet can be deployed by solely applying the compression techniques and the hyperParameter tuning at the expense of unreasonable loss of accuracy. We manage to decrease the loss and achieve a 92% reduction in model size by applying the changes discussed in section 2.4.2. As a result of our modifications to LeNet, it retains almost the same size as the original LeNet, and it can be deployed in all the devices in our study.

Due to the large size of the vanilla AlexNet, it cannot be deployed in any of the devices, including the powerful Raspberry Pi. However, by selectively removing neurons from the intermediate convolution layers and the fully connected layers, as explained in Section 2.4.2, we reduce the size of model by 99% for only 8% loss in accuracy. After combining our modifications with the existing compression techniques, this model can be deployed in Raspberry Pi, ESP32, and Raspberry Pi Pico.

2.5.3 Memory Requirements

The memory requirement of a CNN comprises of the model size and the tensor arena size. The model size depends on the number of parameters in the network. It determines if the model can fit in the flash storage or not. On the other hand, the

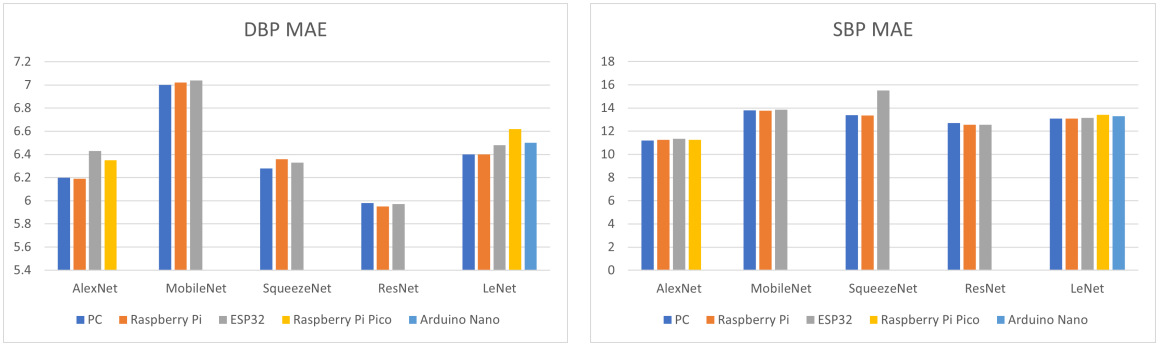
Table 2.6: Memory Requirements of the Architectures

Architecture	**Model Size**	**Tensor Arena Size**
Alexnet	0.95 MB	105 KB
MobileNet	1.04 MB	315 KB
SqueezeNet	0.86 MB	215 KB
ResNet20	0.30 MB	600 KB
LeNet	0.30 MB	70 KB

tensor arena is the memory required by the TensorFlow interpreter to store the input tensors, output tensors, and the intermediate tensors during inference. This represents the RAM required to conduct inference on the device. Table 2.6 describes the memory requirements for the five CNNs.

The tensor arena is the maximum storage needed during any instant at the runtime to store the all the tensors for either input/output feature maps, which depends on the operation types, amount of activation data and the network topology. ResNet20, despite being only 0.3 MB, has the highest tensor arena requirement of 600 KB. This is attributed to the increased simultaneous activation data due to multiple feed-forward connections from a single layer in the ResNet architecture. In contrast, LeNet is a simpler network with regular feed-forward networks that only stores the input and the output data of the current layer. As such, LeNet requires only 70 KB of tensor arena despite having the same model size as ResNet20.

Thus, the device must meet both the memory requirements to fit the model. Based on the requirements in Table 2.6, only the Raspberry Pi and ESP32 could run an inference with all the networks. The other devices fail to fit a model due to shortage in flash storage, RAM, or both. For example, the Pico has 2MB of flash storage to store a MobileNet model of 1.04MB, but its 264 KB RAM is not sufficient



(a) DBP (b) SBP

Figure 2.5: MAE Comparison of Different Networks in Different Devices.

to cover MobileNet's tensor arena requirement of 315 KB. The Nano neither has sufficient flash storage nor RAM to fit MobileNet.

2.5.4 Inference Times

In this section, we investigate the timing inference of the CNNs in different devices recorded in Table 2.5. The inference time depends on the complexity of the network architecture and the device specifications. The complexity of the architecture is manifested by the number of Floating Point Operations (FLOPs) the model will have to perform. We compute the number of FLOPs in each of the network and record them in Table 2.5. We can observe a direct co-relation between the inference time and the number of FLOPs.

With its powerful quad-core processor and large memory capacity, the Raspberry Pi runs at least 1000 times faster inference than the other devices. Among ESP32, Pico, and Nano, the processor is the differentiating factor for the inference time. As the Pico runs on a processor from the lower end of the Cortex M-series, it is slightly slower than the Nano, but it still runs faster than ESP32 which runs an Xtensa processor. For devices running multi-core processors, such as ESP32 and Pico, the

inference time can be improved by multi-core programming techniques.

2.5.5 Model Accuracy

We compare the predictions from the inference with the true values and compute the Mean Absolute Error (MAE), Mean Error (ME), and Standard Deviation (SD). These metrics are determined to evaluate our results with the two most common standards for blood pressure monitoring devices: The BHS [57] and the AAMI[21] standards.

Meeting Medical Grades for BHS Standard. Table 2.7 describes the grading criteria used by BHS. The grades show the cumulative percentage of measurements that are within 5 mm Hg, 10 mm Hg, and 15 mmHg. To achieve a certain grade, the three percentages must be higher than or equal to the numbers in the table. We compute the Cumulative Distribution Function (CDF) of the absolute DBP prediction errors from Raspberry Pi and record the percentages under the standard's thresholds in Table 2.8. All the architectures are well within the limits of grade B, with AlexNet and ResNet20 closely approaching grade A. Since the results from the other edge devices are close to those of Raspberry Pi, as observed in Figure 2.5, we can say that these grades apply to the other devices as well.

Meeting Requirements for AAMI Standard. Furthermore, we compare our results with the AAMI Standards, which states that the ME must be less than or

Table 2.7: BHS Grades and Associated Error Percentages.

	Cumulative Error[mmHg]		
	≤ 5	≤ 10	≤ 15
Grade A	60%	85%	95%
Grade B	50%	75%	90%
Grade C	40%	65%	85%
Grade D	worse than C		

equal to five and its SD must be less than or equal to 8. The comparison is described in Table 2.9. In terms of DBP, AlexNet satisfies both the conditions. As for the other networks, the MEs are well under five, but the SDs are slightly higher than eight, with the worse one being higher by only 1.74 mmHg. In terms of SBP, all the MEs are below five, but the SDs are much greater than the standard limit. It is also worth noting that we have far more subjects than the required number, thus making our results statistically more reliable.

Comparison With Other Solutions. We compare our best SBP and DBP results with server-based solutions (Table 2.10) and tinyML solutions (Table 2.11). For the most apt comparison with server-based works, we pick Neural Network (NN) based solutions that only use PPG signals as input without any calibration methods. Although our models do not perform better than these works, the results are comparable. Due to tinyML considerations, we made trade-offs at every step of our methodology affecting our accuracy. These trade-offs are non-existent in the server-based solutions leading to better performance.

In Table 2.11, we compare our results with the only reported tinyML-based solution. Our CNN models significantly outperform the classical ML algorithms in [3] with 13% and 20% improvement in the MAEs of DBP and SBP respectively.

Table 2.8: Comparison of DBP Results With BHS Standards

	Cumulative Error			
Architecture	$\leq 5mmHg$	$\leq 10mmHg$	$\leq 15mmHg$	**Grade**
AlexNet	51%	82%	94%	B
MobileNet	50%	81%	90%	B
SqueezeNet	58%	82%	91%	B
ResNet20	61%	84%	93%	B
LeNet96*96	50%	81%	91%	B

Table 2.9: Comparison of Raspberry Pi Results with AAMI Standards

	DBP		SBP		
Architecture	**ME**	**SD**	**ME**	**SD**	**# of subjects**
AlexNet	-2.70	7.78	-2.75	14.45	10410
MobileNet	2.24	9.74	-2.24	17.62	
SqueezeNet	2.73	9.04	2.48	17.22	
ResNet20	2.27	8.69	0.64	16.47	
LeNet 96*96	1.91	9.61	2.50	16.89	
AAMI Standard	≤ 5	≤ 8	≤ 5	≤ 8	≥ 85

Table 2.10: Comparison with server-based NN solutions

	DBP		SBP		
Work	**MAE**	**SD**	**MAE**	**SD**	**# of subjects**
[7]	5.95	5.6	10.86	9.54	379
[69]	3.91	4.48	7.34	8.65	21215
Our Work	5.95	6.71	11.27	9.45	10410

2.5.6 Discussion: A tinyML Perspective

Results show that our models satisfy the medical standards. Fig. 5 shows a more fine-grain comparison of accuracy among the different CNN architectures. At a first glance, ResNet20 and AlexNet are the top two accurate models. In a regular machine learning context, we may conclude our discussion by recommending these two networks for our application. However, in a tinyML context, we are severely restricted by memory, power, and processing constraints. As such, we extend our discussion to cover these aspects of tinyML.

The first thing is to determine the if the model can fit in the tinyML device. After applying the architecture modifications, the 0.3 MB ResNet20 model can easily be stored in any of the devices. The AlexNet model could not be stored only in Arduino Nano, even though it has 1 MB of flash storage while AlexNet model is 0.95 MB. This

Table 2.11: Comparison with tinyML-based solutions

	DBP		SBP		
Work	**MAE**	**SD**	**MAE**	**SD**	**# of subjects**
[3]	6.85	9.16	14.08	17.82	52714
Our Work	5.95	6.71	11.27	9.45	10410

is because the device must store the application code alongside the model. Therefore, we recommend prospective tinyML designers to adopt efficient coding practice to minimize the size of the application code and allow more space for the model.

The next step is to ensure that the model can run an inference with the tinyML device. If the device has enough RAM to provide the required tensor arena, the model can successfully run an inference. Here, we highlight the limitation of ResNet20. Despite being the smallest model, it requires the highest tensor arena, which can only be provided by the Raspberry Pi and ESP32. In contrast, AlexNet can be deployed and run on Raspberry Pi, ESP32, and Raspberry Pico. Although ResNet20 model is smaller and provides higher accuracy than AlexNet, we would recommend AlexNet over ResNet20, because from a tinyML perspective, ResNet20's tensor arena of 600 KB maybe a very high requirement for most tinyML devices. Moreover, AlexNet's accuracy is better than that of ResNet for SBP estimations, and slightly worse for DBP estimation.

We also extend our discussion to inference time because most IoT based tinyML applications require fast response rate. The inference time is governed by the complexity of the algorithm and the processing capability of the device. In this thesis, our architecture modifications were mainly targeted to reduce the model size. For applications with hard timing constraints, the tinyML engineer may apply further

architecture modifications targeted to reduce the number of FLOPs, without compromising for accuracy.

For our application of blood pressure estimation, if there is some leeway in the accuracy requirement, we would recommend the LeNet architecture. Apart from meeting the medical standards, its model size is as small as ResNet20, while also requiring the smallest tensor arena, so it was able to fit in all of our devices. It is also a relatively simple model, so it shows the fastest inference time, making it suitable for real-time applications.

2.6 Conclusion

In this study, we introduce a tinyML-based solution methodology to deploy the most common CNNs - ResNet, SqueezeNet, and the models inspired by AlexNet, MobileNet, and LeNet - in a wide range of edge devices with different constraints for real-time estimation of BP with PPG signals. By adjusting the architectures of the networks and adopting compression techniques, we are able to reduce the model size effectively and satisfy the real-time inference requirements on the edge. Finally, we analyze the throughput experimental results from the perspectives of memory cost, inference time and model accuracy on ESP32, Raspberry Pi Pico, and Arduino Nano BLE. While running in an extreme resource-constrained environment, all networks overcome the limitations of computation, memory and power budget, and meet medical standards.

Chapter 3

HW&SW Co-Optimizations For TinyML Inference Time Acceleration

3.1 Introduction

TinyML is an emerging field at the intersection of machine learning (ML) and embedded systems [62]. It makes use of extremely resource-constrained devices with limited power, frequency, and memory capacity to execute complex ML algorithms. TinyML presents a reduced energy consumption and increases the system's reliability as well as data security and privacy by performing local processing [66]. Additionally, it achieves real-time execution by on-device computation [66]. However, the challenges arise from the hardware constraints that impede the tinyML implementation in terms

of the dropping accuracy, increasing inference latency, and the model size limitation. Although developing more powerful hardware devices could solve this problem, the power and form factor constraints of tinyML applications make them impractical to use. Several works have proposed optimizations for the peak memory and computation reduction by simplifying the network inference process [49, 38, 86, 72]. However, improving total inference time achieved less attention.

Since many of the tinyML applications are real-time (eg, healthcare, smart farm, smart city), improving inference time is crucial for enabling tinyML adoption. This thesis addresses this challenge by proposing collaborative HW&SW optimizations towards accelerating tinyML inference on commercial off-the-shelf (COTS) H/Ws. These optimizations are both architecture- and application-aware. Moreover, they do not trade accuracy for inference time; therefore, they do not lead to any accuracy loss. Similar to inference time, accuracy is paramount for many tinyML applications with several of them must adhere to certain standards/policies with regard to accuracy (eg, healthcare domain).

3.2 Related Work

3.2.1 TinyML

Many recent surveys have been focusing on tinyML applications, technologies, and pipelines in [2, 19, 52].

Frameworks

Frameworks like TensorFlow Lite facilitate tinyML development on embedded platforms [15]. It is made up of two primary parts: the interpreter-based approach across hardware architectures for efficient on-device inference and a set of optimization APIs to compress the models. CMSIS-NN is another framework designed specifically for Arm Cortex-M processor cores that provides a collection of highly optimized neural network kernels [45]. [34] provides the tunning solution for the models based on specific hardware characteristics. Works like [63][48][65] introduce the design to automate the searching process for the model architecture and hyperparameters.

Optimized Hardware

The hardware architectures are developed as a necessary foundation for tinyML. To optimize the large machine learning execution and training, Google designed the Tensor Processing Units (TPUs). These units feature a matrix multiply unit (MXU) and a unique interconnect topology, accelerating artificial intelligence (AI) tasks. Edge TPU is an ASIC (Application-Specific Integrated Circuit) form of TPU developed specifically for tinyML that can be integrated into a variety of existing systems [11]. NVIDIA also introduces the custom hardware Jetson [56] to enhance AI applications on the edge. Those works are covered in the survey of [2, 52]. PULP is another academic accelerator built on a parallel ultra-low-power cluster of RISC-V processors, designed for efficient and quantized neural network inference [14].

3.2.2 Applications

TinyML has been widely investigated in environmental problem-solving [6, 60, 59], with some explorations for fields like smart farming [41] and the smart city [54]. Those works are covered in [2]. For the healthcare domain, TinyCare is proposed for continuous BP prediction leveraging classical lightweight machine learning algorithms on the tiny edge devices [3]. In [61], IoT-based hardware equipped with sensors, along with a software deployment flow, is investigated for heart illness classification tasks. One limitation of those research is that they are designed only for lightweight networks, which are inherently small. Deep learning models of a larger scale are not discussed.

[73] investigates the use of PPG signals for BP estimation on various limited-resource devices utilizing five popular CNNs. [42] introduces TinyCES for ECG monitoring, enabling the real-time ECG classification task without relying on cloud-based analysis. The thesis presents a novel lightweight CNN closely modelled on the LeNet architecture, tailored to meet the stringent resource constraints of embedded devices. However, since the system is designed for specific use cases, this approach may face challenges for broader generalization. In this thesis, while using healthcare as a real-world use-case for tinyML, we follow a different approach by focusing on architecture- and application-aware optimizations to best use the MCU resources to address tinyML trade-offs.

3.2.3 CNN Inference Optimization On MCUs

Deploying CNNs on MCUs poses unique challenges due to the hardware's constraints, including the memory and computational capabilities. Currently, most research has

focused on exploring the strategies for inference acceleration or memory footprint reduction.

MCUNetV2 leverages the patch-based approach to have a memory-efficient inference, where the inputs are divided into smaller patches and being processed individually on STM32H7 [49]. The patch-based technique significantly reduces peak memory usage during the inference. It also involves a novel scheduling algorithm that determines the optimal patch size and sequence for each layer. The Demand Layering strategy [38] is proposed to reduce parameter memory usage on GPU-based inference systems when executing Deep Neural Networks (DNNs), specifically on embedded devices. This strategy utilizes a layer-by-layer loading approach to avoid loading the whole model into GPU at once [38]. A pipeline is designed to increase parallelism among the preload, copy, and execution stages during inference. StreamNet is designed to optimize the existing patch-based inference, minimizing the redundant memory and latency overhead [86]. This thesis [72] introduces a new Fused Depthwise Tiling (FDT) strategy and an automated flow for scheduling the tilling operations in DNN. Although these approaches optimize memory usage, they often introduce computational and delay overhead. Our work targets inference latency reduction by maximizing the efficiency of algorithms and device resources, achieving this with no accuracy loss.

3.3 Background

3.3.1 Keras2C Basics

Keras2C (K2C) is an open-source library designed to convert Keras models built on TensorFlow into pure C code [13]. It's developed for real-time applications and supports the majority of Keras model operations. The C code generated by the K2C script relies solely on the standard C library, making it both portable and flexible across various embedded devices [13].

The K2C library comprises three primary components. The first component is a Python script designed to convert the generated .h5 file into C code. In this phase, K2C extracts all relevant parameters, including weights, biases, strides, and other properties of the tensors involved in the inference process. In Keras models, each input, output, and filter in a layer is a tensor. While in K2C, a tensor is an N-dimensional matrix represented as a data type called K2C_tensor, which includes the tensor shape, the elements it contains, and related properties, as depicted in Algorithm 1 [13]. The elements are represented as a one-dimensional array, where all elements are unravelled in a row-major manner. Figure 3.2 provides an example of a [2, 3, 2] tensor, with the numbers on the elements indicating their indices. The second component is the C backend to implement the core functionality for the forward pass through each layer. This part is designed to be clear and easily customizable for optimization during execution. The third component is the generated C code for specific network inference, including several test suites that include sample inputs and outputs for evaluation.

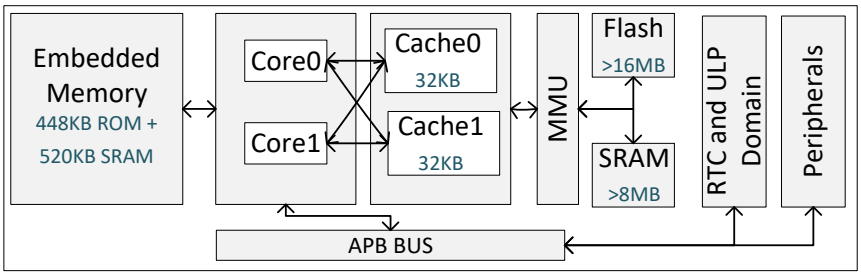


Figure 3.1: An example architecture of an MCU-based edge device

3.3.2 Embedded Device

TinyML targets a portable machine learning model deployment on tiny, low-cost embedded systems. The industry has seen a surge in diverse hardware designs, driven by advances in tinyML. We use ESP32 as an example to show some of the potential general insights for MCUs in Figure 3.1. In general, the characteristics of MCUs can be elaborated on below:

Constrained Resource

Traditional mobile devices, which rely on cloud-based infrastructure, ensure efficient data transfer and are equipped with megabytes of memory and gigahertz frequency processors. In contrast, the MCUs offer limited bandwidth, computation, and memory capacities, all the parameters are stored within the devices. That emphasizes the necessity of exploring appropriate approaches to improve inference efficiency.

Multi-core Architecture

Many MCUs support multi-tasking features by assigning tasks on different cores. Each processor shares the memory resources, allowing two or more cores to run tasks concurrently. For real-time applications, properly scheduling the parallelism of tasks

Device	Processor	Cores	Flash/MRAM	RAM	Cache
ESP32-S3	Xtensa LX7	2	<8MB	<8MB	16KB ICache, 32KB DCache
STM32H7	Cortex-M7 & Cortex-M4	2	<2MB	<1MB	16KB ICache, 16KB DCache
Raspberry Pi Pico	Arm Cortex-M0+	2	<2MB	<256KB	16KB Cache
Alif Ensemble E7	Arm Cortex-A32 & Cortex-M55	4	<5.5MB	<13.5MB	32KB ICache, 32KB DCache

Table 3.1: Technical Specifications of Edge Devices

can significantly speed up the implementation.

Cache

MCUs often include small but fast cache memories to improve efficient data access. These caches store the most recently used memory segments, minimizing the time needed for higher memory hierarchies. The processor's various cache operations facilitate faster data processing and support cache-friendly solutions in real-time applications.

Table 3.1 lists several commonly used embedded devices in academic research, comparing their specifications in terms of processor, core number, cache size, and memory capacity.

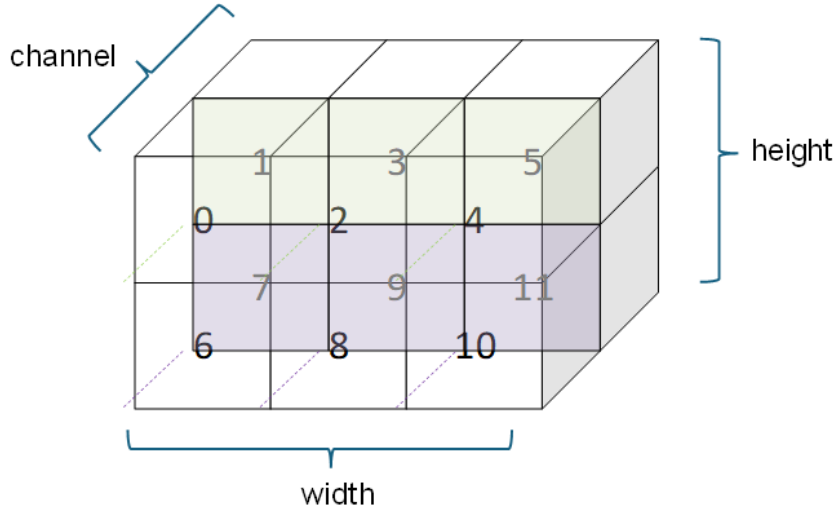


Figure 3.2: [2, 3, 2] Tensor

Algorithm 1 `k2c_tensor` Structure [13]

```
1: struct k2c_tensor {
2:     float* array {Pointer to array of tensor values flattened in row-major order.}
3:     size_t ndim {Rank of the tensor (number of dimensions).}
4:     size_t numel {Number of elements in the tensor.}
5:     size_t shape[K2C_MAX_NDIM] {Size of the tensor in each dimension.}
6: }
```

3.4 Proposed Methodology

In this section, we introduce our architecture- and application-aware optimizations to accelerate tinyML inference time at no accuracy loss. The discussion is following: (1) We present techniques to accelerate inference from both software and hardware perspectives, detailing the applicability for specific layer in CNN. (2) We explain the compatibility of operations across each CNN model.

3.4.1 Overview of Methodologies

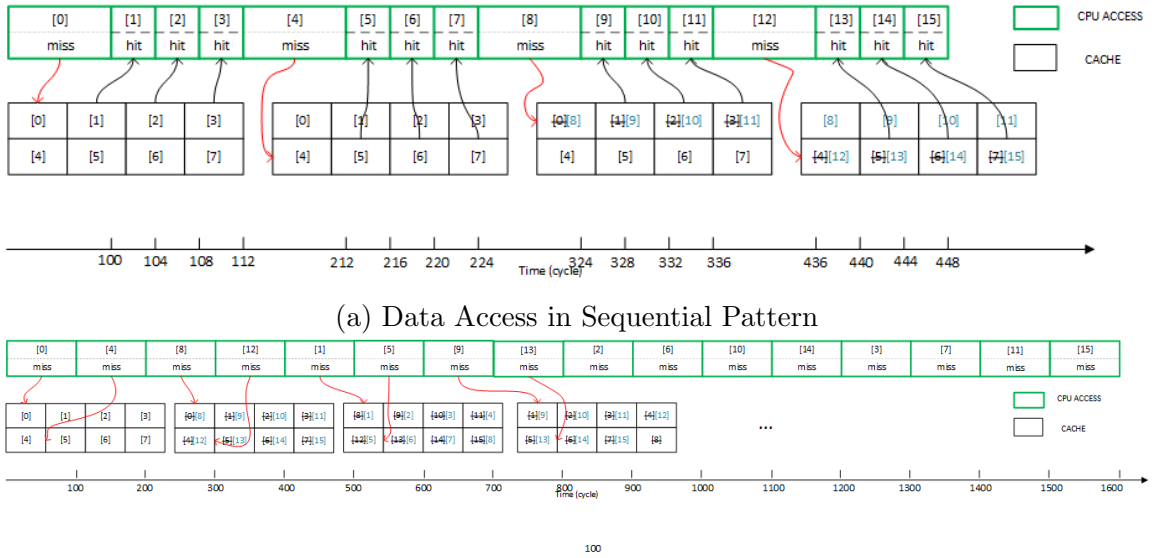
Architecture-aware Optimizations

Cache-friendly Memory Layout. Increasing the cache hit rate is an essential factor for memory hierarchy efficiency. The cache hit rate is defined as the proportion of memory accesses that are directly addressed by the cache. During the CNN inference phase, the CPU conducts numerous read and write operations on input, and output elements, as well as model parameters. The CPU loads data via cache, and it has to find the data from external memory if a miss occurs. MCU processors

are usually in-order [16] and thus, are not able to hide memory latency with computation. Therefore, improving memory access time is crucial for reducing overall inference time, which is clearly more crucial for tinyML than traditional ML algorithms. The latter usually run on powerful machines with massive parallelism and out-of-order processors.

To demonstrate how cache-friendly memory layout impact cache hit rates, we design a simulated scenario without preloading in Figure 3.3. Each green box represents CPU access to a specific data element, while black shows the contents of the cache. The simulation simplifies the scenario with the following constraints: (1) Each element is accessed once. (2) The element size is 4 bytes, and the cache line size is 16 bytes, which allows the CPU to load four elements simultaneously upon a cache miss. Furthermore, the cache is hypothetical to accommodate only two cache lines. (3) The total data size required through the execution is larger than the cache size. (4) Hit is 4 cycles, and miss is 100 cycles. One thing to note is that this simulation closely mirrors the data access process during CNN inference on embedded devices. For instance, the default CNN inference uses the float data type, and the size required for each operation, typically hundreds of kilobytes, far exceeds the cache capacity of MCUs, which is usually within tens of kilobytes as explained in Section 3.3.2.

In Figure 3.3, it's assumed that the initial access is a miss, causing a contiguous data segment to be automatically loaded to the cache by the CPU. With a sequential access pattern as shown in Figure 3.3a, the following requests are hits until element 4. The accesses from element 4 to element 7 would repeat the previous process. Access to element 8 leads to the eviction of an earlier cache line due to cache size limitations, and the following accesses from elements 9 to 11 are hits. In this setup, the cache hit



(a) Data Access in Sequential Pattern

(b) Data Access with Stride

Figure 3.3: Data Access Strategy

rate is 75%. The delay for loading the whole sixteen elements is 448 cycles.

In contrast, Figure 3.3b illustrates a case where elements are fetched with a stride of four. The first and fourth elements are misses. Subsequent requests for elements 8 and 12 lead to the eviction of previously loaded cache lines. This pattern causes repeated cache misses for the next requests, ultimately reducing the cache hit rate to 0%. The delay for loading the whole sixteen elements is 1600 cycles.

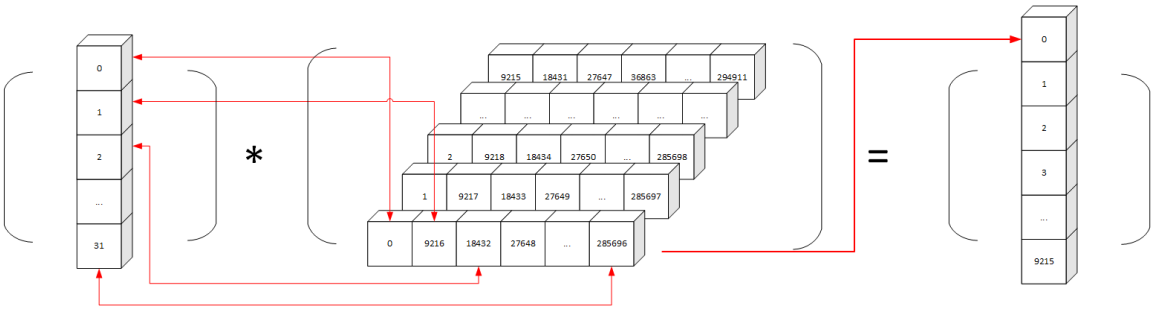
These observations highlight the impact of data access patterns and cache configurations on performance. Specifically, they emphasize the importance of optimizing element placement to maximize cache utilization and efficiency. Strategies that ensure better data access patterns during computation can significantly increase the hit rate, demonstrating a critical area for improvement in cache management.

For the Dense layer, the algorithm performs the procedure of matrix-vector multiplication followed by a bias offset. An example of multiplying an $[1 \times 32]$ input matrix

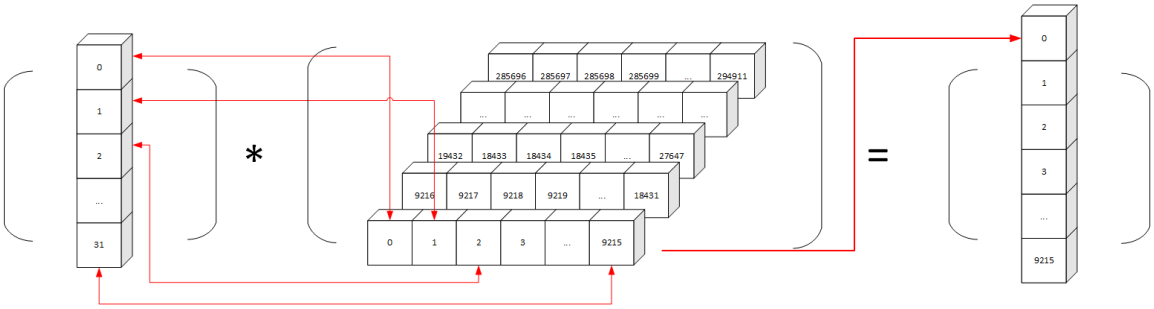
with a $[32 \times 9216]$ kernel matrix is illustrated in Figure 3.4a. The number marked on the elements represents its index within the C array. In the Dense layer of CNNs, inputs and outputs consist of a single channel, thus operating on all elements along this dimension sequentially. However, the kernels are multidimensional, resulting in a non-sequential memory access pattern. To have a cache-friendly access solution, the element locations in the kernel array are rescheduled offline to eliminate the large strides between successively accessed elements as shown in Figure 3.4b. To accommodate the new element arrangement, We also update the algorithm to align with it accordingly, the comparison of Dense inference algorithms is illustrated in Figure 3.5. Furthermore, in CNNs like LeNet, the Dense layer takes up the majority of the network's parameters, making them benefit from such optimizations.

The BN layer and the Add layer are naturally sequential, therefore don't request the cache-friendly access optimization. In the Conv layer, the output is generated in order by iterating the multiplications of neurons and the matched input elements. Thus, both the kernels and outputs are processed sequentially, typically ensuring a high cache hit rate. However, the receptive field is within a small rectangular region, and tensors are represented in the memory of MCUs as a one-dimensional array, mapped in [height, width, channel] order as mentioned in Section 3.3.1, resulting in non-contiguous storage locations. And it would be impractical to reschedule inputs without modifying the previous layer's algorithms, bacause they also represent the output of that layer. Therefore, we employ alternative strategies for the Conv layer, as explained in the following Sections.

Multi-core Parallelism-aware Partitioning. Fused layer partitioning was initially proposed in [71] as a solution utilized in DNN layers for segmenting the static



(a) Dense Inference Baseline



(b) Dense Inference with Cache-friendly Memory Layout

Figure 3.4: Dense Inference

parameters. This technique enables parallel processing of independent computations across different partitions. In the CNN layers, the outputs are computed independently, allowing the division of the computational tasks. Besides, as discussed in Section 3.3.1, most of the MCUs are featured with the multi-core. In our application, we use the ESP32-S3 MCU, which features dual-core processors to support inference with double efficiency. This solution only separates the computation process and the corresponding elements, without impacting the accuracy or the integrity of the model's outputs. The entire process is managed by FreeRTOS, ensuring that operations are synchronized.

Partitioning can be used as a general acceleration strategy on CNNs. Take convolution operation as an example, two partitioning strategies are typically employed:

Algorithm 2 Before

```
1: for i = 0 to outrows − 1 do
2:     for j = 0 to outcols − 1 do
3:         for k = 0 to innerdim − 1 do
4:             output[i][j] += input[i][k] ∗
               weight[k][j]
5:         end for
6:         output[i][j] += bias[j]
7:     end for
8: end for
```

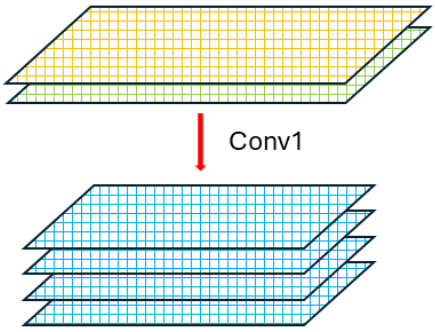
Algorithm 3 After

```
1: for i = 0 to outrows − 1 do
2:     for j = 0 to outcols − 1 do
3:         for k = 0 to innerdim − 1 do
4:             output[i][j] += input[i][k] ∗
               weight[j][k]
5:         end for
6:         output[i][j] += bias[j]
7:     end for
8: end for
```

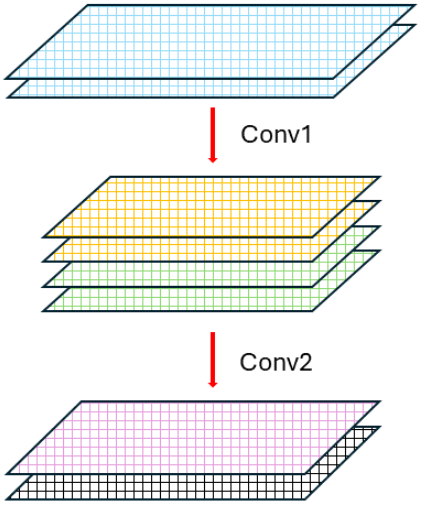
Figure 3.5: Comparison of Dense Layer Inference Algorithms

The first is to divide the inputs in the dimension-wise manner as shown in Figure 3.6a. The upper half of the feature maps are the input tensor, and the lower half maps are the output from the convolution operation. In this approach, not all inputs are required to be operated in each task; instead, only a portion of them is utilized. However, the output arrays are computed only partially. The final output is obtained by summing the outputs from each part. The second method instead divides the outputs in a dimension-wise manner as shown in Figure 3.6b. The three sets of tensors are the input, the output of the Con1 layer (intermediate tensor) and the output of the Conv2 layer. In this scenario, only half of the kernel and output are involved in each task, yet all input is required to be engaged.

In our use case, instances exist where certain inputs only have a single channel. To maximize resource utilization and prevent the coherence issue that arises when multi-core operates on the same output element simultaneously, we chose the second approach. Figure 3.7 illustrates the before-and-after execution processes comparison. The operation is split into two tasks and assigned to separate cores, theoretically reducing the inference time by up to half.



(a) Input Dimension-wise Partitioning



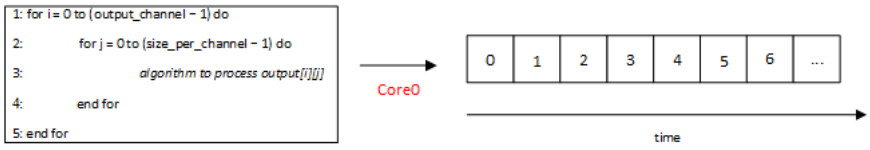
(b) Output Dimension-wise Partitioning

Figure 3.6: Layer Partitioning Strategy

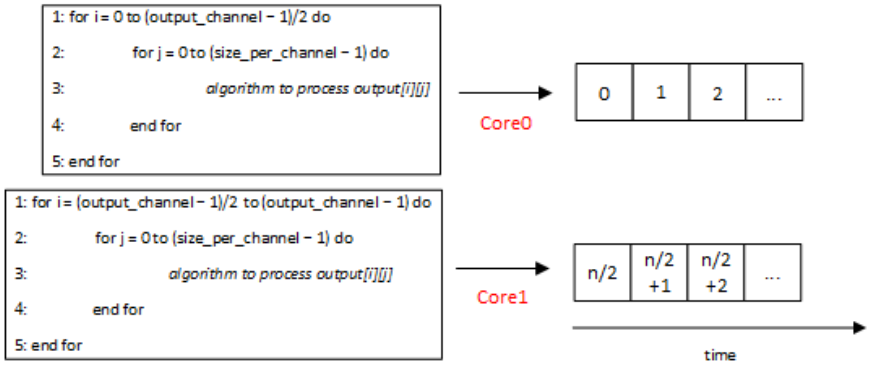
Application-aware Optimizations

Instruction Reordering. From a software perspective, directly reducing the redundant computation is the most straightforward approach to simplify the operations. During each computation, the input elements are found by calculating the index based on multiple parameters, including the kernel size and the corresponding output location across each dimension. However, there is a certain degree of correlation in the successive elements involved in the operation in terms of location. The index can be inferred by the previous element and the stride. Therefore, in these operations, we preserve the replicated portion of the previous index to maximize computation reuse. Additionally, optimizing the order of operations within nested loops can significantly enhance code execution performance as well.

Convolution operations in CNNs involve sliding a kernel block over the input to



(a) Original Execution



(b) Multi-core Parallelism-aware Partitioning Execution

Figure 3.7: Comparision between Original Execution and Multi-core Parallelism-aware Partitioning Execution

perform element-wise multiplications and accumulate the results, sequentially producing output elements. So, the position of the feature map, output, and kernel are inferable from each other. Additionally, the element's index is defined by its width, height, and depth, with overlapping portions of indices from adjacent accesses. Preserving these overlapping segments can reduce replicated index calculations.

Algorithm 4 illustrates that original convolution operations are implemented through nested loops, with all multiplications and index computations occurring in the innermost. This implementation leads to computations and latency overhead. As described in Section 3.3.1, the position of each output element can be located only with its height, width, and channel. During the iteration over all elements of the feature map, the indices for the input and kernel can be calculated based on the output location being generated. Furthermore, by moving the computations of overlapping

index segments computation to outer loops, the process can be further optimized as shown in Algorithm 5.

In the case of Dense layers, the BN layer and the Add layer, as previously stated in Section 3.4.1, process the elements sequentially which eliminates the index calculation.

Linear Computations Fusion. Linear transformations are carried out by iterating over every element and performing linear computations. A set of sequential linear operations in CNN inference can be fused into a single linear combination. This approach avoids multiple iterations for elements and best utilizes the embedded resources.

For the Batch Normalization (BN) layer, Add layer and bias add operation, the code goes through each element within a single loop in turn and performs the linear transformation. Therefore, in the forward pass of the CNN, when there are consecutive linear transformation operations, they can be fused together as shown in Figure 3.9. This fusion strategy effectively reduces loop overhead and best utilizes the embedded resources.

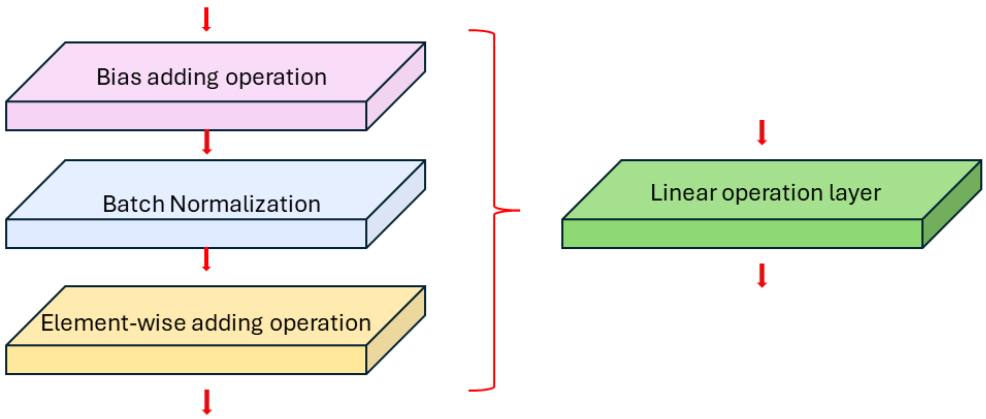


Figure 3.9: Linear Operation Fusing

Algorithm 4 Before Transformation

```
for x0 = 0 to out_rows − 1 do
  for x1 = 0 to out_cols − 1 do
    for z0 = 0 to kernel_rows − 1 do
      for z1 = 0 to kernel_cols − 1 do
        for q = 0 to in_channels − 1 do
          for k = 0 to out_channels − 1 do
            Convert x0, x1, z0, z1, q, k, stride, dilation to index
            Multiplication
          end for
        end for
      end for
    end for
  end for
end for
```

Algorithm 5 After Transformation

```
for x0 = 0 to out_rows − 1 do
  for x1 = 0 to out_cols − 1 do
    Convert output's width and height position to index
    for t = 0 to feature_map_element_num do
      Convert filter's width and height position to index
      Infer input's index with x0, x1, t, and stride/dilation
      for k = 0 to out_channels − 1 do
        Perform multiplication for the k'th output channel
      end for
    end for
  end for
end for
```

Figure 3.8: Comparative View of the Original and Updated Algorithms.

	Cache-friendly Memory Layout	Multi-core Parallelism-aware Partitioning	Instruction Reordering	Linear Computations Fusion	Intermediate Buffer Reuse
Conv layer		✓	✓		✓
Dense Layer	✓	✓			✓
Add Layer		✓		✓	✓
BN Layer		✓		✓	✓

Table 3.2: Metrics Compatibility for Each Layer

Intermediate Buffer Reuse. The intermediate tensors are keep generated during the CNN inference. Usually, each result takes up its own buffer. In the context of the feedforward network, those tensors would not be used after it's lifetime, therefore leading to the memory overhead. For tinyML, efficient storage management is a critical factor in determining whether a model can be successfully deployed on resource-constrained devices. A key part of this intermediate buffer reuse is the over-write mechanism, we allocate a fixed place in heap that can fit the peak memory usage, and over-write the old data with the new forward pass if it's no longer needed for future computations during the conduction. This schedule between the CNN layers can contribute to all the application with intensive storage resources.

Table 3.2 summarizes the compatibility for each operation.

3.4.2 Compatibility Across Each CNN

LeNet

LeNet is one of the revolutionary CNN models developed by Yann LeCun in 1998 [46]. From the high level, the LeNet architecture can be divided into two building blocks, the first part consists of two sets of Conv layers and a set of Max-pooling layers. The Conv layers are used to extract the required features from the input relevant features such as edges, textures, and shapes with a set of filters [33], and the pooling layers

are used to downsample the feature maps, reducing computational complexity, and focus on the most relevant information; the second part contains stacked of Fully Connected layer (Dense layer) that map the flattened output from the first part to the final layer.

The majority of the LeNet inference process occurs within the Conv and Dense layers, which can be accelerated for the overall model performance. Based on the previous Section 3.4.1, the cache-friendly memory layout can be deployed on the Dense layer; instruction reordering and multi-core parallelism-aware partitioning can be deployed on the Conv layer.

AlexNet

AlexNet is built on the ideas of LeNet but significantly expanded the depth and complexity of the architecture. Furthermore, the architecture also employed ReLU for nonlinear activation [43]. AlexNet is composed of eight weighted layers: five Conv layers, followed by three Dense layers. Additionally, we add a BN layer after each of the first four Conv layers for regularization.

In the AlexNet architecture, in addition to employing the cache-friendly memory layout and multi-core parallelism-aware partitioning for optimizing the Conv and Dense layers, the BN layer is integrated with the bias addition operation in the Conv operation. This linear computation fusion is effective because both processes include consecutive linear transformations, allowing for streamlined and efficient computation.

ResNet20

In 2015, Kaiming He introduced the Residual Network [30]. A key feature of ResNet is the use of residual learning to bypass certain layers with skip connections, allowing networks to be significantly deeper without the vanishing problem [31]. However, shortcut connections require additional memory to buffer intermediate feature maps. The element-wise additions after the skip operation increase computational complexity, which impacts the real-time performance on limited resources [32].

We deploy the ResNet20 which has twenty stacked weight layers and nine shortcuts. It has the fewest parameters of the four CNNs, yet it contains many skip connections, requiring the additional intermediate connection data to be preserved in memory as globals throughout the entire inference phase. The requirement leads to memory corruption on the heap during the inference, especially on embedded devices like ESP32, which are typically resource-limited. Therefore, intermediate buffer reuse becomes crucial for efficient implementation. Instead of pre-allocating all intermediate tensors in the heap, we preserve only a portion of the heap memory to store them. It allows the previously used intermediate data to be overwritten by the ongoing intermediate data, thereby minimizing the memory footprint. One of the residual blocks in the ResNet20 network is illustrated in Figure 3.10. During execution, six intermediate tensors are utilized, while only three tensor sizes are maintained. Sharing buffer space between independent layers reduces resource consumption. The Conv layers can be accelerated by instruction reordering and multi-core parallelism-aware partitioning. Additionally, the linear computations fusion in the application-aware strategy further reduces the number of layers and the intermediate tensors generated during these phases.

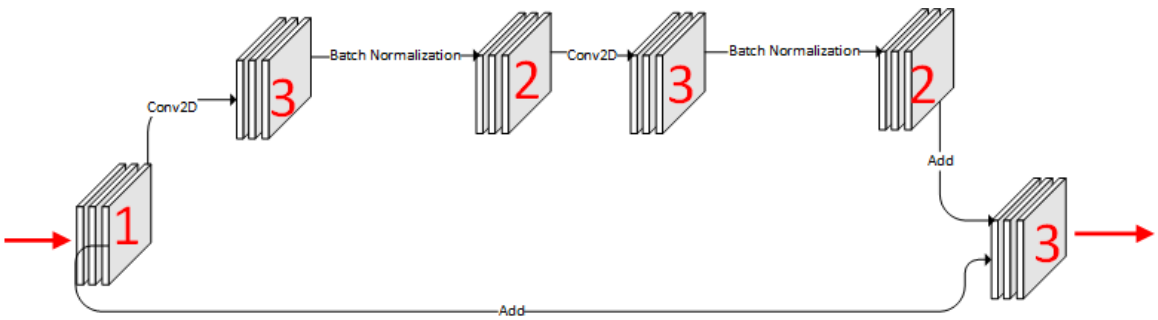


Figure 3.10: A Residual Block in ReNet20

SqueezeNet

Forrest N. Iandola developed SqueezeNet, achieving AlexNet-level accuracy with 50 times fewer parameters [35]. The novel part of SqueezeNet is to include a combination of 1x1 and 3x3 convolutions within the 'fire module'. 1x1 filters facilitate a reduction in the overall network parameters in situations where the filter numbers are less than the input channel numbers. The expend layer performs convolution with the 1x1 and 3x3 filters, subsequently concatenating the output in a dimension-wise manner, and then feeds it into the next block.

SqueezeNet replaces fully connected layers with a global average pooling layer to further reduce the model size. The eight Fire modules mainly consist of the Conv layer and the Concatenate layer. Therefore, we focus on optimizing the Conv layers, with instruction reordering and multi-core parallelism-aware partitioning.

Table 3.3 summarizes techniques applicable in each considered CNN.

	Cache-friendly Memory Layout	Multi-core Parallelism-aware Partitioning	Instruction Reordering	Linear Computations Fusion	Intermediate Buffer Reuse
LeNet	✓	✓	✓		✓
AlexNet	✓	✓	✓	✓	✓
ResNet20	✓	✓	✓	✓	✓
SqueezeNet		✓	✓		✓

Table 3.3: Metrics Compatibility across Each CNN

3.5 Evaluation

3.5.1 Experimental Setup

In this thesis, we employ the proposed optimizations on four CNNs – LeNet, AlexNet, ResNet20, and SqueezeNet on the tiny device. In doing so, we use a real use-case from the healthcare domain. In this use-case, the goal is to predict the SBP from the PPG signals. We use the MIMIC-IV dataset [39], applying an 80%:10%:10% split of the datatset for training, cross-validation, and testing, respectively [73]. The training phase is conducted on a PC machine running an octa-core Intel i7 processor with 32 GB of RAM, and saved in HDF5 format with the TensorFlow framework. After that, we convert it into a pure C program through the K2C to be deployed on the embedded device. The generated C code is mainly organized into two principal parts: the first encompasses a library for each operation's implementation, and the second is the main program that executes these operations. All parameters and data are of the float type. This organization enhances performance optimization by allowing direct modifications to function operations. The measurements derived from the original K2C code may be employed as a baseline. In the context of hardware deployment, the code is implemented on the ESP32-S3 MCU with the ESP-IDF extension. This extension is specifically designed for the development of IoT applications. Though the majority of MCUs nowadays are compatible with our changes, we only evaluate

the strategies on ESP32-S3 to simplify the experiment. The ESP32-S3 is equipped with 8MB of Flash memory, 8MB of PSRAM, and the Xtensa 32-bit LX7 dual-core processor. The performance metrics, including the accuracy and latency, are discussed in the subsequent sections.

3.5.2 Model Accuracy

In the aforementioned section, we adopt both software and hardware strategies to minimize inference latency. From a software perspective, we apply algorithmic optimizations to reduce redundant computations and inefficient memory accesses. On the hardware side, our approach focuses on enhancing cache-friendly memory access patterns to improve hit rates and maximize the utilization of the ESP32-S3 architecture. It is important to note that the model's structure and parameters are preserved, ensuring that the prediction remains unaffected. We compute the MAE between the predictions and the ground truth for both the updated K2C code and the original implementation. The comparison is presented in Table 3.4. One thing to note is that we measure the accuracy using a PC-based ESP32 simulator with a QEMU, whose fork is officially maintained by Espressif. This approach is firstly motivated by the fact that running large datasets on the ESP32 to gather results is a very time-consuming procedure. As detailed in Section 3.4.2, the original ResNet20 program is not able to be deployed on the device. The results indicate that there are no significant differences in accuracy between the two versions, which supports our expectations.

	LeNet	AlexNet	ResNet20	SqueezeNet
Original K2C code	13.36	12.12	NONE	11.81
Optimized code	13.36	12.12	13.80	11.96

Table 3.4: MAE Comparison Before and After Optimizations

3.5.3 Inference Latency

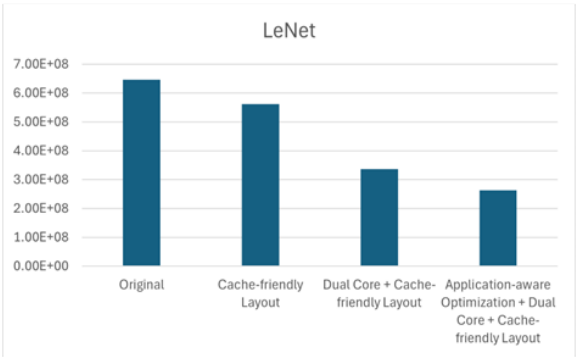
Figure 3.11 shows the inference latency of each solution and their respective groups across the entire model. The latency is recorded as the number of clock cycles in the Xtensa register. Although the intermediate buffer reuse strategy is technically applicable to all networks, it is utilized exclusively for ResNet20 to simplify the evaluation process. The results demonstrate that our strategy can significantly reduce the model's latency, with ResNet20 achieving a remarkable 71% reduction. Among those solutions, instruction reordering and the multi-core parallelism-aware partitioning strategy have the most obvious impact on speedup. This is primarily because the weighted layer in the LeNet model consists entirely of Dense and Conv layers, which are computationally intensive and complex to implement. Therefore, LeNet can be accelerated by 59% with co-optimizations for the Conv and Dense operations.

AlexNet exhibits a more complex structure compared to LeNet, as it features both weighted layers and a variety of layers that incorporate linear operations. Therefore, integrating diverse optimization strategies can greatly accelerate the model's overall speed by 61%.

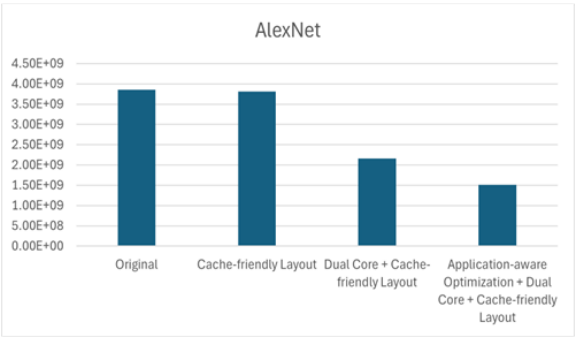
Though failing to deploy the original ResNet20 code on the ESP32 due to the memory corruption issue, we approximate the baseline latency by summing up the delay of the segmented ResNet20 structure. As shown in the diagram, these strategies have achieved a 71% latency reduction.

For SqueezeNet, which lacks Dense layers in its architecture, our optimization efforts are concentrated mainly on the Conv layers. We achieve a 62% improvement in network latency over the baseline.

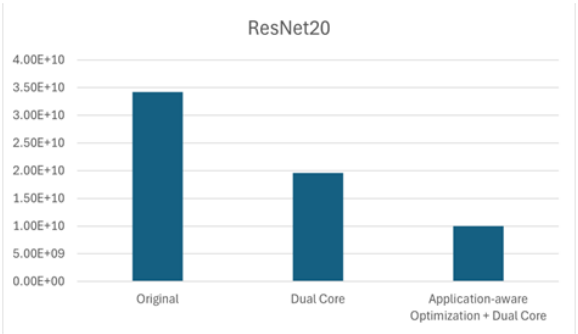
Table 3.5 shows the percentage reduction in latency compared to the baseline



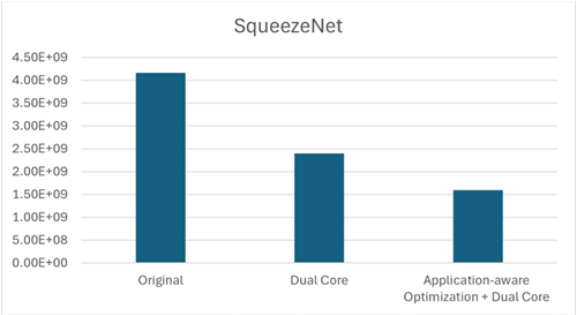
(a) LeNet



(b) AlexNet



(c) ResNet20



(d) SqueezeNet

Figure 3.11: Comparison of CNN Inference Latency

	Cache-friendly Memory Layout	Multicore Parallelsim Design	Application-aware Optimization
LeNet	87%	65%	49%
AlexNet	99%	56%	45%
ResNet20	X	57%	77%
SqueezeNet	X	58%	43%

Table 3.5: Latency Reduction with Optimizations Across CNNs Compared to Baseline

across CNNs under different strategies. These data indicate that different design strategies have significantly varying impacts on different neural network architectures. Overall, the Application-aware Optimization strategy demonstrates the best effect because the software optimizations can bring substantial benefits. Furthermore, combining these designs can further accelerate inference.

Figure 3.12 illustrates the maximum acceleration achieved by each layer under our techniques. For each type of layer, the strategy group that yields the best result is selected. Therefore, there may be more than one optimization for each layer, which can be referenced in Table 3.2. It is worth noting that, although the multi-core parallelism-aware partitioning strategy is employed solely for the Conv layers in latency measurement to simplify the experiment, it can technically be applied to all layers.

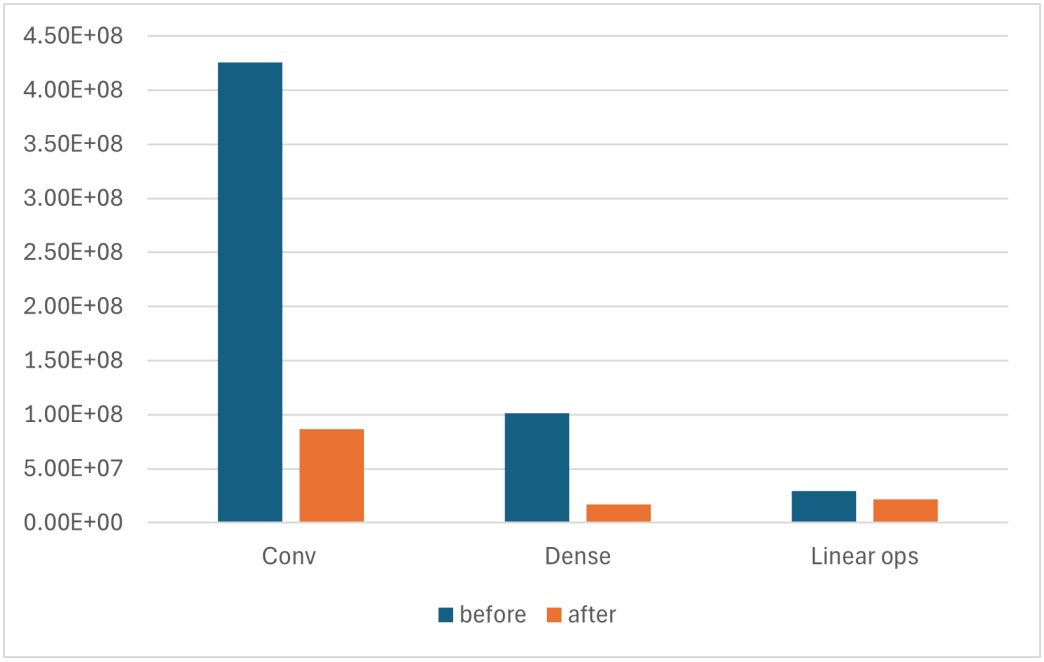


Figure 3.12: Comparison of Layer Inference Delay Before and After Optimizations

3.5.4 Layer Inference Delay

Conv Layer Delay.

Figure 3.13 illustrates a Conv layer delay comparison before and after the multi-core parallelism-aware partitioning and instruction reordering. It displays a series of bars in pairs, where each pair represents a Conv layer inside the four CNNs. The orange bars show performance before the change and the green bars indicate performance after. The Conv layer delay goes up with the FLOPs, a metric indicating the number of computations performed within the layer. Besides, there is a notable time reduction in the majority of cases post-optimization, suggesting significant improvements in computational efficiency across different networks.

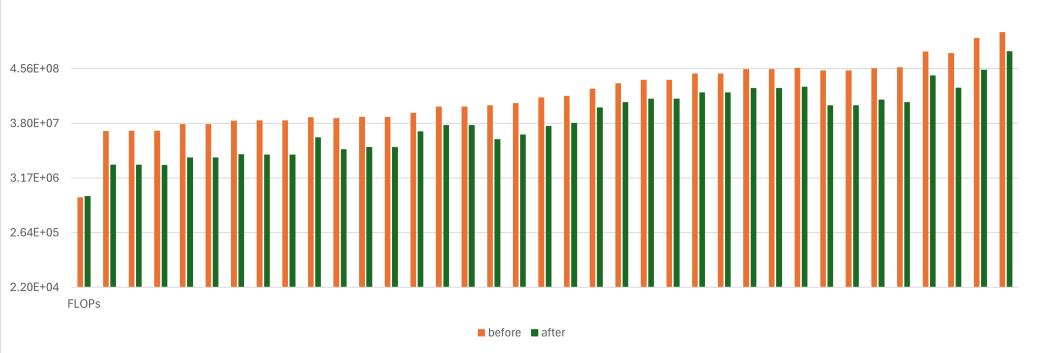


Figure 3.13: Comparison of Conv Layer Delay Before and After Optimizations

Dense Layer Delay.

Figure 3.14 illustrates that the relative change increases with the number of neurons in the Dense layer. Theoretically, several stages can be identified, demonstrating how our cache-friendly memory layout accelerates the Dense layer inference by increasing cache hit rates:

(1) Kernel matrices with a single row produce output sizes of 1×1, which is typical for a network's output layer with minimal parameters. In these cases,

both the input and kernel arrays are one-dimensional arrays. Our strategy does not improve the cache hit rate, given that all elements within the sole column of the kernel are already accessed in a sequential manner. Therefore, as indicated by the chart, the first three groups of bars show a ratio change of approximately 0%.

Figure 3.14: Comparison of Dense Layer Delay Before and After Optimizations

(2) When the kernel matrix is multi-dimensional, as shown in Figure 3.4a, elements are accessed with a stride equal to the output size. All parameters in our model are in float format, occupying 4 bytes each, while the cache line size is 32 bytes. Therefore, after each cache miss, the CPU would bring eight adjacent floats from the flash memory. Under a cache-friendly memory layout, where the stride is 1, 87.5% (7/8) of cache hit rate is obtained. As the stride increases, however, the cache hit rate decreases, reaching 0% when the stride reaches 8 or

higher. A miss occurs for the next access, resulting in the continuous loading of new segments from flash memory until all kernel neurons have been stored in the cache, thereby enabling the following cache hit. As a result, the overall cache hit rate would range from 0% to 87.5%, where our strategy starts showing benefits. This scenario was not represented in the chart due to the absence of applicable conditions within the CNNs used in this study.

(3) As the size of the kernel neurons increases beyond the cache capacity, the newly loaded cache lines begin to evict the initially loaded ones. CPU has to fetch the previously preserved data segment in the flash instead of the cache. It represents the worst-case scenario for data access, where the cache hit rate approaches 0%, posing a significant challenge for inference on the embedded devices. Therefore, the cache-friendly memory layout demonstrates the best optimization performance. This scenario is illustrated by the last three groups of bars.

Linear Operation Delay.

We fuse a series of consecutive linear operations into a single layer. The Chart 3.15 illustrates the comparison of linear operation delay before and after the fusion technique with respect to the count of output elements. The original data set, shown with blue dots and a blue-dotted trend line, indicates a steeper positive linear relationship compared to the updated data set, represented by orange dots and an orange-dotted trend line. This suggests that our approach effectively reduces the linear operation delay.

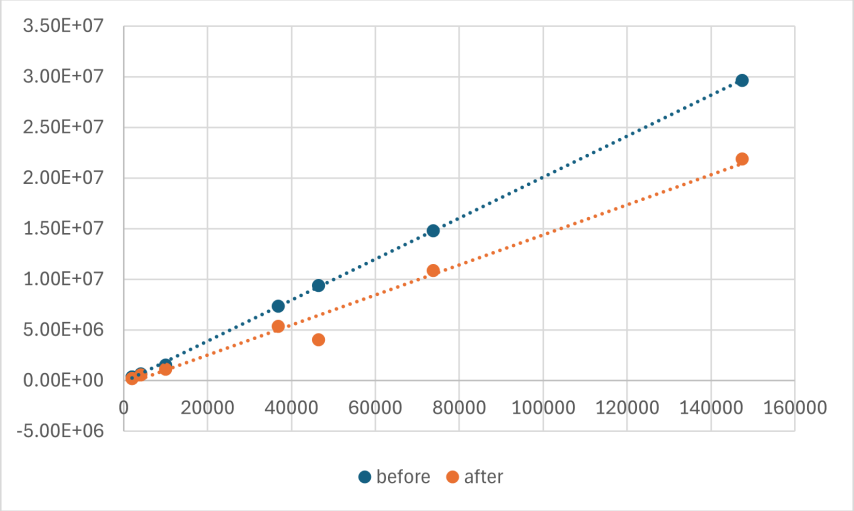


Figure 3.15: Fused Linear Operation Delay

3.6 Conclusion

In this thesis, we address the challenges of accelerating the network inference on the extreme resource-constrained device by proposing several techniques to optimize the inference process of CNNs. We discuss both software and hardware characteristics during the operations and propose approaches to reduce the latency based on that. These solutions demonstrate broad applicability across various networks and devices. We further evaluated four CNNs – LeNet, AlexNet, ResNet20, and SqueezeNet – specifically for SBP estimation tasks on ESP32. This evaluation provides a comprehensive analysis, from overall model performance to the impact of specific metrics on individual layers. The findings of this work highlight the potential of joint architecture- and application-aware optimization in improving the efficiency of tinyML applications across diverse use-cases and MCUs.

Chapter 4

Conclusion

This thesis investigates the implementation of CNNs on extremely limited-resource devices, using real-time blood pressure measurement as a use case. While the embedded device in the tinyML field is broadly referred, the memory capacity of the device used in this study is restricted to only a few megabytes. The on-device inference poses challenges due to its extensive computational demands and high-memory footprint. Therefore, reasonable compression of the model size and careful consideration of the available edge resources are key for successful deployment. Although the growing demand for tinyML drives the development of specialized processor designs that are equipped with the single instruction multiple data (SIMD) support and the neural network accelerator, those solutions rely on the specific platforms, thereby hindering generalization. The strategies developed in this thesis enable the fitting of CNNs on tiny devices with comparable accuracy and improved speed, while ensuring broad applicability across various devices and network architectures. From a high level, the thesis contributes to the tinyML field primarily in two directions: First, by

employing a series of optimization techniques, including the novel architecture modifications and a compression technique, this work shrinks the model's size to support the on-device machine learning without the cloud-cased dependency and the remote data transforming cost. The updated CNNs are evaluated on ESP32, Raspberry Pi Pico, and Arduino Nano BL, using only PPG signal as input, achieving satisfactory accuracy. Second, build on the models evaluated in the first section, the thesis further analyzes the CNNs inference process that is implemented with C backend that allows for hacking. TinyML implementation mainly relies on software and hardware. The thesis optimizes the code to eliminate redundant computation and buffer usage. Additionally, it investigates cache-friendly layout to improve the cache hit rate and multicore-aware methodology to achieve parallelism. As a result, those technologies demonstrate significant acceleration with no drop in accuracy.

Future research directions include latency reduction by carefully designing the preloading during the inference to maximize cache hit rates. Ensuring efficient cache usage remains a valuable topic to be explored for optimizing CNN inference on embedded devices.

Bibliography

[1] STM32Cube.AI, 2021. URL https://www.st.com/en/embedded-software/x-cube-ai.html.

[2] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, and A. S. Hafid. A comprehensive survey on tinyml. *IEEE Access*, 2023.

[3] K. Ahmed and M. Hassan. tinycare: A tinyml-based low-cost continuous blood pressure estimation on the extreme edge. In *2022 IEEE 10th International Conference on Healthcare Informatics (ICHI)*. IEEE, 2022.

[4] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, et al. A low power, fully event-based gesture recognition system. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7243–7252, 2017.

[5] A. Anaya-Isaza, L. Mera-Jiménez, and M. Zequera-Diaz. An overview of deep learning in medical imaging. *Informatics in Medicine Unlocked*, 26:100723, 2021. ISSN 2352-9148. doi: https://doi.org/10.1016/j.imu.2021.100723. URL https://www.sciencedirect.com/science/article/pii/S2352914821002033.

[6] P. Andrade, I. Silva, M. Silva, T. Flores, J. Cassiano, and D. G. Costa. A

tinyml soft-sensor approach for low-cost detection and monitoring of vehicular emissions. *Sensors*, 22(10):3838, 2022.

[7] S. Baek, J. Jang, and S. Yoon. End-to-end blood pressure prediction via fully convolutional networks. *IEEE Access*, 7:185458–185468, 2019. doi: 10.1109/ACCESS.2019.2960844.

[8] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini. Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(5):871–875, 2020.

[9] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018. IEEE, 2019.

[10] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes. Visual wake words dataset. *arXiv preprint arXiv:1906.05721*, 2019.

[11] G. Cloud. Edge tpu—run inference at the edge. `https://cloud.google.com/edge-tpu/`.

[12] CMSIS-NN, 2021. URL `https://arm-software.github.io/CMSIS_5/NN/html/`.

[13] R. Conlin, K. Erickson, J. Abbate, and E. Kolemen. Keras2c: A library for converting keras neural networks to real-time compatible c. *Engineering Applications of Artificial Intelligence*, 100:104182, 2021.

[14] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini. Pulp: A ultra-low power

parallel accelerator for energy-efficient and flexible embedded vision. *Journal of Signal Processing Systems*, 84:339–354, 2016.

[15] R. David, J. Duke, A. Jain, P. Warden, M. Stamatescu, E. Murillo, S. Han, and C. Anderson. Tensorflow lite micro: Embedded machine learning for tinyml systems. In *Proceedings of Machine Learning and Systems*, volume 3, pages 800–811, 2021.

[16] B. D. de Dinechin, C. Monat, P. Blouet, and C. Bertin. Dsp-mcu processor optimization for portable applications. *Microelectronic engineering*, 54(1-2):123–132, 2000.

[17] M. de Prado, M. Rusci, A. Capotondi, R. Donze, L. Benini, and N. Pazos. Robustifying the deployment of tinyml models for autonomous mini-vehicles. *Sensors*, 21(4):1339, 2021.

[18] X. Ding, B. P. Yan, Y.-T. Zhang, J. Liu, N. Zhao, and H. K. Tsang. Pulse transit time based continuous cuffless blood pressure estimation: A new extension and a comprehensive evaluation. *Scientific reports*, 7(1):1–11, 2017.

[19] L. Dutta and S. Bharali. Tinyml meets iot: A comprehensive survey. *Internet of Things*, 16:100461, 2021.

[20] Embedded Learning Library, 2021. URL `https://microsoft.github.io/ELL/`.

[21] A. for the Advancement Instrumentation. American National Standard for Electronic or Automated Sphygmomanometers.

[22] J. F. Gemmeke, D. P. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter. Audio set: An ontology and human-labeled dataset

for audio events. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 776–780. IEEE, 2017.

[23] A. Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2019.

[24] H. Gokul, P. Suresh, B. H. Vignesh, R. P. Kumaar, and V. Vijayaraghavan. Gait recovery system for parkinson's disease using machine learning on embedded platforms. In *2020 IEEE International Systems Conference (SysCon)*, pages 1–8. IEEE, 2020.

[25] J. Gold and K. Shaw. What is edge computing and why does it matter?, May 2022. URL `https://www.networkworld.com/article/3224893/what-is-edge-computing-and-how-it-s-changing-the-network.html`.

[26] I. Goodfellow, Y. Bengio, and A. Courville. Deep learning (adaptive computation and machine learning series). *Cambridge Massachusetts*, pages 321–359, 2017.

[27] B. H. Drug discovery and molecular modeling using artificial intelligence. *Artificial Intelligence in Healthcare*, pages 61–83, 2020.

[28] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[29] F. J. He and G. A. MacGregor. Blood pressure is the most important cause of death and disability in the world. *European Heart Journal Supplements*, 9

(suppl_B):B23–B28, 05 2007. ISSN 1520-765X. doi: 10.1093/eurheartj/sum005. URL https://doi.org/10.1093/eurheartj/sum005.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[31] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*. Springer International Publishing, 2016.

[32] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[33] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[34] S. Hymel, L. Urdaneta, I. Veras, and M. Banzi. Edge impulse: An mlops platform for tiny machine learning. *arXiv preprint arXiv:2212.03332*, 2022.

[35] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[36] T. M. Ingolfsson, A. Cossettini, X. Wang, E. Tabanelli, G. Tagliavini, P. Ryvlin, L. Benini, and S. Benatti. Towards long-term non-invasive monitoring for

epilepsy via wearable eeg devices. In *2021 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 01–04. IEEE, 2021.

[37] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[38] M. Ji, W. Kim, B. Kim, J.-Y. Lee, and H.-J. Yoo. Demand layering for real-time dnn inference with minimized memory usage. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022.

[39] A. Johnson, L. Bulgarelli, T. Pollard, L. A. Celi, R. Mark, and S. Horng IV. Mimic-iv-ed. *PhysioNet*, 2021.

[40] M. Kachuee, M. M. Kiani, H. Mohammadzade, and M. Shabany. Cuffless blood pressure estimation algorithms for continuous health-care monitoring. *IEEE Transactions on Biomedical Engineering*, 64(4):859–869, 2016.

[41] Y. Kalyani and R. Collier. A systematic survey on the role of cloud, fog, and edge computing combination in smart agriculture. *Sensors*, 21(17):5922, 2021.

[42] E. Kim, J.-K. Yoon, S.-S. Shin, S.-H. Kim, K.-H. Kim, D.-S. Seo, and C.-H. Park. Tinyml-based classification in an ecg monitoring embedded system. *Computers, Materials and Continua*, 75(1):1751–1764, 2023.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, volume 25, 2012.

[44] A. L. An Introduction to TinyML, November 2020. URL `https://towardsdatascience.com/an-introduction-to-tinyml-4617f314aa79`.

[45] L. Lai, N. Suda, and V. Chandra. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv preprint arXiv:1801.06601*, 2018.

[46] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[47] K. J. Lee, J. Roh, D. Cho, J. Hyeong, and S. Kim. A chair-based unconstrained/nonintrusive cuffless blood pressure monitoring system using a two-channel ballistocardiogram. *Sensors*, 19(3), 2019. ISSN 1424-8220. doi: 10.3390/s19030595. URL `https://www.mdpi.com/1424-8220/19/3/595`.

[48] E. Liberis, Dudziak, and N. D. Lane. nas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, 2021.

[49] J. Lin, W.-M. Chen, Y. Lin, C. Gan, and S. Han. Mcunetv2: Memory-efficient patch-based inference for tiny deep learning. *arXiv preprint arXiv:2110.15352*, 2021.

[50] R. Lippmann. An introduction to computing with neural nets. *IEEE Assp magazine*, 4(2):4–22, 1987.

[51] S. Lobov, N. Krilova, I. Kastalskiy, V. Kazantsev, and V. A. Makarov. Latent factors limiting the performance of semg-interfaces. *Sensors*, 18(4):1122, 2018.

[52] G. Menghani. Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *ACM Computing Surveys*, 55(12):1–37, 2023.

[53] E. Monte-Moreno. Non-invasive estimate of blood glucose and blood pressure from a photoplethysmograph by means of machine learning techniques. *Artificial intelligence in medicine*, 53(2):127–138, 2011.

[54] A. Mostafavi and A. Sadighi. A novel online machine learning approach for real-time condition monitoring of rotating machines. In *2021 9th RSI International Conference on Robotics and Mechatronics (ICRoM)*, pages 267–273. IEEE, 2021.

[55] Y. Nahshan, B. Chmiel, C. Baskin, E. Zheltonozhskii, R. Banner, A. M. Bronstein, and A. Mendelson. Loss aware post-training quantization. *Machine Learning*, 110(11):3245–3262, 2021.

[56] NVIDIA. Nvidia embedded systems for next-gen autonomous machines. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems`, 2021. Retrieved June 4, 2021.

[57] E. O'Brien, J. Petrie, W. Littler, M. de Swiet, P. L. Padfield, K. O'Malley, M. Jamieson, D. Altman, M. Bland, and N. Atkins. The british hypertension society protocol for the evaluation of automated and semi-automated blood pressure measuring devices with special reference to ambulatory systems. *Journal of Hypertension*, 8(7):607–619, 1990.

[58] L. Oden and T. Witt. Fall-detection on a wearable micro controller using machine learning algorithms. In *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 296–301. IEEE, 2020.

[59] M. M. Ogore, K. Nkurikiyeyezu, and J. Nsenga. Offline prediction of cholera

in rural communal tap waters using edge ai inference. In *2021 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6. IEEE, 2021.

[60] A. Omambia, B. Maake, and A. Wambua. Water quality monitoring using iot & machine learning. In *2022 IST-Africa Conference (IST-Africa)*, pages 1–8. IEEE, 2022.

[61] H. Pandey and S. Prabha. Smart health monitoring system using iot and machine learning techniques. In *2020 sixth international conference on bio signals, images, and instrumentation (ICBSII)*. IEEE, 2020.

[62] D. Pau and P. K. Ambrose. Automated neural and on-device learning for micro controllers. In *2022 IEEE 21st Mediterranean Electrotechnical Conference (MELECON)*. IEEE, 2022.

[63] R. Perego, A. Pastorelli, P. Sironi, G. Danelon, S. Larcher, L. Bocchi, and M. Butti. Tuning deep neural network's hyperparameters constrained to deployability on tiny systems. In *International Conference on Artificial Neural Networks*. Springer International Publishing, 2020.

[64] V. J. Reddi, B. Plancher, S. Kennedy, L. Moroney, P. Warden, A. Agarwal, C. Banbury, M. Banzi, M. Bennett, B. Brown, et al. Widening access to applied machine learning with tinyml. *arXiv preprint arXiv:2106.04008*, 2021.

[65] S. S. Saha, A. Das, A. K. Saha, S. Ghosh, and S. B. Chattopadhyay. Tcing. *ACM Transactions on Embedded Computing Systems*, 2023.

[66] R. Sanchez-Iborra and A. F. Skarmeta. Tinyml-enabled frugal smart objects:

Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 20(3):4–18, 2020.

[67] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[68] SBP Group. 12 Real-World Applications of Machine Learning in Healthcare, February 2020. URL `https://spd.group/machine-learning/machine-learning-in-healthcare/`.

[69] O. Schlesinger, N. Vigderhouse, D. Eytan, and Y. Moshe. Blood pressure estimation from ppg signals using convolutional neural networks and siamese network. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1135–1139, 2020. doi: 10.1109/ICASSP40776.2020.9053446.

[70] S. Shimazaki, H. Kawanaka, H. Ishikawa, K. Inoue, and K. Oguri. Cuffless blood pressure estimation from only the waveform of photoplethysmography using cnn. *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 5042–5045, 2019. doi: 10.1109/EMBC.2019.8856706.

[71] R. Stahl, Z. Zhao, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann. Fully distributed deep learning inference on resource-constrained edge devices. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 19th International Conference, SAMOS 2019, Samos, Greece, July 7–11, 2019, Proceedings 19*, pages 77–90. Springer, 2019.

[72] R. Stahl, D. Mueller-Gritschneder, and U. Schlichtmann. Fused depthwise tiling for memory optimization in tinyml deep neural network inference. *arXiv preprint arXiv:2303.17878*, 2023.

[73] B. Sun, S. Bayes, A. M. Abotaleb, and M. Hassan. The case for tinyml in healthcare: Cnns for real-time on-edge blood pressure estimation. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 629–638, 2023.

[74] X. Sun, L. Zhou, S. Chang, and Z. Liu. Using cnn and hht to predict blood pressure level based on photoplethysmography and its derivatives. *Biosensors*, 11(4), 2021. ISSN 2079-6374. doi: 10.3390/bios11040120. URL `https://www.mdpi.com/2079-6374/11/4/120`.

[75] Y. Sun, J. Zhang, Y. Xiong, and G. Zhu. Data security and privacy in cloud computing. *International Journal of Distributed Sensor Networks*, 10(7):190903, 2014.

[76] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[77] TensorFlow. Tensorflow lite, 2021. URL `https://www.tensorflow.org/lite/`.

[78] tinyML. tinyml foundation, 2021. URL `https://www.tinyML.org`.

[79] V. Tsoukas, E. Boumpa, G. Giannakas, and A. Kakarountas. A review of machine learning and tinyml in healthcare. In *Proceedings of the 25th Pan-Hellenic Conference on Informatics*, pages 69–73, 2021.

[80] I. Ullah and A. Petrosino. About pyramid structure in convolutional neural networks. In *2016 International joint conference on neural networks (IJCNN)*, pages 1318–1324. IEEE, 2016.

[81] Y. Vaizman, K. Ellis, and G. Lanckriet. Recognizing detailed human context in the wild from smartphones and smartwatches. *IEEE pervasive computing*, 16(4): 62–74, 2017.

[82] C. Vuppalapati, A. Ilapakurti, K. Chillara, S. Kedari, and V. Mamidi. Automating tiny ml intelligent sensors devops using microsoft azure. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2375–2384. IEEE, 2020.

[83] P. Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[84] P. Warden and D. Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

[85] J. Weigert. xxd - Unix, Linux Command. URL `https://www.tutorialspoint.com/unix_commands/xxd.htm`.

[86] H.-S. Zheng, Y.-Y. Liu, C.-F. Hsu, and T. T. Yeh. Streamnet: Memory-efficient streaming tiny deep learning inference on the microcontroller. *Advances in Neural Information Processing Systems*, 36, 2024.

[87] T. Zhu, L. Kuang, K. Li, J. Zeng, P. Herrero, and P. Georgiou. Blood glucose prediction in type 1 diabetes using deep learning on the edge. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[88] S. Zixiao, M. Fen, M. Qinghan, and L. Ye. Cuffless and continuous blood pressure estimation based on multiple regression analysis. *2015 5th International Conference on Information Science and Technology (ICIST)*, pages 117–120, 2015. doi: 10.1109/ICIST.2015.7288952.