Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Evaluation and optimization of AI-based sensing for baseband systems

The effects of pruning, clustering, and quantization on line-of-sight blockage prediction

ERIK PERSSON

Evaluation and optimization of AI-based sensing for baseband systems

The effects of pruning, clustering, and quantization on line-of-sight blockage prediction

ERIK PERSSON

Degree Programme in Computer Science and Engineering
Date: March 11, 2025

Supervisors: Archana Thakare, Jorge Garrido Balaguer
Examiner: Mats Nordahl
School of Electrical Engineering and Computer Science
Host company: Huawei Technologies Sweden AB
Swedish title: Utvärdering och optimering av AI-baserad analys för basbandssystem
Swedish subtitle: Effekterna av beskärning, klustring och kvantisering på neuronnätsförutsägelse av siktlinjeblockering

Abstract

Millimeter-wave and sub-terahertz frequencies are emerging as key technologies for future baseband networks in the telecommunications industry, as they enable higher data rates and lower latency, critical factors for both users and service providers. However, these media rely heavily on maintaining a clear line-of-sight between the emitter and the receiver to provide the desired service level. Machine learning and neural network-based models have demonstrated promising results in predicting future line-of-sight conditions between a base station and a client, particularly by leveraging radar to sense the surrounding environment.

Research has been conducted on developing machine learning models for predicting future line-of-sight blockage using radar. However, no real-world implementation has yet been industrialized and deployed, as several challenges remain. Baseband equipment is subject to stringent hardware constraints to minimize energy consumption and production costs, as well as strict timing requirements that significantly influence the feasibility of executing such models on the system.

This thesis specifically examines the hardware requirements associated with running a model for line-of-sight blockage prediction, how these requirements scale with the neural network optimizations: network pruning, weight clustering, and network quantization, and if it is feasible to run this model on real-world deployed hardware. To accomplish this, we tested the model with all different combinations of software optimization, on two different inference platforms based on TFlite and ONNX, on one ARM and one X86-based simulated hardware platform using the gem5 hardware simulator. Each hardware platform was designed to mirror hardware that is suitable for a real-world baseband system.

This thesis examines the hardware requirements for running a model developed by Demirhan and Alkhateeb for line-of-sight blockage prediction, how these requirements scale with the three neural network optimization techniques network pruning, weight clustering, and quantization, and the feasibility of deploying this model on real-world hardware. To evaluate this, the model was tested with various combinations of software optimizations on two different inference platforms, based on TFLite and ONNX, respectively. These tests were conducted on simulated ARM- and x86-based hardware platforms using the gem5 hardware simulator. Each simulated hardware configuration was designed to reflect architectures suitable for real-world baseband systems.

The results in this thesis concluded that neither unstructured pruning nor clustering affected the hardware requirements of running the model. Quantization greatly reduced the memory traffic volume and bandwidth by 44% and 62% respectively, but increased inference time by 48% on TFlite and 50% on ONNX. Structured pruning improved all hardware characteristics across the board, with a 34% to 80% decrease in the number of committed CPU instructions, a 51% to 70% decrease in memory volume, and a 25% decrease to a 45% increase in bandwidth depending on the platform, as well as providing the lowest inference time. Integer instructions, SIMD integer instructions, and SIMD floating-point instructions were the most utilized CPU instructions regardless of the platform, with quantization heavily skewing the results from SIMD floating-point to SIMD integer instructions. Lastly, we concluded that running this model for line-of-sight prediction on hardware suitable for baseband systems would be feasible, as the expected inference time on real hardware would be lower than 30ms, allowing for fast reactions to environmental changes.

Keywords

Machine Learning, Baseband, Radar Sensing, Neural Network Pruning, Weight Clustering, Neural Network Quantization, Gem5

Sammanfattning

Millimetervågor och sub-terahertzfrekvenser är dominanta i framtida basbandsnätverk inom telekomindustrin. Dessa frekvenser möjliggör högre hastigheter och lägre latens, två avgörande faktorer för både användare och operatörer. Samtidigt är dessa tekniker starkt beroende av fri sikt mellan sändare och mottagare för att kunna uppnå önskad prestanda. Användning av maskininlärning och modeller baserade på neurala nätverk har visat stor potential för att förutsäga framtida förhållanden för fri sikt mellan en basstation och en användare, särskilt genom att använda radar för att kartlägga omgivningen.

Tidigare forskning har tagit fram maskininlärningsmodeller för att förutse blockering av fri sikt med hjälp av radar. Än så länge har dock inga lösningar implementerats i verkliga industriella system då det fortfarande existerar flera utmaningar som behövar hanteras. Basbandsutrustning har strikta hårdvarukrav för att hålla energiförbrukning och produktionskostnader nere, samtidigt som hårda tidskrav begränsar vad som faktiskt kan köras på systemen.

I den här projektet har vi undersökt vilka hårdvarukrav som krävs för att köra en modell för att förutsäga blockering av fri sikt. Vi har också studerat hur dessa krav påverkas av olika optimeringar av neurala nätverk, såsom nätverksbeskärning, viktklustring och kvantisering, samt om det är praktiskt möjligt att köra modellen på verklig hårdvara som används i basbandsystem. För att genomföra detta testades modellen med olika kombinationer av mjukvaruoptimeringar på två inferensplattformar baserade på TFlite och ONNX. Dessa kördes på en ARM- och en X86-baserad simulerad hårdvaruplattform med hjälp av hårdvarusimulatorn gem5. Varje plattform var utformad för att efterlikna hårdvara som kan förekomma basbandsystem.

Resultaten visade att ostrukturerad beskärning och klustring inte hade någon större inverkan på modellens hårdvarukrav. Kvantisering däremot minskade både minnesåtkomstvolymen och minnesbandbredden markant, med 44% respektive 62%, men ledde också till en ökning av inferenstiden med 48% för TFlite och 50% för ONNX. Strukturerad beskärning förbättrade däremot alla aspekter av hårdvarukraven, med en minskning av CPU-instruktionerna med 34–80%, en minskning av minnesåtkomstvolymen med 51–70%, samt en förändring av minnesbandbredden som varierade mellan en minskning på 25% och en ökning på 45% beroende på plattform. Dessutom hade denna optimering de kortaste inferenstiderna.

Vi fann också att heltalsinstruktioner, SIMD-heltalsinstruktioner och SIMD-flyttalsinstruktioner var de mest frekvent använda CPU-instruktionerna, oavsett plattform. Kvantisering påverkade kraftigt fördelningen mellan SIMD-flyttals- och heltalsinstruktioner och ledde till en ökad användning av heltalsinstruktioner.

Sammanfattningsvis visar resultaten att det är möjligt att köra en modell för förutspå blockeringar av fri sikt på hårdvara som är lämplig för basbandsystem. Den beräknade inferenstiden för modellen på verklig hårdvara förväntas vara under 30 millisekunder, vilket gör att systemet kan reagera snabbt på förändringar i omgivningen.

Nyckelord

Maskininlärning, Basband, Radar, Neuronnät-beskärning, Vikt-klustring, Neuronnät-kvantisering, Gem5

Acknowledgments

First I would like to thank Huawei Sweden Research and my supervisor at Huawei, Jorge Garrido Balaguer, for hosting me and my thesis and providing all the resources and support needed to make this thesis possible.

Special thanks to Jorge for all the great feedback and support throughout the project and for being my supervisor and mentor.

Lastly, I would like to thank my colleagues and friends for their support and guidance.

Stockholm, March 2025
Erik Persson

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

AI	Artificial Intelligence
CNN	Convolutional Neural Network
FPGA	Field-programmable Gate Array
IPC	Instructions Per Cycle
LOS	line-of-sight
LSTM	Long Short-Term Memory
SDG	Sustainable Development Goal
UN	United Nations

Chapter 1

Introduction

1.1 Background

Millimeter-wave and sub-terahertz frequencies are becoming the dominant direction for future baseband networks, as they can better provide higher-speed communications and reduce latency, key points of interest for both users and providers. However, these medias are highly dependent on having line-of-sight (LOS) between the emitter and the receiver to provide the desired service level. Using the network's capabilities for sensing is a promising direction for both increasing current service performance, as well as enabling new applications.

Artificial Intelligence (AI) is an emergent topic for wireless networks with several potential applications. In particular, prediction techniques are proposed for the optimization of different processes inside baseband systems. Among these potential applications is the prediction of near-future communication channel conditions in order to make better-informed management decisions. In particular, the application of AI in sensing for LOS blockage prediction has shown promising results in baseband scenarios [1] [2] [3]. Predicting future LOS blockage could enable proactive hand-off, proactive resource management, and beam switching in order to continue to provide the user with a high level of service in the event of a blockage. However, limited computing power and strict time requirements dictate what is feasible to run in baseband systems. This project, in particular, looks at what computational resources state-of-the-art methods for sensing in baseband systems require, the feasibility of running these methods from a computational cost to accuracy point of view, and the impact of software optimization strategies (e.g., graph pruning and quantization).

1.2 Problem

In this thesis, we want to look into the usage of machine learning algorithms, utilizing deep neural networks, for sensing in industrial baseband systems applications to see the applicability of running such algorithms in real-world scenarios. Previous research has shown promising improvements to, e.g., condition prediction [1] [2] [3], which could improve the planning and resource utilization of the systems. However, real-world baseband systems impose strict requirements on what hardware is applicable to put in the systems. It needs to be low power enough to reduce cooling issues and electricity costs, cheap enough to be mass-produced and deployed into real scenarios, but also powerful enough to handle all computing tasks associated with its applications. This imposes the question of whether or not it is feasible to run these algorithms on hardware applicable for this kind of deployment, and if we can improve the resource usage by software optimizations without impacting the prediction performance.

1.2.1 Research Question

How do the software optimizations quantization, pruning, and clustering impact the inference characteristics of prediction performance, inference time, CPU instruction usage, and memory usage of a deep neural network model designed for radar-aided blockage prediction in a baseband scenario?

1.3 Purpose

The purpose of this thesis is to evaluate how the neural network optimizations quantization, pruning, and clustering affect the inference characteristics of deep learning models for radar-aided blockage prediction in baseband systems. By analyzing these effects, this study aims to guide future hardware designs suitable for deploying machine learning models within baseband systems. Improved hardware and software integration could enhance service quality by enabling more efficient utilization and planning of transmission resources, potentially leading to greater energy efficiency. Additionally, this thesis provides insights into how these neural network optimizations scale at the hardware level and contributes to the understanding of simulating hardware behavior under AI-driven prediction workloads.

1.4 Goals

The goal of this thesis is to evaluate the usage of deep neural network based algorithms in the real-world industrial case of baseband systems. This has been divided into the following four sub-goals:

1. Evaluate the usage of deep neural networks in baseband systems by improving the understanding of the technology, to see if it is interesting to industrialize and develop into a future product.

2. Characterize the requirements of deep neural networks in an industry application for prediction, and the effects of deep neural network optimizations on a real industrial usecase.

3. Design tests that generate applicable data points. Critical evaluation of results according to defined evaluation criteria. Discuss possibilities of future work and improvement opportunities.

4. Define hardware requirements for inference using deep neural networks, and how these scale with software optimizations, in order to influence future baseband hardware design.

1.5 Research Methodology

This thesis conducts a study with simulation-based experiments. This was chosen to evaluate the problem statement in a realistic scenario while also gathering results and experiences about specific low-level hardware effects. As we want to evaluate the applicability of AI for sensing in baseband systems, we have to develop a test environment that mirrors such a scenario as closely as possible. To do that, we use real-world measurements as input and target data for the AI algorithm and popular production-grade inference platforms. To isolate the parts that we want to measure as much as possible (the AI inference), we are doing the necessary pre-processing of the data beforehand and embedding this into the test environment. The pre-processing step is not an AI task, but rather a signal processing task, and is thus out of scope for this thesis as we are studying the applicability and cost of AI.

1.6 Delimitations

In this thesis, we are not implementing or extending any software for hardware simulations; instead we utilize available open-source software and build a platform around that. We are also not looking into proposing, creating, or implementing new software optimizations for neural networks; rather, we want to utilize already existing optimization techniques. However, we are open to extending existing implementations to provide more functionality and test variants of optimization methods. We are not proposing any new AI based solutions to the problem of blockage prediction in baseband systems but rather use previous work to evaluate the feasibility of such a solution. This project is also limited to a single problem domain, radar-aided blockage prediction for baseband systems, using one dataset from this domain, a single model, and the neural network optimization techniques of quantization, pruning, and clustering.

1.7 Structure of the thesis

Chapter 2 presents relevant background information about neural networks, neural network optimizations, and past work on LOS blockage prediction for baseband systems. Chapter 3 presents the methodology and method used to solve the problem. Chapter 4 presents the development work done as part of this thesis. Chapter 5 presents the results and analyses from the conducted experiments. Chapter 6 presents the discussion of the results. Chapter 7 presents a summary of the work and results of this thesis, and some conclusions and possible future work.

Chapter 2

Background

Neural networks are widely used in modern AI and machine learning. This chapter thus provides basic background information about neural networks. The main topic for this thesis is neural network optimizations, in particular quantization, pruning, and clustering, which are also covered in this chapter. Additionally, related work regarding the usage of AI to solve problems faced in baseband systems is covered as it is foundational for this thesis work. Some software tools used for machine learning and hardware simulations are also covered as they are used throughout the thesis.

This chapter provides the background for this thesis. Section 2.1 presents the fundamental concepts of neural networks. Section 2.2 discusses three neural network optimization techniques: quantization, weight pruning, and weight clustering. Section 2.3 introduces the gem5 hardware simulator utilized in this study. Section 2.4 describes TFLite, one of the libraries used for neural network inference, while section 2.5 covers ONNX, the other inference library used in this work. Finally, Section 2.6 examines related research and its relevance to this thesis.

2.1 Neural Network

A neural network is a layered structure of simple mathematical functions called neurons, where the output from the neurons in one layer is fed into the neurons of the next layer, which is able to model more complex functions by tuning its internal weight values. A neuron in this case is a function that is modeled after a schematic model of a biological neuron. This function takes in a series of input values that are multiplied with some weight values and then reduced by summation, the sum is then usually subtracted from with a bias value

and passed through a non-linear function (called activation function) before being forwarded to the next layer of neurons. A neuron could be expressed by equation 2.1, where x is a vector of input values, w is the vector of weights, b is the bias weight, f is the activation function, and y is the output value of the neuron [4].

A basic graphical representation of a simple neural network can be found in figure 2.1.

$$y = f(\sum_{i=1}^{n} w_i x_i - b) \tag{2.1}$$

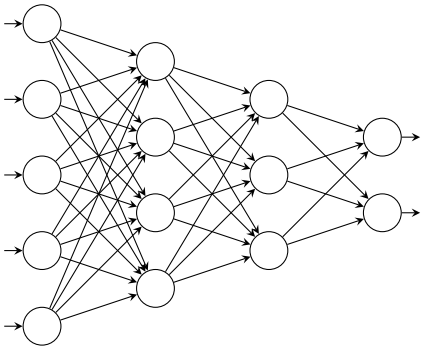


Figure 2.1: A graphical overview of a neural network with four layers of neurons. Each node in the graph represents a neuron, and each directed edge represents the passing of values between neurons. As the arrows are showing, the values are fed from left to right, with the layer farthest to the left receiving the input data, and the layer farthest to the right outputting the output data.

2.2 Neural Network Optimization

As neural networks are used to tackle more and more complex problems, their size (and with it their cost of computation) has increased with some modern models reaching over a billion parameters. Several strategies for lowering the computational cost of neural network based models have been proposed in order to make these models more usable in more deployment scenarios. This section covers three of them, quantization, pruning, and clustering, which are used in this thesis work.

2.2.1 Quantization

Neural network quantization is an optimization strategy that stores weight tensors and activations in a lower bit precision format than the one originally used in the neural network. Usually, these values are stored as 16 or 32-bit floating point values, but quantization can map these values to, e.g., an 8-bit integer type. Going from 32-bit down to 8-bit reduces the memory overhead of storing the tensors by a factor of 4, and the computational cost of doing matrix multiplications by a factor of 16. Studies suggest that neural networks can exhibit a degree of robustness to quantization, meaning that the reduction in numerical precision does not always lead to a significant drop in accuracy. However, the extent of this robustness depends on factors such as the network architecture and application [5].

Quantization is usually defined by three parameters, bit-width, scale factor, and zero point (sometimes locked to 0). The bit-width is the width of the target mapping, e.g., 8-bit. This decides how much precision we lose during the quantization. The scale factor determines how much we want to fit in each step in the quantized space. The zero point makes sure that 0 gets quantized without error, making operations such as zero padding not introduce any quantization error. Quantizing a value x to $\hat{x}$ is done utilizing equation 2.3, where s is the scale factor, z is the zero point, and b is the bit width [5].

$$\text{clamp}(a; l, u) = \begin{cases} a & \text{if } l \leq a \leq u \\ l & \text{if } a < l \\ u & \text{if } a > u \end{cases} \tag{2.2}$$

$$\hat{x} = \text{clamp}(\lfloor \frac{x}{s} \rceil + z; 0, 2^b - 1) \tag{2.3}$$

The clamp function in 2.2 returns the value of the first argument only if it is larger than or equal to the second argument and smaller than or equal to the third argument. Otherwise, it either returns the second or third argument depending on which boundary was broken by the first argument. This limits the output range of the first argument.

It is quite common to try to find these three parameters for each tensor and activation in the neural network, allowing good granularity while keeping the computations needed when running the network simple to implement [5].

There are two main ways to quantize (to find the quantization parameters for) neural networks, post training quantization and quantization aware training.

2.2.1.1 Post Training Quantization

Post training quantization is applied to an already trained network without any retraining. This algorithm can be used with either a small amount of calibration data or without any data at all in order to quantize the weights of the network. The benefit of using calibration data is that the quantization error can be lowered. Having no hyper-parameters associated with it, post training quantization works well as a black box optimization step that does not need to be taken into consideration by the neural network designer [5].

2.2.1.2 Quantization Aware Training

Quantization aware training is a method that applies quantization throughout the training of the neural network, fine-tuning the network to better handle the loss in accuracy that quantization introduces. This is especially needed when targeting lower bit-widths. With the drawback of longer training times and the potential need for additional hyper-parameter tuning [5].

2.2.2 Pruning

Pruning is an optimization strategy that tries to make each layer in the neural network smaller by removing insignificant parts of the neural network, making the resulting model smaller and easier to compute. This process consists of three steps: first, the network is run on reference data to identify weights or segments that appear to have minimal impact on the output. Next, these weights or segments are removed from the neural network. Finally, the network is retrained to fine-tune the remaining weights. This can then be repeated in order to prune the model even more. This method has minimal effect on the accuracy of the model if only the insignificant segments of the model are removed [6].

Figure 2.2 illustrates how a neural network is affected by pruning. The reduction in network complexity should have a negligible effect on accuracy if only insignificant segments are removed.

By reducing the size of the model, pruning directly lowers the memory required to store the model and the memory bandwidth required to run the model. This is an important aspect in both mobile and edge deployments of models where these resources can be quite restrictive and energy-demanding [6].

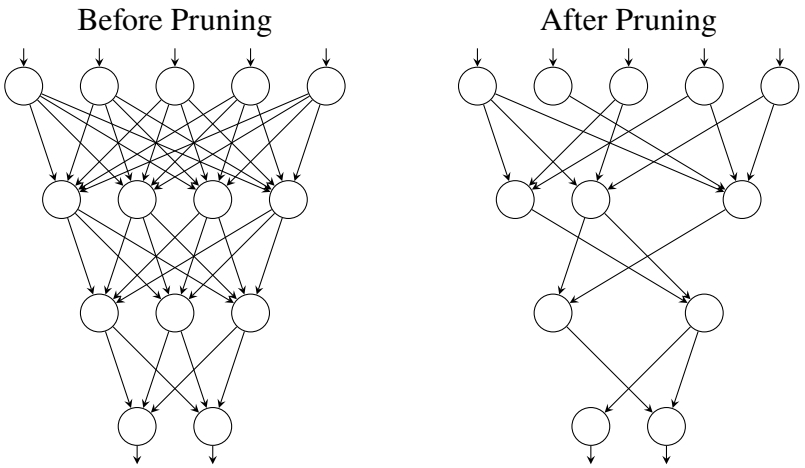


Figure 2.2: The effects of pruning on a neural network. The left graph shows the neural network before pruning, and the right graph shows the neural network after pruning of both synapses and neurons.

2.2.3 Clustering

In a deep neural network, most of the memory needed to be fetched to run a model is composed of the weights of the network. Weight clustering is a way to optimize for this. It works by clustering the unique values of all weights in a deep neural network together into K distinct values and then letting all weights in the deep neural network be represented by an index that can be used to look up the real value of the weight in the K long lookup table. The clustering is typically done using K-means in order to find the appropriate high-precision clustered values [7].

This can greatly reduce the memory bandwidth required to run a model, as the K long lookup table is typically small enough to reside in the cache, and the weight indexes are several times smaller than the actual weights. This does come at the cost of both a small loss in model accuracy and the small cost of looking up the weight values at runtime [7]. But as weight clustering still uses high precision arithmetic for its calculations, it receives a smaller penalty to its accuracy than other methods that use lower precision arithmetic (e.g., using 16-bit floating-points or 8-bit integers) [8].

As an example, the object detection model YOLOv3 [9] consists of about 8GB of weights in the 32-bit floating-point format. Using this model on a video feed of 25 frames per second would put significant pressure on the memory systems as it would require about 200GB/s of memory bandwidth. Hardware

supporting memory bandwidth this high is typically quite expensive and/or draws a lot of power, which makes it inadequate for use in, e.g., self-driving cars or other edge devices. In contrast, if we cluster these weights together in $K = 256$ clusters, we can store all weights in a much smaller table of 256 32-bit floating-point numbers and represent all weights in the network with 8-bit integers for indexing in this lookup table. The lookup table, in this case, would only be about 1kB of data, and the index encoded weights would be about 2GB, effectively decreasing the required bandwidth with a factor of about 4 [7].

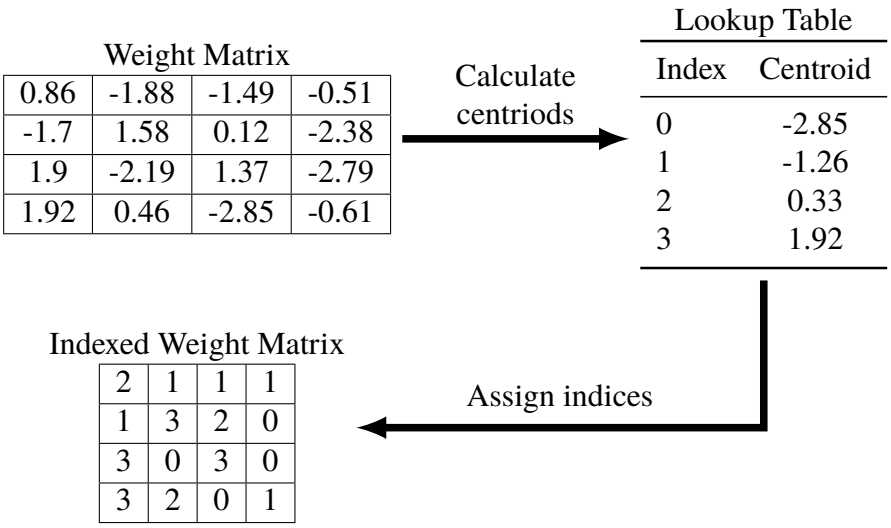


Figure 2.3: Clustering of a weight matrix using four centroids. This allows the weight matrix's values to be encoded into 2-bit indices with the 32-bit weights being stored in the lookup table.

2.3 gem5

The open-source computer systems architecture simulator gem5 is used both in industry and academia and is currently one of the most popular tools for computer architecture research. It is an event-driven functional simulator capable of simulating many different CPU instruction set architectures, e.g., X86, ARM, and RISC-V. Being a functional simulator, it only simulates the functionality of components and not, e.g., performance change with regard to thermal environments. Its modular design allows it to be extended with new components and functionality, allowing new ideas for hardware design to be tested and quickly iterated upon. But it also includes a wide range

of components, e.g., network controllers, multichannel DRAM, and out-of-order processing units. Its capabilities of running an entire unmodified Linux operating system on the simulated hardware, as well as userspace programs with syscall emulation, enables great flexibility in the types of workloads one can run in the simulations [10][11].

For this thesis, gem5 is used to simulate different computer hardware setups. It was chosen because of its permissive open-source license and because of its utilization in academia and industry.

2.4 Tensorflow (TFlite)

Tensorflow is an open-source machine learning framework developed by Google, used to build and train machine learning models, mainly neural networks. It is able to scale from one machine to multiple clustered machines, and across multiple computationally capable devices such as multi-core CPUs, GPUs, and hardware specifically designed for tensor processing. TFlite is a component in the larger Tensorflow framework, designed for running model inference on mobile, embedded, and edge devices in production. It has bindings in several languages like C++, Java, and Python, providing flexibility in its deployment scenarios [12].

For this thesis, TFlite is used as one of the software platforms for machine learning inference. It was chosen because of its widespread adoption and ability to interface with C++.

2.5 ONNX Runtime

ONNX Runtime is an open-source execution platform for machine learning models saved in the ONNX format, it is developed by Microsoft and widely used to power AI inference in their products such as Bing, Azure, Windows, and Office. It is designed to be cross-platform with language bindings in several languages like C++, C#, and JavaScript, and is well suited for several deployment scenarios like cloud, mobile, and web. To accommodate different deployment scenarios and different hardware accelerators, it is designed to utilize different computation backends optimized for, e.g., CPU, CUDA (for Nvidia GPUs), or ROCm (for AMD GPUs) [13].

For this thesis, ONNX is used as one of the software platforms for machine learning inference. It was chosen because of its widespread adoption and ability to interface with C++.

2.6 Related work

This section presents previous research that is foundational to this thesis, as it defines the dataset and model used throughout the work, as well as prior optimization efforts on neural networks. Section 2.6.6 contains a discussion about the related work and how it affects this thesis.

2.6.1 Quantization

There has been a lot of work done on quantization of neural networks, different bit-widths have been explored to find what works from a computational efficiency and an accuracy point of view.

Mazumder *et al.* [14] conducted a survey in 2021 on optimizations on neural network for running inference on edge devices. They report that binary (1-bit) and ternary (2-bit) quantization schemes can provide significant reductions in computational complexity, as arithmetic operations can be expressed as simple binary operations instead, and lowers the memory bandwidth and utilization. However, depending on the model architecture, this can impact the predictive performance of the model negatively. This is especially the case for deep neural networks developed for large-scale datasets such as ImageNet. On the other hand, 16-bit and 8-bit quantization schemes can provide near state-of-the-art performance, with improvements to computational complexity and memory compared to full 32-bit floating point formats.

Chen *et al.* [15] explore in their work how different bit widths used in quantization affect the performance of different models, in order to enable these models to run on weaker edge hardware. Three different models were tested, SA for biomedical image segmentation, VGG-16 [16] for image classification, and Deep Speach for automatic speech recognition. They found that in all cases, quantization was able to improve the model accuracy by 1% to 4%, as well as reduce the memory used by 3.5x to 6.4x. They tested bit widths from 2 to 8-bits, showing that the best model performance was usually achieved with 6 to 7-bits. However, they found that the hardware used (CPU, GPU, FPGA) greatly dictates the gains in inference latency, if there are any at all, depending on what operations are supported by the hardware.

Jacob *et al.* [17] proposed in their publication a quantization scheme that only used integer arithmetic for neural network inference. The advantage of this is that integer arithmetic can be implemented more efficiently than floating-point arithmetic on integer-only hardware that's commonly available.

They evaluated their strategy on some models from the MobileNet model family, already known for their runtime efficiency, running on mobile Snapdragon ARM CPUs. They showed improvements in the tradeoff between latency and predictive performance and a 4x reduction in model size. This quantization scheme has been integrated into TensorFlow Lite (TFlite).

2.6.2 Pruning

There exist several approaches to neural network pruning in the literature, both with regards to lowering the required compute and the storage size.

Han *et al.* [18] utilized a weight-centric approach in their publication, pruning away all weights under a certain magnitude. The weight pruning managed to reduce the number of connections by 9x to 13x in AlexNet [19] and VGG-16 without any reduction in accuracy. Together with a 5-bit quantization scheme and Huffman encoding, they managed to reduce the size of AlexNet and VGG-16 by 35x and 49x respectively. Allowing the whole model to fit into the on-chip SRAM cache rather than off-chip DRAM memory, as well as giving each layer a 3x to 4x speedup and improving the energy efficiency by 3x to 7x.

Liu *et al.* [20] acknowledges the work done by Han *et al.* and concludes that weight-based pruning is capable of greatly reducing the space complexity of neural network models. However, in order to gain any advantages on the computational side, specialized hardware is required for the sparse matrix operations needed for inference. They instead highlight two other approaches to pruning, neuron, and channel pruning, which are a bit more restrictive but do not require any specialized hardware in order to speed up inference. Neuron pruning removes all weights associated with a neuron, allowing the weight matrix to become smaller and dense instead of being the same size and sparse. Channel pruning is similar to neuron pruning, but tweaked to for convolutional neural networks. Channel pruning removes entire filters/channels from convolution layers, allowing for fewer computations during inference.

Hu *et al.* [21] utilized a method of looking at the activations from each neuron in a model over a sample dataset and removed all neurons with outputs that were mostly zero. By this method, they managed to reduce the number of parameters of the VGG-16 model by 2.59x. This can be compared to the work done by Han *et al.* [18], where much greater reductions were achieved. However, as the work by Han *et al.* only introduces sparsity, the number of floating point operations remains the same when dense matrix operations are used. Hu *et al.* managed to reduce the number of floating point operations by

more than 2x.

Jeong *et al.* [22] demonstrates in their publication a channel-based approach to pruning where a channel is pruned depending in the magnitude of the corresponding weights. Doing this iteratively, they managed to reduce the VGG-16 by 53x with only a 7% drop in accuracy. This in turn yielded a 4x speedup in inference time on a basic CPU.

Similar to Jeong *et al.*, He *et al.* [23] explored different means of channel selection when performing channel pruning. They compared three methods of channel selection, first k (selecting the first k channels), max response (selecting the channels based on the absolute sum of weights), and a LASSO regression-based method that they proposed. Their findings show that their proposed method performed better than any of the other methods at the same speed-up ratios (directly correlated with the amount of pruned channels in this case).

The max response method, tested in the publication by Jeong *et al.*, for selecting which channels to prune originates from a publication by Li *et al.* [24]. Their proposed method is to select channels based on the absolute sum of the channels' kernel weights. The channels with the smallest absolute sum are pruned and removed from the network together with the kernels in the following layer corresponding to the pruned feature maps. This allows the network to stay dense and utilize existing efficient library implementations for dense matrix multiplications. In the publication, they show a 34% reduction in computational cost for the VGG-16 Convolutional Neural Network (CNN) and a 38% reduction for ResNet-110 [25] on the CIFAR10 dataset [26].

2.6.3 Clustering

There exists several studies on the practical usages of clustering in order to bring down the bandwidth requirements of a model, and in turn lower the energy requirements for running said model.

Caro *et al.* [7] explored using weight clustering together with accelerators such as systolic arrays in order to reduce the bandwidth and energy requirements of the YOLOv3 object detection model for usage in autonomous driving applications. Several bit-widths for the encoding were tested, with the lowest being 5-bits (32 unique values), and each layer where independently clustered. They measured similar prediction performance to the regular YOLOv3 model (using 32-bit floating-point numbers as weights) to the clustered ones. However, the optimized model only required 30% to 40% of the original bandwidth requirements, as well as bringing the energy

requirement down to 45%.

Pitsis *et al.* [27] looked into using a Field-programmable Gate Array (FPGA) implementation of a CNN model in order to enable the usage of the technology on satellite platforms where bandwidth and energy are restricted resources. To further optimize the bandwidth requirements they explored clustering for weight compression. They measured a reduction in model size from 174 to 11 MB using clustering, enabling parallel on-chip inference execution. Their experiments showed very competitive results against GPU-based platforms regarding both throughput and latency, with a five-fold reduction in energy consumption.

Caro and Abella [28] looked into optimizing the energy and memory bandwidth usage using a combination of weight clustering and reduced precision arithmetic (quantization). In order to test these optimizations on a real-world model used in a wide variety of industries, they used the YOLOv3 model. They noted that bandwidth and energy consumption decreases rapidly while the model accuracy degrades slowly as fewer bits are used for the clustered weights. For weight clustering, they tried using bit widths from 8 to 5 for the weight encoding. For arithmetic precision, they tried 16-bit floating-point and integer representations (compared to the original 32-bit floating-point weights). Using only weight clustering, they observed a 61.5% to 68.5% decrease in memory bandwidth with clustering applied, and a 70.5% to 7.5% with both clustering and reduced arithmetic operations. Similar figures were also observed for the energy consumption of the model inference.

2.6.4 Deepsense 6G

New applications for machine learning in the communications industry are steadily emerging, particularly in the context of multi-modal sensing. However, development and evaluation of these technologies require an appropriately large dataset containing real-world measurements that are synchronized across different modalities and the corresponding communications. The Deepsense 6G dataset by Alkhateeb *et al.* [29] is a real-world dataset with intentions to tackle this issue. It is constructed from several real-world measurements of multiple sensory mediums in connection to data communication tasks.

The dataset is structured into scenarios, where each scenario is a collection of multi-modal sensing and communication information. The different modals used in the scenarios can range from, e.g., wireless receivers, radar, LiDAR, cameras, and GPS receivers, and all sensory and communications data in all

scenarios are synchronized in their sampling rate.

A scenario that is of special interest for this thesis is the ”Radar-Aided Blockage Prediction” scenario (scenario 30 with radar as the medium of sensing). In this scenario, there is a communications receiver and a transceiver with an accompanying radar. The task is to predict future LOS blockages between the receiver and the transceiver using current and past radar samples. Each time-step (100ms) contains a cube of radar measurements, as well as information about whether or not the LOS is currently blocked. This scenario then consists of three sub-problems, predict if the LOS is blocked within the period of one, five, or ten time-steps (100ms, 500ms, or 1s). Figure 2.4 shows a graphical overview of the scenario.

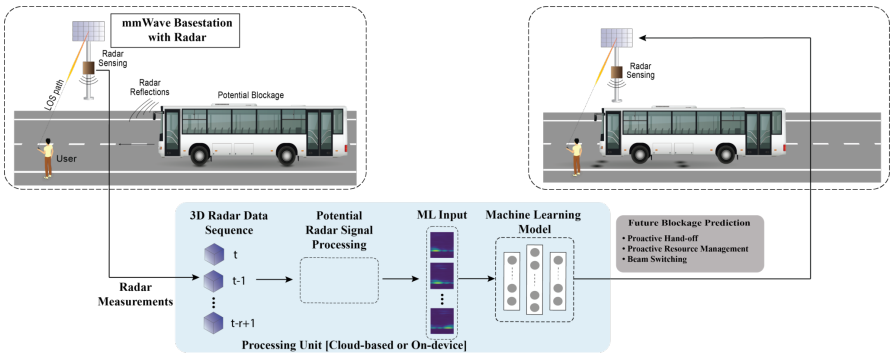


Figure 2.4: A graphical overview of the ”Radar-Aided Blockage Prediction” scenario [30]. The image is licensed under Creative Commons BY-NC-SA 4.0 [31].

Several other studies have been conducted using the Deepsense 6G dataset and its library of different scenarios and sensing mediums. One such study was conducted by Al-Quraan *et al.* [32]. Their study used the data from scenario 30 with radar as the sensing medium (radar-aided blockage prediction) to train a model to predict if a blockage takes place and when it takes place using a federated learning approach. They compared this method of predicting blockages to drive proactive handover to existing reactive solutions. The model achieved a 94% success rate in proactive handover compared to the existing solution which lacked blockage prediction entirely.

Another study by Wu *et al.* [2] looked at using the LiDAR data from the same scenario to predict LOS blockages. They achieved 95% accuracy in predicting blockages within 100ms and 80% accuracy when predicting 1s into the future. This was done by developing an efficient LiDAR data denoising

method (static cluster removal) and training a neural network on the denoised data.

Zhou *et al.* [33] looked at another scenario contained in the dataset, namely beam prediction using radar data for moving vehicles. They proposed a federated learning approach to preserve users' privacy concerning location. Their solution achieved an improved beam selection accuracy of 11.9-33.2% compared to baseline schemes.

Another study that is foundational for this thesis is the publication by Demirhan and Alkhateeb [1]; this publication is explored more in detail in section 2.6.5.

2.6.5 Radar Aided Blockage Prediction

The publication by Demirhan and Alkhateeb [1] proposes the use of radar sensing in order to predict future blockages of LOS in high-frequency baseband systems. In the publication, they used the DeepSense 6G dataset for training and testing. To solve the scenario, they utilized a deep neural network architecture consisting of a feature extracting section composed of a CNN, that then fed into an Long Short-Term Memory (LSTM) layer, that lastly fed into a densely connected neural network that gave the prediction. This architecture is illustrated in figure 2.5. The input data was first reprocessed using fast Fourier transforms in order to create what is called radar maps.

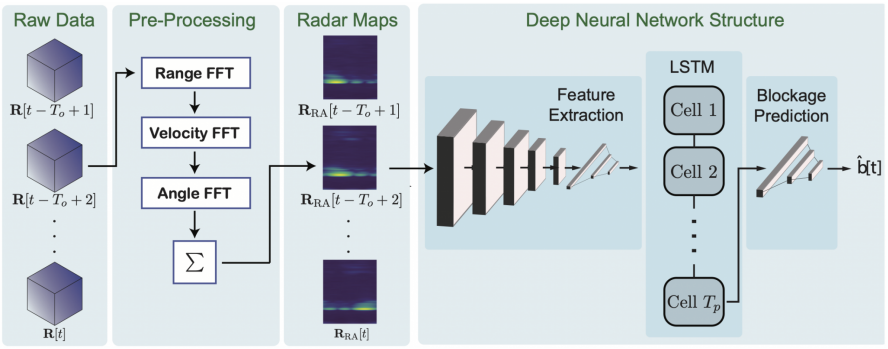


Figure 2.5: A graphical overview of the architecture proposed in the publication by Demirhan and Alkhateeb [1]. The image is licensed under Creative Commons BY-NC-SA 4.0 [31]

Demirhan and Alkhateeb reports promising performance, yielding a prediction accuracy of 92-97%, whilst maintaining a low overall complexity with only 184,015 trainable parameters. The model was able to predict

blockages within a window of 1 second into the future with an F1 score and accuracy above 90%. According to the publication, this shows promising results in using low-cost radar systems in order to predict blockages and increase the reliability of millimeter wave systems. But if the prediction window is changed to 100ms into the future, the solution struggles somewhat with the F1 score, only yielding an F1 score of above 72%, but at the same time keeping the accuracy well above 95%. The publication explains this to be because of the poor angular resolution of the radar.

2.6.6 Discussion

In [1] the Demirhan and Alkhateeb proposes a model for blockage prediction utilizing radar data based on the real-world data collected in [29]. Some pre-processing is done before the data is fed into the model, consisting of fast Fourier transforms and summation. This is a reasonable thing to do as current baseband systems most often have dedicated hardware available for these kinds of signal processing operations.

Surprisingly, in [1], the F1 scores between 100ms and 1s future windows differ quite a lot. However as this thesis is more concerned with the computational cost of the approach, and the same model architecture is used for all window sizes, the predictive performance is not a primary concern for the research in this thesis. The performance numbers are only important when checking if the applied optimizations impact the model's performance. The results are either way promising and motivate further investigations into the topic, such as this thesis work.

The model presented in [1] and its low overall complexity should also prove valuable in a real-world application scenario. A smaller model, by nature, is easier to run than a larger model, particularly when hardware is restricted, as in the baseband systems scenario.

All sources in sections 2.6.1, 2.6.2, and 2.6.3 show that quantization, pruning, and clustering can be valid methods of accelerating neural network performance as well as reducing system requirements. Some of the sources, e.g., [7], [28], and [14], show improvements in memory or bandwidth usage after applying optimizations. However, few have observed CPU usage more than the measured inference time or energy consumption, making this thesis an interesting addition to previous research as it provides more detailed information.

Chapter 3

Method

The purpose of this chapter is to provide an overview of the research method used in this thesis. Section 3.1 describes the research process. Section 3.2 focuses on the data collection techniques used for this research. Section 3.3 describes the experimental design. Section 3.4 assesses the reliability and validity of the method. Finally, Section 3.5 describes the framework selected to evaluate the stated research question.

3.1 Research Process

The following steps were taken in order to answer the research question:

Step 1 plan experiment,

Step 2 model optimizations,

Step 3 develop inference platforms,

Step 4 conduct experiment,

Step 5 analyze data from the experiment, and

Step 6 discuss the results of the analysis.

3.2 Data Collection

All data used in this thesis is collected from the DeepSense 6G project, which in turn set up a real-world isolated scenario using their own hardware. No person, integrity, or privacy has been affected by this collection of data. This also poses minimal environmental impact. No ethical concerns can

thus be raised from the data collection. The dataset is further described in section 3.3.2.

3.3 Experimental design and Planned Measurements

The experiment consists of several parts, the hardware simulator, the inference platforms based on the ONNX and TFlite (from TensorFlow) inference engines, the model optimizer based on tooling from the Tensorflow ecosystem, the model, and the dataset. Both the inference platforms and the model optimizer are developed as part of this thesis.

The model optimizer creates several versions of the model with different combinations of optimizations. These different models are then run on each inference platform over the test data available in the dataset, outputting information about accuracy, F1 score, and total inference time.

Each run either takes place in the gem5 hardware simulator or directly on real-world hardware, both described in section 3.3.1.1. To achieve more accurate results when running directly on the server, each combination of model and platform was run 100 times; then, the average inference time was taken. Due to the effects of uncertainty, such as thermal and processor boosting, not being a factor in the simulations, and due to the available time, each simulation is only run once.

Several metrics were measured during the experiments: accuracy, F1 score, total inference time, total number of CPU instructions, number of floating point instructions, number of integer instructions, number of vector instructions, Instructions Per Cycle (IPC), and memory bandwidth. Accuracy and F1 score are used to make sure that the models' predictive performance is not compromised by either the applied optimizations or the inference platform, compared to the original publication. The other metrics are for looking at how the model optimizations scale on the hardware itself.

3.3.1 Test Environment

This section describes the hardware and software used to create the test environment.

3.3.1.1 Hardware

All simulations and real-world experiments are conducted on a server running two Intel Xeon E5-2680 v2 processors with 264GB of RAM. For testing without simulation, this same server is used.

The simulator is configured to simulate only a single processing core running at 3GHz, with a cache hierarchy consisting of a 64kB L1 (32kB for data and instructions respectively) and a 256kB L2 cache, and 2GB of DDR4 dual channel RAM. This hardware setup was deemed reasonable to reflect the constraints imposed by real-world baseband hardware.

The simulated CPU is configured, depending on the scenario, to use either the X86 or ARM instruction sets. To achieve good simulation speed and simulation accuracy, we are using a technique of switching between the very detailed (but slow) O3 CPU and the simpler (but faster) Atomic CPU, as proposed in [34].

3.3.1.2 Software

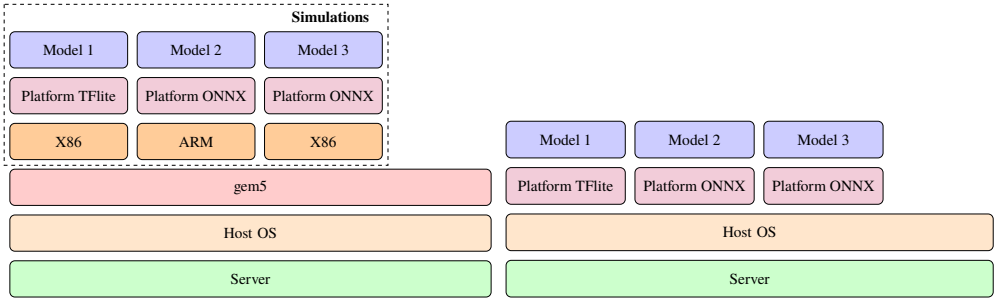
All simulations were run in the gem5 hardware simulator in ”Syscall Emulation” mode, using the latest version (as of testing) from the development branch. All testing outside of the simulations was done on Ubuntu 20.04.6 LTS.

Specific versions of relevant pieces of the software stack are:

- gem5 (commit 5641c5e464)
- Tensorflow (commit 017edc69aee)
- ONNX Runtime (v1.17.0, commit 5f0b62cde5)
- GCC 11.4.0
- Python 3.10

The inference platforms were built in C++ with GCC, Tensorflow (TFlite), and the ONNX Runtime. These are the programs used in the experiments. The model optimization pipeline is written in Python, utilizing several libraries for machine learning and neural network optimization. The most notable libraries are:

- tensorflow 2.15.0
- keras 2.15.0
- tensorflow-model-optimization 0.7.5



(a) Simulated Hardware (b) Real Hardware

Figure 3.1: Software stacks, Simulated versus Real Hardware

- tf2onnx 1.16.1

Figure 3.1 shows the software stacks involved in the experiment. Stack 3.1a shows how the simulations using gem5 affect the software stack; each inference platform was run on top of the simulated hardware running the X86 or ARM CPU architectures provided by gem5. Stack 3.1b shows each inference platform running directly on the server without any layers of simulations.

3.3.2 Dataset

The dataset used in this thesis is the one provided by Alkhateeb *et al.* [29] for line-of-sight blockage prediction between a baseband transceiver and receiver using radar (scenario 30). The dataset consists of input and output pairs sequentially structured in a time series. However, the predefined train, test, and validation sets of the dataset contain shuffled data points. Each radar sample is saved in separate MATLAB matrix (.mat) files and each label is saved in separate text files (.txt) containing a 0 if there was no blockage or 1 if there was a blockage. Each input consists of 8 consecutive radar samples, sampled at a frequency of 10Hz (100ms between each sample). A radar sample consists of a 3D radar cube of measurements, reprecented by a three dimentional vector containing 32-bit floating-point numbers. Each radar cube is of the shape 4 (number of RX antennas) by 256 (number of samples per chirp) by 128 (number of chirps per frame). Each output consists of 10 boolean values, indicating whether or not there was a blockage during that 100ms window. To predict a blockage within K steps (K times 100ms) into the future, the target is defined as whether any of the first K elements in the output vector are *true*. Thus, for a prediction of 100ms into the future, only the first element is

considered; for a prediction of 200ms, the first two elements are considered, and so on. A graphical overview can be seen in fig. 3.2.

The dataset consists only of data from real-world measurements. The setup used a receiver on one side of a two-way street, together with a radar sensor and a transceiver on the other side of the street. Samples were continuously collected with cars of differing sizes, traveling at different speeds (the speed limit being 25mph or 40.6km/h). This results in the diverse pool of LOS blockages contained in the dataset. The dataset is perfect in the sense that there are no missing values. Some processing has already been done, syncing the radar sensor updates with measurements of LOS from the receiver.

In this thesis, we mostly used a 100ms prediction window and predictive performance does not matter much in our evaluation, only the relative changes in performance. However, we do look at prediction windows up to 1000ms to see how the software optimizations affect the predictive performance over several scenarios.

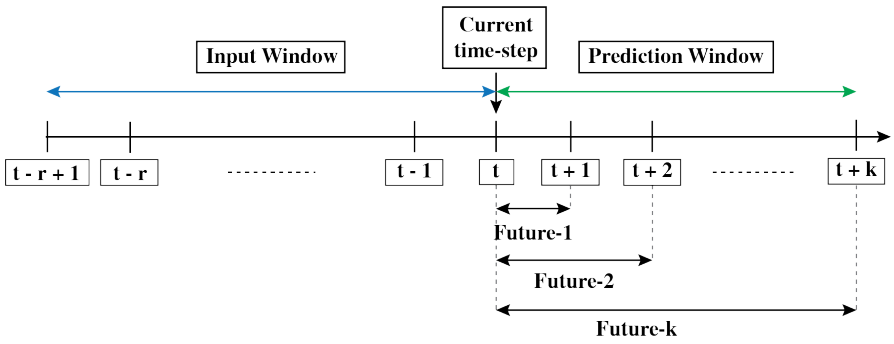


Figure 3.2: A graphical overview of an input/output sample in the Deepsence 6G dataset (scenario 30) [30]. The input window consists of $r = 8$ radar samples and the prediction window consists of k labels. Each step timestep t equals 100ms of time. The image is licensed under Creative Commons BY-NC-SA 4.0 [31]

A preprocessing pipeline was implemented to ensure dataset compatibility with the model architecture, mirroring the approach detailed in the publication by Demirhan and Alkhateeb [1]. This pipeline leveraged fast Fourier transforms to derive a three-dimensional radar cube, encapsulating range, angle, and velocity information from the raw radar samples. The radar cubes were further reduced by summing across all chirp samples, resulting in two-dimensional range-angle maps. These range-angle maps constituted the input feature space for the blockage detection model.

3.3.3 Model

The model that is used as the base for all experiments is the same model as the one proposed in the publication by Demirhan and Alkhateeb [1]. It consists of $183,023$ trainable parameters, and the structure is presented in table 3.1.

Table 3.1: Blockage prediction model structure. The model is fed sequentially from the top down.

Model Section	Neural Network Layer	Specification
Feature Extraction	Input	Size: $1 \times 256 \times 64$
	Convolution	Output Channels: 4 Kernel: (3, 3) Activation: ReLU
	Convolution	Output Channels: 8 Kernel: (3, 3) Activation: ReLU
	Convolution	Output Channels: 16 Kernel: (3, 3) Activation: ReLU
	Average Pool	Kernel: (2, 2)
	Convolution	Output Channels: 4 Kernel: (3, 3) Activation: ReLU
	Average Pool	Kernel: (2, 2)
	Convolution	Output Channels: 2 Kernel: (3, 3) Activation: ReLU
	Average Pool	Kernel: (2, 2)
	Fully Connected	Input Size: 512 Output Size: 256 Activation: ReLU
	Fully Connected	Input Size: 256 Output Size: 64 Activation: ReLU
Temporal Memory	LSTM	Units: 64 Activation: Tanh Recurrent Activation: Sigmoid
Blockage Prediction	Fully Connected	Input Size: 64 Output Size: 1 Activation: Sigmoid
	Output	Size: 1

During inference, the layers are fed sequentially, from top to bottom in the order presented in table 3.1. The first input layer takes in a preprocessed radar cube and the last layer outputs a binary prediction about whether or not there will be a blockage within the next K milliseconds (where K depends on the structuring of the training data). As per the publication by Demirhan and Alkhateeb [1], the network is fed an 8 long sequence of preprocessed radar samples that is then processed by the *feature extraction* and the *temporal memory* section of the model before sending the last produced vector from the *temporal memory* to the *blockage prediction* section for the final prediction. Internally this is done by wrapping each layer in the *feature extraction* section in the TimeDistributed layer from Keras (part of TensorFlow), allowing the user to directly feed 8 preprocessed radar samples into the model and get one prediction.

3.4 Assessing reliability and validity

The method's validity relies heavily on the validity of the gem5 hardware simulator. gem5 is widely used in academia and industry for researching and developing hardware; thus, we can assume that it is a valid approach for the functional simulations we want to do in this thesis. The results are also cross-checked with tests running on real-world server hardware, ensuring we have gotten the correct results.

Regarding switching CPU types during the simulation between the more detailed O3 type and the simpler Atomic type. As shown in previous research by Sandberg *et al.* [34], this way of sampling produces results that are within a small margin of error compared to a fully detailed simulation, making this a valid approach to speed up the simulation time.

The reliability of the method is also highly dependent on the gem5 simulator. As gem5 is a functional simulator, as in it only simulates the functionalities of hardware and not other behavior (e.g., throttling or boosting due to temperature), subsequent simulations of the same setup will produce the same results.

3.5 Evaluation framework

For evaluation, we are comparing the results from each combination of optimizations (10 with a single optimization applied and 4 with stacked optimizations) to the base model without optimizations on each platform

(2) on each hardware architecture (2). This creates 60 different scenarios (15 different models, on 2 software platforms, on 2 hardware platforms) that are compared against one another. The comparison is on a per kind of measurement basis, allowing us to analyze and evaluate how hardware resource requirements scale with each optimization. We are also comparing the accuracy and F1 score with the original publication to evaluate any performance degradation due to any optimizations or inference platforms.

We utilize bootstrap sampling of the original test dataset and pairwise t-tests to evaluate the statistical significance. Bootstrapping is done by creating 100 sampled sets by uniformly sampling the original dataset 977 times per set. T-tests are then performed on the optimized models' accuracy and F1-score against the baseline model over all sampled datasets. For the significance test, the null hypothesis is that there is no significant difference between the optimized model's and the baseline model's prediction performance. We are looking at 95% significance using a p-value of 0.05.

Chapter 4

Development

Several pieces of software were needed to be developed to make the experiments in this thesis possible. In this chapter, we lay out the most major development efforts and explain how these pieces work together to allow us to run and analyze the results from the simulation experiments. A graphical overview of how each piece depends on each other can be seen in fig. 4.1.

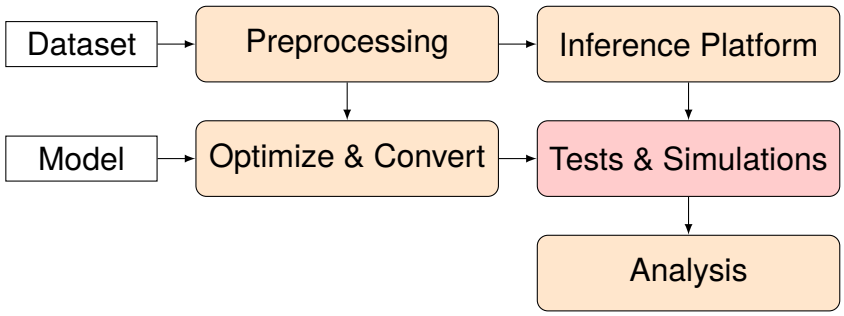


Figure 4.1: Graphical overview of the experiment process

4.1 Data Preprocessing

For the model to be able to interpret the raw radar data from the dataset, each data point needed to be preprocessed the same way as in the publication by Demirhan and Alkhateeb [1], using fast Fourier transforms. For this, we reuse the code from that publication to apply the same preprocessing to the test dataset. This process is implemented in Python; however, to interface with the C++ programs, the processed data must be written to a file in a format that can be easily interpreted by the C++ code. For this purpose, a custom binary

format was chosen.

The structure of the data file is as follows; first, there is a header consisting of three 32-bit integers, denoting the number of data points in the file, the size of the input vector, and the size of the ground truth vector. After that, each input and ground truth vector is laid out one after the other with each value represented in 32-bit floating-point format, for each data point in the test dataset. This is visualized in fig. 4.2

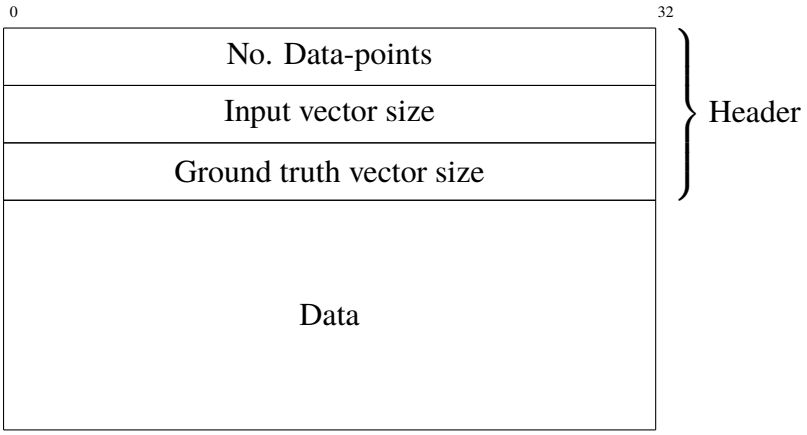


Figure 4.2: Binary structure of dataset dump file.

4.2 Model Optimization and Conversion

In order to perform the desired optimizations on the model, pruning, clustering, and quantization, we used the tensorflow-model-optimization Python library as well as the capabilities of tflite and onnx. To ensure that applying one optimization after another does not undermine the efforts of the previous one, the order in which the optimizations were applied always followed: pruning, clustering, and then quantization. This is, as explained in the documentation for tensorflow-model-optimization, to make sure that applying an optimization does not undo or undermine the work of the previously applied optimization. In order to allow for as many platform-specific optimizations as possible, we decided to quantize the models using the tooling provided by tflite and onnx. While pruning and clustering were performed with tensorflow-model-optimization. As each optimization requires some amount of data to calibrate/fit to, we used the training data available in the dataset (preprocessed) to calibrate/train the model for one epoch for each optimization.

However, for tensorflow-model-optimization to support the models architecture, it was necessary to implement some patches for the library. This was because the model used a wrapping layer from the Keras (part of TensorFlow) ecosystem called *TimeDistributedLayer*. To accomplish this, it was necessary to examine the source code of tensorflow-model-optimization and develop Python code that modifies the library's functionality at runtime to add support for this layer, subsequently enabling the optimization of the model. Most of the changes made were to add appropriate registry entries and make sure that the wrapping layer exposed the underlying layers' weights properly for what the optimization API expected.

As the tensorflow-model-optimization library only supports unstructured pruning, we also implemented support for structured pruning in this thesis work and injected it into the library in order to utilize this optimization strategy.

To test each optimization and its effects, we first tested the model with one optimization applied with several parameter values for each optimization. Then we picked out the most promising parameters and tested how combining the optimizations affected the model performance.

Pruning was tested for both unstructured and structured pruning with the target sparsity set to 25%, 50%, and 75%. Clustering was tested with 4, 8, and 12 clusters (unique values). Quantization was only tested with 8-bit integer quantization as this was the only format that both onnx and tflite supported. One baseline model with no optimization applied was tested as well.

Models for all possible combinations of pruning, clustering, and quantization, as well as a non-optimized model, were generated with the most promising parameters from the single optimization results with only one optimization applied.

Prior to running the models in the test environment, each model was exported and converted to the appropriate formats for tflite and onnx to ensure compatibility with the testing process. This included running some final platform-specific optimizations from tflite and onnx respectively and saving the models in a format specific to each inference platform.

How each optimization is implemented is described in section 4.2.1, section 4.2.2, and section 4.2.3.

4.2.1 Quantization

Both tflite and onnx provide their own 8-bit quantization implementations, using the same general quantization algorithm but some implementation details differ. Both algorithms are presented in section 4.2.1.1 and

section 4.2.1.2.

4.2.1.1 Quantization TFlite

The quantization algorithm from tflite consists of two main steps. First, they feed the model a calibration data set and record the minimum and maximum intermediate values between all operations. Then they iterate through the model, layer by layer, and operation by operation, constructing the new quantized tensors and scaling operations. In general, the weight tensors are quantized channel-wise and symmetrically. However, some layers, such as LSTM, are quantized asymmetrically using the same scale and zero point for both the output and the hidden input state tensor.

4.2.1.2 Quantization ONNX

The quantization algorithm from onnx consists of two main steps. First, they feed the model a calibration data set and record the minimum and maximum intermediate values between all operations. Then they iterate through the model, tensor by tensor, constructing the new quantized tensors and scaling operations. Weight tensors are quantized symmetrically while activations are quantized asymmetrically.

4.2.2 Pruning

The tensorflow-model-optimization library provided the basic algorithm for unstructured pruning. To test the effects of structured pruning, we extended the pruning algorithm provided to support structured pruning based on the max response channel selection method described in [24]. The extended algorithm was then back into the tensorflow-model-optimization library to be used in the optimization workflow provided by the library. Both algorithms are presented in section 4.2.2.1 and section 4.2.2.2.

4.2.2.1 Unstructured Pruning

The algorithm for unstructured pruning works by training/finetuning an already trained model to remove the least significant weights in the network. This is done in an unstructured way, which means that the model structure is not changed, and the selection of weights to remove is not dependent on the model structure. Removing a weight is done by setting it to zero, removing its contribution to the model output. The unstructured pruning algorithm works

by applying the following steps for each weight tensor T in a model, with the target sparsity of S being a percentage between 0% and 100%:

1. Find the weight values L_T in tensor T which absolute magnitude is among the lowest $S\%$.
2. Substitute each weight value in L_T with zero.
3. Run a training epoch.
4. Repeat from step 2 until the training-process is done.

4.2.2.2 Structured Pruning

The algorithm for structured pruning works by training/finetuning an already trained model to remove the least significant weights in the network in a structured way such that sections of the model can be removed afterward. A structure in the model could be, e.g., a node in a dense layer or a channel in a convolutional layer. Removing a weight during training is done by setting it to zero, removing its contribution to the model output. After training, the sections of the model containing only zeros are removed, reducing the size and complexity of the model. The structured pruning algorithm works by applying the following steps for each weight tensor T in a model, with the target sparsity of S being a percentage between 0% and 100%:

1. Find the weight values L_T in tensor T which are part of the structures where the sum of each weight's absolute magnitude is among the lowest $S\%$.
2. Find the weight values corresponding to the structures L_T in tensor T whose cumulated magnitude is among the lowest S.
3. Substitute each weight value in L_T with zero.
4. Run a training epoch.
5. Repeat from step 2 until the training process is done.
6. Iterate through the model layer by layer, removing all structures consisting only of zeros and all connections depending on those structures.

4.2.3 Clustering

The tensorflow-model-optimization library provided the clustering algorithm used in this thesis. This algorithm works by training/finetuning an already trained model to reduce the number of unique weights in each tensor. It takes a number N as an argument, representing the number of unique values each weight tensor is allowed to have. The clustering algorithm then works by applying the following steps for each weight tensor T in a model:

1. Fit N centroids C_T using k-means to the distribution of values in T.
2. Substitute each weight value in T with the closest centroid value in C_T.
3. Run a training step.
4. Take the average derivative for each centroid in C_T.
5. Update the centroids in C_T in the direction of their corresponding reduced derivatives according to the learning rate.
6. Repeat from step 3 until the training process is done.

This results in each tensor being able to be expressed using only N unique values.

4.3 Inference Platforms

Two inference platforms were developed, one built on top of tflite and one on the onnx runtime. As these platforms are the programs responsible for the testing of the models inside of the simulated environment, some special restrictions apply. The programs needed to be statically compiled in order to ensure minimal simulation overhead and maximum comparability regardless of hardware platform, they needed to include the testing data already preprocessed in the executable to prevent additional overhead from reading all data from files, and they needed to be able to interface with both tflite and onnx. For this, we chose to write the programs in C++ as it provides all features needed, is low overhead, and both tflite and onnx had language bindings for C++. Languages like Python would have been many people's first choice when interfacing with these libraries. However, the Python interpreter is dynamically linked and is too resource-intensive and complex to operate efficiently in the simulated environment, rendering it unsuitable for

the purposes of this thesis. tflite does support static linking out of the box, but to statically link onnx we needed to set up a special build process in CMake that statically linked onnx and all its dependencies. The data dump file from section 4.1 was linked into the binary during compilation.

The inference platforms follow a simple program structure: initialize the underlying inference library, load a model, prepare input and output buffers, and then, for each data point in the test dataset (977 entries), record the time taken and the result of each inference. After all inferences, the accuracy and F1 score of the model are calculated and printed out together with the total inference time. A pseudocode overview of how both platforms are implemented can be found in listing 4.1 to illustrate the overall structure of the programs.

Listing 4.1: Pseudocode for inference platforms

```
input: string ModelPath
output: None
begin
  Lib ← inference library
  Lib.initialize()
  Model ← Lib.loadModel(ModelPath)
  Res ← empty list
  StartTime ← getCPUTime()
  foreach X, Y ← embedded dataset
    YPred ← Lib.runInference(Model, X)
    Res ← append(Res, (YPred, Y))
  end
  Duration ← getCPUTime() - StartTime
  Accuracy ← calcAccuracy(Res)
  F1Score ← calcF1Score(Res)
  print(Duration, Accuracy, F1Score)
end
```

4.4 Analysis

Each time a simulation is run in gem5, a statistics file with technical information about the simulation is created. It is structured in key-value pairs, with each simulated hardware component being able to add entries with information about their execution. To efficiently collect and compare data points between simulations we wrote a Python program that scans through all statistics files from all simulations, looking for a given key pattern, and presents the results in a structured table. To group results from multiple matching keys, we aggregate the values by summation or taking the average. This allows us to assess and compare data quickly. As an example, looking for the key "*committedInstType::Simd*" would match

”*board.processor.atomic.core.commitStats0.committedInstType::SimdAdd*” and ”*board.processor.o3.core.commitStats0.committedInstType::SimdFloatMult*”, among others. Summing up all these values gives us the total number of committed SIMD (vector) instructions for all simulations. And if we only want to look at the floating-point-based SIMD instructions we look for the key ”*committedInstType::SimdFloat*”. This allows us to build all the tables needed to evaluate each test result.

Chapter 5

Results and Analysis

There are multiple parts to the results of this thesis work as we are looking at several aspects of model execution that different neural network optimization strategies could impact. Because of this, this chapter is divided into two sections, one for models with one optimization applied (5.1) and one for models with multiple optimizations applied (5.2), with three subsections: prediction performance (5.1.1 and 5.2.1), inference time (5.1.2 and 5.2.2), and hardware utilization (5.1.3 and 5.2.3).

All tables in this chapter show gathered results from the simulations and tests, and all follow the same format. The tables are structured such that each hardware platform (ARM and X86) is denoted by a header, and each software platform (tflite and onnx) by a sub-header. Each variant of the model follows the notation of ”*rabp_*”, with each applied optimization represented by a suffix of ”*pu*” for unstructured pruning, ”*ps*” for structured pruning, ”*c*” for clustering, and ”*q*” for quantization, followed by the optimization parameter value if applicable. The ”*rabp_*” model is the baseline model without any optimizations applied. E.g., a model with unstructured pruning targeting 75% sparsity and clustering using 8 clusters applied would be noted ”*rabp_pu75c8*”.

A value of ”*nan*” in the table indicates that this case did not run successfully. In all tables with results from simulations, we can see that no test cases on X86 running with onnx managed to complete. All of these cases started and ran without issue, but got stuck in model initialization and never completed in the simulations, and thus failed to deliver any results.

Unless specified, all tables are based on a prediction window of 100ms into the future. This does not matter when looking at inference time (5.1.2 and 5.2.2) or hardware utilization (5.1.3 and 5.2.3), but is worth keeping in

mind when looking at the prediction performance sections (5.1.1 and 5.2.1).

5.1 Single optimization

This section shows the results of applying one optimization with different parameters to the baseline model. Section 5.1.1 shows how the prediction performance was affected, section 5.1.2 shows how the inference time was affected, and section 5.1.3 shows how the hardware utilization was affected.

5.1.1 Prediction Performance

Prediction performance regards the models' accuracy and F1 score, indicating how well they perform before and after being optimized.

Table 5.1: Accuracy for 100ms prediction window

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	0.967	0.968	0.967	nan
rabp_c12	0.965	0.966	0.965	nan
rabp_c4	0.967	**0.971**	0.967	nan
rabp_c8	0.963	0.966	0.963	nan
rabp_ps25	0.947	0.954	0.947	nan
rabp_ps50	0.943	0.943	0.943	nan
rabp_ps75	0.964	0.967	0.964	nan
rabp_pu25	0.967	0.964	0.968	nan
rabp_pu50	0.954	0.961	0.954	nan
rabp_pu75	**0.970**	0.970	**0.969**	nan
rabp_q	0.966	0.959	0.966	nan

The accuracy for each model can be seen in table 5.1, which shows that the accuracy of the model's predictions is not greatly affected by any optimization technique, regardless of the hardware or software platform. In the case of tflite, it also shows that model performance barely differs between the two hardware platforms.

Table 5.2: F1 score for 100ms prediction window

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	0.709	0.726	0.709	nan
rabp_c12	0.696	0.703	0.696	nan
rabp_c4	0.652	0.731	0.652	nan
rabp_c8	0.684	0.723	0.684	nan
rabp_ps25	0.133	0.526	0.133	nan
rabp_ps50	0.000	0.000	0.000	nan
rabp_ps75	0.673	0.692	0.673	nan
rabp_pu25	0.660	0.690	0.667	nan
rabp_pu50	0.384	0.548	0.384	nan
rabp_pu75	**0.729**	**0.734**	0.717	nan
rabp_q	0.723	0.692	**0.723**	nan

Regarding the models' F1 score, table 5.2 shows greater differences between the baseline model and the optimized ones. The results also highlight differences between the inference platforms, with tflite generally performing slightly worse than onnx for most models, while *rabp_ps25* and *rabp_pu50* exhibit significantly lower performance. But on the other hand, *rabp_q* on onnx is lagging behind the results tflite by about 0.03 points. The last thing to note is the base model's performance difference, with tflite performing 0.015 worse than onnx.

All data points thus far have only been taken from the scenario with a 100ms prediction window. The model structure does not change between the different windows of prediction, but in order to see how the model's predictive performance is affected at all possible prediction window sizes we look at figures 5.1, 5.2, 5.3, and 5.4.

Looking at fig. 5.1 we see that the general trend follows the one of the base model regardless of the number of centroids used for each layer, sometimes the clustered models event outperforms the base model with regards to both accuracy and F1 score. However, we can also observe some substantial fluctuations in the performance, with the f1 score dropping down to zero at certain window sizes. This behavior was unexpected and is discussed more in chapter 6. There also seems to be a difference between tflite and onnx where the models' performance is more stable when running on onnx compared to tflite with fewer drops in performance.

Figure 5.2 shows how the pruned models using unstructured pruning

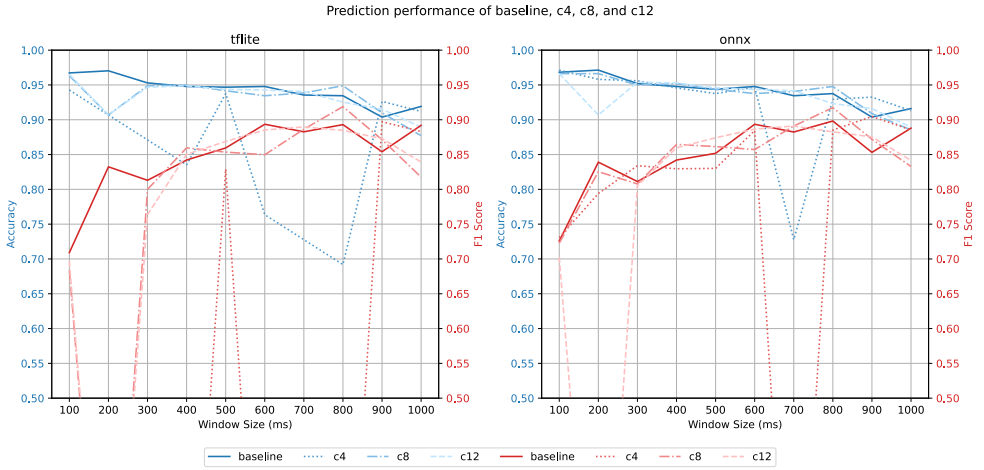


Figure 5.1: Accuracy and F1 score comparison between the base model and the base model with clustering (c) applied. The y-axis is truncated, starting from 0.50.

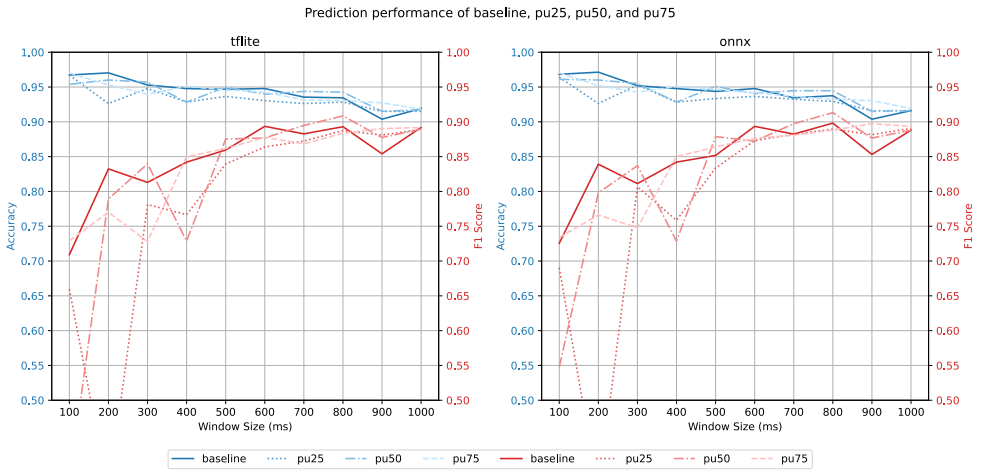


Figure 5.2: Accuracy and F1 score comparison between the base model and the base model with unstructured pruning (pu) applied. The y-axis is truncated, starting from 0.50.

perform compared to the base model. We see some substantial fluctuations in performance in the shorter window sizes, around 100ms to 400ms, however, the performance is quite comparable at the larger window sizes to the base model. The model with the target sparsity set to 50% in particular constantly performs better than the base model at window sizes of 700ms and above.

Once again we can see some difference between the two execution platforms, with the performance on onnx being more stable overall than on tflite.

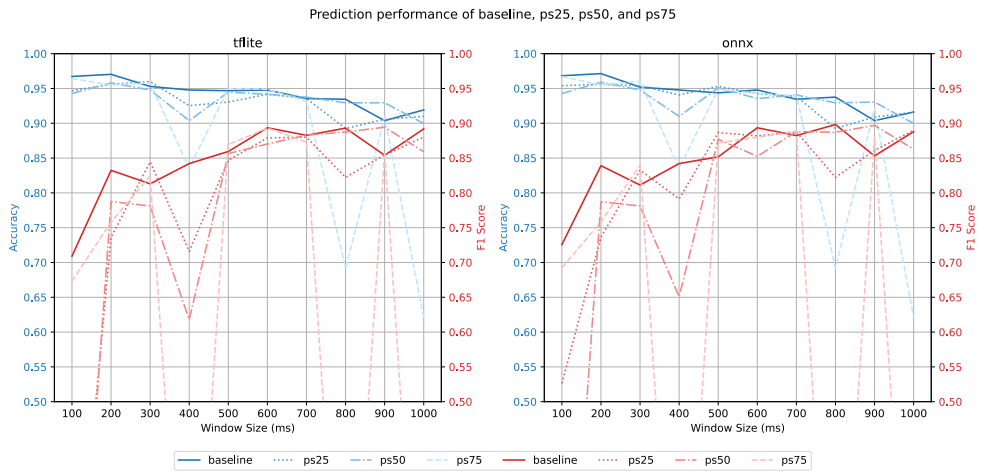


Figure 5.3: Accuracy and F1 score comparison between the base model and the base model with structured pruning (ps) applied. The y-axis is truncated, starting from 0.50.

Figure 5.3 shows how the pruned models using structured pruning perform compared to the base model. We once again see some substantial fluctuations in the performance, however this time it is regardless of the window size. We observe dips at the 400ms, 800ms, and 1000ms prediction windows from the model with the target sparsity set to 75%, where the F1 score goes down to 0. From 500ms, the 25% and the 50% models perform similarly to the base model. The 75% model also performs similarly to the base model at window sizes where the performance does not suddenly drop. This time, the performance difference between the two platforms is quite small, with the only major difference being the 50% model at the 400ms prediction window where the performance dips considerably more on tflite than onnx.

Lastly, fig. 5.4 shows the performance difference between the quantized and base models. We can see that the overall trend of the quantized model tracks the baseline models regardless of the execution platform/quantization framework. The quantized model has performance comparable to the baseline modes. However, it falls off slightly on window sizes larger than 500ms. On onnx, the quantized model performs a bit worse than on tflite for smaller window sizes, with a gap of almost 0.05 points at 100ms.

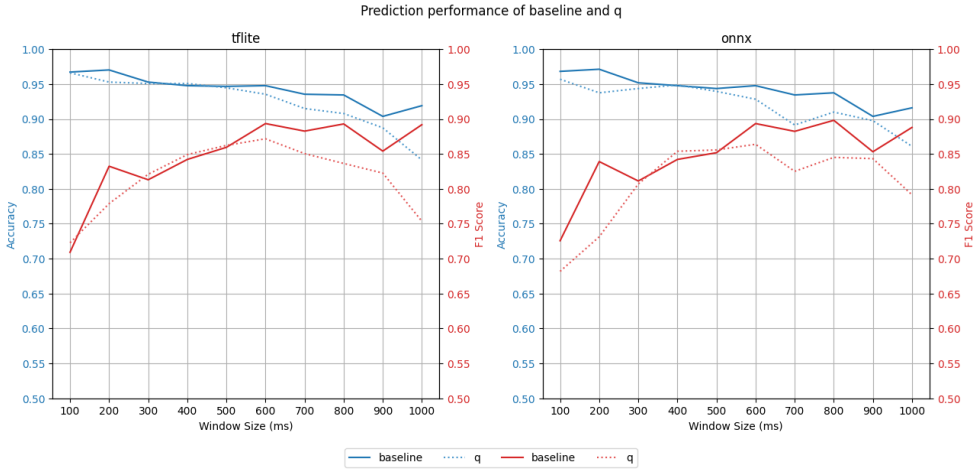


Figure 5.4: Accuracy and F1 score comparison between the base model and the base model with quantization (q) applied. The y-axis is truncated, starting from 0.50.

5.1.1.1 Statistical Significance

The statistical significance of the results was measured and evaluated using the method presented in section 3.5.

Table 5.3: t-test against baseline on tflite for 100ms prediction window

	Accuracy		F1-score	
	t-statistic	p-value	t-statistic	p-value
ps25	3.26e+01	**8.54e-55**	9.51e+01	**3.97e-99**
ps50	3.38e+01	**4.01e-56**	1.43e+02	**1.91e-116**
ps75	8.18e+00	**9.86e-13**	1.08e+01	**2.00e-18**
pu25	2.25e+00	**2.66e-02**	1.02e+01	**3.67e-17**
pu50	1.70e+01	**3.69e-31**	3.56e+01	**3.13e-58**
pu75	-4.69e+00	**8.61e-06**	-3.61e+00	**4.78e-04**
c4	3.38e+01	**4.01e-56**	1.43e+02	**1.91e-116**
c8	1.85e+01	**7.62e-34**	1.71e+01	**2.65e-31**
c12	6.92e+00	**4.47e-10**	5.16e+00	**1.26e-06**
q	4.83e+00	**5.04e-06**	-3.15e+00	**2.15e-03**

Table 5.4: t-test against baseline on onnx for 100ms prediction window

	Accuracy		F1-score	
	t-statistic	p-value	t-statistic	p-value
ps25	2.68e+01	**4.11e-47**	3.40e+01	**2.29e-56**
ps50	3.42e+01	**1.27e-56**	1.53e+02	**2.15e-119**
ps75	1.11e+00	2.68e-01	6.46e+00	**3.97e-09**
pu25	7.89e+00	**4.14e-12**	8.04e+00	**1.99e-12**
pu50	1.22e+01	**1.80e-21**	2.76e+01	**2.67e-48**
pu75	-5.29e+00	**7.39e-07**	-2.73e+00	**7.46e-03**
c4	-7.66e+00	**1.27e-11**	-8.50e-01	3.98e-01
c8	7.64e+00	**1.38e-11**	1.61e+00	1.11e-01
c12	4.91e+00	**3.62e-06**	6.53e+00	**2.81e-09**
q	2.42e+01	**2.54e-43**	1.46e+01	**2.42e-26**

Table 5.3 and table 5.4 shows the t-statistic and p-value for each model on the bootstrapped dataset for tflite and onnx respectively. All p-values that are highlighted are less than 0.05 and are thus considered statistically significant. Thus, rejecting the null hypothesis that there is no significant difference between the optimized model's and the baseline model's prediction performance.

In table 5.3, we see that all p-values are strictly less than 0.05 on tflite, thus rejecting the null hypothesis for all models. In table 5.4, we see that almost all p-values are strictly less than 0.05 on onnx, rejecting the null hypothesis. However, we did not observe any statistical difference between the structurally pruned model at 75% and the baseline model's accuracy. Also, both clustered models using 4 and 8 clusters failed to reject the null hypothesis when looking at the F1-score compared to the baseline model.

Figure 5.5 shows the average accuracy and F1-score of each model over the bootstrapped samples together with their 95% confidence intervals. We can see that the confidence intervals are small (within 0.01 for accuracy and 0.05 for F1-score) for all models. Looking back at table 5.3 and table 5.4, we see that the same scenarios and metrics that did not reject the null hypothesis in the tables, also overlap their confidence intervals with the performance values from the baseline model.

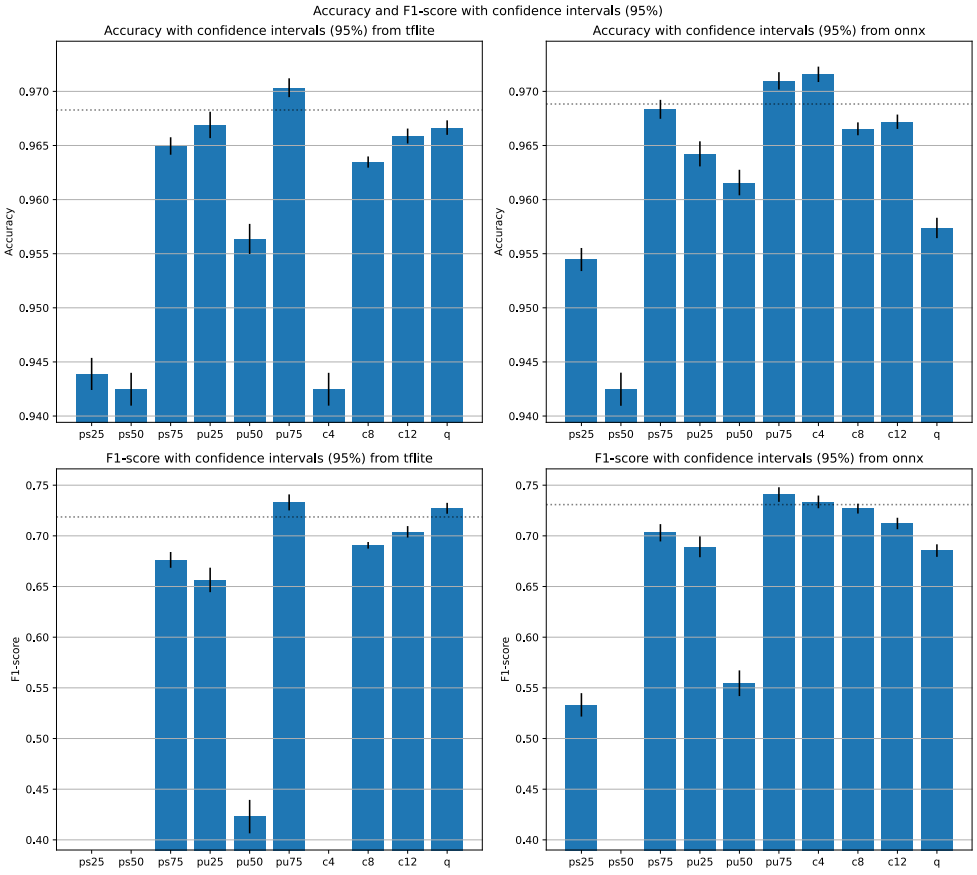


Figure 5.5: Average accuracy and F1-score with 95% confidence intervals from bootstraped samples. The baseline model's performance is marked with a dotted line. Note the truncated y-axes.

5.1.2 Inference Time

Table 5.5: Average time per inference in simulation (ms (simulated time))

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	20.53	28.18	108.92	nan
rabp_c12	20.53	28.18	108.92	nan
rabp_c4	20.53	28.18	108.92	nan
rabp_c8	20.53	28.18	108.92	nan
rabp_ps25	22.49	20.23	92.47	nan
rabp_ps50	16.42	12.34	61.94	nan
rabp_ps75	**12.78**	**5.75**	**41.65**	nan
rabp_pu25	20.53	28.18	108.92	nan
rabp_pu50	20.53	28.18	108.92	nan
rabp_pu75	20.53	28.18	108.92	nan
rabp_q	30.43	48.58	164.15	nan

In table 5.5 we can see the average time of an inference. This table shows that neither clustering nor unstructured pruning affected the inference time, however both structured pruning and quantization did. Quantization increased the latency for each inference by about 48% on tflite, and 50% on onnx. This was unexpected as one of the goals with quantization is to improve the inference time. This is discussed further in chapter 6. Structured pruning on the other hand greatly improved the latency, with the target sparsity of 25% increasing the latency by 9.4%, 50% decreasing the latency by 20%, and 75% decreasing the latency by 37.7% on tflite on ARM. Greater savings on tflite can be seen on X86, with 25% sparsity, 50% sparsity, and 75% sparsity decreasing the latency by 15.1%, 43.1%, and 61.8% respectively. Similar trends can be seen for onnx running on ARM, with 25% sparsity, 50% sparsity, and 75% sparsity decreasing the latency by 28.2%, 56.2%, and 79.6% respectively. We can also see a great difference between the simulated ARM and X86 platforms, with the ARM platform being more than 5 times faster.

Table 5.6: Average time per inference on server (ms)

	Server (X86)	
	tflite	onnx
rabp_	29.62	24.82
rabp_c12	29.62	24.86
rabp_c4	29.77	24.70
rabp_c8	29.66	24.91
rabp_ps25	24.12	nan
rabp_ps50	19.30	16.19
rabp_ps75	**15.37**	**10.53**
rabp_pu25	29.63	24.84
rabp_pu50	29.79	24.99
rabp_pu75	29.64	24.65
rabp_q	38.00	48.17

Regarding potential real-world inference time, table 5.6 shows the average time for an inference when each software platform is running natively on the server without a simulated hardware environment. Similar trends are seen in table 5.6 as in table 5.5, with clustering and unstructured pruning having no discernible effect in inference time. But an increase in inference time with quantization enabled, about an 28.3% increase with tflite and 94% with onnx. The latency for structured pruning follows a similar trend as before with 25% sparsity, 50% sparsity, and 75% sparsity decreasing the latency by 18.6%, 34.8%, and 48.1% respectively on tflite. On onnx, the model pruned to 25% sparsity failed to run, while the 50% and 75% sparsity decreased the latency by 34.8% and 57.6% respectively.

5.1.3 Hardware Utilization

5.1.3.1 CPU instructions

Table 5.7: Number of instructions executed per CPU cycle (IPC)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	0.341506	0.971663	0.545471	nan
rabp_c12	0.341410	0.970622	0.544630	nan
rabp_c4	0.341404	0.971806	0.544853	nan
rabp_c8	0.341533	0.971940	0.544682	nan
rabp_ps25	**0.594450**	0.863188	0.485233	nan
rabp_ps50	0.315840	0.871608	0.393626	nan
rabp_ps75	0.312446	0.860715	0.327986	nan
rabp_pu25	0.341548	0.970774	0.544575	nan
rabp_pu50	0.341343	**0.974228**	0.545428	nan
rabp_pu75	0.341670	0.971385	0.545550	nan
rabp_q	0.397855	0.834170	**0.806304**	nan

Table 5.7 shows that IPC is mostly not affected by any specific optimization. However, we can see a dip across all hardware and software platforms for the structurally pruned models, about 6% to 11% on ARM and 11% to 40% on X86. Then we have two ”outliers” *rabp_ps25* on ARM with tflite and *rabp_q* on X86 on onnx. These entries exhibit a substantial increase in IPC compared to all other values in their respective columns.

Table 5.8: Total number of instructions committed during simulation (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	48,965,992,442	59,618,453,790	127,637,026,519	nan
rabp_c12	48,965,992,267	59,619,601,773	127,636,551,247	nan
rabp_c4	48,965,973,601	59,618,608,907	127,636,729,860	nan
rabp_c8	48,965,992,853	59,618,060,983	127,636,954,928	nan
rabp_ps25	53,003,904,394	41,575,469,463	113,547,722,190	nan
rabp_ps50	39,059,373,490	25,088,392,721	78,191,642,372	nan
rabp_ps75	**31,916,405,516**	**11,826,620,343**	**54,367,848,542**	nan
rabp_pu25	48,965,985,558	59,618,528,693	127,636,763,961	nan
rabp_pu50	48,965,992,250	59,617,771,617	127,636,725,598	nan
rabp_pu75	48,965,985,422	59,618,025,009	127,636,880,249	nan
rabp_q	73,593,079,189	131,824,330,119	174,961,719,582	nan

Looking at table 5.8, we can see the total number of instructions executed during each simulation. There is a clear trend in the table, with only the quantized and structurally pruned models affecting the total number of instructions in any meaningful way. For all instances of the quantized model, regardless of hardware or software platform, the number of committed instructions increased. On tflite the increase is about 50.3% and 37% on ARM and X86 respectively, and 121.1% with onnx running on ARM. Structured pruning on the other hand steadily decreased the total number of committed instructions when the target sparsity was increased, with the most substantial difference achieved at 75% sparsity resulting in a 34.8% and 57.4% reduction with tflite on ARM and X86 respectively, and an 80.2% reduction with onnx on ARM.

Table 5.9: Number of committed floating-point instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	18,911,499	28,783,310	40,681,045,592	nan
rabp_c12	18,911,484	28,783,237	40,681,037,016	nan
rabp_c4	18,897,453	28,783,238	40,681,008,502	nan
rabp_c8	18,911,542	28,783,245	40,681,045,005	nan
rabp_ps25	18,741,230	22,561,886	40,002,132,224	nan
rabp_ps50	18,746,012	20,701,624	23,681,242,377	nan
rabp_ps75	18,811,243	**16,967,788**	**15,515,917,747**	nan
rabp_pu25	18,906,313	28,783,258	40,681,036,250	nan
rabp_pu50	18,911,484	28,783,207	40,681,045,027	nan
rabp_pu75	18,906,134	28,783,230	40,681,039,663	nan
rabp_q	**4,183,203**	3,982,551,873	34,972,111,284	nan

Table 5.9 breaks down the results in table 5.8 a bit and only looks at floating-point instructions. Looking at tflite on ARM, we see that all models except the quantized one are within 1% of one another. However, the quantized model decreased the number of floating-point instructions by about 77.9%. With tflite running on X86 we still see a decrease for the quantized model, about 14%. However, the structurally pruned model also shows a decrease in committed floating-point instructions, with the sparser models showing a greater difference, of about 61.9% at 75% sparsity. All other models are still within 1% of one another. For onnx running on ARM, a 41.1% decrease at 75% sparsity can be seen for the structurally pruned model. However, the quantized model increased the number of floating-point instructions by about 137.4 times. This major increase was unexpected and seems abnormal when comparing to the results when running on tflite. This is further discussed in chapter 6. All other models are within 1% of one another.

Table 5.10: Number of committed integer instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	26,432,875,665	29,509,391,495	88,755,858,058	nan
rabp_c12	26,432,875,555	29,510,329,531	88,755,035,580	nan
rabp_c4	26,432,875,591	29,509,529,590	88,755,375,811	nan
rabp_c8	26,432,875,895	29,509,084,538	88,755,728,395	nan
rabp_ps25	29,453,313,703	22,592,725,760	81,807,998,509	nan
rabp_ps50	24,429,523,308	14,776,385,574	61,563,104,601	nan
rabp_ps75	**20,816,654,024**	**7,745,284,323**	**44,507,746,470**	nan
rabp_pu25	26,432,875,683	29,509,450,927	88,755,436,493	nan
rabp_pu50	26,432,875,541	29,508,839,813	88,755,356,770	nan
rabp_pu75	26,432,875,751	29,509,041,658	88,755,642,683	nan
rabp_q	40,445,690,865	78,066,529,731	142,040,234,676	nan

Table 5.10 looks only into the number of committed integer instructions. Once again, all models except the pruned and structurally pruned ones are within 1% of one another regardless of software or hardware platform. The quantized model increased the number of integer instructions by about 53% and 60% running with tflite on ARM and X86 respectively, and 164.5% with onnx on ARM. Generally, the structurally pruned model decreased the number of integer instructions, except for one case of *rabp_ps25*, when running with tflite on ARM the number of instructions increased by about 11.3%. Once again the model targeting 75% sparsity achieved the largest difference. We can see a 21.3% and a 49.9% decrease with tflite on ARM and X86 respectively, and a 73.8% decrease with onnx on ARM.

Table 5.11: Total number of committed vector (SIMD) instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	20,261,191,895	27,752,009,110	129,665,914,941	nan
rabp_c12	20,261,191,867	27,752,009,043	129,665,906,365	nan
rabp_c4	20,261,187,190	27,752,009,040	129,665,877,851	nan
rabp_c8	20,261,191,983	27,752,009,051	129,665,914,354	nan
rabp_ps25	21,519,743,254	17,218,494,042	95,934,000,741	nan
rabp_ps50	13,108,895,308	9,070,398,161	56,353,613,999	nan
rabp_ps75	**7,358,490,764**	**3,124,277,953**	**28,043,180,585**	nan
rabp_pu25	20,261,190,180	27,752,009,056	129,665,905,599	nan
rabp_pu50	20,261,191,867	27,752,009,013	129,665,914,376	nan
rabp_pu75	20,261,190,142	27,752,009,028	129,665,909,012	nan
rabp_q	29,060,575,169	18,075,930,432	216,370,354,207	nan

Table 5.11 only looks at the number of committed vector instructions for each simulation. Again, all models except the pruned and structurally pruned ones are within 1% of one another regardless of software or hardware platform. Similarly to table 5.10, we can see that the structurally pruned model decreased the number of vector instructions, except in the case of *rabp_ps25* when running with tflite on ARM the number of instructions increased by about 6.2%. The greatest difference was achieved by the model targeting a sparsity of 75%, with a 63.7% and 78.4% decrease with tflite on ARM and X86 respectively, and an 88.7% decrease with onnx on ARM. The quantized model increased the number of vectorized instructions when running with tflite, 43.4% on ARM, and 66.9% on X86, but decreased the number of vectorized instructions when running with onnx on ARM by 34.9%.

Table 5.12: Number of committed floating-point based vector (SIMD) instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	16,848,358,144	24,678,051,793	62,273,979,015	nan
rabp_c12	16,848,358,128	24,678,051,793	62,273,970,439	nan
rabp_c4	16,848,358,128	24,678,051,793	62,273,941,925	nan
rabp_c8	16,848,358,186	24,678,051,793	62,273,978,428	nan
rabp_ps25	20,310,767,968	14,848,275,025	76,591,167,351	nan
rabp_ps50	12,208,780,528	7,400,314,833	45,337,290,827	nan
rabp_ps75	6,616,506,087	2,479,361,233	23,676,867,259	nan
rabp_pu25	16,848,358,150	24,678,051,793	62,273,969,673	nan
rabp_pu50	16,848,358,128	24,678,051,793	62,273,978,450	nan
rabp_pu75	16,848,358,163	24,678,051,793	62,273,973,086	nan
rabp_q	**1,225,178,518**	**695,110,273**	**4,376,733,440**	nan

Table 5.12 further breaks down the results and looks only at the number of committed floating-point-based vector instructions for each simulation. Again, all models except the pruned and structurally pruned ones are within 1% of one another regardless of software or hardware platform. Once again, we can see the same trend with the structurally pruned model as in table 5.11, with an increase of 20.5% and 23% for *rabp_ps25* running with tflite on ARM and X86 respectively. The greatest difference was once again achieved by *rabp_ps75*, with a 60.7% and a 62% decrease with tflite on ARM and X86 respectively, and a 90% decrease with onnx on ARM. The quantized model decreased the number of committed floating-point-based vector instructions, 92.6% and 93% with tflite on ARM and X86 respectively, and 97.2% with onnx on ARM.

Table 5.13: Number of committed integer based vector (SIMD) instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	3,412,833,751	3,073,957,317	67,391,935,926	nan
rabp_c12	3,412,833,739	3,073,957,250	67,391,935,926	nan
rabp_c4	3,412,829,062	3,073,957,247	67,391,935,926	nan
rabp_c8	3,412,833,797	3,073,957,258	67,391,935,926	nan
rabp_ps25	1,208,975,286	2,370,219,017	19,342,833,390	nan
rabp_ps50	900,114,780	1,670,083,328	11,016,323,172	nan
rabp_ps75	**741,984,677**	**644,916,720**	**4,366,313,326**	nan
rabp_pu25	3,412,832,030	3,073,957,263	67,391,935,926	nan
rabp_pu50	3,412,833,739	3,073,957,220	67,391,935,926	nan
rabp_pu75	3,412,831,979	3,073,957,235	67,391,935,926	nan
rabp_q	27,835,396,651	17,380,820,159	211,993,620,767	nan

Looking at table 5.13, we can see the number of committed integer-based vector instructions for each simulation. Again, all models except the pruned and structurally pruned ones are within 1% of one another regardless of software or hardware platform. We can see a clear trend with the structurally pruned models, where the sparser the model, the greater the decrease in committed integer-based vector instructions. The greatest difference can be seen for 75% sparsity with a 78.3% and a 93.5% decrease with tflite on ARM and X86, and a 79% decrease with onnx on ARM. The quantized model heavily increased the usage of integer-based vector instructions, with a 715.8% and a 214.6% increase with tflite on ARM and X86, and a 465.6% increase with onnx on ARM

Table 5.12 and table 5.13 together show a clear change in the distribution of vector instructions executed when quantization is applied, with floating-point based vector instructions having a much greater presence in un-quantized models compared to integer based once. The opposite is true when quantization is used, with integer-based vector instructions having a much greater presence.

5.1.3.2 Memory

Table 5.14: Utilized memory bandwidth (MB/s)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	2749.85	1267.94	518.22	nan
rabp_c12	2749.85	**1267.83**	518.22	nan
rabp_c4	2749.86	1267.85	518.22	nan
rabp_c8	2749.86	1267.93	518.22	nan
rabp_ps25	2135.62	1316.43	519.17	nan
rabp_ps50	2456.54	1495.29	650.66	nan
rabp_ps75	2068.41	1843.74	633.44	nan
rabp_pu25	2749.86	1267.94	518.22	nan
rabp_pu50	2749.85	1267.87	518.22	nan
rabp_pu75	2749.86	1267.98	518.22	nan
rabp_q	**1039.63**	1335.44	**192.14**	nan

Table 5.14 shows each test case's total utilized memory bandwidth. We can see similar trends as before, with only structured pruning and quantization having any effects on the results compared to the baseline model. Quantization slightly increased the memory bandwidth used with onnx on ARM by 5.4%, while decreasing the bandwidth when running on tflite by 62.2% on ARM and 62.9% on X86. The structurally pruned models saw two different trends, with tflite on ARM the bandwidth decreased when the model became sparser, but on X86, and with onnx on ARM the bandwidth increased as the model became sparser. With tflite on ARM, the greatest decrease was 24.8% at 75% sparsity. The greatest increase with tflite on X86 was 25.6% at 50% sparsity, and 45.4% at 75% sparsity with onnx on ARM.

Table 5.15: Memory traffic volume per inference (MB)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	57.78	36.57	57.77	nan
rabp_c12	57.78	36.57	57.77	nan
rabp_c4	57.78	36.57	57.77	nan
rabp_c8	57.78	36.57	57.77	nan
rabp_ps25	49.16	27.26	49.14	nan
rabp_ps50	41.29	18.89	41.25	nan
rabp_ps75	**27.06**	**10.85**	**27.0**	nan
rabp_pu25	57.78	36.57	57.77	nan
rabp_pu50	57.78	36.57	57.77	nan
rabp_pu75	57.78	36.57	57.77	nan
rabp_q	32.38	66.4	32.28	nan

Using tables 5.5 and 5.14 we can infer the total volume of memory traffic at each inference. We present this in table 5.15. As with most results previously, only structured pruning and quantization affected the results compared to the baseline model. Quantization used in combination with tflite decreased the memory volume by 44% on both hardware platforms, while it increased the volume by 81.6% on onnx. Structured pruning decreased the traffic volume regardless of software or hardware platform, with the highest savings achieved by the model with 75% sparsity. With tflite, we saw a 51.2% decrease on ARM and X86, and with onnx, we saw a 70.3% decrease on ARM.

5.2 Multiple optimizations

This section shows the results of stacking the optimizations from section 5.1. The most promising parameters for each optimization were chosen based on the results in section 5.1. The optimizations chosen were clustering with 8 centroids, structured pruning with a target sparsity of 75%, and full integer (8-bit) quantization.

5.2.1 Prediction Performance

Prediction performance regards the models' accuracy and F1 score, indicating how well they perform before and after being optimized.

Table 5.16: Accuracy for 100ms prediction window

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	0.967	0.968	0.967	nan
rabp_c8	0.963	0.966	0.963	nan
rabp_c8q	0.965	0.965	0.965	nan
rabp_ps75	0.964	0.967	0.964	nan
rabp_ps75c8	**0.972**	**0.969**	**0.972**	nan
rabp_ps75c8q	0.970	**0.969**	0.970	nan
rabp_ps75q	0.960	0.965	0.960	nan
rabp_q	0.966	0.959	0.966	nan

The accuracy for each model can be seen in table 5.16, which shows that the accuracy of the model's predictions is not greatly affected by any stacking of optimization techniques, regardless of the hardware or software platform. In the case of tflite, it also shows that model performance is not dependent on the hardware platform.

Table 5.17: F1 score for 100ms prediction window

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	0.709	**0.726**	0.709	nan
rabp_c8	0.684	0.723	0.684	nan
rabp_c8q	0.712	0.712	0.712	nan
rabp_ps75	0.673	0.692	0.673	nan
rabp_ps75c8	**0.748**	0.712	**0.748**	nan
rabp_ps75c8q	0.724	0.722	0.724	nan
rabp_ps75q	0.629	0.673	0.629	nan
rabp_q	0.723	0.692	0.723	nan

Regarding the models' F1 score, table 5.17 shows greater differences between the baseline model and the optimized ones. It also shows a difference between the inference platforms, with tflite lagging behind onnx by about 0.02 to 0.04 points for most models. But on the other hand, "*rabp_q*" and "*rabp_ps75c8*" are performing better on tflite by about 0.04 points.

Figure 5.6 shows the prediction performance of the models with only one optimization applied compared to the base model. These results are the same

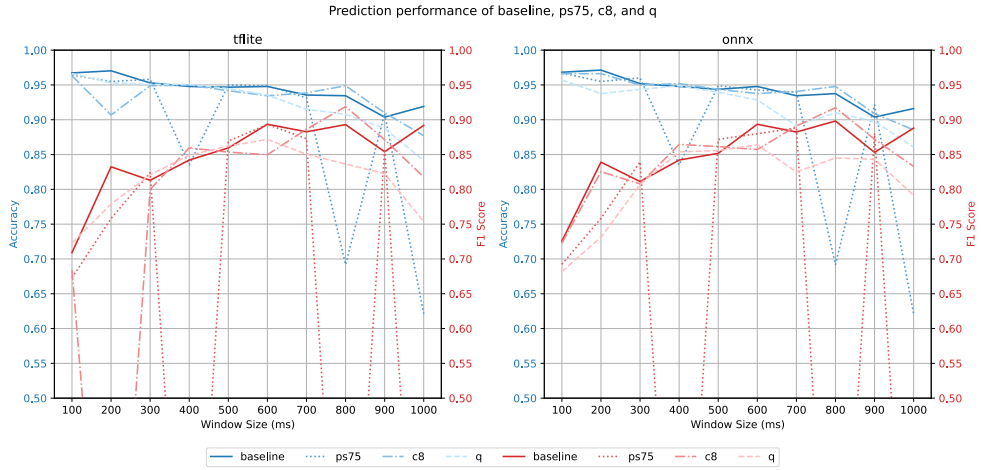


Figure 5.6: Accuracy and F1 score comparison between the base model with either structured pruning (ps), clustering (c), or quantization (q) applied. The y-axis is truncated, starting from 0.50.

as can be extrapolated from section 5.1.1.

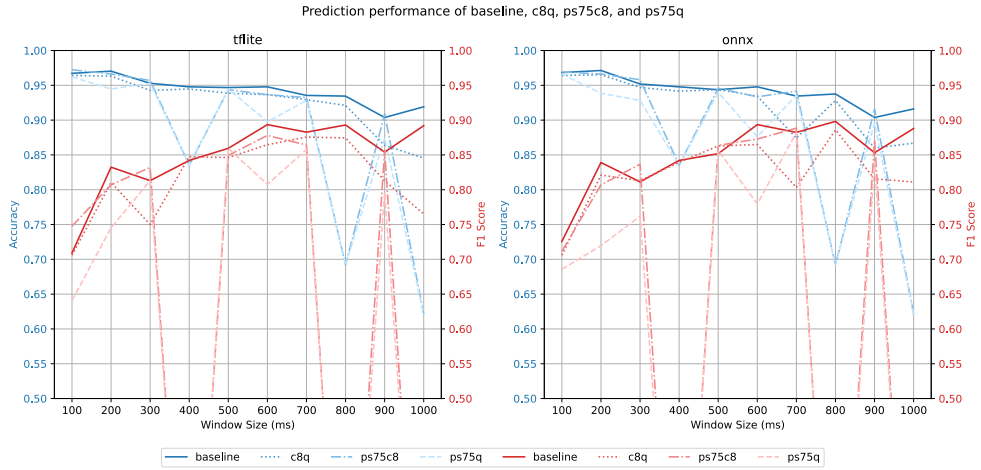


Figure 5.7: Accuracy and F1 score comparison between the base model and the base model with structured pruning and clustering (ps and c), clustering and quantization (c and q), and structured pruning and quantization (ps and q) applied. The y-axis is truncated, starting from 0.50.

Figure 5.7 shows the prediction performance of the models with two optimizations applied. We observe substantial drops in prediction

performance observed in fig. 5.3 manifests in all models with pruning applied, (*"ps75c8"* and *"ps75q"*). However, the optimized models perform similarly to the baseline model outside of the drops on window sizes smaller than 700ms. Sometimes they even outperform the baseline, e.g., *"ps75c8"* on the 100ms prediction window. This time the difference between the execution platforms is not as significant as we have observed previously.

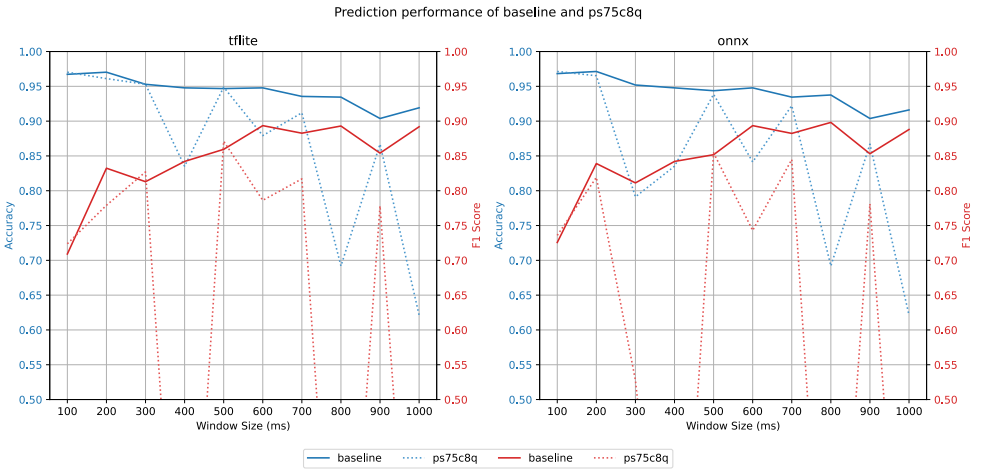


Figure 5.8: Accuracy and F1 score comparison between the base model and the base model with structured pruning, clustering, and quantization (ps, c, and q) applied. The y-axis is truncated, starting from 0.50.

Figure 5.8 shows the model's prediction performance with three optimizations applied. Regardless of platform, we once again see some instability in the curve. At some window sizes, the optimized model performs as well as, or even slightly better than, the baseline.

As in section 5.1.1, we see the same unexpected drops in F1-score for the pruned and clustered models in fig. 5.6, fig. 5.7, and fig. 5.8. This behaviour is consistent for all graphs of models with structured pruning applied. However, the clustered model only experiences this at 200ms if no other optimization is applied. This will be further discussed in chapter 6.

5.2.1.1 Statistical Significance

The statistical significance of the results was measured and evaluated using the method presented in section 3.5.

Table 5.18: t-test against baseline on tflite for 100ms prediction window

	Accuracy		F1-score	
	t-statistic	p-value	t-statistic	p-value
ps75	8.18e+00	**9.86e-13**	1.08e+01	**2.00e-18**
c8	1.85e+01	**7.62e-34**	1.71e+01	**2.65e-31**
q	4.83e+00	**5.04e-06**	-3.15e+00	**2.15e-03**
c8q	1.13e+01	**1.61e-19**	3.14e+00	**2.20e-03**
ps75c8	-1.25e+01	**4.26e-22**	-8.94e+00	**2.29e-14**
ps75q	1.38e+01	**1.06e-24**	1.70e+01	**3.75e-31**
ps75c8q	-6.99e+00	**3.19e-10**	-2.89e+00	**4.69e-03**

Table 5.19: t-test against baseline on onnx for 100ms prediction window

	Accuracy		F1-score	
	t-statistic	p-value	t-statistic	p-value
ps75	1.11e+00	2.68e-01	6.46e+00	**3.97e-09**
c8	7.64e+00	**1.38e-11**	1.61e+00	1.11e-01
q	2.42e+01	**2.54e-43**	1.46e+01	**2.42e-26**
c8q	1.81e+01	**4.43e-33**	1.35e+01	**4.25e-24**
ps75c8	-3.64e+00	**4.33e-04**	2.56e+00	**1.20e-02**
ps75q	3.47e+00	**7.76e-04**	8.08e+00	**1.64e-12**
ps75c8q	-8.53e+00	**1.70e-13**	-3.03e+00	**3.09e-03**

Table 5.18 and table 5.19 shows the t-statistic and p-value for each model on the bootstrapped dataset for tflite and onnx respectively. All p-values that are highlighted are less than 0.05 and are thus considered statistically significant. Thus, rejecting the null hypothesis that there is no significant difference between the optimized model's and the baseline model's prediction performance.

In table 5.18 we see that all p-values are strictly less than 0.05 on tflite, thus rejecting the null hypothesis for all models. In table 5.19 we see that almost p-values are strictly less than 0.05 on onnx, rejecting the null hypothesis. As in table 5.4, we did not observe any statistical difference between the structurally pruned model at 75% and the baseline model's accuracy. Also, the clustered model using 8 clusters failed to reject the null hypothesis when looking at the F1-score compared to the baseline model.

Figure 5.9 shows the average accuracy and F1-score of each model over the bootstrapped samples together with their 95% confidence intervals. We

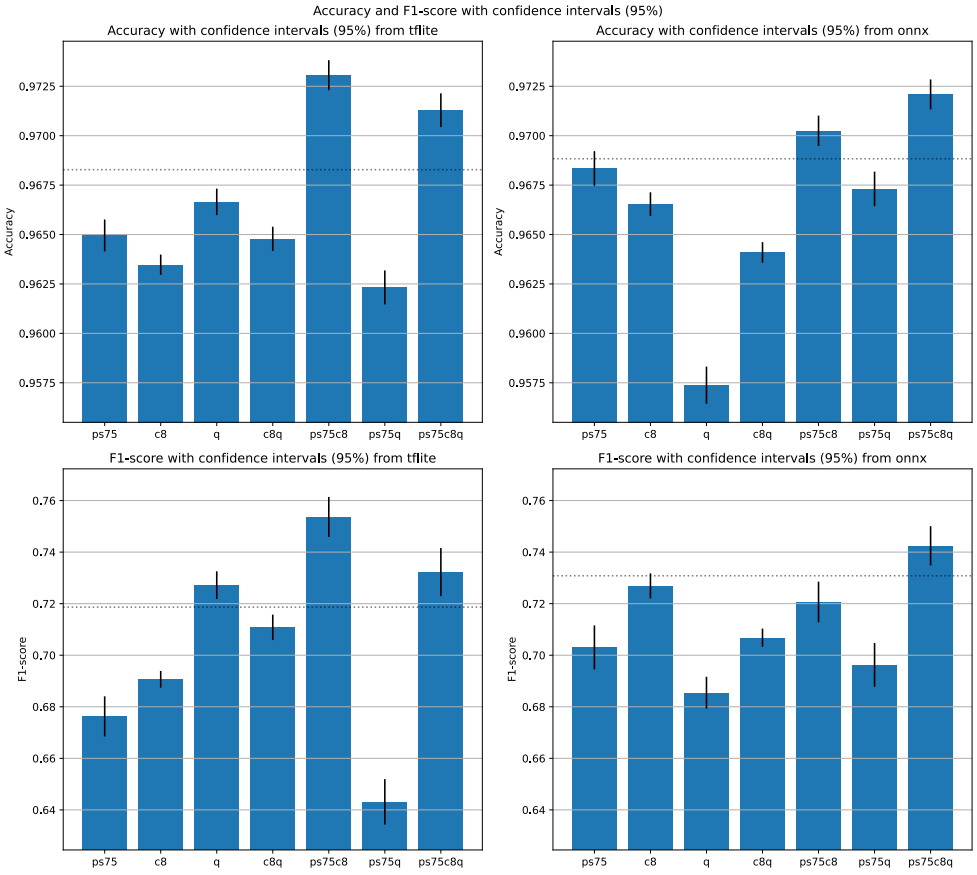


Figure 5.9: Average accuracy and F1-score with confidence intervals from bootstraped samples. The baseline model's performance is marked with a dotted line. Note the truncated y-axes.

can see that the confidence intervals are small (within 0.01 for accuracy and 0.05 for F1-score) for all models. Looking back at table 5.18 and table 5.19, we see that the same scenarios and metrics that did not reject the null hypothesis in the tables, also overlap their confidence intervals with the performance values from the baseline model.

5.2.2 Inference Time

Table 5.20: Average time per inference in simulation (ms (simulated time))

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	20.53	28.18	108.92	nan
rabp_c8	20.53	28.18	108.92	nan
rabp_c8q	30.43	48.58	164.15	nan
rabp_ps75	**12.78**	**5.75**	**41.65**	nan
rabp_ps75c8	**12.78**	**5.75**	**41.65**	nan
rabp_ps75c8q	23.40	35.16	80.87	nan
rabp_ps75q	23.40	35.16	80.87	nan
rabp_q	30.43	48.58	164.15	nan

In 5.20 we can see the average time of an inference. This table shows that clustering does not affect the inference time regardless of combination with other optimizations. Quantization increased the latency for each inference by about 49% to 51% on tflite, and 72% on onnx. We can once again see a great difference between the simulated ARM and X86 platforms, with the ARM platform being more than 5 times faster for the baseline model. However, that gap in performance shrinks as optimizations are applied.

Table 5.21: Average time per inference on server (ms)

	Server (X86)	
	tflite	onnx
rabp_	29.62	24.82
rabp_c8	29.66	24.91
rabp_c8q	21.23	28.54
rabp_ps75	15.37	10.53
rabp_ps75c8	**8.81**	**5.18**
rabp_ps75c8q	10.89	18.54
rabp_ps75q	10.89	18.54
rabp_q	38.00	48.17

Regarding potential real-world inference time, table 5.21 shows the average time for an inference when each software platform is running natively

on the server without a simulated hardware environment. Similar trends are seen on 5.21 as in 5.20, with clustering having no discernible effect on inference time. Quantization increased the inference time by about 23% with tflite and 185% with onnx. All models with structured pruning applied provided great reductions in inference time. Combining quantization and structured pruning made the time penalty introduced by quantization less noticeable

5.2.3 Hardware Utilization

5.2.3.1 CPU instructions

Table 5.22: Number of instructions executed per CPU cycle (IPC)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	0.341506	0.972290	0.544935	nan
rabp_c8	0.341533	0.971877	0.545338	nan
rabp_c8q	0.396496	0.835021	**0.807153**	nan
rabp_ps75	0.312446	0.863976	0.327986	nan
rabp_ps75c8	0.312783	0.861629	0.328165	nan
rabp_ps75c8q	**0.975393**	1.022178	0.742727	nan
rabp_ps75q	0.972560	**1.022210**	0.744103	nan
rabp_q	0.397855	0.834398	0.806427	nan

Table 5.22 shows that IPC mostly is not affected by clustering. We can see that stacking quantization and structured pruning yields great improvements to the IPC for tflite and onnx running on ARM and on X86. However, just using quantization on X86 yields the highest IPC.

Table 5.23: Total number of instructions committed during simulation (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	49,015,109,429	59,740,688,214	128,014,784,409	nan
rabp_c8	49,015,113,857	59,741,488,860	128,014,704,913	nan
rabp_c8q	73,680,243,399	132,076,338,761	175,506,335,120	nan
rabp_ps75	31,945,044,969	**11,850,948,000**	54,463,283,460	nan
rabp_ps75c8	**31,944,881,376**	11,851,422,944	**54,462,715,904**	nan
rabp_ps75c8q	58,427,837,465	96,969,319,578	93,655,089,858	nan
rabp_ps75q	58,427,702,442	96,969,121,904	93,654,936,326	nan
rabp_q	73,680,186,518	132,076,826,511	175,511,774,191	nan

Regarding the number of instructions executed in each simulation, we look at table 5.23. Here we can also see a clear distinction between the quantized models and the others, with the quantized models needing more CPU instructions to run. Applying structured pruning lowers the number of instructions, mitigating some of the increases imposed by quantization. On ARM, combining quantization and structured pruning yields a 19.2% and a 60.6% increase on tflite and onnx respectively. However, stacking these two optimizations decreased the number of committed instructions compared to the baseline model on X86. Tables 5.24, 5.25, and 5.26 further dissects which kind of instructions each case used.

Table 5.24: Number of committed floating-point instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	18,911,499	28,783,253	40,681,045,594	nan
rabp_c8	18,911,542	28,783,237	40,681,045,009	nan
rabp_c8q	4,183,203	3,982,551,815	34,972,111,280	nan
rabp_ps75	18,811,243	16,969,747	15,515,917,747	nan
rabp_ps75c8	18,790,135	**16,967,774**	**15,515,840,542**	nan
rabp_ps75c8q	**4,182,379**	1,431,423,998	19,607,994,840	nan
rabp_ps75q	**4,182,379**	1,431,423,948	19,607,994,846	nan
rabp_q	4,183,203	3,982,551,867	34,972,111,287	nan

Table 5.24 shows the total number of floating-point instructions executed

for each simulation. Looking at the numbers for tflite, we can see a clear trend that quantization lowers this number significantly regardless of any other optimizations, about a 77.9% reduction on ARM and a 14% reduction on X86. These reductions are helped further when structured pruning is applied. However, quantization on onnx increases the number of committed floating-point instructions so much that applying structured pruning still leaves the number of instructions two orders of magnitudes larger than the baseline.

Table 5.25: Number of committed integer instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	26,432,875,665	29,509,002,281	88,755,858,044	nan
rabp_c8	26,432,875,895	29,509,662,952	88,755,728,266	nan
rabp_c8q	40,445,690,889	78,065,862,286	142,026,279,191	nan
rabp_ps75	20,816,654,024	**7,745,536,562**	44,507,746,470	nan
rabp_ps75c8	**20,816,548,529**	7,746,036,667	**44,506,631,687**	nan
rabp_ps75c8q	33,845,053,259	67,402,126,723	70,665,706,651	nan
rabp_ps75q	33,845,053,265	67,401,978,024	70,666,606,774	nan
rabp_q	40,445,690,865	78,066,287,714	142,040,234,676	nan

Looking at 5.25, we can again see some clear trends regarding quantization. Regardless of the hardware or software platform, we see a clear increase in the number of integer instructions used whenever quantization is applied. Only on tflite on X86 were these increases mitigated by structured pruning enough to decrease the number of integer instructions compared to the baseline.

Table 5.26: Total number of committed vector (SIMD) instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	20,261,191,895	27,752,009,053	129,665,914,917	nan
rabp_c8	20,261,191,983	27,752,009,033	129,665,914,306	nan
rabp_c8q	29,060,575,172	18,075,930,377	216,370,354,200	nan
rabp_ps75	7,358,490,764	3,124,279,910	28,043,180,585	nan
rabp_ps75c8	**7,358,448,546**	**3,124,277,933**	**28,043,103,380**	nan
rabp_ps75c8q	19,575,233,886	9,167,300,049	95,679,730,234	nan
rabp_ps75q	19,575,233,886	9,167,300,005	95,679,730,240	nan
rabp_q	29,060,575,169	18,075,930,430	216,370,354,207	nan

In table 5.26 we can see the number of vector instructions utilized by each case. We see that clustering still does not affect the results. Looking at the quantization number we can see that its trend differs between tflite and onnx, on tflite did quantization increase the usage of vector instructions whiles it decreased it on onnx. Structured pruning always decreased the number of vector instructions compared to the baseline model, even when combined with quantization. Combining these results with the results in table 5.23 we can see that the ratio of vector instructions differs when we stack quantization and structured pruning. For both platforms running on ARM, both quantization and structured pruning lower the ratio of vector to non-vector instructions. However, quantization applied on tflite on X86 increased this ratio.

Looking more closely into what kind of vector instruction is being executed, we look at table 5.27 and 5.28.

Table 5.27: Number of committed floating-point based vector (SIMD) instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	16,848,358,144	24,678,051,793	62,273,979,015	nan
rabp_c8	16,848,358,186	24,678,051,793	62,273,978,428	nan
rabp_c8q	1,225,178,518	695,110,273	4,376,733,447	nan
rabp_ps75	6,616,506,087	2,479,361,233	23,676,867,259	nan
rabp_ps75c8	6,616,484,980	2,479,361,233	23,676,790,054	nan
rabp_ps75c8q	**964,686,870**	**596,691,201**	3,620,269,044	nan
rabp_ps75q	**964,686,870**	**596,691,201**	**3,620,269,038**	nan
rabp_q	1,225,178,518	695,110,273	4,376,733,440	nan

The usage of floating-point-based vector instructions follows a clear trend in table 5.27, both structured pruning and quantization greatly decreased the number of floating-point vector instructions. Applying both quantization and structured pruning further decreased this number beyond what just a single optimization was able to do, with *rabp_ps75q* and *rabp_ps75c8q* utilizing the least amount of floating-point vector instructions on all software and hardware platforms.

Table 5.28: Number of committed integer based vector (SIMD) instructions (total over 977 inferences)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	3,412,833,751	3,073,957,260	67,391,935,902	nan
rabp_c8	3,412,833,797	3,073,957,240	67,391,935,878	nan
rabp_c8q	27,835,396,654	17,380,820,104	211,993,620,753	nan
rabp_ps75	741,984,677	644,918,677	**4,366,313,326**	nan
rabp_ps75c8	**741,963,566**	**644,916,700**	**4,366,313,326**	nan
rabp_ps75c8q	18,610,547,016	8,570,608,848	92,059,461,190	nan
rabp_ps75q	18,610,547,016	8,570,608,804	92,059,461,202	nan
rabp_q	27,835,396,651	17,380,820,157	211,993,620,767	nan

Looking at table 5.28, we can see a trend of quantization greatly increasing the usage of integer-based vector instructions while structured pruning greatly decreases the usage, with clustering having no discernable effect. When both

quantization and structured pruning are applied, we can see that the utilization of integer-based vector instructions is higher than on the baseline model, but in some cases less than half of what just quantization utilized.

Similarly to what we could see in section 5.1.3.1, we can still see a great shift in the utilization of integer and floating-point based vector instructions when quantization is applied in table 5.27 and table 5.28, regardless of other applied optimizations. Quantization shifts the usage towards integer-based vector instructions while structured pruning lowers the number of committed instructions overall.

5.2.3.2 Memory

Table 5.29: Utilized memory bandwidth (MB/s)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	2749.85	1267.92	518.22	nan
rabp_c8	2749.86	1267.92	518.22	nan
rabp_c8q	1039.63	1335.42	192.14	nan
rabp_ps75	2068.41	1843.70	633.44	nan
rabp_ps75c8	2068.39	1844.09	633.45	nan
rabp_ps75c8q	**833.59**	1238.75	240.01	nan
rabp_ps75q	833.60	**1238.57**	240.01	nan
rabp_q	1039.63	1335.37	**192.14**	nan

Table 5.29 shows each case's total utilized memory bandwidth. Looking at the data, we see some differing trends between the different hardware and software platforms. Running with tflite on ARM made the baseline model use the highest bandwidth while both quantization and structured pruning lowered the bandwidth, with *rabp_ps75q* and *rabp_ps75c8q* using the least amount of memory bandwidth. On X86 however, we see that structured pruning increases the bandwidth. Making the quantized model utilize the lowest amount. Looking at onnx on ARM, we can see that both quantization and structured pruning increased the bandwidth when applied alone. However, combining the two yielded an overall lower bandwidth and resulted in the lowest bandwidth for that platform.

Table 5.30: Memory traffic volume per inference (MB)

	ARM		X86	
	tflite	onnx	tflite	onnx
rabp_	57.78	36.57	57.77	nan
rabp_c8	57.78	36.57	57.77	nan
rabp_c8q	32.38	66.4	32.28	nan
rabp_ps75	27.06	**10.85**	27.0	nan
rabp_ps75c8	27.06	**10.85**	27.0	nan
rabp_ps75c8q	**19.97**	44.58	**19.87**	nan
rabp_ps75q	**19.97**	44.57	**19.87**	nan
rabp_q	32.38	66.4	32.28	nan

Using tables 5.20 and 5.29 we can infer the volume of the memory traffic at each inference. We present this in table 5.30. As in table 5.15, we see that both quantization and structured pruning lowers the memory volume on tflite regardless of hardware platform, resulting in the models with both of these optimizations applied to have the lowest used memory volume. We can also see that the memory volume for tflite is the same regardless of the hardware platform. With onnx we instead see that quantization greatly increases the used memory volume, resulting in all models using quantization having a higher memory volume than baseline, regardless of other applied optimizations. This resulted in the structurally pruned model (with and without clustering) achieving the lowest memory volume.

Chapter 6

Discussion

This thesis needs to discuss two main points: the effects of software optimizations on hardware utilization and the applicability of a system like this to the real world. This is done in the two sections following the next section, which summarizes the results.

6.1 Summary of Results

Chapter 5 provides some key findings, which we have summarized in this section.

The model accuracy was not affected much by any optimization technique in the 100ms prediction window as seen in table 5.1 showing the prediction accuracy of each model. However, greater fluctuation was observed for the F1-score in table 5.2 with all pruning targets lower than 75% for both structured and unstructured pruning, yielding substantially lower predictive performance. We also saw greatly fluctuating scoring depending on the window size in the graphs showing the prediction performance over each window size, fig. 5.1, fig. 5.2, fig. 5.3, fig. 5.4, fig. 5.6, fig. 5.7, and fig. 5.8.

Only structured pruning and quantization affected the inference time in an observable way, as seen in table 5.5 and table 5.20 showing the inference time of each model. Structured pruning decreases the inference time with higher pruning percentage targets, and quantization increases the inference time across all hardware and software platforms. Table 5.20 also shows that combining the two optimizations yields slightly higher inference time than the baseline model but improves the inference time greatly compared to just using quantization.

The total number of committed instructions seen in table 5.8 and table 5.23

follows the same trend as the inference time, with only structured pruning and quantization having any significant impact on the total. Structured pruning decreases the total number of committed instructions across all platforms, while quantization increases the total. However, quantization shifts the distribution of committed instructions from heavily using vectorized floating-point instructions to using integer and vectorized integer instructions, as seen in section 5.1.3.1 and section 5.2.3.1 showing CPU instruction utilization.

The patterns in utilized memory bandwidth seen in table 5.14 and table 5.29 vary between software and hardware platforms. However, multiplying the bandwidth with the inference time, resulting in the memory traffic volume in table 5.15 and table 5.30, show clearer patterns. Quantization decreased the traffic volume when running on tflite and increased the traffic volume when running on onnx. Structured pruning decreased the traffic across all platforms, with greater pruning percentage targets resulting in higher reductions.

Some unexpected phenomena were encountered as well. In section 5.1.1 and section 5.2.1, we observed significant dips in F1-score for 200, 400, 800, and 1000 millisecond prediction windows for all models except the baseline and the quantized models. What is causing this is not known to us or further explored in this thesis, but a hypothesis is that the base model could be somewhat unstable at certain window sizes, perhaps due to some characteristics of the dataset, and is thus more sensitive to the modifications caused by some optimizations.

When looking at the inference time in both the simulated scenarios seen in table 5.5 and table 5.20, and the real-world tests seen in table 5.6 and table 5.21, we see that the quantized model always runs slower than any other model by 28.3% to 94%. This was unexpected as quantization is generally used to speed up inference time. The underlying causes of these slowdowns are not further explored in this thesis. However, one explanation can be found in table 5.8, showing the total number of committed instructions. Quantization adds operations at each layer in the network to handle quantization, dequantization, and scaling of input and output vectors. For the hardware tested in this thesis, it seems like the overhead of adding these extra instructions is greater than using just floating-point operations.

The quantized model running on onnx on ARM showed three orders of magnitude higher number of committed floating-point instructions in table 5.9 and table 5.24 compared to the baseline model. This was unexpected as quantization is generally used to lower or eliminate the amount of floating-point instructions to adapt the model to certain hardware configurations. The

cause is not explored in this thesis, but one hypothesis is that when quantizing the model, we still feed it input and expect output data in floating-point format, letting the inference framework take care of the conversion. This might lead the onnx framework to use a great amount of non-vectorized floating-point instructions to convert between floating-point and integer vectors.

6.2 Effects of Software Optimizations

Each optimization affected the results in different ways, but the main differentiating factor between most test cases when it comes to hardware utilization was whether or not quantization and/or structured pruning was applied. With the knowledge from the background study, it was surprising to see that quantization increased the inference time across the board, no matter the hardware or software platform used, or if the hardware was simulated or real. One explanation for this can be found in the instructions counts, with quantization the number of executed instructions increases dramatically. We expected a reduction in the number of floating-point instructions executed, which we only saw on tflite, but the major increase in integer instructions clearly overshadowed any possible gain we got on the floating-point part. This might be due to tflite not being able to vectorize the integer instructions as much as the floating-point instructions, and not utilizing the CPUs vector (SIMD) capabilities as much as would have been needed.

Quite unexpectedly, the onnx-based inference platform did not only increase the number of integer instructions but also increased the number of floating-point instructions by more than 100 times. What this is due to is hard to say, as it is highly dependent on the implementation details of the runtime. The only real improvement that quantization brought to the table was regarding memory handling. As seen in table 5.14 and 5.29, running the model with tflite gave a real improvement to the amount of bandwidth used by about 62% compared to the baseline. But this came with a cost of inference time (as seen in table 5.5 and 5.20 where quantization increases the inference latency by 48%-50%).

To make sure that the reduction in bandwidth is greater than the increase in inference time, we can look at the inferred tables 5.15 and 5.30. This shows that the slower inference time is not the only contributor to the reduction in bandwidth, but that less memory was read and written to during an inference as the memory traffic goes from the baselines 57.78MB to the quantized models 32.28MB.

Applying structured pruning always improved the hardware characteristics

and inference time. However, looking at fig. 5.3 (and all figures in section 5.2.1) we can observe that the F1-score is quite a bit lower than the baseline model and that the results are quite inconsistent over different prediction windows. This could be attributed to several factors. One possible explanation is the channel selection method used, max response. This isn't necessarily the best channel selection method as some previous research has shown [23]. It was mostly chosen as part of this thesis for its ease of implementation and seemingly good enough results. However, other channel selection methods could perform better and thus preserve the prediction performance better.

An alternative explanation may lie in the broader optimization framework provided by tensorflor-model-optimization, within which the implementation from this thesis was integrated. This library prunes all layers to the same sparsity individually. However, previous research has shown that this might not be the best method if one wants to preserve predictive performance. The publication by He *et al.* [23] showed that the earlier layers of the model are less subjective to performance degradation than deeper layers in the network when pruned and can thus be pruned more aggressively and deeper layers. Thus pruning all layers the same amount might not be optimal with regards to model prediction performance.

Neither unstructured pruning nor clustering had any significant impact on the hardware utilization. This is surprising as the background material shows great promise for these methods. An explanation for this could be the inference backends (tflite and onnx) used and their support for different kinds of optimizations. Both pruning and clustering require special software implementations in order to gain efficiency. For pruning to be effective, algorithms specialized for sparse matrix multiplication must be used, as many of the weights are zeroed out. Additionally, the model must be resilient to the removal of many of its components. Similarly, for clustering to be effective, matrix multiplication algorithms capable of looking up values in a lookup table during execution must be used, rather than simply unpacking the tabulated matrix and applying standard dense matrix multiplication methods.

From what the results show, it seems like neither tflite nor onnx has the tools to recognize that these kinds of optimizations are possible or the tools to utilize these optimizations. For this thesis, we used the default CPU backends for both tflite and onnx, but these optimizations/tools might be available on other more specialized compute backends. There are also other methods for "shrinking" the size of the model other than making the model more sparse as with pruning, knowledge distillation/transfer learning might be a more

successful approach to achieve this according to Sanh *et al.* [35], as noted on an issue on one of tensorflow's GitHub repositories [36].

6.3 Applicability for Real World Sensing

Deploying a sensing system like this one in a real-world scenario, would one find any advantage in using software optimizations on the model itself? The results in this thesis show arguments against using some kinds of software optimizations, unstructured pruning and clustering showed no effects on the runtime characteristics of the model inference, and thus provided no advantages over the baseline model. These methods of optimization might provide some advantages when it comes to compressing and storing the model. However, this was not explored in this thesis work as we were only concerned about inference time characteristics.

If memory bandwidth is a strict restriction, then quantization can be utilized. To ensure that quantization achieves the intended effects, it is advisable to conduct multiple experiments on the relevant hardware and software stacks. As we have shown in the previous chapters, the results may vary quite a bit between different setups. Structured pruning had a substantial impact on the inference characteristics, lowering almost all measured data points. We showed that combining quantization and structured pruning could greatly decrease the memory volume and bandwidth used compared to the baseline model. Using structured pruning could also offset some of the loss in inference time introduced by quantization, making this combination especially interesting for embedded systems where quantization might be necessary to run the model.

However, as seen in sections 5.1.1 and 5.2.1 where the structurally pruned models F1-scores range between 0 and 0.673 compared to the baselines score of 0.709, can structured pruning heavily impact the model's prediction performance. Proper testing should be conducted to evaluate if this is okay for the application in question. But regardless of the optimizations used, no model faced any issues running on the restricted simulated hardware.

The real-world inference time of a model depends greatly on the specific CPU and how well it handles integer, float, and vector operations. From tables 5.9, 5.10, 5.11, 5.25, 5.24, and 5.26, we see that integer and vector-based instructions are especially important to handle well as they accounted for about 54% and 41% of committed instructions respectively. The vector instructions were over 80% floating-point-based. However, that shifted when applying quantization to 95% and 62% integer-based vector instructions on

tflite and onnx respectively. Looking at the simulated numbers in tables 5.5 and 5.20 and the real-world numbers in 5.6 and 5.21, we can see that most models took about 20ms to 30ms per inference when running on simulated and real hardware. Thus, it seems reasonable for the sake of this argument to assume that an inference with this model would take about 30ms in a real-world scenario. To see if this would be usable, let us consider an example. Let us say that we have a bus traveling down a city street at 50 km/h. In 30ms that bus travels about 0.417 meters, or about 41.7 cm. This is not an especially long distance and would allow the system to react relatively fast to changes in the environment.

Having an inference time of about 30ms (or anything lower than 50ms) would potentially allow us to run several versions of the model back to back on the same data frame from the radar before the next frame arrives. This could open up some interesting possibilities regarding usage. One could potentially only run the model to predict a blockage within the next coming 1000ms, every 500ms or so to save power. If a blockage is predicted one can start anticipating it by running predictions for the next coming 100ms and reassuring that it will happen by predicting the next 1000ms at the same time (or rather one after the other). When the blockage goes away, one could start doing the longer time-frame predictions again every 500ms to once again save power.

The applicability of this solution is not only dictated by the models themselves but also by the limitations in the sensing medium itself. In the dataset used for this thesis, the difference between time-step t and $t+1$ has been 100ms. This was a limitation by the radar that was used in the construction of the dataset, as it only produces 10 data-frames a second. In the world of telecommunications and baseband, 100ms is considered relatively low-resolution. Current baseband networks use 1ms time slices when dictating traffic to the users, and only knowing when a blockage will happen in 100ms windows could lead to some inefficiencies in the system. Let us say that we detect a LOS blockage within the next 100ms, so we decide to change to a slower but more obstruction-tolerant transmission channel. If the actual blockage does not happen until 90ms into the prediction window, have 90ms of potential high bandwidth communications with the client been ”wasted”. The same goes for when a blockage passes, if we predict that the LOS blockage will continue to exist within the next 100ms, but the blockage ends after 15ms, we once again have a 85ms period before we decide that the blockage is over and switch to a higher bandwidth channel. As also mentioned in the publication by Demirhan and Alkhateeb [1], the low angular resolution of the radar makes it hard to predict blockages within small time-frames. From an execution time

and memory point of view, there is no technical limitation to using the model for predicting shorter intervals, but rather a business decision if it is worth the development effort.

Chapter 7

Conclusions and Future work

To conclude this thesis, we draw some conclusions from the results and discussions in the previous two chapters, examine some limitations of the work, and consider what needs to be done to build upon the knowledge gathered in this thesis.

7.1 Conclusions

In this thesis, we have shown that it is possible to run deep neural network based algorithms for LOS blockage sensing on hardware suitable for baseband systems. Using radar as the sensing medium, the deep neural network model provides good accuracy and can be executed in a time span that is shorter than the update period of the radar itself, potentially allowing for multiple predictions to take place in one radar update.

We have also shown that applying pruning or clustering as a means for software optimization of a model only yields a small difference in prediction performance but does not affect the hardware requirements of the model. Quantization on the other hand was shown to greatly impact the total number, as well as the type of CPU instructions used for inference, and the memory bandwidth requirements, depending on the software platform used. With the quantized model running on tflite having an overall lower utilization of memory traffic and memory bandwidth, as well as a higher utilization of integer and vector instructions. But having an overall worse execution time than the unoptimized model.

The results also allow us to conclude that integer and vector instructions for floating-point numbers and integers play an essential role in the execution performance of the deep neural network model. Using a CPU capable of

handling these kinds of instructions effectively is essential to achieving good real-world performance and, in turn, low inference times.

The solution seems applicable for a real-world baseband scenario, as long as the somewhat coarse-grained temporal resolution of 100ms is an acceptable restriction for the system.

7.2 Limitations

Due to both the time and availability of models, this thesis only looked at one base model implementation. The model is non-trivial, utilizing several convolution layers and an LSTM layer. However, it is still a limiting factor of the thesis as the improvements each optimization can bring will differ somewhat depending on the model's structure.

A simulator was needed for hardware testing as access to real hardware was not possible. Though the simulator provided great analytical data on the execution, it is only a simulator and will differ from reality somewhat as it does not contain a perfect model of the world.

Originally, we wanted to include tests using a GPU accelerator and see how that was utilized and affected CPU utilization. But we never managed to get the GPU component of gem5 to work properly, making it impossible to run these kinds of tests. This limited us to only look at CPU and memory utilization.

7.3 Future work

Due to the breadth of the question about whether deep neural network-based algorithms are applicable to run in baseband systems, this thesis has covered only one small case. This section focuses on some of the work required to answer the overarching question further.

7.3.1 Optimization scaling over model sizes

In this thesis, we were only concerned with one model of a limited size. Applying the optimizations explored in this thesis might yield different results or at least scale differently if the model was of a different size. To investigate this, applying optimizations to several models similar to one another but of different sizes needs to be investigated. Especially large-size models would be interesting as these are more computationally expensive to run.

7.3.2 Apply software optimizations to other problems

There are several challenges within baseband systems besides sensing, where solutions based on deep neural networks need to be evaluated for real-world applicability. The same can also be said about sensing, where other sensing mediums might require different models. Optimizations' effects and utility might depend on the model structure, and this is something that can be further investigated in future work.

7.3.3 A more complex scenario

The model and dataset used in this thesis are only concerned with predicting the future LOS blockage between one base station and one client. A more realistic, but also more complex, scenario would be if one base station were serving multiple clients at once. This makes the problem more complex by requiring the blockage prediction model to be able to predict where the blockage takes place, not only when. This would require more data from this specific scenario in order to be properly evaluated and would require more work in creating a model as well. This new model would then also need to be evaluated whether it is applicable to run in a baseband system, and to what extent.

7.4 Sustainability

The main concern of this thesis is to evaluate AI as a solution to improve the efficiency and capacity of future baseband infrastructure. This has both economic and environmental value. Improved efficiency in the system allows carriers to utilize less hardware to serve more customers, reducing the cost of buying and deploying, as well as the environmental impact of producing the baseband hardware. This contributes directly to United Nations (UN) Sustainable Development Goals (SDGs) number 9, which is concerned with sustainable and resilient infrastructure.

Even though this thesis is only concerned with infrastructure efficiency, some ethical issues could be raised around this thesis. Firstly, simulating hardware is way more inefficient than running the test on real hardware. The simulations consume a lot more energy to run. This is partly due to the hardware used to run the simulations requiring more power to operate than the low-power hardware found in basestations. As well as the simulations requiring more time to run, running just a few simulated seconds can take

hours of real-world time. Running these inefficient simulations for each scenario adds up, especially when developing and as the simulations are tested multiple times. One could ask oneself if doing these kinds of experiments in simulated environments is ethically sound. It could be justified by creating knowledge that can improve the efficiency of large infrastructure systems (such as baseband) that run around the clock all over the world for several years, making the possible savings much greater than the cost of the experiments.

Another ethical issue one could raise is the impact of running machine intelligence on such a large scale. It could enable actors to apply machine intelligence in other areas that do not necessarily contribute to people's well-being and standard of living, such as mass surveillance or weaponry, as much as it could enable actors to apply machine intelligence in ways of improving efficiency and helping people. This issue boils down to whether or not AI is a force of good or bad for humanity, something that is highly dependent on perspective and values and is hard to answer without speculation. This is more of an issue of ethics and politics in the end. The industry as a whole is pushing for more AI in lots of sectors, contributing to making the future of AI deployments a bit more sustainable and efficient could thus be considered a good thing.

References

[1] U. Demirhan and A. Alkhateeb, "Radar aided proactive blockage prediction in real-world millimeter wave systems," in *ICC 2022 - IEEE International Conference on Communications*, Seoul, Korea, Republic of: IEEE, May 16, 2022, pp. 4547–4552, ISBN: 9781538683477. DOI: 10.1109/ICC45855.2022.9838438. [Online]. Available: https://ieeexplore.ieee.org/document/9838438/ (visited on 01/29/2024).

[2] S. Wu, C. Chakrabarti, and A. Alkhateeb, "LiDAR-aided mobile blockage prediction in real-world millimeter wave systems," in *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, Austin, TX, USA: IEEE, Apr. 10, 2022, pp. 2631–2636, ISBN: 9781665442664. DOI: 10.1109/WCNC51071.2022.9771651. [Online]. Available: https://ieeexplore.ieee.org/document/9771651/ (visited on 01/30/2024).

[3] G. Charan and A. Alkhateeb, "Computer vision aided blockage prediction in real-world millimeter wave deployments," in *2022 IEEE Globecom Workshops (GC Wkshps)*, Rio de Janeiro, Brazil: IEEE, Dec. 4, 2022, pp. 1711–1716, ISBN: 9781665459754. DOI: 10.1109/GCWkshps56602.2022.10008524. [Online]. Available: https://ieeexplore.ieee.org/document/10008524/ (visited on 01/30/2024).

[4] O. Calin, *Deep Learning Architectures: A Mathematical Approach* (Springer Series in the Data Sciences). Cham: Springer International Publishing, Feb. 13, 2020, ISBN: 9783030367206 9783030367213. DOI: 10.1007/978-3-030-36721-3. [Online]. Available: https://link.springer.com/10.1007/978-3-030-36721-3 (visited on 04/23/2024).

[5] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," Jun. 15, 2021. DOI: 10.48550/ARXIV.2106.08295. [Online]. Available: https://arxiv.org/abs/2106.08295 (visited on 01/29/2024).

[6] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, vol. 28, Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/hash/ae0eb3eed39d2bcef4622b2499a05fe6-Abstract.html (visited on 02/20/2024).

[7] M. Caro, H. Tabani, and J. Abella, "At-scale evaluation of weight clustering to enable energy-efficient object detection," *Journal of Systems Architecture*, vol. 129, p. 102 635, Jun. 30, 2022, ISSN: 13837621. DOI: 10.1016/j.sysarc.2022.102635. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S138376212200159X (visited on 07/12/2024).

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017, ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2017.2761740. [Online]. Available: http://ieeexplore.ieee.org/document/8114708/ (visited on 01/29/2024).

[9] J. Redmon and A. Farhadi, *YOLOv3: An incremental improvement*, 2018. DOI: 10.48550/ARXIV.1804.02767. [Online]. Available: https://arxiv.org/abs/1804.02767 (visited on 12/17/2024).

[10] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 31, 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. [Online]. Available: https://dl.acm.org/doi/10.1145/2024716.2024718 (visited on 05/14/2024).

[11] J. Lowe-Power *et al.*, "The gem5 simulator: Version 20.0+," 2020. DOI: 10.48550/ARXIV.2007.03152. [Online]. Available: https://arxiv.org/abs/2007.03152 (visited on 05/14/2024).

[12] TensorFlow Developers, *TensorFlow*, version v2.15.1, Mar. 8, 2024. DOI: 10.5281/ZENODO.4724125. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.4724125 (visited on 05/14/2024).

[13] ONNX Runtime developers, *ONNX runtime*, version v1.17.0, Feb. 3, 2024. [Online]. Available: https://onnxruntime.ai/.

[14] A. N. Mazumder *et al.*, "A survey on the optimization of neural network accelerators for micro-AI on-device inference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 4, pp. 532–547, Dec. 2021, ISSN: 2156-3365. DOI: 10.1109/JETCAS.2021.3129415. [Online]. Available: https://ieeexplore.ieee.org/document/9627710 (visited on 06/14/2024).

[15] W. Chen *et al.*, "Quantization of deep neural networks for accurate edge computing," *J. Emerg. Technol. Comput. Syst.*, vol. 17, no. 4, 54:1–54:11, Jun. 30, 2021, ISSN: 1550-4832. DOI: 10.1145/3451211. [Online]. Available: https://doi.org/10.1145/3451211 (visited on 07/11/2024).

[16] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2014. DOI: 10.48550/ARXIV.1409.1556. [Online]. Available: https://arxiv.org/abs/1409.1556 (visited on 12/17/2024).

[17] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Salt Lake City, UT: IEEE, Jun. 2018, pp. 2704–2713, ISBN: 9781538664209. DOI: 10.1109/CVPR.2018.00286. [Online]. Available: https://ieeexplore.ieee.org/document/8578384/ (visited on 02/14/2025).

[18] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2015. DOI: 10.48550/ARXIV.1510.00149. [Online]. Available: https://arxiv.org/abs/1510.00149 (visited on 07/10/2024).

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, Curran Associates, Inc., 2012, pp. 1106–1114, ISBN: 9781627480031.

[20] S. Liu, D. S. Ha, F. Shen, and Y. Yi, "Efficient neural networks for edge devices," *Computers & Electrical Engineering*, vol. 92, p. 107 121, Jun. 1, 2021, ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2021.107121. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790621001257 (visited on 06/14/2024).

[21] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," 2016. DOI: 10.48550/ARXIV.1607.03250. [Online]. Available: https://arxiv.org/abs/1607.03250 (visited on 07/10/2024).

[22] T. Jeong, E. Ghasemi, J. Tuyls, E. Delaye, and A. Sirasao, "Neural network pruning and hardware acceleration," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, Dec. 2020, pp. 440–445. DOI: 10.1109/UCC48980.2020.00069. [Online]. Available: https://ieeexplore.ieee.org/document/9302790/ (visited on 06/14/2024).

[23] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*, Venice: IEEE, Oct. 2017, pp. 1398–1406, ISBN: 9781538610329. DOI: 10.1109/ICCV.2017.155. [Online]. Available: http://ieeexplore.ieee.org/document/8237417/ (visited on 02/14/2025).

[24] H. Li, H. Samet, A. Kadav, I. Durdanovic, and H. P. Graf, "Pruning filters for efficient convnets," in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Jun. 2016, pp. 770–778. [Online]. Available: https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html (visited on 12/17/2024).

[26] K. Alex and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[27] G. Pitsis *et al.*, "Efficient convolutional neural network weight compression for space data classification on multi-fpga platforms," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Brighton, United Kingdom: IEEE, May 2019, pp. 3917–3921, ISBN: 9781479981311. DOI: 10.1109/ICASSP.2019.8682732. [Online]. Available: https://ieeexplore.ieee.org/document/8682732/ (visited on 07/08/2024).

[28] M. Caro and J. Abella, "Energy-efficient object detection: Impact of weight clustering for different arithmetic representations," *Journal of Signal Processing Systems*, vol. 96, no. 4, pp. 287–300, May 2024, ISSN: 1939-8018, 1939-8115. DOI: 10.1007/s11265-024-01917-8. [Online]. Available: https://link.springer.com/10.1007/s11265-024-01917-8 (visited on 07/08/2024).

[29] A. Alkhateeb *et al.*, "DeepSense 6g: A large-scale real-world multi-modal sensing and communication dataset," *IEEE Communications Magazine*, vol. 61, no. 9, pp. 122–128, Sep. 2023, ISSN: 0163-6804, 1558-1896. DOI: 10.1109/MCOM.006.2200730. [Online]. Available: https://ieeexplore.ieee.org/document/10144504/ (visited on 03/10/2024).

[30] *Radar-aided blockage prediction overview*. [Online]. Available: https://www.deepsense6g.net/wp-content/uploads/2022/02/main_fig_radar_blockage-2048x800.png (visited on 03/10/2024).

[31] "CC BY-NC-SA 4.0 deed | attribution-NonCommercial-ShareAlike 4.0 international | creative commons." (), [Online]. Available: https://creativecommons.org/licenses/by-nc-sa/4.0/ (visited on 03/10/2024).

[32] M. Al-Quraan *et al.*, "Enhancing reliability in federated mmWave networks: A practical and scalable solution using radar-aided dynamic blockage recognition," *IEEE Transactions on Mobile Computing*, vol. 23, no. 10, pp. 10 146–10 160, Oct. 2024, ISSN: 1536-1233, 1558-0660, 2161-9875. DOI: 10.1109/TMC.2024.3373529. [Online]. Available: https://ieeexplore.ieee.org/document/10460129/ (visited on 11/20/2024).

[33] Q. Zhou, Y. Gong, and A. Nallanathan, "Radar-aided beam selection in MIMO communication systems: A federated transfer learning approach," *IEEE Transactions on Vehicular Technology*, vol. 73, no. 8, pp. 12 172–12 177, Aug. 2024, ISSN: 1939-9359. DOI: 10.1109/TVT.2024.3373496. [Online]. Available: https://ieeexplore.ieee.org/document/10463115/ (visited on 12/02/2024).

[34] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *2015 IEEE International Symposium on Workload Characterization*, Atlanta, GA, USA: IEEE, Oct. 2015, pp. 183–192, ISBN: 9781509000883. DOI: 10.1109/IISWC.2015.29. [Online]. Available: http://ieeexplore.ieee.org/document/7314164/ (visited on 05/06/2024).

[35] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2019. DOI: 10.48550/ARXIV.1910.01108. [Online]. Available: https://arxiv.org/abs/1910.01108 (visited on 05/21/2024).

[36] TensorFlow Developers, *Sparsity runtime integration with TF/TFLite for latency improvements*, May 21, 2024. [Online]. Available: https://github.com/tensorflow/model-optimization/issues/173.

TRITA-EECS-EX-2025:70
Stockholm, Sweden 2025

www.kth.se