MASTER'S THESIS 2020

Optimizing Machine Learning Inference for MCU:s

Josefine Myllenberg, Jens Johansson

ISSN 1650-2884
LU-CS-EX: 2020-46

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-46

Optimizing Machine Learning Inference for MCU:s

Josefine Myllenberg, Jens Johansson

Optimizing Machine Learning Inference for MCU:s

Josefine Myllenberg
josefine@myllenberg.se

Jens Johansson
dat15jj1@student.lu.se

July 30, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Flavius Gruian, flavius.gruian@cs.lth.se
Supervisor: Johan Björnstedt, johan.bjornstedt@acconeer.com
Supervisor: Martin Löwegren, martin@lowegren.nu

Examiner: Jörn Janneck, jorn.janneck@cs.lth.se

Abstract

Deep neural networks come with high demand for storage and computational resources, which makes it difficult to deploy deep convolutional neural networks on limited resource devices. This thesis investigates different approaches of how to reduce the size of a network in order to run it on limited resource devices, while keeping the accuracy close to the original network. The network used is a gesture detection network consisting of six convolutional layers, which is converted to C code using X-CUBE-AI and Keras2C in order to run on an MCU. The pruning techniques channel-based pruning and magnitude-based pruning are then applied to reduce the network size and inference time. Results show that the execution time can be reduced by up to $8\times$ and memory usage by up to $4.5\times$ for flash and $2.8\times$ for RAM. Quantization is also applied, reducing the network parameters from 32-bit floating points to 8-bit integers which results in a reduction in execution time by $2.5\times$ and memory usage by $4\times$ for flash and $3.7\times$ for RAM. Both the pruning and quantization optimizations are applied while losing no more than 1.5 percentage points of accuracy compared to the original network.

Keywords: Convolutional Neural Networks, Embedded Systems, Micro Controller Unit (MCU), Inference, Memory optimization, Speed optimization

Acknowledgements

We would like to thank Acconeer for letting us write our master's thesis here. Helpful, supportive, and interested colleagues has made this a great place to write our thesis. We also want to thank our supervisors at Acconeer, Martin Löwegren and Johan Björnstedt, for showing great dedication and eagerness to help. You always took the time to answer our questions and help us when we got stuck.

Finally, we want to thank our supervisor at LTH, Flavius Gruian, for always pointing us in the right direction and providing us with relevant sources of information. We really appreciate your advice.

Contents

1 Introduction **7**
1.1 Problem Description 7
1.1.1 Trade-off: Cost - Performance 7
1.1.2 Research Questions 8
1.2 Contributions 8
1.3 Division of Work 8
1.4 Outline of the Report 9

2 Background **11**
2.1 Convolutional Neural Networks 11
2.1.1 Convolutional layers 11
2.1.2 Filters, Kernels, and Channels 12
2.2 Pruning 13
2.2.1 Magnitude-Based Weight Pruning 13
2.2.2 Channel-Based Pruning Using L1-norm 14
2.3 Quantization 14

3 Related Work **17**
3.1 Pruning 17
3.2 Quantization 18

4 Method **21**
4.1 Approach 21
4.1.1 Optimizing the X-CUBE-AI Model 22
4.1.2 Optimizing the Keras2C Model 22
4.2 Methods for Measuring 23
4.3 Tools 23
4.3.1 Nucleo F722ZE 24
4.3.2 Acconeer Python Exploration Tool 24

4.3.3 TensorFlow and Keras 24
4.3.4 STM32CubeIDE and X-CUBE-AI 25
4.3.5 Keras2C 25
4.3.6 Keras-surgeon 25
4.3.7 Perf 25

5 Experiments **27**
5.1 Setup 27
5.1.1 Our Model 27
5.1.2 Tool Versions 29
5.1.3 Optimization Techniques Setup 30
5.1.4 Experiments Setup 32
5.2 Results 33
5.3 Discussion 39
5.3.1 Answer to Research Questions 41
5.3.2 Ethical Aspects 41
5.3.3 Threats to Validity 42

6 Conclusions **43**
6.1 Summary 43
6.2 Future Work 44

Bibliography **47**

Appendix A Data tables **53**

Appendix B Counting clock cycles on the Cortex M7 **59**

Appendix C Model **61**

Chapter 1

Introduction

Convolutional Neural Networks (CNNs) are becoming more and more popular due to their high accuracy and machine efficiency for performing image predictions [6]. Moreover, with deep neural networks becoming more sophisticated, their size and complexity increase as well. An ongoing challenge is to bring machine learning inference to embedded devices with limited computational power and memory size. Storage complexity as well as computational complexity and energy consumption has become important considerations when designing neural networks, particularly when deploying the network at limited resource devices.

1.1 Problem Description

Acconeer is a radar company in Lund, Sweden, that offers high range resolution radar sensors with low power consumption. They offer products such as radar sensors, radar modules, and development kits. In previous work, Acconeer has developed machine learning models for gesture-based control [4]. These models were only run on a PC, which usually has sufficient flash memory and RAM as well as overall computing power. We want to run these models on a Micro Controller Unit (MCU), which have limited resources when it comes to memory and computing power. To run these models in products with limited resources it needs to be investigated if and how the models can be optimized to make good enough predictions on lower spec MCU:s.

1.1.1 Trade-off: Cost - Performance

The MCU cost and MCU performance are connected in the terms that when the performance requirements increase, the price of the MCU increases as well. In this case, performance requirements correspond to CPU clock frequency, as well as RAM and flash memory. The

result will therefore be a trade-off between MCU cost and inference performance. Metrics to consider:

- **Cost:** The MCU cost shall be kept as low as possible. This will impact the performance of the MCU, specifically in terms of CPU performance and memory size
- **Inference Performance:** The user experience shall be similar to the model used before the MCU optimizations. The prediction accuracy shall be about the same

1.1.2 Research Questions

This master's thesis will investigate if it is possible to reduce the size of a machine learning model and the resources needed for inference without losing too much accuracy. This means that the machine learning model should still be able to predict gestures correctly and in real time, however, it is acceptable if the network loses some certainty about its decision.

In order to achieve this, the following research questions are composed:

RQ1: How does pruning impact inference regarding performance and memory footprint?

RQ2: How does quantization impact inference regarding performance and memory footprint?

RQ3: How is the accuracy affected when applying pruning and quantization to a neural network?

1.2 Contributions

This master's thesis contributes to:

- Knowledge in using tools for pruning and quantization to optimize Keras machine learning models for embedded systems
- A better understanding of system requirements, such as CPU speed and memory usage, when running machine learning algorithms on resource limited edge devices
- Finding a suitable MCU for gesture-based control with Acconeer radars

1.3 Division of Work

Most of the work in this Master's Thesis has been carried out in close collaboration between the authors. In the experiment phase, both authors have been working on the different optimizations. Sometimes pair programming was applied, and sometimes different approaches were tested in parallel, particularly when we did not exactly know how to take on a task and needed to try different approaches until success. The result of this is that Jens Johansson has been more responsible for the TensorFlow/Keras code, and Josefine Myllenberg more responsible for the C code. In the end, however, both of the authors have ultimately been involved in all parts of the experiments and coding.

In the report writing phase, Jens Johansson has mainly been responsible for writing the related work, results, and appendices. Josefine Myllenberg wrote most of the abstract, introduction, and background theory. Discussion and conclusions have been written together, as well as the method chapter and the setup section.

1.4 Outline of the Report

Here in the introduction chapter, we have explained the background to the problem as well as our research questions. We have stated what our work can contribute to, and how we divided the work during this project. In the second chapter, *Background*, we introduce the background theory used in this thesis. The third chapter, *Related Work*, talks about other work in the field of optimizing neural networks that is somehow related to our work. In chapter 4, *Method*, we explain the approach we used to answer the research questions and the tools used in our experiments. Chapter 5, *Experiments*, shows the setup for all experiments that were carried out, the results, and discuss them and any threats to validity that may have occurred. The final chapter, *Conclusions* contains a summary, our conclusions from the experiments, and points out some future work.

Chapter 2

Background

In this chapter, we introduce the theoretical background and theories used in this thesis. We start with the concepts of Convolutional Neural Networks in section 2.1, which are needed for understanding the optimizations that follow. Pruning is explained in section 2.2, which is a way of reducing the size of the network by removing parameters that adds little to the classification result. In section 2.3 the optimization technique called quantization, which mainly aims to reduce the memory usage of the model, is explained.

2.1 Convolutional Neural Networks

Convolutional Neural Networks are networks specialized in processing data with a grid-like topology [5], and are therefore commonly used in image recognition. The input to the CNN can typically be an image, and each convolutional layer in the network will then extract certain features from this image. In this thesis, the radar data is converted to colored images by a preprocessing algorithm, before being fed to the network. Convolutional layers are explained in section 2.1.1. The feature extraction is done by a filter, this is further explained in section 2.1.2.

2.1.1 Convolutional layers

All neural networks consist of layers, but what differs in CNNs are that they use convolutional layers. Convolutional layers convolve the input - hence the name - and pass the result to the next layer [13].

Convolution is a mathematical operation on two integrable functions f and g that creates a third integrable function h. The function h can be seen as a weighted average of the function

f at the moment t where the weighting is given by g. As t changes, the weighting function emphasizes different parts of the input. However, t does not have to represent the time domain.

Convolution is written as the following $h(t) = f(t) * g(t)$ where the $*$ denotes the convolution operation, and $f(t) * g(t) = \int_{-\infty}^{\infty} f(t - \tau)g(\tau)d\tau$.

In convolutional networks the first argument (the function f described above) is often referred to as *input*, and the second argument (function g described above) as the *kernel* [5].

2.1.2 Filters, Kernels, and Channels

Each input to a layer in a CNN has a height and width, but also one or several *channels*, where each channel contains some part of information about the input. Only together, the channels will represent the input. As an example, a colored image may contain three channels: R (red), G (green), and B (blue). Hence, we can view input data as 3D-objects that each layer operates on, with the dimensions $H \times W \times C$, where H = image height, W = image width, and C = number of channels. Depending on layers used in the network, the height, width, and number of channels can reduce or expand when moving from one layer to another [23].

Every convolutional layer acts as a detection *filter*, which searches for specific patterns (usually called features) in the input image. A feature can typically be angles, straight lines, or corners, and the deeper the network, the more sophisticated the filters become. Once the filter have been applied on the input image, we get an output which is called *feature map* [5][13]. The concept of a convolutional layer with kernel, input, and feature map is shown in figure 2.1.

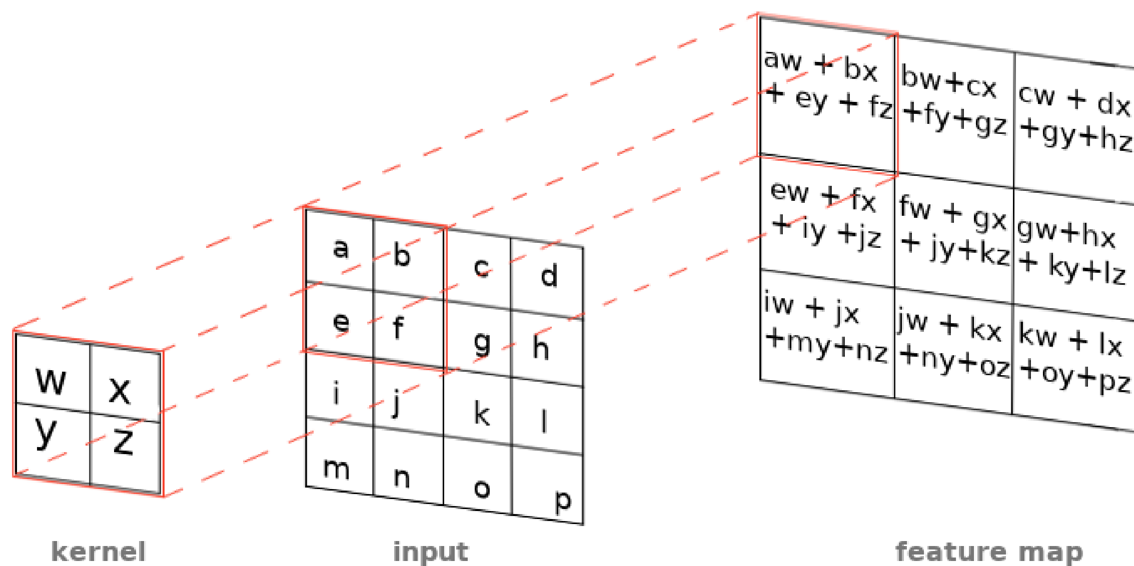


Figure 2.1: The convolution operation for one channel in a CNN.

Sometimes the word filter and kernel are used interchangeably, but here we view a filter as a concatenation of one or more kernels, which are stacked on top of each other. They can be viewed as a pile of papers where each kernel is represented by a single paper, the whole pile then represents the filter, and hence the filter is also a 3D-object. All kernels in a filter have

the same width and height, and the number of kernels decides the depth of the filter. Kernels are $w \times h$ sized matrices containing weights (where $h \leq H$ and $w \leq W$, and each kernel slides over the corresponding input channel to compute the feature map using convolution. The depth of the convolution filter (number of kernels) must be equal to the number of channels in the input feature map, since each kernel is assigned to a corresponding channel in the input [12].

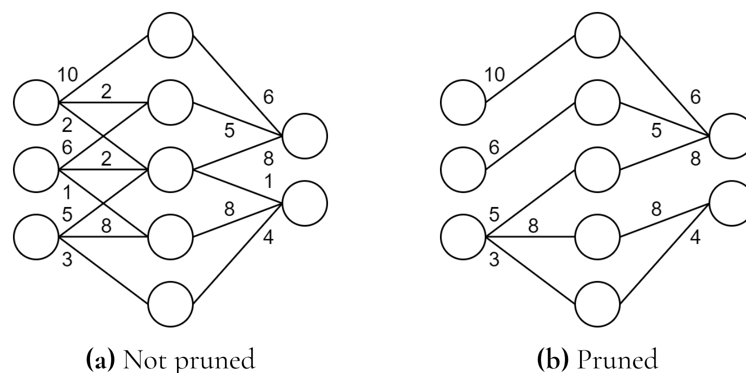
2.2 Pruning

When it comes to neural networks, pruning refers to the action of removing or compressing layers in a network with the goal of reducing the model while losing as little of the original accuracy as possible. This allows the network to become smaller and faster[14] which results in fewer computations in inference and a reduced physical size [26]. For this experiment, we have explored two different approaches to pruning called channel-based pruning [7] and magnitude-based weight pruning [26].

2.2.1 Magnitude-Based Weight Pruning

Magnitude-based pruning is the simplest weight pruning algorithm and uses an element-wise approach; it applies a threshold function on each weight tensor in each layer. After each training epoch, the link with the smallest weight is removed. It is possible to apply different thresholds on different layers if needed. See figure 2.2 for a visual explanation of how the network can be illustrated before and after removing the weights.

It has the benefit over the other pruning technique explored in this thesis, channel-based pruning, being more generally applicable, and does not depend on the network structure. This is because the weights to be removed are chosen by their significance to the final result of inference [26].



(a) Not pruned

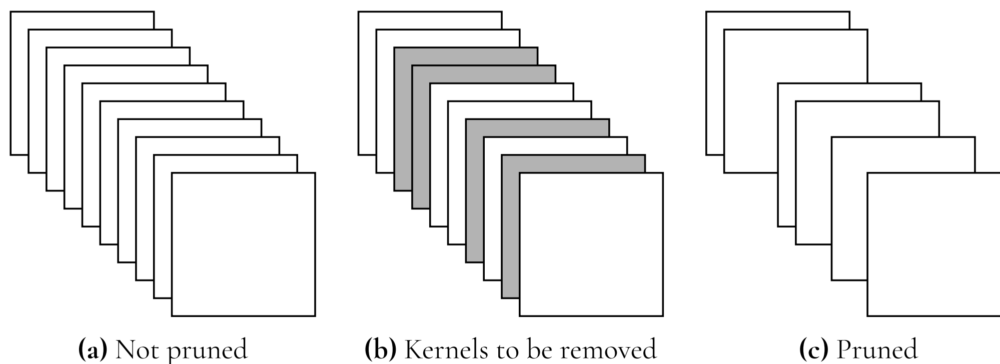
(b) Pruned

Figure 2.2: Illustration of a CNN that is pruned with magnitude-based weight pruning. Connections with a weight lower than 3 are removed.

2.2.2 Channel-Based Pruning Using L1-norm

Channel-based pruning is when channels are physically removed from layers in a model. In this thesis, the L1-norm is used to select the layers to be removed. The L1-norm is equivalent to summarizing the absolute value of a filter's weights. This tells us that the filter with the lowest value of the L1-norm contributes the least to our final result and is therefore most appropriate to remove. See figure 2.3 for a visual explanation of how channels are removed from a filter, where the rectangles represent kernels in a filter and the kernels with the lowest L1-norm are removed.

Channel-based pruning has the advantage over weight-based pruning that the benefits of lower computation times can be exploited in more cases since special libraries or implementations of layers are not needed. This is because the general structure of the network stays the same. In the case of weights pruning the weights that are removed are often selected in an irregular order, which makes it more difficult to physically remove the weights while exploiting the benefits of a sparse matrix. For example, a tensor where only some of the weights are set to 0 by weight pruning, cannot be removed from the network structure. Correspondingly, if all weights in the tensor are set to 0 it can be removed from the network, resulting in a smaller network that executes faster. There will therefore be a need for special libraries that exploit sparse matrices or modifications to the inference back-end when using magnitude-based weight pruning [15].



(a) Not pruned **(b)** Kernels to be removed **(c)** Pruned

Figure 2.3: Illustration of channel pruning a filter in a neural network.

2.3 Quantization

Quantization is the process of converting a continuous, often large, range of numbers to a smaller constrained range of discrete numbers. This means that several numbers from the continuous range can be represented by the same number in the discrete range.

The most commonly used numerical format in neural network models are 32-bit floating points. However, using 16-bit floating points or even 8-bit integers to represent weights

and activation functions has proven to be possible without a significant loss in accuracy for inference [9]. For the methods used in this thesis, the model is quantized after training and some of the training data is used for optimizing the quantization parameters. There is also an option to retrain the quantized model to achieve a higher inference accuracy. However, retraining is not used for the quantized model in this thesis since the accuracy was kept high after quantization [11].

With decreasing bit-width, the memory storage requirements decrease as well since fewer bits are needed to store the information. By using 8-bit integers instead of 32-bit floating points, the storage requirements for a model can theoretically be reduced by $4\times$. Furthermore, the number of computations during inference decrease, making the execution time faster, since integer computations often are less resource-demanding for the processor than floating point computations. The results will depend on the platform used and how well that platform operate using floating points compared to integers [3].

Chapter 3

Related Work

In this chapter, we present some previous research conducted in the fields of optimizing machine learning models for inference, and running machine learning inference on MCUs. There have been many studies in these fields and we choose to present some that are relevant to our optimization methods, pruning and quantization, in sections 3.1 and 3.2 respectively.

3.1 Pruning

Various studies have explored the results of pruning neural networks in order to lower the computation power and memory usage required for inference. *To prune, or not to prune: exploring the efficacy of pruning for model compression* by Zhu et al. [26] is one of them. It confirms that pruning a model using weight pruning can reduce the execution time and model size significantly without losing much accuracy. The study was conducted with the aim to compare how a previously large model that is pruned, compares to a smaller dense model in terms of execution time, memory usage, and accuracy. One important conclusion was that a large model that has been pruned to become sparse, outperforms the smaller dense model in terms of accuracy and efficiency on a wide range of existing network architectures. This means that it is possible to achieve better results by pruning a large model than training a small model from the start. This is useful for the work carried out in our thesis, since it proves that pruning can be useful even though you have the possibility to design your model from the beginning. If the model is designed to perform with as high accuracy as possible and then is pruned, it can outperform a model that was optimized more in the design stage. In the pruning experiments carried out in our thesis, both channel pruning and magnitude-based weight pruning are conducted. The conclusions from this paper argue for the advantages of magnitude-based weight pruning over channel-based pruning since channel pruning does not produce sparse matrices.

Convolution-Weight-Distribution Assumption: Rethinking the Criteria of Channel Pruning by Huang et al. [7], which compares different techniques for the pruning criteria in channel pruning, is another study on the area of pruning that connects to our thesis. The report finds that the filters in a convolutional layer approximately follows a Gaussian-like distribution, which means that most of the primary pruning criteria that exist today will achieve similar pruning results. The L1-norm criteria used for channel pruning in our thesis is included in those. Furthermore, for networks with a large number of filters, and therefore a lot of redundancy between the convolutional layers, the tested criteria can not determine the importance of the filters. This paper was used in our work when selecting the pruning criteria for channel pruning. It confirms that our method of using the L1-norms for selecting the channels to be pruned has no significant impact on the pruning results compared to using different criteria.

We read other papers about pruning as well, they were however not very useful for us in this work. One of those papers is *Runtime Neural Pruning* by Ji Lin et al [17], where they use a decision network that decides at runtime how a second neural network should be pruned. This is not applicable in our case since our platform is an embedded system with a very small memory space.

Another paper addressing the same area, but not applicable in our case, is *Pruning by Explaining: A Novel Criterion for Deep Neural Network Pruning* by Yeom et al. [25], where a new pruning criterion for CNN pruning is proposed. The criteria use the activations from each layer in inference to determine the relevance of an element in the network. However, the machine learning model used in our thesis is simple enough that simpler and more widely used criteria for pruning will lead to good results. Our thesis is not about finding the optimal criteria for pruning, as much as studying the effects of pruning on our machine learning model when deployed on an MCU.

3.2 Quantization

There have been several different studies on quantization of neural networks, that explore the possibility to improve the execution time and memory usage of inference using different quantization techniques. One of them is *Quantizing deep convolutional networks for efficient inference: A whitepaper* by Raghuraman Krishnamoorthi [11], where different quantization techniques are benchmarked. Some conclusions were that quantization, in general, can reduce the execution time by up to $3\times$ and memory usage by $4\times$ when quantizing from 32-bit floating points to 8-bit integers.

Another article, *FQ-Conv: Fully Quantized Convolution for Efficient and Accurate Inference* by Verhoef et al. [24], shows that specific techniques can be applied, to reduce the drop in accuracy when performing a full model quantization. The network is quantized during training in order to achieve an accuracy as high as possible. To achieve this they introduce a technique called gradual quantization, which gradually lowers the bit-width of the quantized weights and connections for the network to more easily recover from the accuracy loss of quantization during training. The results showed that the accuracy was very close to the original accuracy and that quantization therefore can be applied, if done correctly, without impacting the accuracy of a network significantly.

These two papers were helpful since they both show that quantization has a positive impact on execution time and memory usage of inference. This is one of the reasons we decided to use quantization despite the possible negative impacts on the accuracy of the model. Other articles, also confirming the benefits of quantization, are *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference* by Benoit Jacob et al. [8] and *Low-bit Quantization of Neural Networks for Efficient Inference* by Yoni Choukroun et al. [3]. They both show that quantization has a positive impact on the memory usage of the model as well as the execution time of inference.

Another paper on quantization, *Fixed Point Quantization of Deep Convolutional Networks* by Darryl D. Lin et al. [16], explores the benefits of using fixed point implementations of CNNs to optimize them for deployment on embedded hardware. The results were promising with no reduction in accuracy for the CIFAR-10 benchmark. However, this study was not useful for us since our model is simple enough to use integer quantization.

Chapter 4

Method

In this chapter, we describe the method used in this thesis. The approach to which the experiments were conducted is described in section 4.1, how we measure the results is described in section 4.2, and the tools used are described in section 4.3.

4.1 Approach

The neural network used in this thesis is equivalent to the network designed in the master's thesis *Hand Gesture Classification using Millimeter Wave Pulsed Radar* by Eda Dagasan [4], which is designed specifically for gesture prediction on Acconeer radar data. Since the focus of this thesis is optimizations, our approach was to use an already defined model design for gesture detection with Acconeer radar and train it to fit data collected by us. Hence we chose the hand gesture model by Dagasan. We collect training, test, and validation data and train the model using the Acconeer Exploration Tool with a Keras backend. This approach is chosen by us in order to have complete knowledge and control of the gestures when conducting the experiments.

We then save the trained model as a Keras model that has to be converted to C code to run on the selected MCU. We explore different options and their strengths and limitations, by selecting two different ways of converting the model, X-CUBE-AI and Keras2C. These are two different approaches to run a Keras model on an MCU, and a further description of them can be found in section 4.3.4 and 4.3.5. We also explored other options but found that Keras2C and X-CUBE-AI are the most well-documented as of today. A flowchart of our approach can be seen in figure 4.1. The work in Acconeer Exploration Tool is done once, to get a baseline. The main work in this thesis is done in Python with Keras and Tensorflow, and in STM32CubeIDE with X-CUBE-AI and Keras2C.

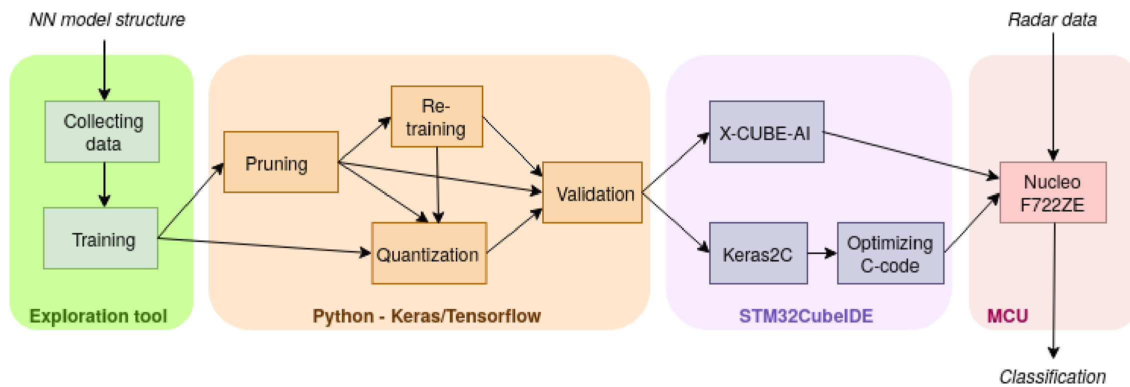


Figure 4.1: A flow chart of the workflow in this project.

For the models to run more efficiently and reduce the execution time and memory footprint, the optimization techniques mentioned in chapter 2 are explored. The approaches of these optimizations vary between the X-CUBE-AI and Keras2C models and the approach of these optimizations are further described in sections 4.1.1 and 4.1.2

For all optimizations, execution time and memory usage data are collected and calculated according to section 4.2. The results are then presented, compared, and discussed in chapter 5.

4.1.1 Optimizing the X-CUBE-AI Model

Since X-CUBE-AI is an already compiled library there are no possibilities for manual optimizations of the C code of the model. Both pruning and quantization apply to this approach. It is, however, not possible to apply further optimizations by manually altering the C code.

The pruning techniques tested by us are magnitude-based weight pruning in TensorFlow, and channel-based pruning using Keras-surgeon. However, since there is no access to the C code of the model, the benefits of magnitude-based weights pruning can not be exploited. We test various pruning parameters, and evaluate the execution time and memory footprint of the models.

In order to quantize the models from 32-bit floating points to 8-bit integers, we use the TensorFlow Lite conversion tool. We quantize all of the pruned models, and evaluate the execution time and memory footprint. At last, we compare the quantized models to the non-quantized models to achieve an understanding of how an 8-bit model compares to the 32-bit model.

4.1.2 Optimizing the Keras2C Model

It is possible to modify the generated C code from the Keras2C library, which means that the inference can be optimized after conversion. We profile the model using perf [18] while running the model on a Linux PC, to give a clear hint of where the optimizations should be applied. Since perf shows that over 98% of the total amount of computations are done in the

convolutional layers, time is spent on optimizing the convolution function. We also profile the model on the MCU by counting clock cycles for each layer, as explained in section 4.2.

The model is not quantized for Keras2C since we were not able to find any ready-to-use tools to quantize a Keras model, and because of the time frame of this thesis, we do not have time to implement one ourselves.

The pruning techniques we use are both weight-based pruning using the TensorFlow Optimization Tool, and channel-based pruning using Keras-surgeon.

4.2 Methods for Measuring

We measure the execution time and memory footprint of a model using STM32CubeIDE when connected to the MCU.

The build analyzer in STM32CubeIDE displays the statically allocated memory of both flash and RAM. In order to measure the memory used by only the model, we create a reference project containing all features except the features related to the model. The memory usage for that project is then subtracted from the final memory usage of the projects containing the models. The memory usage of a model is also available from X-CUBE-AI in the configuration section.

Using 20 test frames from the validation data, five for each gesture, we calculate the execution time by counting the average clock cycles during their execution. We use the same 20 test frames for all experiments, to give a fair comparison between the different setups. The number of 20 test frames is chosen since it allows for a wide enough spread between the gestures while not consuming too much of the MCU memory. We count the clock cycles using the `DWT_CYCCNT` register, described in appendix B, and calculate the execution time according to equation 4.1, where the clock speed is equal to $216 \cdot 10^6$ Hz for the Cortex M7 CPU used in all experiments. The execution time is interesting when compared with the real-time inference requirements mentioned in section 5.1.1.

$$\text{execution time} = \frac{\text{clock cycles}}{\text{clock speed}} \tag{4.1}$$

In order to determine the accuracy of the Keras models, we use the TensorFlow python module function `evaluate()`. The accuracy is measured before the models are converted to C code. For the TensorFlow Lite quantized models the evaluate function is not available and an equivalent method to calculating the accuracy is provided by TensorFlow [23] and is implemented by us.

4.3 Tools

There are a lot of different tools for machine learning available on the market, many of them are open-source. In this thesis we use TensorFlow with Keras for the CNN since it is building on the Keras models constructed in a previous thesis at Acconeer, *Hand Gesture Classification using Millimeter Wave Pulsed Radar* by Eda Dagasan [4]. The CNN is deployed on an ST board

called *Nucleo F722ZE*. To be able to run the model on a microcontroller, such as the Nucleo F722ZE, we use two different converters, one from ST called X-CUBE-AI, and another called Keras2C that is open-source. All of the tools used are described in the subsections below and referenced from chapter 5.

4.3.1 Nucleo F722ZE

The Nucleo F722ZE is a development board from ST equipped with Arms Cortex M7 CPU, running with a clock frequency of 216 MHz. It has a 512 kB flash memory and a 256 kB RAM, as well as an L1-cache with 8 kB I-cache and 8 kB D-cache [21].

4.3.2 Acconeer Python Exploration Tool

The Acconeer Python Exploration Tool is a tool developed by Acconeer AB used for streaming radar sensor data to a local machine with the help of a graphical user interface. Using the `-ml` option a, currently experimental, machine learning toolbox is available. Using this toolbox it is possible to create, train, and use Keras machine learning models together with the radar sensor data [1].

4.3.3 TensorFlow and Keras

TensorFlow is an open-source machine learning platform created and developed by Google. In recent years TensorFlow has launched a compressed version called TensorFlow Lite for usage on smaller devices, such as mobile phones or embedded devices [23].

Keras is an open-source neural-network library created as a high-level neural networks API capable of running on top of TensorFlow, amongst others. As opposed to using TensorFlow directly, Keras is considered more user friendly [10]. Keras is also implemented in and used as a light-weight API by TensorFlow [23].

TensorFlow Model Optimization Toolkit

TensorFlow provides a pruning API that uses magnitude-based pruning. This API is used throughout the pruning experiments that focus on magnitude-based weight pruning. The tool iteratively removes connections, by setting them to zero based on their magnitude while training the model. It is important to note that creating sparse tensors by setting weights to zero does not result in faster execution time or lower memory footprint, without an implementation of the inference with the capability to skip calculations when the weights are set to zero.

When using the pruning API the parameters *Epochs*, *Initial Sparsity*, *Frequency*, *Begin Step* and *Final Sparsity* needs to be set. *Epochs* decides for how many epochs the network is re-trained after each pruning step. *Initial Sparsity* decides how much of the network should be pruned in the first pruning iteration, before any re-training. *Frequency* states how often the pruning should be applied. *Begins Step* decides at which step in the pruning process to start the actual pruning. Finally, *Final Sparsity* determines the sparsity at which the pruning ends, i.e. how much of the total network that is discarded during pruning.

The TensorFlow Model Optimization toolkit provides a quantization technique for already trained models, that is performed during a TensorFlow Lite conversion. Weights are quantized from 32-bit floats to 16-bit floats or 8-bit integers [23]. This quantization technique quantizes the weights of the model before runtime. In addition to this, quantization and dequantization of the input and output data are applied for each layer during inference.

4.3.4 STM32CubeIDE and X-CUBE-AI

STM32CubeIDE is a development platform based on the Eclipse CDT framework and GCC toolchain for development and GDB for debugging. It enables the configuration of STM32 MCU:s and the development of C and C++ projects to be generated and deployed on the MCU [20]. Most important for this thesis is that it enables the use of the X-CUBE-AI tool.

X-CUBE-AI is an STM32CubeIDE extension package that allows for automatic conversion of Neural Network inferences and generation of optimized libraries for integration into an STM32CubeIDE project. It supports conversion from various deep learning frameworks such as Keras and TensorFlow Lite [22].

4.3.5 Keras2C

Keras2C is an open-source python library licensed under the GNU GPLv3 license that can be used for converting Keras neural network inference to C99 code, using only standard libraries [19].

4.3.6 Keras-surgeon

Keras-surgeon is a python library used for modifying trained Keras models. It allows for the user to delete neurons or channels, and delete, insert or replace layers [2].

4.3.7 Perf

Perf is a performance analyzing tool for Linux (version 2.6+) based systems which is capable of statistical profiling of the entire system. The tool supports e.g. CPU performance counters, tracepoints, software performance counters, and dynamic probes. The tool uses several different subcommands and comes with a terminal user interface [18].

Chapter 5

Experiments

In this chapter, we first explain the model used and the setups for the various experiments, which can be read about in section 5.1. In section 5.2 we show the results for the different experiments and setups. Finally, we discuss the results in section 5.3

5.1 Setup

The model we use was developed at Acconeer as described in section 4.1, and the setup will be further described below. We decided to use Acconeer Exploration Tool since it provides us with a simple GUI which makes it easy to both train models, connect with the sensor to collect data, and try out an NN inference in real-time on a PC. STM32CubeIDE has been used before at Acconeer, which provides us with useful experience with the IDE when setting up and debugging the project. We had a range of MCU:s to chose between, and decided to start with an MCU with a large amount of flash memory and RAM, to surely be able to fit the inference on it. The articles mentioned in chapter 3 motivated us to try out the pruning and quantization techniques, since they seemed somewhat used before but still can give very different results depending on the network they are used on. All of our chosen setups are further described in the sections below.

5.1.1 Our Model

The model used in this master's thesis is a CNN with 13 layers, as seen in figure 5.1. A detailed summary of all the layers in this model can be seen in appendix C. This network is used to differentiate between three different hand gestures and one empty, or non-gesture, background:

- Swipe: the hand moves quickly from one side to another over the sensor
- Press: an open hand moves down and up once above the sensor, as to press an imaginary button
- Pick: the hand is held steadily above the sensor while the thumb and index finger meet once
- Empty: there is no hand or other object above the sensor

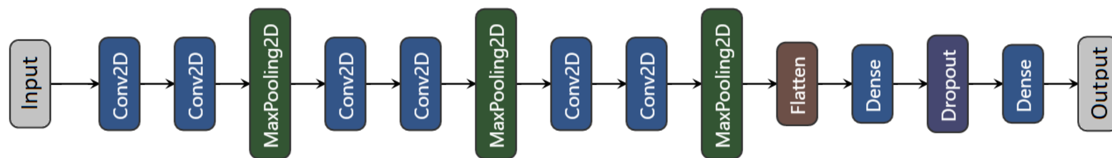


Figure 5.1: The layers of the CNN used in this thesis.

Each batch of data from the sensor, called a *sweep*, consists of a series of integers, where a higher number indicates a stronger reflector in front of the sensor. This means that a piece of metal will generate a higher number than pure air.

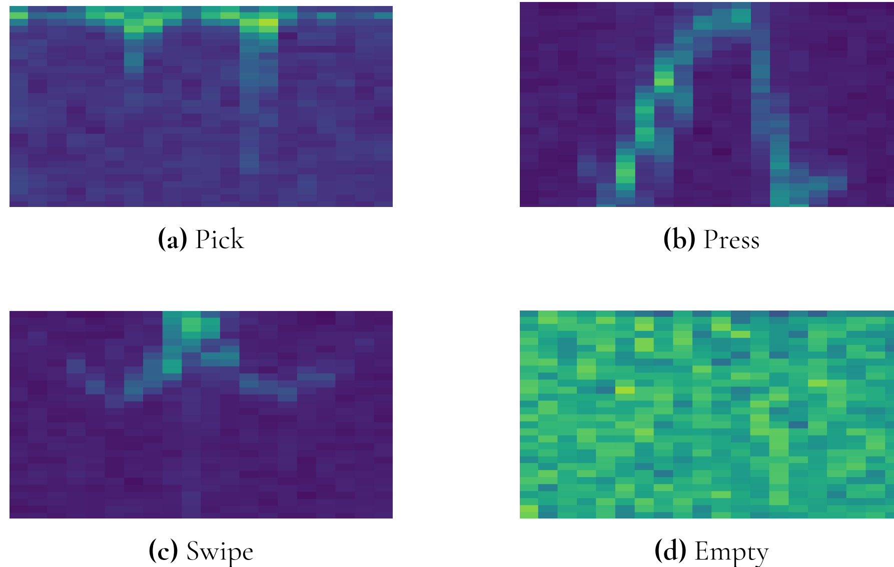
In order to perform inference on the radar data, it has to be preprocessed. Without preprocessing the sensor data, the network would not be able to classify the gestures correctly, no matter how good accuracy it had in the validation state. This is due to that the network was trained at data that was preprocessed this way, which is decided by Acconeer Exploration Tool [1].

In the preprocessing step, we use a buffer where sensor data is saved. This buffer can be seen as a sliding window; each time the sensor produces new data, the oldest sweep is removed, replaced with the second oldest data, and so on. The buffer has space for 20 sweeps. 20 sweeps together are called a *frame*. The frame passes through a Hanning window and an FFT to build an image of the sensor data.

Examples of what the different gestures look like as images can be seen in figure 5.2. Lighter pixels mean stronger reflector (i.e. a hand or object in front of the sensor at that position) relative to darker pixels which indicates a weaker reflector (e.g. such as air). The brightness of the pixels does however depend on what information the sensor data contains; as seen in figure 5.2(d) the empty air reflects roughly the same no matter where you look at the data, hence the whole picture lights up.

We use the sensor with an update rate of 18 Hz, which means that every $\frac{1}{18}$ second we get new data from the sensor. This means that an image represents a time frame of $\frac{1}{18} \cdot 20 \approx 1.111$ seconds. Hence, we capture an image of a gesture, by rebuilding sensor data from a small time series of 20 sweeps.

A processed frame gets a width of 20 and a height of 30, and just 1 channel. This is the input image for our network. In the convolution layers, the input will get 32 channels, and the height and width will shrink with each max pooling that it passes. This is defined by how Dagasan [4] built the network. Each image will then pass through the CNN inference, and the output result will be a classification of what gesture the network thinks the image contained.



(a) Pick (b) Press

(c) Swipe (d) Empty

Figure 5.2: Illustration of the four gestures that can be classified by our network.

Since the sensor is configured with an update rate of 18Hz, we have approximately $\frac{1}{18} = 0.05556$ seconds to preprocess the data and identify a gesture without missing any new data. The preprocessing takes around 27077 clock cycles (0.00013 seconds) when running on a Cortex M7 CPU. This means that the inference cannot be slower than 0.05543 seconds in order to perform inference in real-time and not miss new sensor data.

5.1.2 Tool Versions

In this section, the versions of operating systems and tools used during the experiments are presented.

Operating Systems

During the experiments, we used two operating systems: Windows 10 and Ubuntu 18.04 LTS. Acconeer Exploration Tool and all python scripts regarding the model are mainly executed in the Windows environment. STM32CubeIDE is run on both operating systems. Profiling with perf is done in the Linux environment. GCC version 7.4.0 was used.

Acconeer Exploration Tool

We use Acconeer Exploration Tool with version 2.0.0. However, since this version can only save models in a numpy format, we altered the source code to be able to save the models as Keras files as well.

Python Versions

We use Python 3.6 together with TensorFlow and Keras. When we use them together with Acconeer Exploration Tool, TensorFlow version 1.15.2 and Keras version 2.2.4 are needed. For

quantization, we use TensorFlow 2.1.0. We use Keras-surgeon version 0.1.3 throughout the experiments.

IDE and Converter Versions

Throughout all experiments with the Keras2C converter, we use the version with the latest commit from February 20th, 2020 (commit df61022). For the X-CUBE-AI experiments, we use X-CUBE-AI version 5.0.0 to convert the Keras and TensorFlow Lite models to C code. Throughout all experiments with STM32CubeIDE, we use IDE version 1.2.1 in combination with STM32CubeMX 5.5.0-RC6 to be able to build and run the code on chosen MCU.

5.1.3 Optimization Techniques Setup

The following subsections will describe the detailed setup for each optimization technique used.

Magnitude-Based Weights Pruning

We use the TensorFlow tool for magnitude-based weight pruning with re-training in Keras, described in section 4.3.3, in the magnitude-based weight pruning experiments. Different settings for the parameters *Epochs*, *Initial Sparsity* and *Frequency* are tested for the optimal setup of pruning our model. The final settings are shown in table 5.1.

Setting	Value
Epochs	15
Initial Sparsity	0
Begin Step	0
Frequency	50

Table 5.1: Magnitude-based pruning settings.

It is decided that *Begin Step* is set to 0 since the model is already trained. The *Final Sparsity*, corresponding to how much of the network to remove, is varied over the experiments to test how it affects the accuracy.

Channel-Based Pruning

We use KerasSurgeon version 0.1.3, described in section 4.3.6, for channel-based pruning. Channels with the lowest L1-norm are removed, according to section 2.2.2, using the method `delete_channels()`. In order to explore how many channels can be removed without losing accuracy, we remove channels in an iterative manner, one by one. We do channel-based pruning both with and without re-training in our experiments. At last, we apply re-training to all pruned models to examine how the models recover from the accuracy loss.

Quantization

We perform quantization using the TensorFlow model optimization toolkit, described in section 4.3.3, while converting the model to TensorFlow Lite. In order to achieve a better understanding of how quantization affects a model, we quantize the initial model as well as all models pruned with channel pruning. At last, we import the quantized models into X-CUBE-AI and run them on the MCU. Keras2C does not support converting TensorFlow Lite models.

Optimizing Model C Code

First, we import the C code generated with Keras2C to a STM32CubeIDE project configured for the Nucleo-F722ZE. When we have converted a Keras model to C code, we modify the code to store all the weights in the flash memory instead of RAM, since the weights are static. This is done to save some space in the RAM, which is needed for other (volatile) variables. It is not possible to flash our chosen MCU with the inference software without applying this modification, for the simple reason that we otherwise run out of RAM.

Furthermore, the generated C code from Keras2C uses unique output arrays for each layer to store the result, before sending it to the next layer. This means that a new array is allocated for each layer, resulting in 14 arrays for our model. Since the output from one layer can be removed after it has been used as input to the next layer, we optimize the code to alternate between two arrays instead. For example, the first layer of a model can use the first array to store its output, and the second layer can use the second array, the third layer can now use the first array since the output of the first layer is no longer needed, and so forth. This means that only two arrays need to be allocated, instead of 14. This optimization was also necessary for the model to fit in the RAM of the MCU used in this thesis. Now the code is runnable on the Nucleo-F722ZE.

Using the compiler settings O0, OFast, and O3 we determine how they affect the performance. We choose to test with these compiler settings since, in order to meet the real-time inference requirements mentioned in section 5.1.1, the execution time is the biggest obstacle. OFast and O3 are the optimization levels that optimize the cost most and there is, therefore, no need to test with optimization levels that are not as fast. O0 is used as a baseline for comparing the results.

Optimizing Convolution Layer C Code

We optimize the convolution layer to take advantage of the magnitude-based weight pruned model. We do this, by removing the weights that are set to zero from the arrays in which they are stored, and reduce the array size correspondingly. This allows for smaller arrays which reduce the computations needed for inference and consume less memory.

When the convolutional layer performs the convolution, it loops over the kernel array one time for each index in the output array. The convolutional layer then calculates output values, by using the kernel value and input value for each value in the kernel array. This corresponds to a convolution as described in section 2.1.1.

In order to speed up the convolutional layer, we now loop over the reduced kernel array,

instead of the original kernel containing the zeros. The convolutional layer uses data from where in the kernel array the current weight is stored, for calculating the index to the input and output array where data should be stored and retrieved. This data is now lost since we only loop over the weights that were not zero. In order to still save and collect data from the correct indices in the input and output arrays, these index parameters have to be stored for each weight, in an additional array corresponding to the new, reduced, kernel array. An additional array containing information about the corresponding indices for the input and output tensor for each weight not set to zero, is therefore stored for each convolution layer. This optimization allows for fewer computations since we now skip all computations with zero. A visual example of how the weight arrays are reduced can be seen in figure 5.3. In this figure, the array `weights` contains the original kernel weights. The array `reduced_weights` contains the reduced kernel array, and the array `index_parameters` contain the index parameters needed for calculating the indices for the input and output arrays. In this case `idx_params_0` contains the index parameters for the first weight, `idx_params_1` contains the index parameters for the second weight, etc. This means that the length of the `reduced_weight` array and `index_parameters` array are equal.

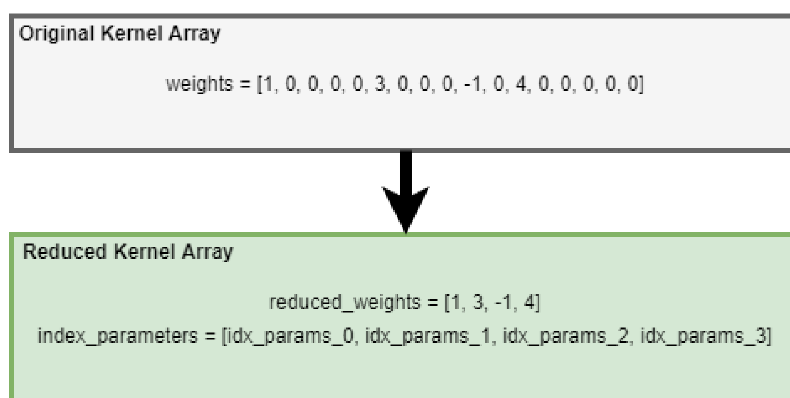


Figure 5.3: A visual explanation of how kernel arrays are reduced.

It is important to note that, while we reduce the number of computations performed in the convolutional layer, we also increase the number of memory accesses. This could potentially result in higher execution time. If the time for memory accesses increases more than the time for the weight calculations decrease, the execution time becomes higher.

5.1.4 Experiments Setup

Each experiment conducted can be seen in table 5.2. Setups 1-6 are experiments carried out with X-CUBE-AI. Setups 7-10 are experiments carried out with Keras2C. An "initial model" (i.e. not optimized original model) was used in both X-CUBE-AI and Keras2C to set a baseline for the rest of the experiments (see Setup 1 and 7 respectively). The model is evaluated and measured according to section 4.2 in all experiments.

Setup	Weight Pruning	Channel Pruning	Quant-ization	C-code Opt.	Array Opt.	Cube-AI	Keras2c
1						✓	
2		✓				✓	
3	✓					✓	
4			✓			✓	
5		✓	✓			✓	
6	✓		✓			✓	
7							✓
8		✓		✓			✓
9	✓			✓			✓
10	✓			✓	✓		✓

Table 5.2: The setup for all techniques and tools used.

5.2 Results

Here we present the results for each setup described in table 5.2.

Results for Setup 1 - Baseline for X-CUBE-AI

The result of Setup 1 serves as a baseline for the other experiments with X-CUBE-AI. The original model, no optimizations applied, uses 265.06 kB flash memory and 84.74 kB RAM. It executes one inference at 0.2239 seconds and has an accuracy of 1.0 on the test data set, meaning that 100% of the predictions were correct.

Results for Setup 2 - Channel Pruning

In Setup 2, channel pruning was used. When re-training is not applied, it is possible to remove 8 channels (25% of the convolutional layers) without losing a significant amount of accuracy. Furthermore, with re-training, it is possible to remove 20 channels (62.5% of the convolutional layers) before the accuracy starts to decline. With re-training, we loose at most 4% points when pruning almost 94% of the channels. The accuracy concerning the channel pruning is shown in figure 5.4. Measured numbers are available in appendix A.

Another result of channel pruning is that when the number of channels decreases, the execution time decrease as well as memory usage. This is expected since fewer parameters require fewer computations and less memory. When removing 8 channels (without re-training), the inference executes at 0.1230 seconds with an accuracy of 0.9916. The memory uses 164.74 kB of flash and 63.55 kB of RAM. When removing 20 channels (with re-training) the inference executes at 0.0265 seconds, with an accuracy of 0.9976. The memory uses 57.46 kB of flash and 31.78 kB of RAM. The execution time and memory usage are displayed in figure 5.5.

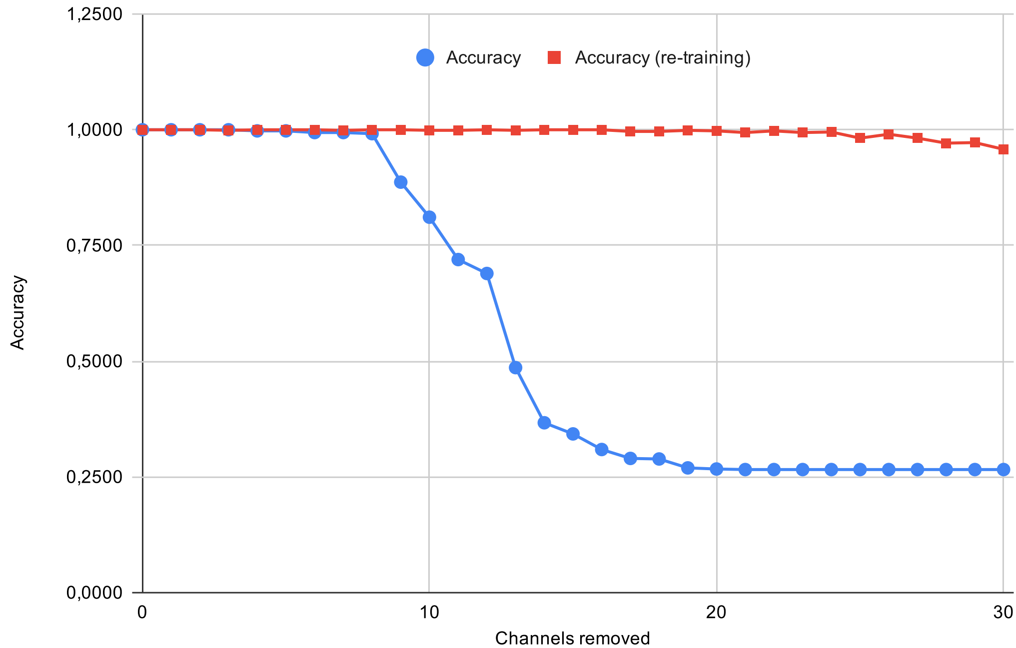


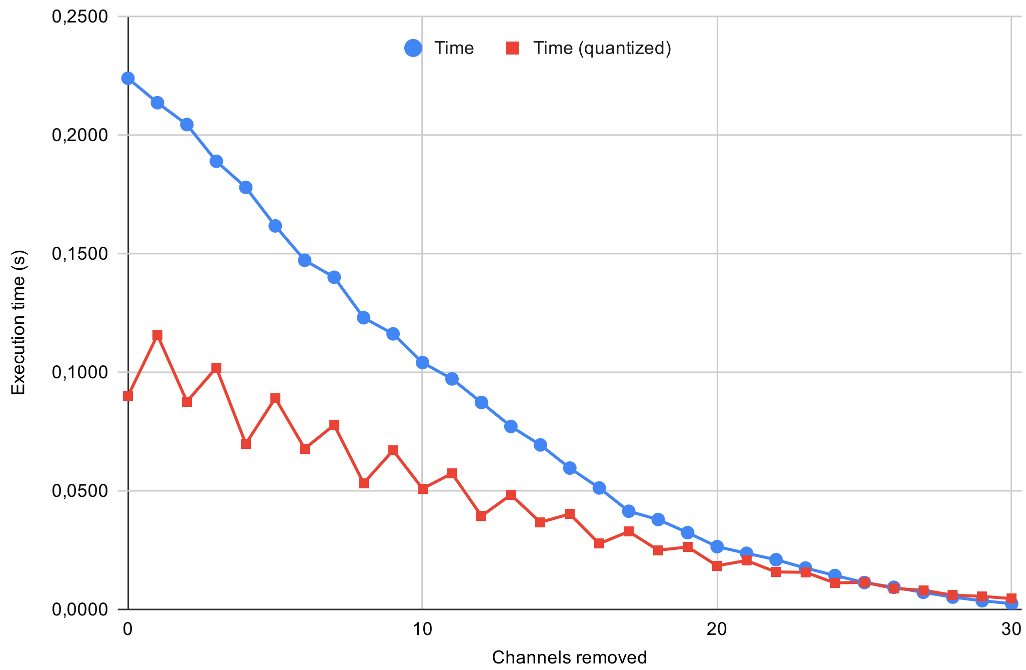
Figure 5.4: Accuracy after pruning for setup 2 and 8. The baseline model has 32 channels.

Results for Setup 3 - Weight Pruning

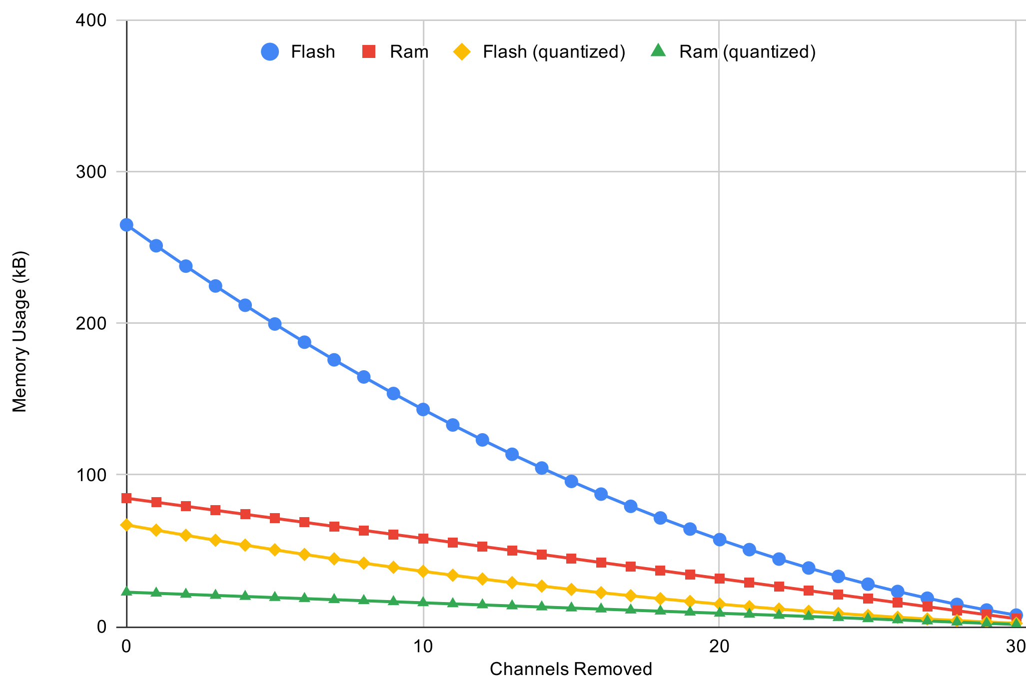
With Setup 3, it is possible to prune up to 90% of the network using weight-based pruning without losing a significant amount of the original accuracy. However, it is not possible to take advantage of the pruned model with X-CUBE-AI, since there is no access to modify the source code. The pruning tool only sets the values of the pruned weights to zero, they are not removed physically. This means that it is not possible to reduce clock cycles needed for inference or memory usage by pruning since all calculations in the convolution layer will be done, and weights are stored regardless if they are zeros or not. Hence, the result regarding memory usage and execution time equals the baseline. The accuracy of the 90% weight pruned model is 0.9855. The results for 5 tested sparsities can be seen in table 5.3.

Final Sparsity (%)	Accuracy
82	0.9927
86	0.9843
90	0.9855
94	0.7063
98	0.3213

Table 5.3: Accuracy of the weight pruned model at different sparsity levels.



(a) Execution Time



(b) Memory Usage

Figure 5.5: Execution time and memory usage after pruning and quantization for the models converted with X-CUBE-AI in setup 2, 4 and 5. The baseline model has 32 channels.

Results for Setup 4 - Quantization

In Setup 4, quantization was successfully applied to the original model. This means that it is possible to quantize from 32-bit floating point to 8-bit integers without losing more than 1% of the original accuracy. Quantization reduced the memory requirements to 67.15 kB flash and 22.78 kB RAM. It also reduced the inference execution time to 0.0900 seconds.

In figure 5.5 the quantization results can be seen both on its own, and combined with different stages of channel pruning. Pruning zero channels correspond to no pruning at all, hence quantization combined with pruning zero channels represents quantization alone. All numbers can be seen in appendix A.

Results for Setup 5 - Quantization and Channel Pruning Combined

In Setup 5, when applying quantization together with channel pruning, the execution time for i number of channels ($i \in \mathbb{N}$: i mod 2 $\neq 0$) is longer than for i+1 numbers of channels, as can be seen in 5.5(a). This is interesting and connected to that some optimizations the ST team have done only apply to layers with an even number of channels. However, since we cannot access the X-CUBE-AI source code, we cannot confirm how this works.

Results for Setup 6 - Quantization and Weight Pruning Combined

In Setup 6, weight pruning is combined with quantization. Since it is not possible to take advantage of the weight pruning with X-CUBE-AI, the improvements for experiment 6 regarding memory usage and execution time rely on the quantization alone which was presented for Setup 4 above. The accuracy of this combination is 0.9795.

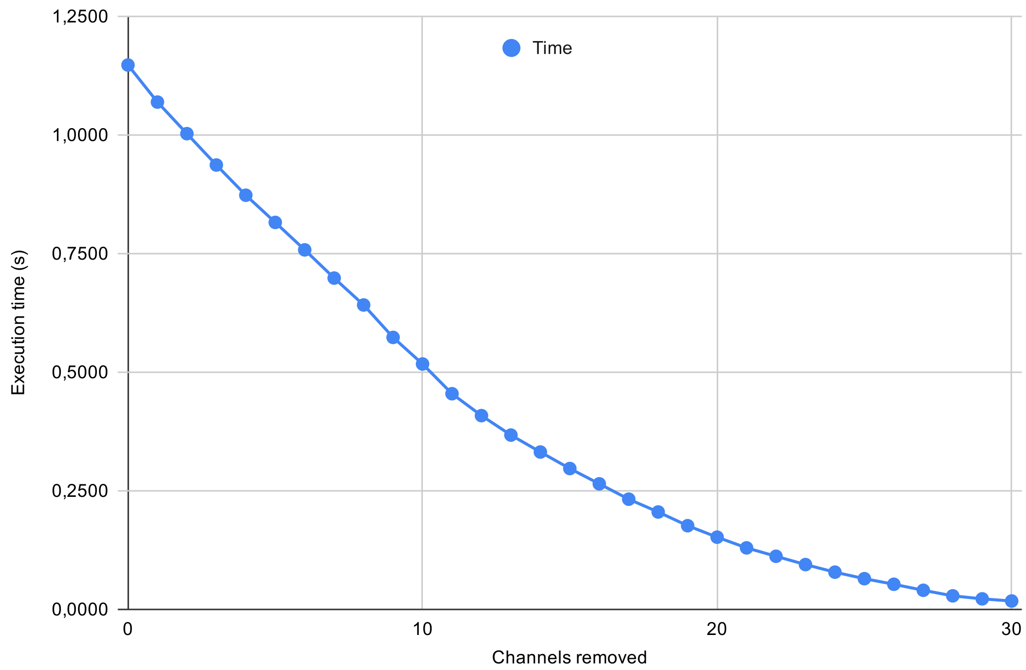
Results for Setup 7 - Baseline for Keras2C

The results of Setup 7 serves as a baseline for the other experiments with Keras2C. The original model, no optimizations applied and compiled with -O0, uses 312.89 kB flash memory and 167.83 kB RAM. It executes one inference at 17.61 seconds with an accuracy of 1.0.

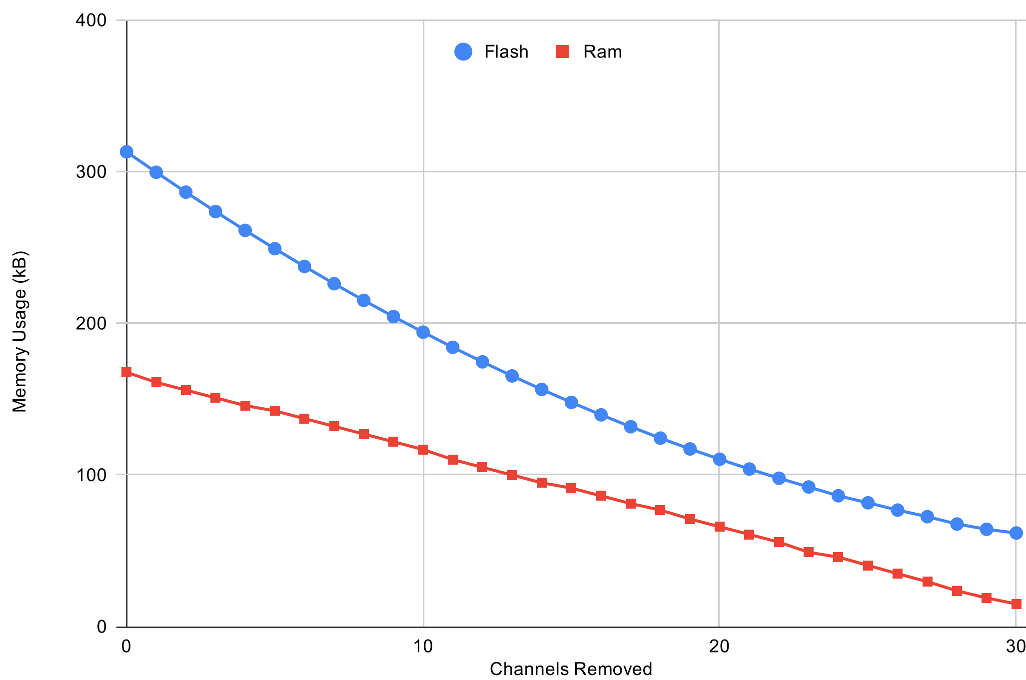
Results for Setup 8 - Channel Pruning

In Setup 8, the original model is channel pruned. Since the accuracy is measured before conversion to C code, the accuracy is the same as presented for Channel Pruning with X-CUBE-AI in the results for Setup 2, and as can be seen in figure 5.4.

When pruning 8 channels without re-training, the model needs 231.38 kB flash memory and 130.09 kB RAM, and needs 0.6418 seconds to execute one inference. When pruning 20 channels, with re-training, the memory needed decreases to 126.62 kB flash and 68.97 kB RAM, while executing one inference in 0.1524 seconds. The memory usage and execution time for different levels of channel pruning can be seen in figure 5.6. All the numbers from this experiment can be found in appendix A.



(a) Execution Time



(b) Memory Usage

Figure 5.6: Execution time and memory usage after pruning and quantization for the models converted with Keras2C in setup 8. The baseline model has 32 channels.

Results for Setup 9 - Weight Pruning

In Setup 9, weight pruning is applied to the original model. Since the weight pruning is done before the conversion to C code, the accuracy of the weight pruned model is the same as for the X-CUBE-AI converted model, which is 0.9855. The results can be seen in table 5.3.

On its own, it is not possible to take advantage of the weight pruning with Keras2C, since all computations in the convolution layer are done regardless of whether the numbers are zero or not. This means that the memory usage and execution time for an inference equals the original model (baseline). However, in Keras2C there is access to the source code which can be modified to take advantage of weight pruning. The results of this are elaborated below, in the results for Setup 10.

Results for Setup 10 - Weight Pruning and Optimizing C code arrays

In this setup, weight pruning is combined with further optimizations in the convolution layer in the C code, which when compiled with -O3 decreases the execution time to 0.4905 seconds. The accuracy is the same as presented in section Magnitude-Based Weight Pruning above, 0.9855, and the memory usage is reduced to 166.30 kB for flash and 166.26 kB for RAM. A summary of the execution time for the different compiler optimizations can be seen in table 5.4.

Model	O0	OFast	O3
Initial	17.6062	1.8016	1.1476
Weight pruned	17.6062	1.8015	1.1406
Optimized for weight pruning	2.1253	0.5816	0.4905

Table 5.4: Execution time of the optimizations for the different compiler settings tested in setup 7, 9, and 10.

Using this C code optimization, we can see that the clock cycles are reduced from roughly 383 million to 98 million for one inference. Correspondingly only 96% of the inference time is used in the convolution layer, as opposed to without optimizations where 98% of the inference time is spent in the convolution layer. The profiling can be seen in table 5.5.

	No optimizations		Optimized for pruning	
Layer	Clock Cycles	Quota	Clock Cycles	Quota
Padding	4895120	0.01254	1104568	0.01086
Convolution	383225162	0.98149	98045158	0.96393
Maxpool	1503385	0.00385	1736898	0.01708
Flatten	27872	0.00007	6751	0.00007
Dropout	23	0.00000	23	0.00000
Dense	800979	0.00205	821030	0.00807

Table 5.5: Perf profiling result of the Keras2C model.

5.3 Discussion

Here we extend the discussion of some of the results further. The research questions will be answered and some ethical aspects, as well as threats to validity, will also be discussed.

Exploiting Sparse Matrices With TensorFlow Lite

TensorFlow states in the guide of how to prune a network with their tool that *"By itself, a sparse model will not be faster to execute. It just enables backends with such capability. In the near future, however, TensorFlow Lite will take advantage of the sparsity to speed up computations"* [23]. This confirms our result that there was no difference in execution time or memory usage between the initial model and the magnitude-based weights pruned model when using X-CUBE-AI. X-CUBE-AI has not implemented a way to exploit the sparse matrices yet.

As shown by the study *To prune, or not to prune: exploring the efficacy of pruning for model compression* by Zhu et al. [26] creating a large model and pruning it to become sparse can outperform a dense model in terms of accuracy. It will, therefore, be interesting to investigate the TensorFlow Lite optimization that will be available in the future. If X-CUBE-AI implements a backend that can exploit the sparse TensorFlow Lite models it will probably outperform our Keras2C solution since X-CUBE-AI has proven to be faster than Keras2C in all of the experiments carried out in this thesis. Furthermore, it may very well be the case that the weight pruned model can outperform the channel pruned model on inference accuracy, execution time, and memory usage.

Increased Execution Time for Small Models

In figure 5.5 and table A.2 we can see that the execution time for the 32-bit floating point model is about as short or shorter than the quantized 8-bit integer model when the number of channels removed exceeds 25. This can be explained by the quantization technique used by TensorFlow Lite, where the input data is quantized and dequantized for each layer at runtime, as mentioned in section 4.3.3. This technique allows for a more accurate quantization of the model but will have an impact on the execution time if the layers are small. The time saved by using integer operations will be lost to the quantization and dequantization of the inputs and outputs between layers. It is therefore important to take this aspect into consideration when quantizing a smaller model using the TensorFlow Lite tool.

Even Number of Channels Provides Faster Inference in X-CUBE-AI

As shown in figure 5.5 the execution time for the quantized model varies greatly when channels are removed, and for cases when the channels are decreased from an even number to an odd the execution time increase. This does not comply with the theory that removing channels and creating a smaller model will lead to faster inference time. For instance, removing one channel from 32 to 31 the execution time should theoretically decrease since the model has fewer parameters and therefore should require fewer computations. However, on the contrary, we can see that the execution time increase.

Furthermore, we can see that the execution time does not decrease at the same rate between the even channels. Removing 2 channels from 32 to 30 results in an execution time that is fairly close to using 32 channels. However, removing 2 channels from 30 to 28 results in a bigger difference in execution time. This means that removing channels when using a quantized model does not always provide a speedup that is worth the possible loss in inference accuracy. One has to be careful and measure how the optimizations affect the model to be sure that the optimization is as efficient as requested.

A possible explanation for this is that X-CUBE-AI uses an optimization that only applies to convolutional layers with an even number of channels. This optimization could potentially be loop unrolling, where the feature maps are calculated two and two in parallel, instead of one and one, combined with using bitshift whenever possible. We can not see this result for the Keras2C converted models because Keras2c does not use this optimization.

Overdimensioned Models

As can be seen in the graph of accuracy after pruning, in figure 5.4, the accuracy starts to drop after around 20-25 channels are removed with retraining. Furthermore, about 90% of the weights were removed in the experiments with magnitude-based weight pruning, without losing any significant accuracy. This makes us suspect that the model used in this thesis is over-dimensioned and that a smaller model could have been trained from the beginning. It could also mean that our training and test data set is not large and diverse enough and that a larger data set would have resulted in an earlier accuracy drop. Since all people are unique and would perform the gestures somewhat different from each other, it is possible that data collected with people not used in the original data set would result in lower accuracy. It is possible that the smaller model only performs well on the test data used in the accuracy measurements for our model.

In order to investigate this further when designing, training, and optimizing a model there is a need for more testing on more data. When testing it in real time the larger model may make better predictions of hand gestures that produce data not similar to those in the train and validation data set. By using a well-defined data set when training the model and validating the accuracy, the impacts of pruning and quantization can be determined with greater certainty.

Comparing X-CUBE-AI With Keras2C

X-CUBE-AI and Keras2C were both able to convert the model to C code and perform inference successfully on the MCU. However, the tools are different in both usability, as well as the resulting inference execution time and model memory usage.

Our results in this thesis show that the X-CUBE-AI converted models are faster and consume less memory than the Keras2C converted models. For instance, the Keras2C converted model needs to prune around half of the channels in the convolutional layers in order to perform as well as the baseline X-CUBE-AI converted model, in terms of execution time. Furthermore, the memory usage is around 85% of flash and 50% of RAM for the X-CUBE-AI converted models compared to Keras2C.

That being said, Keras2C still has the advantage over X-CUBE-AI in the case that the C code used for inference is modifiable. If the converted model has possibilities of being optimized in a unique way that is not implemented in X-CUBE-AI, the user can optimize the code by hand which may result in improvements, making the Keras2C converted model inference more efficient than X-CUBE-AI.

5.3.1 Answer to Research Questions

In section 1.1.2 the research questions of this thesis are stated.

RQ1: How does pruning impact inference regarding performance and memory footprint?

RQ2: How does quantization impact inference regarding performance and memory footprint?

RQ3: How is the accuracy affected when applying pruning and quantization to a neural network?

The work carried out in this thesis has shown two possible ways of converting machine learning models to run on an MCU: X-CUBE-AI and Keras2C. Furthermore, two optimization techniques have been explored, pruning and quantization. The results from these experiments lead to the answers of RQ1 and RQ2 as follows. Using pruning, the inference time can become up to $8\times$ faster, flash memory usage up to $4.5\times$ lower, and RAM memory usage up to $2.8\times$ lower. Similarly, quantization can improve the execution time by up to $2.5\times$, reduce flash memory usage by up to $4\times$, and RAM memory usage by up to $3.7\times$.

Considering the accuracy as mentioned in RQ3, the pruning and quantization that was applied to the model used in this thesis are considered. The magnitude-based weight pruning experiments showed that pruning 90% of the network result in an accuracy loss of 1.5 percentage points. Channel-based pruning only affects the accuracy marginally when retraining is applied, dropping at most 4 percentage points when pruning almost 94% of the channels. If no re-training is applied the accuracy starts to decrease after around 25% of the channels are removed. Finally, quantization from 32-bit floating points to 8-bit integers only had a minor effect on the accuracy with a drop of at most 1%.

5.3.2 Ethical Aspects

When using the optimization techniques discussed in this thesis it is important to consider the possible effects of a decreased accuracy. If the users are unaware of the effects on accuracy, and a network that is used in some existing application is pruned and quantized, it is possible that the accuracy drop causes the application to behave differently. For example, an application used for image recognition might start to classify images differently after pruning and quantization is applied.

Furthermore, these techniques allow for neural networks to become smaller and therefore also deployable on smaller devices. This allows for usage in areas not thought about before, as well as an extended area of use of applications that previously required larger hardware. For example, a face recognition application that previously was deployed at one location because

of hardware constraints, can now be deployed at numerous different locations on smaller devices.

5.3.3 Threats to Validity

In this master's thesis, only one model has been used and evaluated. The results for this model do not necessarily apply to other models since other networks potentially could be more fragile to quantization and pruning than the one used in this thesis.

The size of the data set which the models are trained and re-trained on, as well as validation and test data sets, are relatively small. The accuracy of the different optimized models and the original model could probably be lower when using larger data sets, but we believe the proportions would stay relatively the same when it comes to the results. However, collecting data and training the model was outside the scope of this thesis, and was done just enough to be able to carry out the experiments.

As mention in section 5.3, the over-sizing of the model could potentially also be the reason for being able to prune the network by 90%.

Chapter 6

Conclusions

In this master's thesis, we have successfully reduced a network regarding memory footprint and execution time. We used the Acconeer Exploration tool to train a gesture classification network. This network was then successfully pruned both with magnitude-based weight pruning and channel-based pruning. It was also successfully quantized. All the network reducing techniques was carried out in a Python environment using Tensorflow Lite and Keras. In order to deploy the network on an MCU, STM32CubeIDE was used to set up a C code project. To be able to convert the Python written network to C code, two different approaches were used: X-CUBE-AI and Keras2C. The different network reducing techniques was the main focus of this thesis.

6.1 Summary

X-CUBE-AI and Keras2C are two viable ways of converting a Keras machine learning model to run on an MCU. As mentioned before, Keras2C is a good choice if one wants to optimize the inference by altering the C code. On the other hand, our baselines show that using X-CUBE-AI will give a faster inference before even applying any optimizations. A consequence of not being able to modify the X-CUBE-AI inference source code is that the network is not able to take advantage of any weight pruning. This means that magnitude-based weight pruning does not have an impact on the execution time and memory usage unless the backend supports it. Exploiting a weight pruned model is possible in Keras2C and can improve the execution time by $2.3\times$ and reduce the flash memory usage by $2\times$. RAM memory usage is however unaffected.

Applying channel-based pruning was able to reduce the execution time by up to $8\times$, flash memory usage by up to $4.5\times$, and RAM memory usage by up to $2.8\times$ without having a nega-

tive impact on the accuracy. However, when performing pruning, re-training the model can significantly improve the accuracy of the pruned model. Channel-based pruning only affects the accuracy marginally when re-training is applied, and, in our case, only dropped at most 4 percentage points when pruning almost 94% of the channels.

The quantization technique we used proved to be able to reduce execution time by $2.5\times$, flash memory usage by $4\times$, and RAM memory usage by $3.7\times$. This shows that quantization is a good choice both when it comes to reducing execution time, but mainly when one wants to reduce the memory footprint of a network.

We were not able to use a less powerful MCU in this particular case with this model, despite the results we received with the optimization techniques applied. However, we have shown that pruning and quantization are two good alternatives, that can be used on its own or together, in order to reduce a network.

6.2 Future Work

Here we present some ideas for future work that could build upon the results in this thesis.

Hardware Accelerated Architectures

One interesting way to optimize machine learning inference for MCUs is by using hardware accelerated architectures designed specifically for the network deployed. For instance, if the network used in this thesis were to use a hardware accelerated architecture it could be focused on speeding up the convolutional layer since it constitutes more than 90% of the clock cycles in inference. One could compare the execution time of a convolutional layer implemented in hardware to the C code equivalent in this thesis.

Different Ways to Prune a Network

There are many ways to prune a network, channel pruning and magnitude-based weight pruning are just two of them. It would be interesting to try out other pruning alternatives to see how they behave.

TensorFlow Lite will have a back-end that supports speed-up for the weight pruned model soon (according to TensorFlow). When that is possible and if X-CUBE-AI supports it one can look at deploying the weight pruned model on X-CUBE-AI again. Since X-CUBE-AI is the better alternative of the two approaches explored in this thesis when it comes to execution time only, it would be very interesting to see how much faster a weight pruned model can become when X-CUBE-AI supports it.

Quantize a Model to Use with Keras2C

Quantization was omitted for the Keras2C approach due to lack of time since there is no simple way to perform the quantization. Implementing such a function would be out of the scope for this thesis. If a good quantization technique is implemented for Keras models, it

would be interesting to see how the Keras2C network would perform in comparison with X-CUBE-AI. Today, there is no such quantization implemented.

Bibliography

[1] Acconeer AB. Acconeer Python Exploration docs. https://acconeer-python-exploration.readthedocs.io. Accessed 2020-02-04.

[2] Ben Whetton. Keras-surgeon documentation. https://github.com/BenWhetton/keras-surgeon. Accessed 2020-04-02.

[3] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit Quantization of Neural Networks for Efficient Inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018, 2019.

[4] Dagasan Eda. Hand Gesture Classification using Millimeter Wave Pulsed Radar. Master's thesis, Lund University, Mathematical Statistics, 2020.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[6] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher Wardlaw Fletcher. UCNN: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings - 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture, ISCA 2018*, pages 674–687. Institute of Electrical and Electronics Engineers Inc., July 2018.

[7] Zhongzhan Huang, Xinjiang Wang, and Ping Luo. Convolution-Weight-Distribution Assumption: Rethinking the Criteria of Channel Pruning. *ArXiv*, abs/2004.11627, 2020.

[8] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2017.

[9] Qing Jin, Linjie Yang, and Zhenyu Liao. Towards Efficient Training for Neural Network Quantization. *ArXiv*, abs/1912.10207, 2019.

[10] Keras. Keras documentation. https://keras.io/. Accessed 2020-03-01.

[11] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *ArXiv*, abs/1806.08342, 2018.

[12] Souvik Kundu, Saurav Prakash, Haleh Akrami, Peter A. Beerel, and Keith M. Chugg. pSConv: A Pre-defined Sparse Kernel Based Convolution for Deep CNNs. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 100–107, 2019.

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recogniziton. *Proceedings of the IEEE*, 86(11):2278 – 2324, November 1998.

[14] Yann Lecun, John Denker, and Sara Solla. Optimal Brain Damage. In *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, January 1989.

[15] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning Filters for Efficient ConvNets. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[16] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *ICML*, 2016.

[17] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime Neural Pruning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 2178–2188, Red Hook, NY, USA, 2017. Curran Associates Inc.

[18] Linux Kernel Organization. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed 2020-04-29.

[19] Rory Conlin. Keras2C documentation. https://github.com/f0uriest/keras2c. Accessed 2020-02-20.

[20] STMicroelectronics. Stm32CubeIDE. https://www.st.com/en/development-tools/stm32cubeide.html. Accessed 2020-01-25.

[21] STMicroelectronics. STM32F722ZE. https://www.st.com/en/microcontrollers-microprocessors/stm32f722ze.html. Accessed 2020-05-30.

[22] STMicroelectronics. X-CUBE-AI. https://www.st.com/en/embedded-software/x-cube-ai.html. Accessed 2020-01-25.

[23] Tensorflow. Tensorflow documentation. https://www.tensorflow.org. Accessed 2020-03-01.

[24] Bram-Ernst Verhoef, Nathan Laubeuf, Stefan Cosemans, Peter Debacker, Ioannis A. Papistas, Arindam Mallik, and Diederik Verkest. FQ-Conv: Fully Quantized Convolution for Efficient and Accurate Inference. *ArXiv*, abs/1912.09356, 2019.

[25] Seul-Ki Yeom, Philipp Seegerer, Sebastian Lapuschkin, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Pruning by Explaining: A Novel Criterion for Deep Neural Network Pruning. *ArXiv*, abs/1912.08881, 2019.

[26] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *ArXiv*, abs/1710.01878, 2018.

Appendices

Appendix A

Data tables

This chapter presents the raw data used for the graphs from the experiments with channel pruning.

Table A.1 presents the accuracy for each step in the channel pruning process, both for the models with no re-training and the models that were re-trained for 3 epochs.

X-CUBE-AI

Table A.2 presents the clock cycles and execution times of one prediction of a gesture as well as the consumed flash and RAM for each step in the channel pruning process.

Table A.3 presents results from the quantization of the channel pruned models. The accuracy for both the re-trained models and the models with no re-training, as well as the execution time and memory usage.

Keras2C

Table A.4 presents the clock cycles and execution times of one prediction of a gesture as well as the consumed flash memory and RAM for each step in the channel pruning process.

Channels removed	Accuracy	Accuracy (re-training)
0	1.0000	1.0000
1	1.0000	1.0000
2	1.0000	1.0000
3	1.0000	0.9988
4	1.0000	1.0000
5	0.9976	1.0000
6	0.9940	1.0000
7	0.9940	0.9988
8	0.9916	1.0000
9	0.8869	1.0000
10	0.8111	0.9988
11	0.7196	0.9988
12	0.6895	1.0000
13	0.4862	0.9988
14	0.3670	1.0000
15	0.3430	1.0000
16	0.3093	1.0000
17	0.2900	0.9964
18	0.2888	0.9964
19	0.2696	0.9988
20	0.2671	0.9976
21	0.2659	0.9940
22	0.2659	0.9976
23	0.2659	0.9940
24	0.2659	0.9952
25	0.2659	0.9819
26	0.2659	0.9904
27	0.2659	0.9819
28	0.2659	0.9711
29	0.2659	0.9723
30	0.2659	0.9579

Table A.1: Accuracy for channel pruning.

Channels removed	Clock-cycles	Time (s)	Flash (kB)	RAM (kB)
0	48368230	0.2239	265.06	84.74
1	46143775	0.2136	251.26	82.09
2	44155595	0.2044	237.82	79.44
3	40809781	0.1889	224.74	76.79
4	38432933	0.1779	212.02	74.14
5	34927260	0.1617	199.06	71.50
6	31795139	0.1472	187.66	68.85
7	30250631	0.1400	176.02	66.20
8	26564211	0.1220	164.74	63.55
9	25095141	0.1162	153.82	60.90
10	22478782	0.1041	143.28	58.26
11	20995052	0.0972	133.06	58.26
12	18846673	0.0873	123.22	55.61
13	16663913	0.0771	113.74	52.96
14	14979988	0.0694	104.67	50.31
15	12871868	0.0596	95.86	45.02
16	11062079	0.0512	87.46	42.37
17	8943298	0.0414	79.42	39.72
18	8185565	0.0379	71.74	37.07
19	6994035	0.0324	64.42	34.42
20	5719290	0.0265	57.46	31.78
21	5114236	0.0237	50.86	29.13
22	4531161	0.0210	44.62	26.48
23	3772016	0.0175	38.74	23.83
24	3092222	0.0143	33.22	21.18
25	2449292	0.0113	28.06	18.54
26	2016701	0.0093	23.26	15.89
27	1557508	0.0072	18.82	13.24
28	1121035	0.0052	14.74	10.59
29	790803	0.0037	11.02	7.94
30	520389	0.0024	7.66	5.30

Table A.2: Execution time and memory usage for channel pruning. X-CUBE-AI converted models.

Channels removed	Accuracy	Accuracy (re-training)	Clock	Time (s)	Flash (kB)	(RAM (kB)
0	1.0000	1.0000	19439494	0.0900	67.15	22.78
1	1.0000	1.0000	24987978	0.1157	63.70	22.08
2	1.0000	0.9988	18930209	0.0876	60.31	21.36
3	1.0000	0.9988	22016828	0.1019	57.04	20.65
4	0.9988	1.0000	15082635	0.0698	53.82	19.94
5	0.9976	1.0000	19245027	0.0891	50.73	19.23
6	0.9940	1.0000	14648251	0.0678	47.70	18.54
7	0.9940	0.9988	16793673	0.0777	44.78	17.85
8	0.9904	1.0000	11495013	0.0532	41.93	17.16
9	0.8869	1.0000	14483190	0.0671	39.20	16.47
10	0.8111	0.9988	10997723	0.0509	36.52	15.78
11	0.7160	0.9988	12389384	0.0574	33.97	15.09
12	0.6883	1.0000	8507728	0.0394	31.48	14.40
13	0.4777	0.9976	10446784	0.0484	29.10	13.71
14	0.3646	1.0000	7938166	0.0368	26.79	13.02
15	0.3394	0.9976	8696488	0.0403	24.60	12.33
16	0.3117	1.0000	5994612	0.0278	22.46	11.64
17	0.2900	0.9940	7093928	0.0328	20.45	10.95
18	0.2888	0.9952	5392248	0.0250	18.50	10.26
19	0.2708	09988	5689615	0.0263	16.67	9.57
20	0.2671	0.9976	3982400	0.0184	14.89	8.88
21	0.2659	0.9940	3982400	0.0206	13.24	8.19
22	0.2659	0.9928	4455056	0.0158	11.65	7.50
23	0.2659	0.9916	3411188	0.0157	10.18	6.79
24	0.2659	0.9916	3385856	0.0112	8.76	6.03
25	0.2659	0.9819	2481101	0.0115	7.47	5.28
26	0.2659	0.9880	2481101	0.0089	6.24	4.52
27	0.2659	0.9819	1914820	0.0081	5.12	3.78
28	0.2659	0.9663	1745611	0.0060	4.07	3.02
29	0.2659	0.9687	1302366	0.0055	3.14	2.27
30	0.2659	0.9567	986888	0.0046	2.26	1.51

Table A.3: Accuracy, execution time and memory usage of the quantized channel pruned models. X-CUBE-AI converted models.

Channels removed	Clock-cycles	Time (s)	Flash (kB)	RAM (kB)
0	247874028	1.1476	313.20	167.83
1	231007666	1.0695	299.72	161.16
2	216626937	1.0029	286.59	156.07
3	202355098	0.9368	273.82	150.97
4	188602601	0.8732	261.39	145.88
5	176213887	0.8158	249.32	142.37
6	163716500	0.7579	237.61	137.27
7	150874769	0.6985	226.24	132.18
8	138625832	0.6418	215.22	127.08
9	123860370	0.5734	204.56	121.99
10	111759095	0.5174	194.25	116.90
11	98220565	0.4547	184.29	110.22
12	88249353	0.4086	174.68	105.13
13	79390233	0.3675	165.42	100.04
14	71673998	0.3318	156.51	94.94
15	64161803	0.2970	147.96	91.43
16	57196006	0.2648	139.75	86.33
17	50207773	0.2324	131.90	81.24
18	44371145	0.2054	124.40	76.93
19	38127232	0.1765	117.25	71.05
20	32909865	0.1524	110.46	65.96
21	28036704	0.1298	104.01	60.87
22	24210872	0.1121	97.92	55.77
23	20430305	0.0946	92.18	49.10
24	16977202	0.0786	86.37	45.91
25	14008574	0.0649	81.75	40.49
26	11467006	0.0531	76.93	34.97
27	8725447	0.0404	72.59	2974
28	6195988	0.0287	67.74	23.68
29	4823184	0.0223	64.27	19.00
30	3842445	0.0178	61.82	15.02

Table A.4: Execution time and memory usage for channel pruning. Keras2C converted models.

Appendix B

Counting clock cycles on the Cortex M7

The different defines and functions used for counting clock cycles on the Cortex M7 CPU are shown below.

```
#include <stdint.h>

#define KIN1_DWT_CONTROL (*((volatile uint32_t*)0xE0001000))
#define KIN1_DWT_CYCCNTENA_BIT (1UL<<0)
#define KIN1_DWT_CYCCNT (*((volatile uint32_t*)0xE0001004))
#define KIN1_DEMCR (*((volatile uint32_t*)0xE000EDFC))
#define KIN1_TRCENA_BIT (1UL<<24)

#define KIN1_InitCycleCounter() \
  KIN1_DEMCR |= KIN1_TRCENA_BIT
#define KIN1_ResetCycleCounter() \
  KIN1_DWT_CYCCNT = 0
#define KIN1_EnableCycleCounter() \
  KIN1_DWT_CONTROL |= KIN1_DWT_CYCCNTENA_BIT
#define KIN1_DisableCycleCounter() \
  KIN1_DWT_CONTROL &= ~KIN1_DWT_CYCCNTENA_BIT
#define KIN1_GetCycleCounter() \
  KIN1_DWT_CYCCNT
```

Appendix C

Model

When running the command `summary()` on a Keras model, we get a summary of the different layers of the model. The summary tells which layer types the model has, the shape, and its parameters. The shape refers to the shape of the output tensor for that specific layer. The number of parameters refers to the number of weights for each layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 30, 20, 1)]       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 30, 20, 32)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 20, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 10, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 15, 10, 32)        9248
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 10, 32)        9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 5, 32)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 7, 5, 32)          9248
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 7, 5, 32)          9248
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 3, 2, 32)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 192)               0
_________________________________________________________________
dense_1 (Dense)              (None, 100)               19300
```

```
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 404
=================================================================
Total params: 66,264
Trainable params: 66,264
Non-trainable params: 0
_________________________________________________________________
```

EXAMENSARBETE Optimizing Machine Learning Inference for MCU:s
STUDENT Josefine Myllenberg & Jens Johansson
HANDLEDARE Flavius Gruian (LTH), Johan Björnstedt (Acconeer AB), Martin Löwegren (Acconeer AB)
EXAMINATOR Jörn Janneck (LTH)

Optimizing machine learning classification for edge devices

POPULÄRVETENSKAPLIG SAMMANFATTNING **Josefine Myllenberg & Jens Johansson**

Imagine that you are food shopping in the middle of the Covid-19 pandemic and managed to avoid coming in contact with other people. At the register, however, you need to touch the keypad on the card reader, potentially contracting the virus. But what if we used radar based gesture detection and AI to create a contact free option?

Such technology as our example with the contact free keypad, requires running machine learning networks on data from the radar sensor to detect which button has been pressed. Running a machine learning network is often very resource demanding and not suitable for smaller systems that are mass produced, such as the keypad in our example.

To actually be able to run a machine learning network on a small device, it is beneficial to make the network use as little memory as possible, since these systems become very expensive when the memory increase. It is also important to make sure that we have a real-time gesture classification, because who would want to wait several minutes before the network realizes that you pressed button 3? Other benefits by reducing a network could, for example, be that fewer computations mean less power consumption, i.e. the environment would be happier.

In our thesis, we have applied two optimization techniques for optimizing machine learning networks, pruning and quantization. Pruning is an optimization technique that prunes a network by removing nodes, layers, or links between the nodes, in order to make the network less memory consuming and potentially faster. Quantization is a technique where we reduce the size of the data types used to store network information. This results in a potentially faster network with lower memory consumption. Since the network information now needs less memory to be stored, the whole network needs to use less memory.

In our work we applied these optimization techniques, both on their own and together, to reduce a hand gesture classification network. The network has learned to classify three different gestures, as well as just empty space, with an accuracy of 100%, meaning that it never classify any gesture wrong.

When we applied pruning, we managed to make the network use 4.5× less memory, with an accuracy of 96%. Quantization made the network 4× smaller with an accuracy of 99%. Our optimizations also lowered the execution time for one gesture classification by 10×. In other words, we successfully managed to make the network both smaller and faster, while keeping the accuracy not lower than 96%.