ETH zürich

QUANTIZED CONVOLUTIONAL NEURAL NETWORKS FOR EMBEDDED PLATFORMS

TIMO PASCAL FAREI-CAMPAGNA

Supervised by Matthias Meyer

Co-supervised by Dr. Jan Beutel

Advised by Prof. Lothar Thiele

A thesis submitted for the degree of Master of Science

Department of Information Technology and Electrical Engineering
ETH Zurich

March 2018 – version 0.1

ABSTRACT

We present an energy efficient implementation of a convolutional neural network (CNN) for acoustic event (AE) classification on a low power microcontroller. This system may be adopted on the edge nodes of a wireless sensor network (WSN) where energy-efficiency is of utmost importance. The benefits of a classification on the sensor side rather than in the cloud are, for example, diminished data transmission costs, lower latency and reduced network congestion. For the implementation, incremental network quantization (INQ) is used to reduce the memory footprint of the CNN model. The implementation is optimized for a fast forward inference by taking into account hardware limitations like the number of general purpose registers available in the CPU core. A pipelining-like data movement strategy is adopted in order to perform the complete forward inference only on energy-efficient on-chip memory.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Matthias Meyer for his outstanding support during this project. The countless meetings we had were always very inspiring and helped me a lot to solve the challenges we faced. I am also very grateful for his smooth nature even when time was running out while still a lot of work had to be done.
I would also like to thank Dr. Jan Beutel for his excellent supervision in this project.
Furthermore, I would like to thank Prof. Lothar Thiele who gave me the opportunity to take up this very interesting project where I could learn a lot about both machine learning and microcontroller programming.
I am very grateful to the whole team at TEC, especially Akos Pasztor and Jonathan Candel for their help with all questions I had.
Special thanks to Reinhard Schuler which made me look at the generated code on assembly level by saying: "Lass dich nicht vom Compiler verarschen!" which is German for "Don't let you be fooled by the compiler!".
Finally I would like to thank André Miede for his wonderful LaTeX report style that I used to write this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AE	acoustic event
ALU	arithmetic logic unit
avg	average
CCM	core coupled memory
CNN	convolutional neural network
conv	convolutional
CPU	central processing unit
DAQ	data acquisition
DMA	direct memory access
DSP	digital signal processing
DWT	data watchpoint trigger
FFT	fast Fourier transform
FIFO	first in first out
FPGA	filed-programmable gate array
FPU	floating point unit
GPU	graphics processing unit

HAL hardware abstraction layer

INQ incremental network quantization

LL low level

MAC multiply accumulate

MCU microcontroller unit

NN neural network

UART universal asynchronous receiver-transmitter

ReLu rectified linear unit

RF receptive field

SDRAM synchronous dynamic random-access memory

SRAM static random-access memory

SIMD single input multiple data

WSN wireless sensor network

1 INTRODUCTION

Currently, WSNs are extensively used in a variety of monitoring and tracking applications [1]. Monitoring may for example facilitate the development of new early warning systems [2]. Within the PermaSense project [3] such a WSN was developed for environmental monitoring in the Swiss Alps where geological phenomena are analyzed. An overview of this system is shown in figure 1.1.

Multiple sensor nodes are mounted on the rock wall. Among other sensors, every node features a geophone that senses acoustic signals in the low frequency range up to 500 Hz. The sampled data is continuously streamed to a server through the WSN. A CNN for AE classification is used to analyze the received data in order to automatically discard unwanted signals.

Because edge devices like the sensor nodes of the WSN are usually battery-driven it is of utmost importance for them to work as energy-efficiently as possible. However, wireless data transmission is an energy-consuming task [4].
Therefore, AE classification directly on the sensor nodes using an embedded platform may reduce data transmission and thus energy consumption considerably. Instead of sending the raw data, only the corresponding label, which signifies the class of a given AE, has to be sent through the WSN. With a reduced payload that must be transmitted per sensor node the WSN may also comprise a larger number of nodes without the risk of network congestion. Additionally, the system may become more responsive because the data of the different nodes can be processed in parallel rather than sequential like on the server.

Because the classification task itself is a power hungry computation a low power MCU is a very attractive choice for this application [5]. However, state of the art CNNs that achieve reasonable prediction accuracies often consist of millions of parameters which on the one hand cause a large memory footprint and on the other hand involve an extensive amount of arithmetic and data access operations that need to be performed. Thus, they are usually implemented on powerful general purpose CPUs or GPUs.

In contrast, MCUs usually have very limited computation power which makes real-time applications with CNNs challenging. Apart from the

limited flash memory these platforms provide to store CNN model parameters the deployment of CNNs on MCUs is further exacerbated because of limited working memory. This makes it challenging to handle the large inputs and intermediate results that occur during the classification task and may require the use of off-chip memory. However, off-chip memory accesses consume about two orders of magnitude more energy than on-chip SRAM accesses [6].

In recent years, various quantization techniques have been developed. One of their goals is to reduce the memory footprint of the CNN model parameters while maintaining their original classification accuracy. Other techniques focus on lowering the necessary computation power to allow real-time processing [6]. However, many of the developed techniques require dedicated hardware like FPGAs. There has only been limited research in implementing CNNs on MCUs [5, 7, 8].

In this work we first use INQ in order to reduce the memory footprint of the acoustic CNN mentioned above. With a reduced model size we implement the CNN on the STM32F469NI MCU.
Thereby, we focus on an efficient implementation of the convolution algorithm which is known to be the bottleneck regarding the execution time of forward inference. A thorough analysis of the implementation will be performed on assembler level showing the need to take into account subtle hardware characteristics like the number of registers that are available in the MCU core.
We then address the challenge of small on-chip memory with a dedicated data movement strategy that helps to avoid using power-hungry off-chip memory.
Finally, an estimation will give a first impression whether performing the classification directly on the edge nodes of the WSN effectively reduces the overall energy that is consumed on a given sensor node.

Although the focus of this thesis will be on low frequency geophone data we will also consider more general audio data which may greatly increase the scope of application of such a WSN.

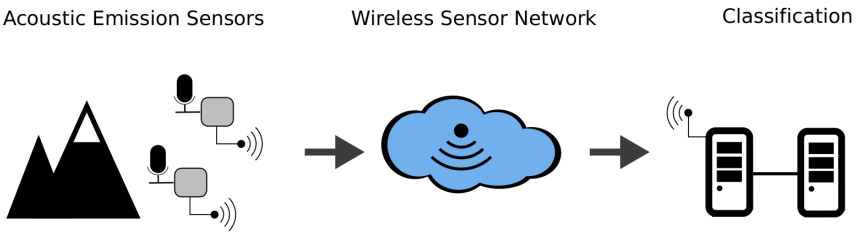

Figure 1.1: Simplified overview showing the current state of the PermaSense WSN. Geophone data is sampled and streamed continuously to a server for classification.

2

THEORY

2.1 NEURAL NETWORKS

Inspired by the biological nervous system, a NN, in machine learning, consists of artificial neurons and synapses. Consider for example the NN shown in figure 2.1 which is structured into three layers, an input layer, a hidden layer and an output layer. The neurons are represented by circles, while the synapses are represented by the lines between them.

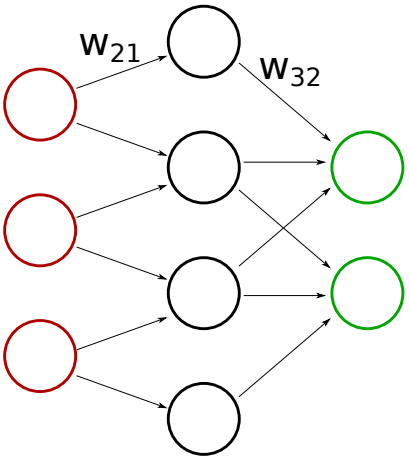

Figure 2.1: Illustration of a small neural network that consists of an input layer (red), a hidden layer (blue) and an output layer (green).

This network is essentially a function that maps a set of input numbers to some output. Each neuron contributes to this function by scaling its inputs by the corresponding weights, which represent the strength of the synapse. A nonlinear function is then applied to the sum of these products. Mathematically this can be expressed as $y = f(\sum_{i=1}^{n} x_i \cdot w_i)$ where y is the neuron's output, f is the nonlinear function and x_i and w_i are the inputs and weights respectively.

In order to compute something useful, the NN must first learn the desired input output relation. This can be achieved by training the network which means that a set of input data as well as the corresponding desired output values are presented to the network. Starting with random weights, the network can compute its output for a given input. The resulting output is taken to compute the loss which measures the distance of the actual from the desired output. Based on

this loss, the weights are updated using the well known backpropagation algorithm [9] in order to minimize the loss. Repeating the process with a large number of training samples, the network may eventually find appropriate weights which allow not only to compute the correct output on the training data but also to predict the output for previously unseen input data. Computing the output for a new input is also referred to as forward inference.

2.1.1 *Convolutional Neural Network*

A CNN is a special kind of NN which usually contains more than 3 hidden layers. The particularity of a CNN is that they mainly consist of conv layers. While a neuron in a fully connected layer is connected to every neuron of the preceding layer, the neurons in conv layers are only connected to a subset of them. Additionally, all neurons of a given layer share the same set of weights. These weights build the so-called filter kernel which is used in a mathematical operation called convolution which coined the name of this layer type. The set of neurons in the input layer that affect the given neurons output are reffered to as the receptive field (RF) of that neuron.

During training, the depth of this architecture allows the network to learn features of increasing complexity, similar to the neurons in the visual cortex of the brain. Therefore, the output of a given layer is usually called feature map which in turn serves as input for the next layer. The nonlinear function in CNNs is often the ReLu which maps negative values to zero.

2.2 INCREMENTAL NETWORK QUANTIZATION

Various techniques have been developed with the goal to manipulate NNs in order to reduce their memory footprint for both model parameters and intermediate results. Reducing the memory footprint may help to fit a CNN smaller and thus cheaper MCUs thereby also increasing the number of MCUs that can be used for a given application.

The difficulty of this task is to maintain a reasonable network accuracy. Recently, INQ has been developed which enables to reduce the memory footprint of a given NN from the original 32-bit values to 4 bits without accuracy loss [10]. In fact, for some models INQ achieves even an improved model accuracy compared to the original 32-bit versions. The strategy of INQ is to only quantize small portions of

the model parameters at a time while using re-training between successive quantization steps. Thereby, the original weights are mapped onto a limited set of numbers which are the different quantization levels. This mapping allows to introduce an encoding for the quantized weights such that each weight can be represented by a reduced number of bits. The number of quantization levels is defined by the bit-width parameter b.

The algorithm can be divided into three steps, namely weight partition, group-wise quantization and re-training which are illustrated in the top row of figure 2.2. In order to quantize a given model these three steps are repeated until all weights are either powers of two or zero as illustrated in the bottom row of the figure.
During the partition step a subset of the weights in each layer is selected to be quantized. In our work this selection is always random. In [10], however, they also used a selection mechanism where smaller weights are selected first. This idea is based on [11] where it is argued that smaller weights are less important than larger ones.
During quantization the weights are mapped to powers of two or zero, therefore enabling to replace multiplication by cheaper bit-shifting which may be interesting for cheap MCUs that are not equipped with digital signal processing (DSP) units.
After each quantization step the network is re-trained for a certain number of epochs. These three steps are repeated until, ultimately, all weights of the network are quantized.

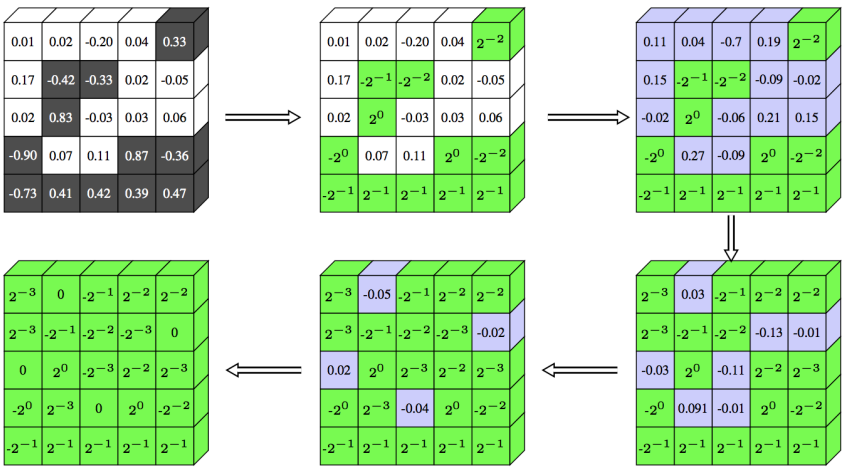

Figure 2.2: Illustration of the the three steps in INQ on a 2D weight matrix. First, a subset of weights is selected to be quantized. Second, the selected weights are quantized. Third, the remaining weights are re-trained to account for the quantization error in the previous step. These three steps are repeated until finally the complete weight matrix consists only of quantized weights. Figure from [10].

3

SYSTEM OVERVIEW

The main goal of this thesis is to implement a CNN for AE classification efficiently on a MCU. This system may then be adopted on the edge nodes of the WSN introduced in section 1. This section describes the working principle of a given sensor node and introduces both the acoustic CNN and the MCU that are used.

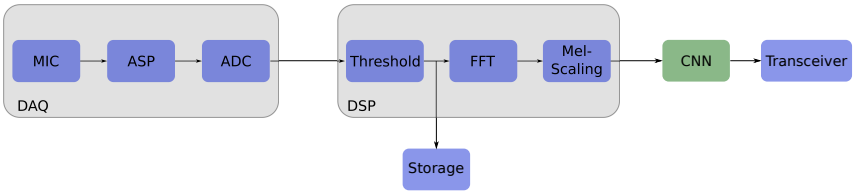

Figure 3.1: Block diagram showing the simplified processing chain of a sensor node which is equipped with the CNN to perform data classification in real-time.

The block diagram in figure 3.1 shows the simplified working principle of a given sensor node. During the data acquisition (DAQ) phase the analog voltage levels that are generated by a geophone sensor are amplified, filtered and converted into a digital signal.
A digital thresholding scheme is then applied to the samples. Only a signal that exceeds the threshold is on the one hand stored on memory and on the other hand subject to a fast Fourier transform (FFT) followed by Mel-scaling.
The resulting spectrogram is then processed by the CNN for classification and the determined label is sent through the WSN.

As already mentioned in section 1, we are interested to not restrict ourselves to the low frequency geophone data. We also consider the scenario of more general microphone data which contain higher frequency components.
A summary of all relevant signal information and DSP parameters is shown in table 3.1.

3.1 ACOUSTIC CONVOLUTIONAL NEURAL NETWORK

For the AE classification task we use the CNN-CNP network that was developed in [12]. This network was inspired by the CNNs introduced in [13]. Two modifications, however, resulted in a new architecture that is optimized for embedded platforms. An overview of this net-

	GEOPHONE DATA	MICROPHONE DATA
Sample rate	1 kHz	16 kHz
FFT window size	1000 ms	32 ms
FFT hop size	512 ms	16 ms
Mel-frequency bins	64	64

Table 3.1: Signal information of geophone and microphone data.

work is shown in table 3.2.

The network can be divided into two sections, one for feature extraction and one for classification. The first section consists of four conv layers. The usual max pooling operation was removed and instead, the stride parameter of the respective layers has been set to two. This is the first of the above mentioned modifications which was proposed in [14] and resulted in a reduced number of MAC operations. The second section uses three conv instead of fully connected layers for classification. Together with a global average (avg)-pooling layer this is the second of the above mentioned modifications which drastically reduced the number of model parameters and hence the memory footprint of the CNN. The softmax activation function generates the output vector that consists of 28 elements which correspond to the 28 classes present in the acoustic event classification dataset that was generated in [13]. The complete network has about 452k parameters and achieves an accuracy of 85.1% [12]. After every conv layer a ReLu activation is performed.

Throughout this work we assume an input shape of $(FxT) = (64x400)$, where F is the number of Mel-bins and T is the number of Mel-frequency vectors. For this input the resulting number of MAC operations is 1239M.
Given the network architecture, the biggest intermediate result is defined by the output of the first conv layer which has the shape (64x400x64). This intermediate result is in turn the input for the second network layer which computes a new output of shape (32x200x64). In order to compute this output the memory of a MCU must at least contain the former intermediate result while providing some space for the new output. Therefore, the necessary amount of memory is at least 6.25 MB[1].
To estimate the computation power that is necessary for real-time processing the duration of an AE must be known. With a hop size $t_{h,fft}$ and a window size $t_{w,fft}$ the duration can be calculated as $L = t_{w,fft} + (T-1) \cdot t_{h,fft}$. For the geophone and microphone data this

1 32-bit data

LAYER TYPE	# PARAMS.	# MAC
conv 3, 1, 64	640	14.8M
conv 3, 2, 64	36.9k	236.0M
conv 3, 1, 128	73.9k	471.9M
conv 3, 2, 128	147.6k	236.0M
conv 3, 1, 128	147.6k	236.0M
conv 1, 1, 128	16.5k	26.2M
conv 1, 1, 28	3.6k	5.7M
avg pool	0	0
softmax	0	0
Total:	452k	1239M

Table 3.2: Structure, parameter count and number of required MAC operations of the acoustic CNN. Convolutional layers are defined as filter size, stride, number of filters. The horizontal line separates the section for feature extraction from the classification section.

	GEOPHONE DATA	MICROPHONE DATA
Model parameters	1.73 MB	1.73 MB
Intermediate results	6.25 MB	6.25 MB
MAC-speed	6M MAC/s	193M MACs/s

Table 3.3: The first two rows show the size of the model parameters and the biggest intermediate result during forward inference. The third row shows the MAC-speed required for real-time processing for both geophone and microphone data. Data size is shown in 32-bit format.

formula evaluates to 205.312 s and 6.416 s respectively (table 3.1). Hence, the necessary MAC-speed for real-time processing is about 6M MACs/s and 193M MACs/s respectively. To summarize, the most relevant numbers are listed in table 3.3.
Remember that one of the motivations to deploy the CNN on the sensor nodes rather than on the server is the reduced amount of data that needs to be streamed through the WSN which may reduce the energy consumption of a sensor node considerably. With the above computed AE-duration and the sample rates provided in table 3.1 the size of an AE can be found to be 401 kB and 802 kB for microphone and geophone data respectively[2]. Compared to the 5 bit label needed to encode the 28 classes this is a reduction by more than factor 10^5.

2 32-bit data

3.2 MICROCONTROLLER UNIT

For our investigations we use an STMF469NI MCU which is integrated on the STM32F469I discovery board from ST [15]. This MCU features a 32-bit Cortex-M4 CPU with 12 general purpose registers, a floating point unit (FPU) with additional 32 single precision registers and it implements a full set of DSP instructions including the MAC operation for fixed point numbers, that is the operation that must be performed excessively in a CNN. Furthermore, the MCU features 320 kB SRAM, 64 kB core coupled memory (CCM) and 2 MB Flash memory. The CPU clock ranges up to 180 MHz. At a 3.3 V power supply the typical current consumption of the MCU lies between 2 mA and 103 mA depending on the selected clock speed and the peripherals that are enabled. The discovery board provides an additional 16 MB off-chip synchronous dynamic random-access memory (SDRAM).

Given the above specifications the question arises whether the acoustic CNN from the previous section can be implemented while satisfying the processing and memory requirements of the application. The relevant information to answer this question can be found in table 3.3.

Because the model parameters are essentially part of the forward inference algorithm they can be stored on flash memory. The 2 MB flash memory of the STM32F469NI MCU are enough to store the acoustic CNN which needs about 1.73 MB as we have seen earlier. There are, however, different variants of this MCU [16]. For example the STM32F469NE MCU is very similar but features only 512 kB of flash memory. The benefit of a smaller flash memory is that the area cost is reduced which makes the MCU cheaper [17]. Here, we only use the STM32F469NI for our investigations while in the end another version may effectively be used. Therefore, it is still of great interest to reduce the memory footprint of the acoustic CNN.
Considering the memory footprint of the forward inference the 320 kB SRAM is clearly too small to store the biggest intermediate result on on-chip memory. Remember that our goal is to only use on-chip SRAM because the use of off-chip SDRAM comes with two major drawbacks. First, SDRAM must be continuously recharged which may cause a lot of energy consumption. Second, during the recharge process the memory cannot be accessed which may result in delays during forward inference [18]. Additionally, SRAM accesses cost about two orders of magnitude less energy than SDRAM accesses [6]. The challenge of reducing the inference memory footprint will be addressed in section 4.3.

4 METHODS

We first use INQ to prepare a CNN in order to implement it with a lowered memory footprint on a MCU. Once our INQ implementation is verified we implement three code variants which will be optimized with respect to inference time which is of utmost importance in real-time applications.

The first variant uses floating-point arithmetic, thereby exploiting the microcontroller's FPU. Given that NNs are usually implemented with 32-bit floating-point data on general purpose CPUs, this variant additionally serves as a baseline to verify a correct implementation of the complete algorithm.
Both other variants use fixed-point arithmetic which generally consume lesser energy compared to floating-point operations[4]. The first one takes advantage of the single-cycle MAC instruction provided by the DSP unit of the MCU. The second one is designed to be executed on the arithmetic logic unit (ALU). This version takes advantage of INQ by using binary bit-shifting which may be more energy-efficient than standard fixed-point multiplication [10]. Note that this is only possible because INQ generates convolution filter coefficients that are either powers of two or zero. Such an implementation may be of special interest on more basic MCUs which do not feature a FPU or DSP unit.
Throughout this work the three code variants will be referred to as V0, V1 and V2 respectively.

Instead of using our target CNN (from section 3.1) to implement and compare the three code variants we use an example network from [19]. This CNN allows to ignore the problem of a too small SRAM in a first step, as will be described below.
An evaluation of the inference time shows whether the necessary MAC-speed for real-time processing can be achieved practically.

With an optimized implementation of the convolution algorithm we address the memory problem that arises for the acoustic CNN. Here, a measurement of the inference time will verify whether the real-time constraints of the AE classification system can be met and a measurement of the energy consumption will help estimating whether it is worthwhile to perform the classification on the sensor node rather than on the server.

For the INQ-implementation we use Keras, one of the available machine learning frameworks written in python, together with Tensorflow as backend. If not stated otherwise, training and quantization of the CNNs is performed on an Intel Core i7-4710MQ CPU on a lenovo ThinkPad T440p machine, running ubuntu 16.04 LTS. Implementations on the MCU are made on Windows 10 OS using the Eclipse IDE set up as described in [20].
The clock tree of the MCU is programmed with STMCube to provide a 168 MHz CPU clock. The complete code is written in the popular C programming language. For critical code sections, which must be as efficiently as possible, we rely on the CMSIS drivers to manipulate MCU registers. For other code sections we use the hardware abstraction layer (HAL) and low level (LL) drivers which are not as efficient but simplify programming substantially while making the code portable to other devices.

4.1 INQ-IMPLEMENTATION

The python code is divided into three files. INQ.py is used to execute the usual functions like loading a model and dataset, data preprocessing and controlling the training procedure of a given model. Additionally, it features a function called customize_model() which is used to customize a standard Keras model for our needs. Thereby, the architecture of a given model is analyzed and both conv and fully connected layers are replaced by custom layers which are defined in INQ_layers.py. During this process it is important to keep the original weights in order to not loose the accuracy of the pretrained CNN.
INQ_layers.py contains customized versions of the standard *Conv2D* and *Dense* classes which are essentially the layers used to build a CNN in Keras. Each of the customized classes provide two functions with the names partitioning_weights() and quantize() which perform the actual weight partitioning and quantization steps that were described in section 2.2.
The last file is called INQ_model.py and defines a class that derives from the *Sequential* class provided in Keras to define a model and ist only used during the customization process and to easily loop through all layers of the model during the quantization steps.

As explained in section 2.2 the model is retrained, once a given portion of weights has been quantized. In order to avoid that Keras overwrites the already quantized weights with re-trained values we chose to adopt the computation graph illustrated in figure 4.1.
We defined three kernel variables. One is called kernel_nontrainable which is used to store quantized weights during the quantization procedure. This variable is saved as a non-trainable variable in the Keras model. A second variable is called kernel_trainable. According

to its name this variable is set as trainable in the Keras model and is therefore updated with the retrained weights during backpropagation. In order to provide the correct combination of trainable and quantized weights for the forward inference during training, a mask variable was introduced as illustrated in the figure. The mask has the same shape as the two kernel variables. Its values are equal to one at the indices where the weight is still trainable. At indices where the kernel has already been quantized, however, its values are equal to zero. Through the arithmetic operations shown in the computation graph Keras uses the correct combination of trainable and quantized weights for the forward inference.
According to [10] the weights of the layers are quantized separately. The reason for this is that the dynamic range of the weight distribution may differ substantially in different layers. Thus, a quantization over all layers together may result in a large quantization error.
However, no information was provided about the biases. Therefore, in our implementation we decided to also quantize the biases separated from the weights for the same reason.
Another difference to the original implementation is that we do not make use of the pruning-inspired approach during the partitioning step of INQ.

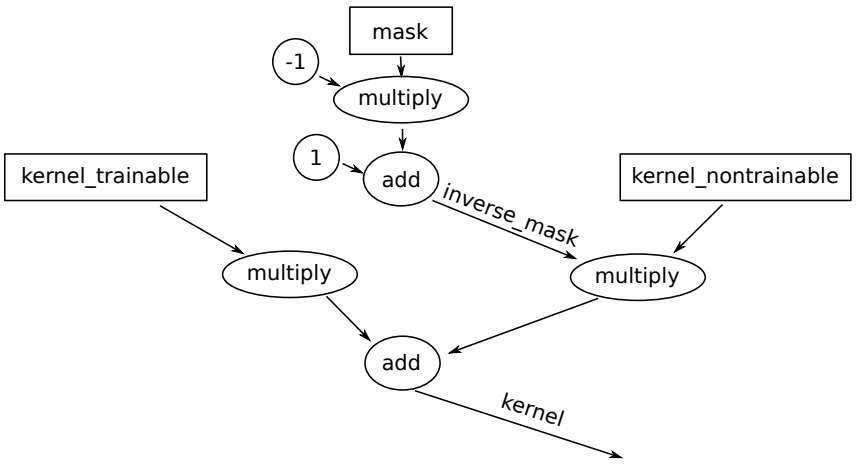

Figure 4.1: Illustration of the computation graph that was used to mask the backpropagation updates during when performing INQ. The non-trainable kernel is used to hold quantized weights. The trainable kernel is updated with the gradients during model training. The mask is used to provide the right combination of quantized and trainable weights for the forward inference.

4.2 REDUCING THE INFERENCE TIME

Once the INQ-algorithm is verified the three code variants V0 - V2 can be implemented in order to optimize the inference time and compare them in terms of energy consumption.

Because our target CNN from section 3.1 is purely convolutional and the convolution algorithm is essentially identical for every CNN we use another network for this optimization task as mentioned above. This network is designed for the CIFAR-10 dataset [21] and is one of the example CNNs that can be found on the Keras website[19].
The architecture of the network is shown in table 4.1. With a maximum intermediate result of 128 kB we can allocate two static arrays of this size on SRAM. This allows a very simple data flow. In particular, the two arrays can be used as input and output buffers of any given layer during forward inference. This approach enables us to neglect the challenge of the inference memory footprint for now in order to focus completely on optimizing the convolution algorithm.

As can be seen the number of parameters of the CIFAR-10 network is 1.3M which would actually require 4.96 MB of flash memory[1]. Here, the power of INQ becomes evident. With a bit-width of b = 8 bits per weight the model size is reduced to about 1.27 MB which easily fits on our 2 MB flash memory.

The inference time in CNNs is usually dictated by the convolutional layers due to there large number of MAC operations. This can be seen when comparing the 19M MAC operations of the first network section with the 6M MAC operations of the second section. Together with the fact that our target CNN is fully convolutional we restrict the optimization of the three code variants to the implementation of the convolution operation.

In case of V0 we implement the convolution according to the pseudocode shown in algorithm 1. Notice, that every filter channel is loaded exactly once outside the two most inner loops, that is, the respective filter coefficients are re-used as long as possible and only the input channels are loaded inside the loops. This is very important especially given the fact that the weights need to be determined indirectly via a look-up table because of the encoding introduced by INQ. That is, loading a weight essentially causes two memory accesses while loading an input costs only one memory access.

As mentioned above, the two code variants V1 and V2 do not make use of the FPU. Therefore, they do not have the extra register bank that is available in the FPU (section 3.2). Instead, they must rely on the 12 general purpose registers of the CPU. One might think that 12 registers are enough to store 9 filter coefficients and only a single register is needed to load the inputs sequentially. However, a certain amount of registers is needed for other purposes like for example to store the

1 32-bit data

LAYER TYPE	# PARAMS.	# MAC
conv 3, 1, 32	896	885k
max-pooling	0	0
conv 3, 1, 32	9.2k	8.3M
max-pooling	0	0
conv 3, 1, 64	18.5k	4.1M
max-pooling	0	0
conv 3, 1, 64	36.9k	6.2M
max-pooling	0	0
fc, 512	1.2M	1.2M
fc, 10	5.1k	5.1k
softmax	0	0
Total:	1.3M	20.7M

Table 4.1: Architecture, parameter count and number MAC operations of the CIFAR-10-CNN that is used to optimize the convolution algorithm.

break conditions of the loops. Therefore, we split the two most inner loops from algorithm 1 into three separate sections where the rows of a given 3x3 kernel are loaded individually as shown in algorithm 2. With this strategy we make sure that the filters are still loaded exactly once outside the two most inner loops in order to avoid redundant memory accesses.

Algorithm 1 : Pseudocode for the floating-point version V0 showing 2D convolution with 3x3 filter kernels. With the additional register bank of the FPU it is possible to load all filter coefficients two most inner loops.

```
initialize;
for each output channel do
    for each input channel do
        load 3x3 filter channel;
        for each row do
            for each column do
                load 3x3 input fragment at the current position;
                do convolution;
                store result;
```

Algorithm 2 : Pseudocode for the fixed-point versions V1 and V2 showing 2D convolution with 3x3 filter kernels. With only 12 general purpose register available in the CPU it is necessary to load the filters row by row to avoid redundant load instructions within the two most inner loops.

```
initialize;
for each output channel do
    for each input channel do
        load 1st 1x3 filter row;
        for each row do
            for each column do
                load 1x3 input fragment at the current position;
                do convolution;
                store result;
        load 2nd 1x3 filter row;
        for each row do
            for each column do
                load 1x3 input fragment at the current position;
                do convolution;
                store result;
        load 3rd 1x3 filter row;
        for each row do
            for each column do
                load 1x3 input fragment at the current position;
                do convolution;
                store result;
```

4.3 REDUCING THE INFERENCE MEMORY FOOTPRINT

In section 3.2 we have seen that the SRAM of 320 kB is too small to store a 6.25 MB intermediate result. One technique to reduce the necessary memory is to use 16-bit instead of 32-bit data format which effectively halves the necessary memory footprint. In our case, however, the data would still be too large to be stored on SRAM only.

A possible approach to overcome this problem is to store the input on off-chip memory and employ a DMA controller to transfer the input in smaller portions to the on-chip SRAM where it is efficiently processed by the CPU. The computed partial intermediate results may then be transferred back to the off-chip memory to provide space for the next portion of the input. This technique, however, results in a large memory-bandwidth on both SDRAM and SRAM which may cause a considerable energy consumption.

In order to investigate for another strategy to effectively reduce the inference memory footprint let us consider a very simple CNN. On the left, figure 4.2 shows the common visualization of the forward inference in case of a CNN that consists of a single conv$(3,1,1)$ layer with ReLu activation. Now, assume that the input data is provided over a certain period of time. In this case it would be nice to start the process of forward inference already when only a fraction of the input is stored in memory rather than waiting for the complete, possibly very large, input. This scenario is illustrated on the right side of the figure. Given that each output neuron has a RF of size three, an input buffer with three elements is necessary. Starting at the top right in the image, the input buffer is pre-initialized to zeros, waiting to be filled with the input. Additionally, an output buffer with the expected output size of four elements is also initialized with zeros. Having in mind that the input vector is provided over a certain amount of time we can imagine that it is shifted into the input buffer with a given update rate. As the stride is equal to one in this example, the input is shifted element by element through the input buffer.
With this process the first output neuron can be computed in the fourth step, as indicated by the green numbers. From this moment on, every new input element "pushes" the leftmost element out of the input buffer like in a FIFO queue. Each such update allows to compute another element of the output buffer. Note that in each step also the computed output elements are shifted by one element which will become clear below. This process is repeated until, in the last row, the same output as on the left side of the figure is computed. Hence, instead of storing the complete input of six elements, the correct output was now computed by only storing three input elements at a time.
Note, that the second output element on the left is set to zero by

ReLu activation which corresponds to the fifth step on the right. Note also, that the amount by which the inference memory footprint is decreased depends on the length of the input vector. That is, the larger the input vector on the left, the more memory is saved using the buffer system on the right.

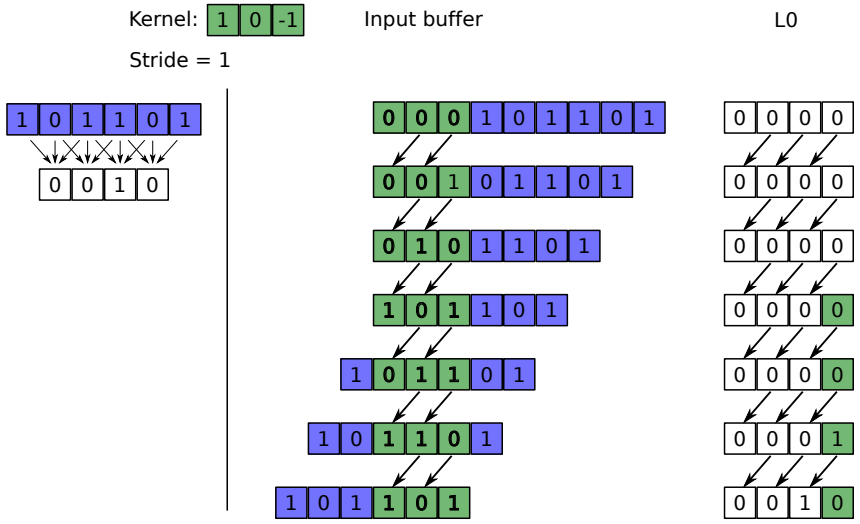

Figure 4.2: Illustration of the buffer system we use to run the forward inference only using on-chip memory. The inference for a conv(3,1,1) single-layer CNN with ReLu activation is shown. The usual representation on the left is compared with the buffered system on the right.

With the same idea we can consider the CNN from the previous paragraph but with the stride parameter increased to two. This scenario is illustrated in figure 4.3. On the left, an input of nine is mapped to an output of four elements. On the right, the output is again computed step by step in the same manner as described before. Note that due to the increased stride, however, two input elements are shifted into the buffer at a time. For simplicity the input is not shown anymore.
There are two small differences compared to the previous example. First, with a stride equal to two, the input buffer must consist of four rather than three elements. Otherwise, the first input element would be shifted out of the buffer before it has been utilized to compute the corresponding output element. Second, there is only a single number left at the end of the input stream because the input has an odd number of elements while it is always shifted in groups of two elements into the buffer for a given step. Since the programmer of a CNN knows the expected input shape for a given network layer it is easy to take into account this edge effect by forcing an additional zero at the end of the input.
Similar to the previous example, the necessary memory to store the input is reduced but here, it consists of four rather than the previous three elements.

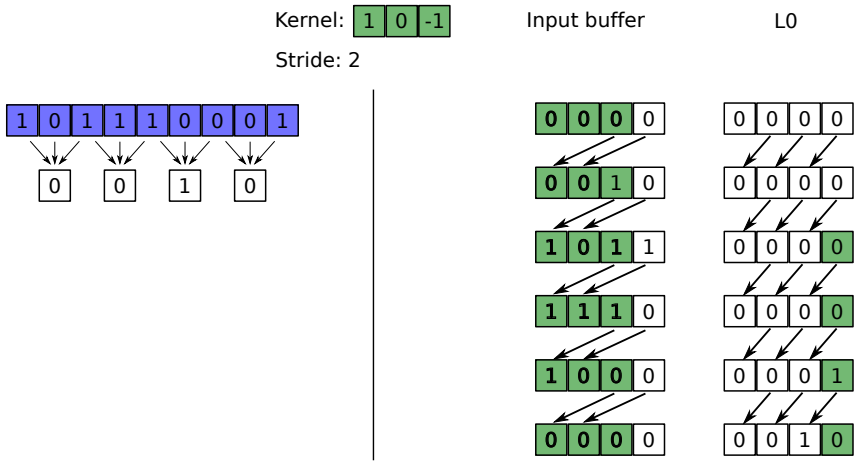

Figure 4.3: Illustration of the buffer system for a conv(3,2,1) single-layer CNN with ReLu activation is shown. The usual representation on the left is compared with the buffered system on the right.

In both examples we observe that the number of MAC operations does not change when comparing the two different approaches to do the forward inference.

In order to see what happens for multi-layer networks, consider figure 4.4 where the above two networks are stacked on each other. Again, the output of the left scenario can be computed step by step when using the correct buffer size for each layer.
In contrast to the first example, however, the input buffer consists of four elements rather than three even though the first layers of both networks have the same stride parameter. The reason for this can be found when considering the RF of the output neurons in the left scenario. The RF of the first output neuron consists of the first five input elements as indicated by the red bounding box. The RF of the second output neuron is indicated by the green bounding box which is essentially a shifted version of the red RF with a shift amount of two elements. More generally, the RFs of two successive output neurons are shifted versions of each other with a shift amount of two in this network. For the scenario on the right this means that, like in the previous example, two input elements are provided per step in order to compute a new output neuron in every step. Because the first layer has its stride parameter equal to one, however, two new values have be computed per step in its output buffer. This explains why this buffer has four elements rather than three.

In general the input must be shifted by the product of all layers' stride parameters starting from the output layer. This observation can be generalized to the input buffers of hidden layers as follows. The shift amount at the input of a given layer is the product of the stride parameters of all layers, starting from the network output up to the

layer that is considered. This number describes by what amount the local RF of two successive output neurons, seen in the input of the considered layer, are shifted from each other.
Mathematically, this number can be expressed by equation 4.1 where s_l is the shift in layer l, L is the total number of weighted layers and str_i is the stride parameter in layer i. Note that layer indexing is from zero to $L-1$ rather than from 1 to L. In the present example this formula evaluates to $s_0 = 2 \cdot 1 = 2$ and $s_1 = 2$. The number of elements that need not be computed but, instead, are only copied from the previous step is equal to $k-1$ where k is the filter size of the succeeding layer. In this example, k is equal to three in both layers, thus always $k-1=2$ elements are copied per step.
Hence, the parameters k and str of a given layer determines the shape of its input buffer which is illustrated by the two colours in the respective buffers. Note that the output buffer has always the size of the expected output and only one element in this buffer is computed per step which is equivalent to say that $s_{L-1} = 1$. Notice also that in this example not only the memory, needed to store the input but also the memory that is needed for the input of the second network layer is reduced. In general, the size of every buffer except for the last one can be reduced in this way.

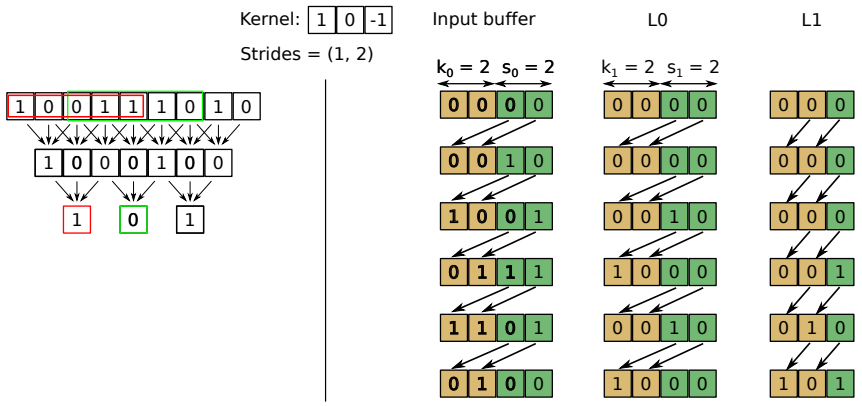

Figure 4.4: Illustration of the buffer system for a two-layer NN consisting of a conv(3,1,1) and conv(3,2,1) layer with ReLu activation.

$$s_l = \prod_{i=l}^{L-1} str_i \tag{4.1}$$

The same idea can be adopted for layers that produce more than a single output channel.
One may also imagine the input to consist of multiple rows like in an image. In this case all rows of the input are shifted together into a buffer system of appropriate size similarly as described above. In fact, the examples above can be considered to be a special case where the inputs are images that consist of only a single row of pixels.

If padding is applied in the layers, special attention must be paid at the edges of the inputs. Apart from some small modifications, however, the same idea can still be applied.

The question arises whether we can use the above method to process the FxT-Matrix which is the input in our target CNN as mentioned in section 3.1. Note that the Mel-spectrogram of an acoustic signal can also be interpreted as an image. The difference to an actual image, however, is that the individual columns of the input are computed sequentially over time which is exactly the input property we may exploit with the buffer system introduced above. Therefore, we adopt this idea for our acoustic CNN which will be described in section 5.3.4.

5 EXPERIMENTS

5.1 INQ VERIFICATION

To verify the our INQ implementation the well known VGG16 image classification CNN [22] was quantized similarly as described in [10]. A description of the dataset used to train VGG16 is provided in the following section. The relevant quantization parameters are provided in section 5.1.2.

5.1.1 *Data Set*

The VGG16 model was trained on the ImageNet 2012 dataset [23]. This data set consists 1.3 million images each containing one of 1000 object classes. The validation set consists of 50000 images.

5.1.2 *Quantization*

For quantization the nework was loaded from keras. Quantization was performed with similar parameters as described in [10]. The bit-width was chosen as b = 5. The model was retrained for one epoch after each quantization step. The learning rate was chosen as 10^{-5} which is the final learning rate that was used during the initial training in [22]. The train batch size was set to 32 and the momentum to 0.9. The cumulative portion of quantized weights is set to [0.5, 0.75, 0.875, 1].

To provde the data using a custom ImageGenerator in keras, the images were stored in a hdf5 ffile. To ease this process the original images were first rescaled isotropically and then center cropped to (256x256) images. During retraininig a random crop and during evaluation a centercrop of size (224x224) was taken from the images stored in the hdf5 file. Although the documented top5 accuracy in keras is 90.1% the model achieved an accuarcy of 89% on the validation set. A possible explanation for this may be the way we stored the data set in the hdf5 file. Because we only want to verify the correct implementation of INQ, however, the accuracy is good enough. Data preprocessing was performed as described in [22] but no RGB colour shifts were applied to the images during training.

The quantization was performed on a NVIDIA TITAN X GPU featuring 12 GB working memory.

5.2 REDUCING INFERENCE TIME

Before the CIFAR-10-CNN can be implemented on the MCU it must be trained and then quantized appropriately. The next two subsections describe the initial training and quantization procedures for this network. Section 5.2.4 describes how the MCU implementation is analyzed.

5.2.1 *Data Set*

The data set consists of 60000 32x32 RGB images which contain one of 10 classes [21]. The data set is split into a training set containing 50000 images and a test set with 10000 images.

5.2.2 *Initial Training*

For the training procedure the input images are normalized but no further preprocessing of the data is done. The learning rate is 10^{-4} and the weight decay is set to 10^{-6}. The train batch size is 32 images.

5.2.3 *Quantization*

The quantization process is performed with a bit-width of b = 8. The accumulated portions of quantized weights is selected as [0.25, 0.5, 0.75, 0.875, 0.9, 1] and the number of epochs in each step is set to 2. All other parameters are identical to the initial training parameters.

5.2.4 *Experiments on the MCU*

The network is implemented with the convolution algorithms described in section 4.2. For the fixed-point versions V1 and V2 the number of decimal places is set to 14.

The input data is streamed to the MCU through a UART connection. A DMA controller is set up to move the received data from the UART peripheral to a specified SRAM address. After a complete input image has been received together with its label an interrupt is triggered for the CPU to process the new input.

First, the network implementation on the MCU is verified on the complete CIFAR-10 test set. Then, the implementation is discussed on assembly level and the number of clock cycles needed to execute the most inner loop of the convolution algorithm is measured. Ultimately, the inference time and energy consumption are measured as an av-

erage over 100 input images. To measure these quantities we use a RocketLogger [24] at a sampling rate of 1kHz.

5.3 REDUCING THE INFERENCE MEMORY FOOTPRINT

5.3.1 *Data Set*

In order to train the acoustic CNN an AE data set was used that was developed in [13]. It consists of 28 event types that are harvested from a free online source. The total length of the 5223 audio files is 768.4 minutes sampled at 16 kHz. The data set is split into training and test set with a ratio of 0.75.

5.3.2 *Initial Training*

For the training procedure of the acoustic CNN the above data set is preprocessed as follows. The audio files of a given class are first normalized and glued together to build a single file. The Mel-spectrogram of this compound of AEs is then calculated with a window length of 32 ms, a hop size of 50% and 64 Mel-bins. Finally, this spectrogram is divided into $64x400$ matrices.
The network is trained for 21 epochs. The learning rate is set to 0.001. The train batch size is 32.

5.3.3 *Quantization*

The acoustic CNN is quantized using INQ with the quantization bit width set to b = 8. The cumulative portions of quantized weights is set to [0.25, 0.5, 0.75, 0.875, 0.9, 0.95, 1]. All other training parameters as well as data preprocessing are identical to the initial training procedure.

5.3.4 *Experiments on the MCU*

In this experiment we adopt the buffer system that was introduced in section 4.3 for our acoustic target CNN. The input data is again continuously streamed via a UART connection. In this scenario, however, the input is processed in small fragments instead of waiting for the complete input to be received. The DMA controller is programmed to trigger an interrupt after each input fragment has been received. Here, the baud-rate of the UART connection can be used to simulate the real-time nature of our AE detection application.
Note that we only use microphone data for this experiment. Because for geophone data the expected input shape is identical and only the duration of an AE is different, the same measurements can be used to

make statements about the system when considering geophone data.

The convolution is implemented using the floating-point implementation Vo. First the prediction accuracy is verified. Then, the average energy consumption and inference time as well as the effective MAC-speed is analyzed.
Additionally, the average energy consumption over a set of three AEs is measured to take into account the energy consumption during the sleep cycles between the processing of the input fragments. This measurement is used to estimate whether it is worthwhile to employ the acoustic CNN on the sensor nodes of the WSN.

6 RESULTS

6.1 INQ VERIFICATION

The figure shows the top1 and top5 validation accuracy during network quantization. Starting with an initial top5 accuracy of 89% the accuracy drops by about 1% in the first quantization step. After this step the top5 accuracy stays at about the same level as the initial value and the final accuarcy is 88.8%. As this experiment was only executed for verification of our INQ implementation these results are satisfying.

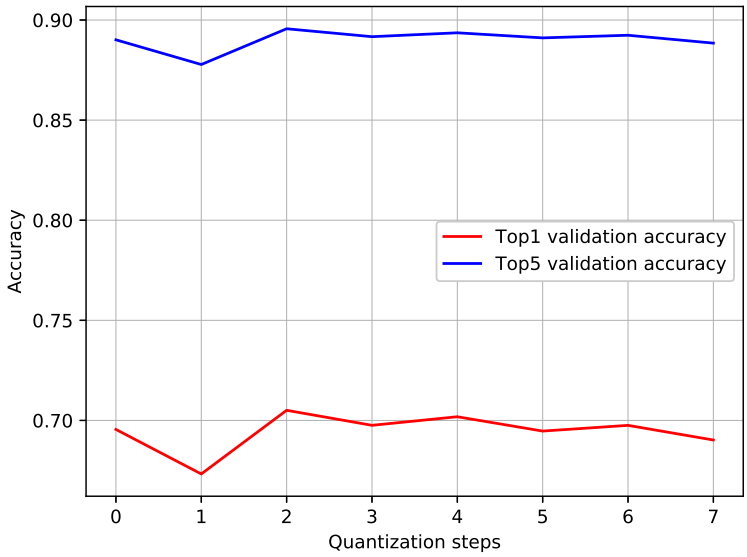

Figure 6.1: Top1 and top2 validation accuracy versus quantization steps for the VGG16 CNN. At x = 0 the accuracy of the pre-trained network is shown. The subsequent values show the network accuracy after quantization and retraining in an alternating manner.

6.2 REDUCING THE INFERENCE TIME

6.2.1 *Initial Training*

Figure 6.2 shows the training progress of the CIFAR-10 CNN. According to [19] the network accuracy should achieve 79%. In our training procedure, however the accuracy achieved 78.25%. The main reason for this is that we did not perform any data augmentation.

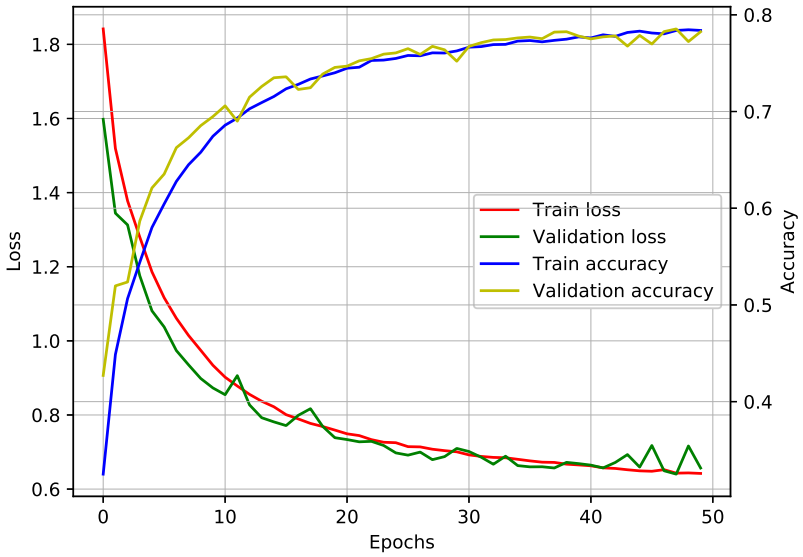

Figure 6.2: Training and test accuracy and loss versus training epochs of the CIFAR-10-CNN.

6.2.2 *Quantization*

Figure 6.3 shows the quantization history of the CIFAR-10 CNN. Although the network accuracy drops by about 6% and 8% during the first and second quantization steps respectively it is successfully increased during the following retraining steps. During later quantization and retraining steps the quantization error is much smaller because smaller portions of weights are quantized. With a final accuracy of 78.1% the result is satisfying for our planned implementation on the MCU.

6.2.3 *Test on MCU*

6.2.3.1 *Network Accuracy*

The three code variants V0, V1 and V2 all achieved an accuracy of 78.1% on the MCU which ensures that the implemented algorithms are correct.

6.2.3.2 *Clock Cycle Analysis*

Recall that the convolution is implemented with four nested loops. Because the most inner loop computes the actual convolution it is this code section which forms the bottleneck of execution time. Therefore, we focus only on these code segments in the three implementations here.

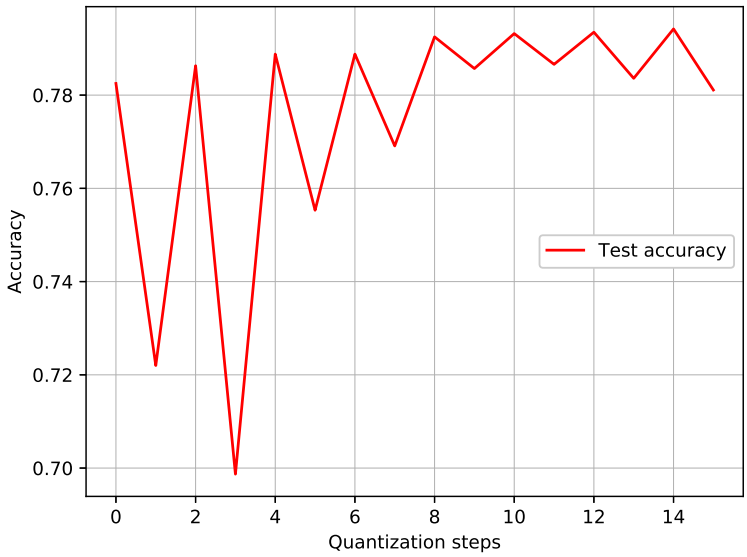

Figure 6.3: Test accuracy versus quantization steps for CIFAR-10-CNN. At x = 0 the the accuracy of the pre-trained model is shown. The subsequent values show the network accuracy after quantization and retraining in an alternating manner.

Disassembling the executable that was built for the floating-point implementation results in the assembly code shown in listing 6.1. For a given 3x3 filter kernel there are nine corresponding input elements that must be loaded from SRAM. This is done by the vldr instrucions. A tenth vldr instruction is necessary to load the content of the output memory cell to which the current convolution result is added to. The corresponding instruction can be seen in the second code line which loads the content of the memory address stored in register r0 into the FPU register s15. There are nine vfma.32 instructions perform the actual MAC operations.
This implementation was found to consume the least number of CPU clock cycles when using three pointer variables. Each one of them contains the start address of one of the three rows in the current input fraction. Incrementation of the pointers which is necessary for the next loop cycle is done with the three adds instructions. The vstmia instruction at the bottom of the code stores the computed result to SRAM. The cmp instruction checks the loop termination condition and sets a hardware flag either to true or false. The final bne.n instruction checks this flag and jumps out of the loop if it is true, or to the loop's start address otherwise.

The generated assembley instructions for the two fixed-point implementations are shown in listings 6.2 and 6.3 respectively. Remember that the convolution filters are loaded row by row in both versions

(algorithm 2).
Regarding the cycle count, the code is identical for all three rows of a given filter. Thus, the code listings show only the code that processes a single row. The ldr, str, cmp and bne instructions can be explained similarly as before. The difference of the two versions is that V1 makes use of the MAC instruction (mla) while V2 uses the combination of a shift (asr or lsl) and add or subtract (sub) instruction.
Although the technical reference manual [25] provides a lot of information about how many clock cycles a given instruction needs it is difficult to take into account the effect of the pipeline and branch instructions. Fortunately, the MCU is equipped with a data watchpoint trigger (DWT) which we used to measure the number of clock cycles needed for the above code sections.
To run the respective code in V0 this measurement resulted in 45 clock cycles. For the two fixed-point versions V1 and V2 the number of clock cycles was measured to be 13 and 18 respectively. However, the fixed-point versions both contain the most inner loop three times. Therefore these numbers can be multiplied by three which results in 39 and 54 clock cycles respectively. Hence, V1 should be the fastest and V2 the slowest of the three implementations.

The main contributor to the smaller number of cycles needed in V1 is the single-cycle MAC instruction provided by the DSP unit. In contrast, the floating-point MAC instruction needs three clock cycles. The drawback of V2 is obviously the additional add or sub instruction involved after each binary bit-shift operation when compared to V1. Compared V0, both fixed-point versions execute the fix costs of a loop (cmp and bne) three times more often. Also, the two fixed-point versions execute 3x more str instructions compared to V0.

6.2.3.3 *Inference Time and Energy Consumption*

The previous section indicated that V1 would be the fastest of the three code variants. In the following, the duration for the three implementations to process a single CIFAR-10 image is analyzed. Remember that our acoustic target CNN is fully convolutional. Therefore, the measurement is only performed for the conv section of the CIFAR-10 network here.
The results are listed in table 6.1. On average V1 needs 598 ms to process an image which is about 13% faster compared to V0 which needs 687 ms. V2 needs 763 ms and is clearly the slowest of the three. These measurements are in agreement with the cycle counts from the previous section. Given these numbers we can see that only very few extra cycles inside the convolution algorithm can significantly slow down the forward inference.
The effective MAC-speed for the different implementations can be computed as 28.38M, 32.61M and 25.56M MACs/s. Therefore, on this

MCU all three code variants satisfy the real-time constraints of the geophone scenario with a large margin. For the microphone data, however, a substantial speed-up is required.
Regarding the current consumption an average of 67.9 mA was measured for V1. Both V0 and V2 draw a little less current with 61.4 mA and 65.8 mA respectively. To determine the energy that is consumed purely by the respective algorithms, however, the current that is consumed during sleep mode should be subtracted. During sleep mode the Cortex-M4 core is disabled and hence this current amounts to the energy that is consumed by the peripherals like SRAM, DMA controller and UART and was measured to be 35.4 mA. Therefore, the currents that are effectively drawn by V0, V1 and V2, when excluding the current of the other peripherals, are 26 mA, 32.5 mA and 30.4 mA respectively. With these currents and the respective execution times we see that V0 is slightly more energy efficient than the other two versions as shown in table 6.1. Since floating-point operations are considered to be more power-hungry than fixed-point operations [17], these results are unexpected.
However, the two fixed-point versions have the advantage that they can use 16-bit or even 8-bit data. Working with 16-bit instead of 32-bit data effectively reduces the memory bandwidth and hence the energy consumed by load and store operations while not having a significant effect on the prediction accuracy of CNNs. [17]
Additionally, V1 can exploit the parallel computing capabilities of the microcontroller's DSP unit. With the SMLAD instruction [26] two 16-bit MAC operations are computed in parallel. Therefore, the inference speed may be increased by about a factor 2x. However, single input multiple data (SIMD) instructions require a dedicated data alignment which can result in complex algorithms. Note that a speed-up by factor 2x would still not be enough to satisfy the real-time constraints of the scenario with microphone data. Therefore, yet another speed-up technique must be used. A possible approach is mentioned in section 8.

6.3 REDUCING THE INFERENCE MEMORY FOOTPRINT

6.3.1 *Initial Training*

Figure 6.4 shows the training history of the acoustic CNN. The final test accuracy is 86.6% which proves a successful training procedure.

6.3.2 *Quantization*

The quantization history of the acoustic CNN is shown in figure 6.5. The final test accuracy is 86.9%. Hence, the network was successfully

	DURATION	# MACS/S	CURRENT	ENERGY
V0	687 ms	28.38M	26 mA	58.9 mJ
V1	598 ms	32.61M	32.5 mA	64.1 mJ
V2	763 ms	25.56M	30.4 mA	76.5 mJ

Table 6.1: For V0, V1 and V2 the average duration and effective MAC-speed are shown in the first two columns. V1 is the fastest and V2 the slowest of the three code variants. The last two columns show the effective current and energy consumption to process the convolutional network section of the CIFAR-10 CNN for a single input image. The floating-point version V0 consumes slightly less energy than the two fixed-point versions.

quantized without loss in accuracy while reducing the memory footprint by factor 4x.

6.3.3 *Test on MCU*

6.3.3.1 *Network Accuracy*

The implementation on the MCU achieves a prediction accuracy of 87.2%. Note, however, that due to the long streaming process this test was performed on a smaller test set that consists of only 179 AEs. This explains the 0.3% increase compared to the 86.9% from the previous section.
The results, however, verify the MCU implementation and the idea of the buffer system that was introduced in section 4.3.

6.3.3.2 *Inference Memory Footprint*

Figure 6.6 shows the architecture of the buffer architecture that was used in our implementation. Notice that the total memory is 230.3 kB which is by factor 28x smaller than the 6.25 MB that would be needed to store the largest intermediate result in memory[1]. Therefore, this technique facilitates computing the complete forward inference on SRAM without using any off-chip memory.
The number of MAC operations that must be be computed per input fragment is about 12.26M. With 100 fragments per AE this results in an overal 1226M MAC operations which shows that the number of overall MAC operations has not been increased by the buffer system. Notice that the slightly higher value in table 3.2 is due to the fact that the intermediate number of MAC operations were rounded before computing their sum. For each input fragment the processing time is $4 \cdot t_{hop,fft}$ because the shift parameter $s_0 = 4$ in this CNN. Together with the 12.26M MAC operations per fragment this leads to the same

1 In 32-bit format.

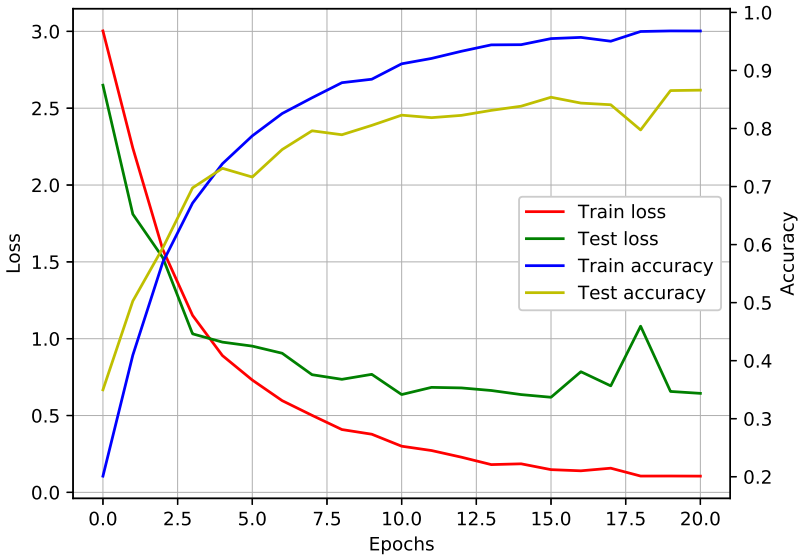

Figure 6.4: Training and test accuracy and loss versus training epochs of the acoustic target CNN.

real-time constraints as shown in 3.1.
Note that the buffer for network layer L0 has only a single green column instead of two which is a result of padding that is applied in the acoustic CNN.

6.3.3.3 *Inference Time*

Here we analyze the effect of the buffer system on the inference time and energy consumption. Specifically, we compare these two quantities with the corresponding results of the V0 implementation from table 6.1.
The processing time for a single (64x4) input fragment is measured as an average over two AEs. On average, the processing needs 521 ms. With 12.26M MAC operations per input fragment, the average MAC-speed is found to be 23.5M MACs/s. This is about 5M MACs/s slower compared to the V0 implementation with the CIFAR-10 CNN. This may be explained by the fact that, in our implementation, we copy the memory contents as illustrated in section 4.3. An improvement of the implementation may therefore include the use of circular buffers to reduce the possibly redundant memory accesses. However, the real-time constraint for the geophone data scenario is still satisfied.

In our acoustic CNN the data was zero padded in every layer. Considering that the last layer before softmax activation is a global avg-pooling the responsivity of the system may be increased from about 205 s to $4 \cdot t_{hop,ftt}$, that is 2.048 s for geophone data. This may be

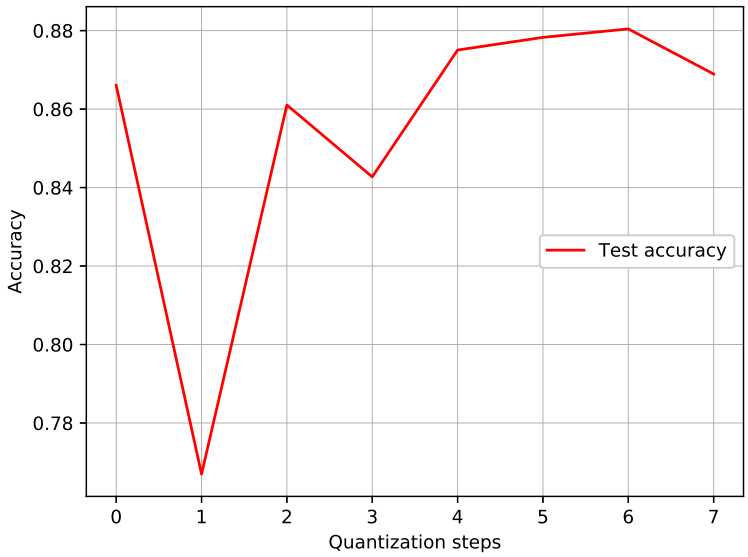

Figure 6.5: Test accuracy versus quantization steps for the acoustic target CNN. At x=0 the accuracy of the pre-trained network is shown. The subsequent values show the network accuracy after quantization and retraining in an alternating manner.

achieved by avoiding any zero padding within the network. This approach would guarantee that the contents of the "Average(F)" buffer in figure 6.6 would never be corrupted by zero-padding and, hence, the output of the "Average(T)" buffer could be computed in every single step.

6.3.3.4 *Energy Consumption*

The MCU current that is drawn at 3.3 V during forward inference is shown in figure 6.7. At them moment where an input fragment is processed the momentary average current of the MCU is at 59 mA. However, between two successive input fragments the MCU is set to sleep mode to save power. During sleep mode the MCU consumes no more than 35.4 mA when only the UART and DMA peripherals are enabled. The overall average current is thus only 41.5 mA.

6.3.3.5 *Comparison to the Current System*

To make a rough estimation whether it is worthwhile to adopt the acoustic CNN on the sensor nodes of the WSN the scenarios illustrated in figure 6.8 are considered.
A CentAUR data acquisition system is sampling a geophone signal during 24 hours.
In scenario (a) the sampled data is continuously streamed through the WSN by a core station. This is the current state of the WSN. With

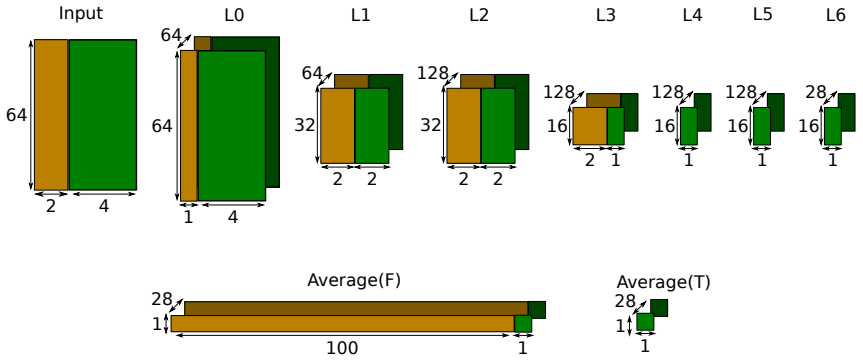

Figure 6.6: A detailed overview of the buffer system for the acoustic CNN is shown. These buffers enabled to perform the forward inference only using SRAM. Lo - L6 are the output buffers of the six conv layers of the acoustic CNN. The two buffers at the bottom are used to realize the global avg-pooling. The green elements need to be computed during forward inferene whereas the brown elements are used like a FIFO buffer as described in section 4.3.

an average current consumption of approximately 390 mA at 14.27 V the core station consumes about 480.8 kJ per day.
In the second scenario the data is processed in real-time on the MCU and per AE only a label that consists of say one byte is passed to the SX1211 transceiver. With an AE duration of 205.312 s (section 3.1) there are about 421 events within 24 hours. Thus, 421 labels are computed per day which may for example be sent with the energy efficient SX1211 transceiver using the the Dozer networking protocol [27]. With this protocol a one byte payload may be sent together with a seven byte header. At a bit rate of 100 kbit/s sending the 8 bytes would take 0.64 ms. However an additional startup and shut down time of 2 ms each should be included, that is, the SX1211 transceiver is active for 4.64 ms to send a single label. At an average current of 15.5 mA at 3.3 V sending 421 labels would thus cost about 100 mJ. However, the classification task must also be included in this estimation. With an average current of 41.5 mA at 3.3 V from the previous section, running the MCU costs about 11.8 kJ per day. Note that the energy consumed by the CenAUR station is the same for both scenarios and can thus be neglected.
Note that the above calculations are only showing the general trend as, for example, we have not considered the energy that is consumed to compute the Mel-spectrogram. Given that the energy consumption in scenario (a) is higher by about factor 40x, however, it may be that deploying the CNN on the sensor node may improve the energy-efficiency of the sensor nodes considerably.

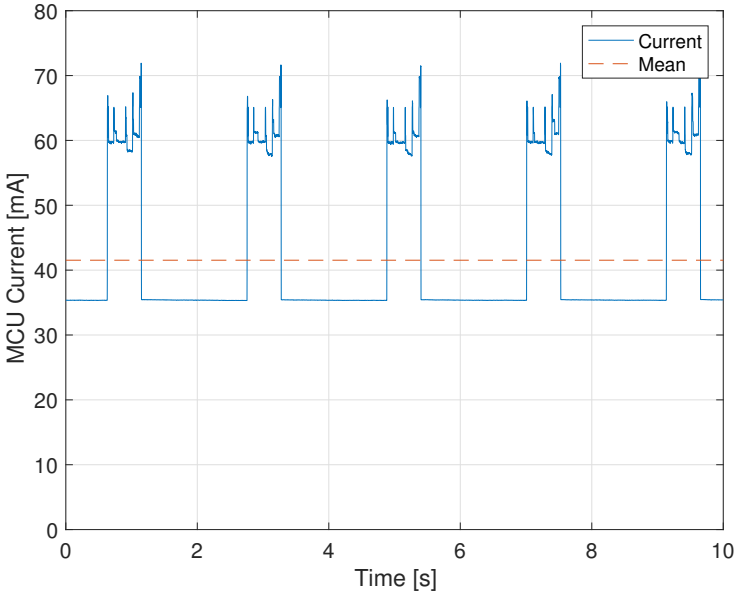

Figure 6.7: The momentary and average current consumption of the MCU is shown during forward inference of the acoustic CNN. During sleep mode the MCU consumes 35.4 mA. Note that during sleep mode only the UART and DMA peripherals are enabled.

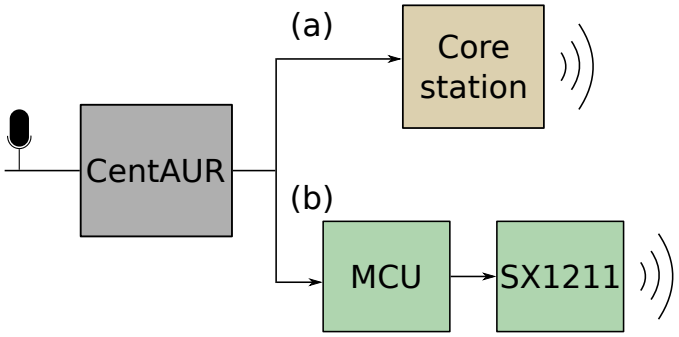

Figure 6.8: The scenario where raw data (a) and single labels (b) are sent through the WSN are compared.

Listing 6.1: Assembly code of the floating point baseline implementation V0. Only the code segment of the most inner loop is shown.

```
080013a4:   vldr                s14, [r3]
080013a8:   vldr                s15, [r0]
080013ac:   vldr                s7, [r3, #4]
080013b0:   vldr                s8, [r3, #8]
080013b4:   vldr                s9, [r1]
080013b8:   vldr                s12, [r2]
080013bc:   vldr                s10, [r1, #4]
080013c0:   vldr                s11, [r1, #8]
080013c4:   vldr                s13, [r2, #4]
080013c8:   vfma.f32            s15, s17, s14
080013cc:   adds                r3, #4
080013ce:   adds                r1, #4
080013d0:   adds                r2, #4
080013d2:   vfma.f32            s15, s16, s7
080013d6:   cmp                 r4, r3
080013d8:   vldr                s14, [r2, #4]
080013dc:   vfma.f32            s15, s0, s8
080013e0:   vfma.f32            s15, s1, s9
080013e4:   vfma.f32            s15, s2, s10
080013e8:   vfma.f32            s15, s3, s11
080013ec:   vfma.f32            s15, s4, s12
080013f0:   vfma.f32            s15, s5, s13
080013f4:   vfma.f32            s15, s6, s14
080013f8:   vstmia              r0!, {s15}
080013fc:   bne.n               0x80013a4
```

Listing 6.2: Assembly code of the fixed-point implementation V1. Only the code segment of the most inner loop is shown.

```
08001370:   ldr     r3, [r1, #0]
08001372:   ldr     r4, [r0, #0]
08001374:   ldr.w   r2, [r1, #4]!
08001378:   mla     r4, r3, r12, r4
0800137c:   ldr     r3, [r1, #4]
0800137e:   mla     r2, r2, r7, r4
08001382:   mla     r3, r3, r6, r2
08001386:   cmp     r5, r1
08001388:   str.w   r3, [r0], #4
0800138c:   bne.n   0x8001370
```

Listing 6.3: Assembly code of the fixed-point implementation V2. Only the code segment of the most inner loop is shown.

```
0800203a:   ldr.w   r0, [lr]
0800203e:   ldr.w   r6, [lr, #4]!
08002042:   asr.w   r11, r0, r3
08002046:   ldr.w   r0, [r8]
0800204a:   asr.w   r10, r6, r4
0800204e:   ldr.w   r6, [lr, #4]
08002052:   sub.w   r0, r0, r11
08002056:   add     r0, r10
08002058:   asrs    r6, r5
0800205a:   add     r0, r6
0800205c:   cmp     r9, lr
0800205e:   str.w   r0, [r8], #4
08002062:   bne.n   0x800203a
```

7 CONCLUSION

Using INQ we have successfully reduced the memory footprint of the acoustic CNN by a factor 4x without loss in classification accuracy.

The network was efficiently implemented on the STM32F469NI MCU. In order to achieve a maximum inference speed we reduced the number of clock cycles needed to execute the time consuming convolution. At a clock frequency of 168 MHz an effective MAC-speed of 32.61M MACs/s was achieved with a fixed-point implementation that uses a single-cycle MAC instruction. We have shown the importance to take into account the available number of CPU registers during algorithm design in order to avoid possibly redundant CPU instructions.

The necessary memory to store intermediate results was reduced by more than factor 27x using a pipelining-like data movement strategy. This enables to compute the forward inference using only the SRAM memory of the device and thus increases the energy-efficiency because no power-hungry off-chip memory is necessary. With this data flow an average power consumption of about 137 mW was achieved in a floating-point baseline implementation.

Although real-time processing of geophone data can be established with our implementation, a large speed-up is needed to do the same with microphone data. Suggestions on how to accomplish further accelerations are presented in section 8.

A first estimation indicates that performing the classification on the sensor nodes may help to reduce the energy consumption by a factor of about 40x.

8 FUTURE WORK

In this project we implemented the acoustic CNN on the STM32F469NI MCU. Although the implementation allows to perform AE classification in real-time for geophone data it is still too slow to process the audio signals from a microphone. This section presents some ideas how to speed-up the forward inference.

The current implementation of the acoustic CNN is only a baseline implementation that uses floating-point arithmetic. That is, the convolution algorithm is implemented using the code variant V0. In section 6.2.3.3 we have shown that the fixed-point version V1 is faster than V0. Therefore, a first way to decrease the inference time is to implement the convolution of the acoustic CNN according to V1.
Additionally, V1 can exploit the parallel computing capabilities of the microcontroller's DSP unit. With the SMLAD instruction [26] two 16-bit MAC operations can be computed in parallel, thereby only using a single clock cycle. Therefore, the inference speed may be increased by about a factor 2x. However, SIMD instructions require a dedicated data alignment which can result in very complex algorithms. Given that a speed-up by factor 2x would not be enough to satisfy the real-time constraints of microphone data, yet another speed-up is necessary.
A possible technique may be pruning where a large number of synapses are removed from the CNN by setting the corresponding weights to zero [11]. This essentially reduces the number of weights and thus also the number of MAC operations that need to be performed which, ultimately, results in a faster forward inference. In order to exploit the resulting zeros, however, dedicated hardware is necessary which is not present on general MCUs. Therefore, the idea of structured pruning [28] is particularly interesting. With this approach the weights may be pruned in such a way that for a given 9x9 kernel the zeros build a structure like a 3x1 row or column for example.
Using the implementation V1, which loads the individual rows separately, the benefit of this idea is that filter rows that contain only zero-elements can be detected and the computationally intensive convolution can be omitted. Instead, the algorithm can immediately load the next filter row.
Together with SIMD instructions this technique may decrease the time that is needed for classification such that even microphone data can be processed in real-time.

The above mentioned techniques mainly address the problem of a

too slow forward inference. Note however, that using SIMD instructions may also improve the energy-efficiency of the classification because only a single memory access is needed to load two 16-bit inputs while two memory accesses are necessary to load two 32-bit inputs.

BIBLIOGRAPHY

[1] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. "Wireless Sensor Network Survey." In: *Comput. Netw.* 52.12 (Aug. 2008), pp. 2292–2330. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2008.04.002. URL: http://dx.doi.org/10.1016/j.comnet.2008.04.002.

[2] Jan Beutel et al. "PermaDAQ: A Scientific Instrument for Precision Sensing and Data Recovery in Environmental Extremes." In: *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*. IPSN '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 265–276. ISBN: 978-1-4244-5108-1. URL: http://dl.acm.org/citation.cfm?id=1602165.1602190.

[3] Igor Talzi, Andreas Hasler, Stephan Gruber, and Christian Tschudin. "PermaSense: Investigating Permafrost with a WSN in the Swiss Alps." In: *Proceedings of the 4th Workshop on Embedded Networked Sensors*. EmNets '07. Cork, Ireland: ACM, 2007, pp. 8–12. ISBN: 978-1-59593-694-3. DOI: 10.1145/1278972.1278974. URL: http://doi.acm.org/10.1145/1278972.1278974.

[4] Artem Vasilyev. "CNN optimizations for embedded systems and FFT." In: (2015).

[5] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. "Hello Edge: Keyword Spotting on Microcontrollers." In: (Nov. 2017). eprint: 1711.07128. URL: https://arxiv.org/abs/1711.07128.

[6] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." In: (Oct. 2015). eprint: 1510.00149. URL: https://arxiv.org/abs/1510.00149.

[7] Bradley McDanel, Surat Teerapittayanon, and H.T. Kung. "Embedded Binarized Neural Networks." In: (Sept. 2017). eprint: 1709.02260. URL: https://arxiv.org/abs/1709.02260.

[8] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices." In: (July 2017). eprint: 1707.01083. URL: https://arxiv.org/abs/1707.01083.

[9] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." In: *Nature* 323 (Oct. 1986), 533 EP –. URL: http://dx.doi.org/10.1038/323533a0.

[10] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. "Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights." In: (Feb. 2017). eprint: 1702.03044. URL: https://arxiv.org/abs/1702.03044.

[11] Yann Le Cun, John S. Denker, and Sara A. Solla. "Optimal Brain Damage." In: *NIPS* 89 (1989).

[12] Matthias Meyer, Lukas Cavigelli, and Lothar Thiele. "Efficient Convolutional Neural Network For Audio Event Detection." In: (Sept. 2017). eprint: 1709.09888. URL: https://arxiv.org/abs/1709.09888.

[13] Naoya Takahashi, Michael Gygli, Beat Pfister, and Luc Van Gool. "Deep Convolutional Neural Networks and Data Augmentation for Acoustic Event Detection." In: (Apr. 2016). eprint: 1604.07160. URL: https://arxiv.org/abs/1604.07160.

[14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. "Striving for Simplicity: The All Convolutional Net." In: (Dec. 2014). eprint: 1412.6806. URL: https://arxiv.org/abs/1412.6806.

[15] *STM32F469 data sheet:* http://www.st.com/content/st_com/en/support/resources/resource-selector.html?querycriteria=productId=LN1876$$resourceCategory=technical_literature$$resourceType=datasheet.

[16] *STM32F469/479:* http://www.st.com/en/microcontrollers/stm32f469-479.html?querycriteria=productId=LN1876.

[17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." In: (Aug. 2017).

[18] Alejandro Valero, Salvador Petit, Julio Sahuquillo, Pedro López, and José Duato. "Design, Performance, and Energy Consumption of eDRAM/SRAM Macrocells for L1 Data Caches." In: *IEEE TRANSACTIONS ON COMPUTERS* 61.9 (2012).

[19] *Example CNN from keras:* https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py.

[20] *Mastering STM32:* https://leanpub.com/mastering-stm32.

[21] *CIFAR-10 dataset:* https://www.cs.toronto.edu/~kriz/cifar.html.

[22] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: (Sept. 2014). eprint: 1409.1556. URL: https://arxiv.org/abs/1409.1556.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database." In: (2009).

[24] Lukas Sigrist and Lothar Thiele. "Design Support for Energy Harvesting Driven IoT Devices." In: (2017).

[25] *Arm Cortex-M4 Processor, Technical Reference Manual:* http://infocenter.arm.com/help/topic/com.arm.doc.100166_0001_00_en/arm_cortexm4_processor_trm_100166_0001_0_en.pdf.

[26] *Programming Manual:* http://www.st.com/content/st_com/en/support/resources/resource-selector.html?querycriteria=productId=LN1876$$resourceCategory=technical_literature$$resourceType=programming_manual.

[27] Nicolas Burri, Pascal von Rickenbach, and Rogert Wattenhofer. "Dozer: Ultra-low Power Data Gathering in Sensor Networks." In: *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. IPSN '07. Cambridge, Massachusetts, USA: ACM, 2007, pp. 450–459. ISBN: 978-1-59593-638-7. DOI: 10.1145/1236360.1236417. URL: http://doi.acm.org/10.1145/1236360.1236417.

[28] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. "Learning Structured Sparsity in Deep Neural Networks." In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett. Curran Associates, Inc., 2016, pp. 2074–2082.

DECLARATION

Put your declaration here.

Zurich, March 2018

Timo Pascal Farei-Campagna