Master Thesis

Extension of an Embedded AI Framework for RISC-V Systems

Pascal Pfeiffer

Matr. Nr.: 5182994

Technische Hochschule Mittelhessen

Department

Informationstechnik, Elektrotechnik und Mechatronik (IEM)

Fraunhofer Institute for Photonic Microsystems

Department

Data Communication & Computing

June 2024

First supervisor: Prof. Dr.-Ing. Hartmut Weber

Second supervisor: Dr. Andreas Weder

Abstract

This master’s thesis presents the extension of an embedded AI framework for floating-point and quantized neural networks. The framework is extended with 1D convolutions, batch normalization, batch normalization folding, concatenation and global pooling while also lowering the memory requirements for quantized neural networks.

At the beginning of this thesis, the challenges of embedded AI are explained and the RISC-V IP core used is presented. The embedded AI framework Emmi is introduced, the fundamentals for understanding convolutions and batch normalization are presented, and a brief overview of its features and the quantization technique used is given.

Before implementing the extensions, Emmi is analyzed and the requirements are summarized. During the test-driven implementation, unit and integration tests ensure the functionality of the framework. After testing and benchmarking it on a non-vector RISC-V system, it is compiled with automatic vectorization and tested in a RISC-V simulator that supports vector instructions.

Finally, it is shown that the framework can be used in a predictive maintenance application.

Declaration of academic integrity

I hereby assure that the work presented is my own work. The work has been carried out independently and exclusively using the stated sources. Contents of this work, taken verbatim or analogously from the sources are made identifiable. The work has not yet been done before in this form and has not been submitted for examination or published.

X

Pascal Pfeiffer

Table of Contents

Table of Figures

Table of Tables

Table of Listings

List of Abbreviations

AI	**A**rtificial **I**ntelligence
AIfES	**A**rtificial **I**ntelligence **f**or **E**mbedded **S**ystems
CISC	**C**omplex **I**nstruction **S**et **C**omputer
DDR	**D**ouble **D**ata **R**ate
e5AIsuite	**E**MSA**5** **AI Suite**, today: Emmi
DTCM	**D**ata **T**ightly **C**oupled **M**emory
DYINQ	**Dy**namic **In**ference **Q**uantization
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FPU	**F**loating-**P**oint **U**nit
GCC	**G**nu **C** **C**ompiler
GPIO	**G**eneral **P**urpose **I**nput **O**utput
HAL	**H**ardware **A**bstraction **L**ayer
I2C	**I**nter-**I**ntegrated **C**ircuit
ISA	**I**nstruction **S**et **A**rchitecture
IP core	**I**ntellectual **P**roperty **C**ore
ITCM	**I**nstruction **T**ightly **C**oupled **M**emory
JAQ	Efficient Integer-Arithmetic-Only implemented after **Jac**ob et al.
LSTM	**L**ong **s**hort-**t**erm **m**emory
RAM	**R**andom **A**ccess **M**emory
RISC	**R**educed **I**nstruction **S**et **C**omputer
SPI	**S**erial **P**eripheral **I**nterface
TFLM	**T**ensor**F**low **L**ite **M**icro
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter

1 Introduction

In today's age artificial intelligence is not limited to the cloud but is seamlessly integrated into the fabric of our devices. Embedded AI holds the potential to revolutionize various fields, including smart cities, medical devices with advanced privacy requirements, autonomous vehicles, and the factory floor. It integrates intelligence into everyday devices, acting as a silent orchestrator behind the scenes. [1, 2, 3]

Embedded AI systems are essential in addressing the demand for smart and diverse low power, low cost and high efficiency applications. Embedded AI can be used to create intelligent monitoring systems that operate on sensor signals while fully processing the data on embedded systems. This sensor data may include information from a privacy critical application, a time-sensitive factory floor, or an agricultural device without dedicated power supply.

The core of an embedded device capable of running neural networks is its processor. Currently, this is often a device ISA compatible to RISC-V, which has gained popularity in recent years due to its flexibility, scalability and, simplicity. [4]

At the Fraunhofer IPMS in Dresden the RISC-V compatible IP core EMSA5 is developed. It provides support for running neural networks using TensorFlow Lite Micro (TFLM) [5], Artificial Intelligence for Embedded Systems (AIfES) [6] and Emmi (formerly known as EMSA5 AI Suite) [7], an embedded AI Framework developed at the Fraunhofer IPMS. Emmi features a vectorizable codebase, this means that it is designed in-order to be automatically vectorized by the compiler and executed using vector instructions. For embedded integer only RISC-V systems this is the ISA extension Zve32x. [8]

1.1 Scope of Work

This thesis will explain the fundamentals of embedded AI and is going to deal with the framework Emmi. During this thesis the functionality and code-base of Emmi will be extended, improved and refactored. In particular, support for 1D convolutions, 1D average and max pooling, global average and max pooling, and the concatenation of tensors required for some residual networks will be added. In addition, several ways to support neural networks using batch normalization will be presented and implemented in program code. All implemented features should take into account the compiler requirements for automatic vectorization of C code.

Besides the expansion of Emmi the thesis will evaluate Emmi in a simulator, capable of running vectorized RISC-V code. The challenges of using a simulator will also be presented.

1.1.1 Goal and Research Questions

One goal of this master thesis is to extend the functionality of the inference-only embedded AI framework Emmi. It is currently limited to a small number of layers, which will be extended and improved.

Namely the following layers will be implemented:

- 1D convolution
- 1D average pooling
- 1D max pooling
- Global 1D & 2D poolings
- Concatenation
- Batch normalization

In addition, this work should answer the following questions:

Q1. How to implement batch normalization for Emmi and should the folding of batch normalization parameters be considered, and if so, how can it be implemented?

Q2. How to reduce the memory footprint of quantized Emmi models?

Q3. How to improve the accuracy of quantized networks?

Q4. How to use a RISC-V simulator to run Emmi when compiling with automatic vectorization?

Besides the questions above, the following secondary points can also be addressed:

S1. Embedded AI Application Example
An Example covering the process from training to a real-world embedded AI application

S2. How to implement a depth-wise convolution?

S3. How integrate a Strassen-based matrix multiplication, replacing the current row-column approach?

1.2 Contents of following chapters

After the introduction, the second section of this thesis gives an overview of the technical fundamentals related to the embedded AI framework Emmi. It starts with an introduction to RISC-V, which is the main platform for Emmi, introduces the IP core EMSA5, and gives an overview of the fundamentals of embedded AI and the Emmi framework. It also presents the theory of the layers and operations implemented in this work, as well as the quantization technique DYINQ. The third section analyzes the current state of Emmi and discusses the design decisions made to extend and improve the framework. This is followed by section four, which briefly presents the implementation of the layers and the integration of batch normalization folding. Section five evaluates the framework's extensions in terms of accuracy and performance. It also covers the use of a RISC-V simulator. Section six presents an embedded AI application and is followed by section seven, which concludes the thesis.

2 Fundamentals

The chapter provides a comprehensive overview of the fundamental concepts and technologies used in this work. It begins by introducing the RISC-V instruction set architecture and the IP Core EMSA5, for which the embedded AI Framework Emmi was developed initially. It gives a brief overview of Embedded AI, its workflow and all operations implemented during this thesis. Concluded is the chapter with and introduction of the used quantization technique DYNQ.

2.1 RISC-V

The Instruction Set Architecture (ISA) RISC-V, pronounced "risk five," was introduced by UC Berkeley in 2010. Its manual is licensed under the Creative Commons Attribution 4.0 International License. In literature RISC-V is described as the world's first open-source ISA with widespread commercial backing [9]. Currently, RISC-V International, a non-profit organization comprising members such as SiFive, Intel, and Fraunhofer, oversees its development. [10]

The name RISC-V hints a Reduced Instruction Set Computer (RISC) architecture. It contrasts with Complex Instruction Set Computers (CISC) like x86, which employ varying instruction lengths and aim to cover every special case with a dedicated instruction. Despite this foundational simplicity, RISC-V processors can incorporate more than the basic 47 instructions through ISA extensions. These extensions introduce additional, standardized sets of instructions that hardware designers can implement to tailor specific functionalities or enhance performance. RISC-V International has already standardized various extensions, including the "F" extension, which introduces single-precision floating-point capabilities. There are numerous other extensions either in development or already standardized. [11]

Vendor-specific extensions are also allowed, fostering a modular system that accommodates specialized cores and enables designers to fine-tune their cores based on performance, power, and size considerations.

The RISC-V Vector Extension (RISC-V V) extends the instruction set to approximately 300 instructions. It defines two subsets for embedded processors: Zve32x for 32-bit integer operations and Zve32f for 32-bit floating-point computations. The extension Zve32x is partially supported by the IP core EMSA5, which is introduced in the following section. [12]

2.2 EMSA5

The EMSA5 is a 32-bit RISC-V IP core developed in-house at the Fraunhofer IPMS located in Dresden. This core offers support for a range of RISC-V instruction sets, including I, M, C, Zicsr, and Zifencei. Furthermore, it partially accommodates the embedded vector instruction set, Zve32x.

The EMSA5 IP core comes in various configurations, offering different peripheral and memory options. The version utilized in this thesis incorporates a set associative 64KB/4-way cache, particularly in response to matrix multiplication benchmarks utilizing vector instructions. [7]

Moreover, the version employed in this thesis includes:

- 256 KB ITCM dedicated for program code
- 4 KB DTCM dedicated for data
- 4 KB SRAM usable for data and instructions
- 256 MB DDRRAM for data and instructions

- Two 32-bit timers, I2C, SPI, UART, a watchdog, and eight GPIO pins.

As evaluation platform the FPGA development board Arty A7-100T is used.

2.3 Embedded AI

Embedded AI is the name for a field of technologies dealing with artificial intelligence on embedded systems. The TinyML foundation describes embedded AI as “hardware, algorithms and software capable of performing on-device sensor data analytics at extremely low power” [13]. When developing an embedded AI application, a specific workflow (described in section 2.3.1) is required. It includes the development and optimization of neural networks for specific hardware using techniques such as approximation, batch normalization folding (section 2.5.5) and quantization (section 2.6). [14]

Currently embedded AI frameworks are primarily focused on inferences (a known exception is AIfES [6]) and deployment to embedded devices rather than the creation and training of neural networks. These frameworks provide tools to convert and run trained models on the destination platform. Due to heavily limited resources, compromises may be necessary when performing inferences on embedded devices. For instance, some devices lack a floating-point unit, which necessitates converting models to integer or fixed-point numbers, potentially resulting in reduced accuracy but a smaller size and an improved inference time.
[35, 7]

Table 1 gives an overview of embedded AI Frameworks, many of them are vendor specific and bundled with a manufacturers hardware, most of them are proprietary.

Table 1: Overview of embedded AI Frameworks

Framework	**Vendor**	**Officially Supported Platforms**	**Vendor Specific**	**Proprietary**	**Ref.**
AIfES	Fraunhofer IMS	ARM, Atmel, RISC-V	No	No	[6]
Cube.AI	STMicroelectronics	ARM	Yes	Yes	[15]
Emmi	Fraunhofer IPMS	RISC-V	Yes	Yes	[16]
DRP-AI	Renesas	ARM / DRP-AI Accelerator	Yes	Yes	[17]
KRAI	KRAI	ARM	No	Yes	[18]
Plumerai	Plumberai	ARM	No	Yes	[19]
TFLM	Google	ARM, RISC-V, Xtensa	No	No	[5]

2.3.1 Embedded AI Workflow

The workflow when developing an embedded AI application can be divided into three major steps:

1. Use case definition and requirements analysis
2. Model development
3. Embedded system deployment

All steps may vary depending on the used framework. The here described steps can be applied to TFLM, AIfES and Emmi.

Step 1: Use Case Definition and Requirements Analysis

A typical embedded AI workflow begins with defining the application's use case and requirements. Next, available resources are gathered, and the hardware platform that best suits the desired use case and requirements in terms of cost, performance and power requirements is selected. This platform may consist of a microcontroller or FPGA with storage and connectivity options. Additionally, a preselection of possible embedded AI frameworks is made by considering their features and supported platforms.

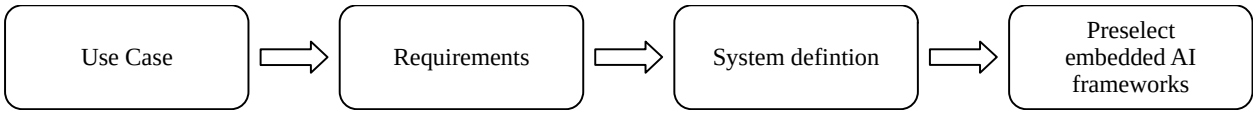

Figure 1: First steps when developing an embedded AI application

Step 2: Model Development

To build and train the neural network, sensor data is collected from the selected real-world scenario and preprocessed. If not enough data is available, data augmentation, or a simulation of the scenario should be considered. Once enough data has been gathered a model is built, trained, and tested using desktop machine learning frameworks such as TensorFlow or PyTorch. The future embedded AI use case is always considered while building the model, this means that the number of parameters is reduced as much as possible, and operations and layers used are supported in at least one of the preselected embedded AI frameworks.

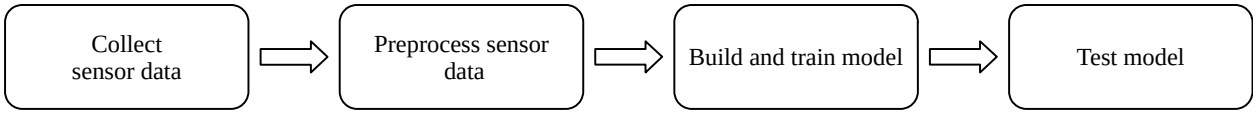

Figure 2: Data preprocessing and model creation

Step 3: Embedded System Deployment

Following the development and training of the model, the embedded AI framework is selected. Criteria include the usability, support of the used network architecture, memory requirements, and the support for the used platform. The converter tool of the selected embedded AI framework is started, and the conversion options, which include a range of optimizations from quantization to the selection of (faster) approximated activation functions, are set. Once the model is converted, it is integrated into the embedded application. If required, the pipeline for preprocessing and using the recorded data is implemented. Afterwards the model is ready to be tested on the embedded system. If the generated model is too big or too slow, the process is restarted from step “Build and train model”.

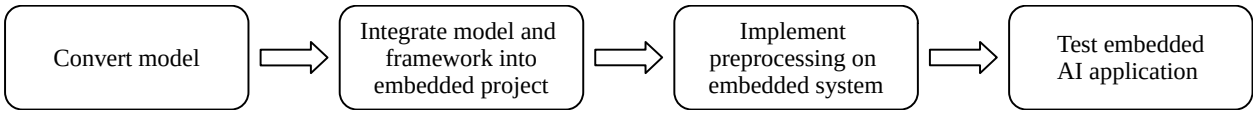

Figure 3: build embedded AI application

2.4 Emmi

Emmi is an inference-only AI framework designed to run neural networks on embedded systems. It offers a layer-based interface for neural networks and is completely written in C. It supports floating-point and integer operations with quantized weights, biases and feature-maps. Models for Emmi are converted from TensorFlow Keras, while sequential and functional models are supported.

2.4.1 Components

Emmi compromises three different components as shown in Figure 4. The first component is the model converter, which takes trained TensorFlow Keras models and converts them into C code invoking the functionalities presented by the Emmi framework. The converter runs checks on the model, such as for unsupported operations or model parameters. The model undergoes optimization and is exported as C code, which consists of four files: two C implementation files (one containing the weights and one containing the model) and two headers. When using the converted model, only a single header needs to be included.

The second component of the framework is known as the Emmi Core and represents the interface to the AI functionalities used by the generated models. It implements all supported activation functions (including approximations) and layers, such as dense, convolution or pooling. The Emmi Core is included by the C files of the converted model.

The implementations of the layers and activation functions utilize the methods of the numerical library, which represents the third component, namely vmath. It implements all mathematical operations for floating-point and the quantized data types. Like the layers implemented in the Emmi Core, the functionality implemented in vmath can be automatically vectorized by the RISC-V compiler.

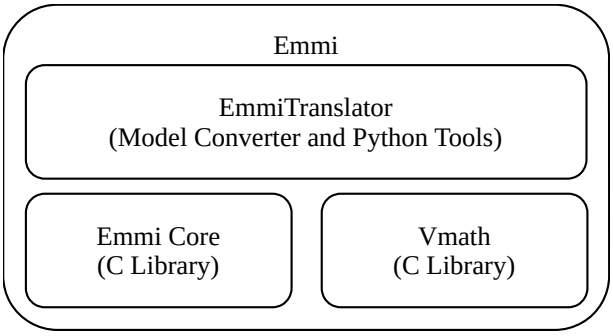

Figure 4: Emmi Components

2.4.2 Features

Emmi supports various layers, activation functions and model architectures. One of its outstanding features is, that each layer implemented in Emmi is optimized to be automatically vectorized by the compiler. This results in a faster execution time, while still being compatible to a wide range of platforms thanks to its C codebase. Emmi is tested with convolutional and residual networks and supports quantized and floating-point models. When converting a model using the EmmiTranslator, optimizations such as the removal of dropout layers from the inference function of the model are supported.

A complete overview of all supported layers, if the quantization is supported for those layers, and if the implemented quantization is a true integer quantization is given in Table 2. All listed layers are

implemented for floating-point. Layers marked with a capital ‘X’ are implemented during the course of this thesis.

Table 2: Layers supported by Emmi

	Layer	**Quantization Support**	**True Integer Quantization**
	Dense	✓	✓
X	Convolution 1D	✓	✓
	Convolution 2D	✓	✓
X	Depth-wise Convolution 1D	✓	✓
X	Depth-wise Convolution 2D	✓	✓
X	Max Pooling 1D	✓	✓
	Max Pooling 2D	✓	✓
X	Average Pooling 1D	✓	✓
	Average Pooling 2D	✓	✓
X	Global Max Pooling 1D	✓	✓
X	Global Max Pooling 2D	✓	✓
X	Global Average Pooling 1D	✓	✓
X	Global Average Pooling 2D	✓	✓
	Flatten	✓	
	Add	✓	✓
X	Concatenation	✓	
X	Batch Normalization	✓	

2.5 Machine Learning Operations

In machine learning models are represented as graphs, where the nodes represent the layers and the edges the dataflow. Layers are used to build the model and may contain a single or multiple operations in the form of a subgraph. [20]

This subsection introduces all operations added, and implemented to Emmi during this thesis. Already implemented operations are not covered. Detailed explanations for already implemented operations can be found in “Development of a Machine Learning Framework for Quantized Neural Networks on Embedded RISC-V Systems” [7].

2.5.1 Convolutions

Convolution 1D

Discrete convolutions enable the extraction of features from input data. They are local operations, meaning that they only consider selected groups of elements instead of the entire sequence at once. Each feature in the output sequence is computed based on the corresponding element in the original sequence and its neighboring elements. This approach determines the value of the resulting feature based on its

local context. The resulting sequence is formed progressively by moving the filter across the original sequence. [21, 22, 23]

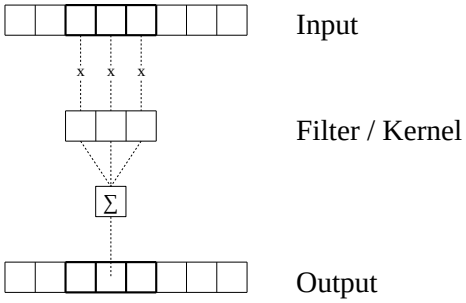

Figure 5: Convolution for 1D inputs

A convolution involves a kernel (also called filter), which is slided across the input features, performing element-wise multiplication, and summing the results to produce a feature-map, which represents the output vector of the convolutional operation.

$$S(i) = I * K = \sum_{n}^{N} I\left(i + n - \left\lfloor \frac{N}{2} \right\rfloor\right) \cdot K(n) \tag{1}$$

Where I is the feature-map of the input, K is the kernel and $S(i)$ the output vector at position i. Variable n is the index of the kernel, N is the number of elements in the kernel.

Within a neural network, the weights of the kernel are determined during the training. In a convolutional layer within a neural network, an additional bias can be incorporated.

$$S(i) = I * K + b = b + \sum_{n}^{N} I\left(i + n - \left\lfloor \frac{N}{2} \right\rfloor\right) \cdot K(n) \tag{2}$$

Where b is the bias of the kernel.

Especially in an embedded system the required number of multiplications is of interest, since these operations often consume more than a single clock cycle. The number of multiplications per output feature depends on the number of channels and the number of elements per kernel. In general, larger kernels require more multiplications.

$$Number\ of\ Muliplications\ per\ Output\ Feature = N * c \tag{3}$$

Where N is the number of Kernel elements per channel and c the number of channels.

Depth-wise Convolution 1D

Standard convolutional operations use the same kernel for all input channels, while the depth-wise convolution involves a separate kernel for each channel to preserve the separation of channels. This approach allows for more precise feature extraction while reducing the required number of multiplications, as can be concluded from formula (3) since channel c is always one when performing a depth-wise convolution.

Figure 6 compares the standard convolution with a depth-wise convolution using a sequential input with five elements and three channels. The kernel has a size of three elements. Notice that the kernel used for the standard convolution is as deep as the number of channels, wheres the kernel of the depth-wise convolution is split into three kernels: one for each channel. Also the output of the depth-wise convolution

differs from the standard convolution: It outputs a feature-map for each channel, merged to a single tensor. [24]

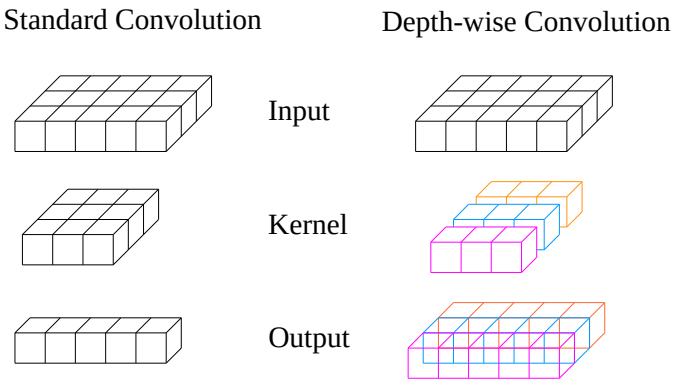

Figure 6: Comparison of standard and depth-wise convolution on a three channel input with five elements

Equation (4) shows how an element in the depth-wise convolution is calculated.

$$S(i,c)=I*K^{(c)}=\sum_{n}^{N} I\left(i+n-\left\lfloor\frac{N}{2}\right\rfloor,c\right)\cdot K^{(c)}(n) \tag{4}$$

Where $S(i,c)$ is the output tensor at position i in channel c and $K^{(c)}(n)$ is the kernel of the channel c at position n.

The depth-wise convolution can also be applied to two-dimensional inputs:

$$S(i,j,c)=I*K^{(c)}=\sum_{m}^{M}\sum_{n}^{N} I\left(i+m-\left\lfloor\frac{M}{2}\right\rfloor,j+n-\left\lfloor\frac{N}{2}\right\rfloor,c\right)\cdot K^{(c)}(m,n) \tag{5}$$

Where $S(i,j,c)$ is the output feature-map at position (i,j) in channel c. Variables m and n index the current position of kernel $K^{(c)}$, M and N are the elements per kernel dimension.

When using a bias, the equation (4) used for a one dimensional depth-wise convolution evolves to:

$$S(i,c)=I*K^{(c)}+b^{(c)}=b^{(c)}+\sum_{n}^{N} I\left(i+n-\left\lfloor\frac{N}{2}\right\rfloor,c\right)\cdot K^{(c)}(n) \tag{6}$$

And the equation for a two dimensional depth-wise convolutions (5) evolves to:

$$S(i,j,c)=I*K^{(c)}+b^{(c)}=b^{(c)}+\sum_{m}^{M}\sum_{n}^{N} I\left(i+m-\left\lfloor\frac{M}{2}\right\rfloor,j+n-\left\lfloor\frac{N}{2}\right\rfloor,c\right)\cdot K^{(c)}(m,n) \tag{7}$$

Notice that a bias for each channel is used.

2.5.2 Pooling Operations

Pooling is a local operation used to down-sample the input. Pooling involves applying a sliding window across the input, where a specific pooling function (such as max or average pooling) is applied to the values within each window, resulting in a single output value for each window. This technique not only helps in reducing the size of the input data but also enhances the model's ability to recognize features

regardless of their position within the input, thereby improving the model's robustness and generalization capabilities. [21, 22]

Average Pooling 1D

Average pooling calculates the average value of all elements within the pooling window. After calculating the average, the window is moved. Figure 7 shows a one dimensional average pooling with a window size of four and a stride of four, meaning, that the window is moved four elements at a time. [25]

Max Pooling 1D

Max pooling writes the highest value within the pooling window into the output element. After determining the highest value within the pooling window, the window is moved. Figure 7 shows max pooling with a window size of four and a stride of four. [26]

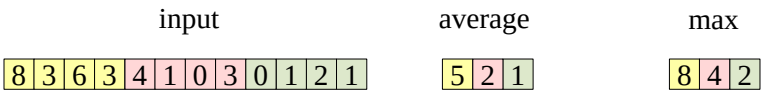

Figure 7: 1D average and max pooling

Global Average Pooling 1D

Global pooling operations operate on the input of a channel instead of using a dedicated pooling window. In global average pooling 1D, the average of the complete input series is calculated for each input channel. Figure 8 illustrates various global pooling operations. [27]

Global Max Pooling 1D

Global max pooling 1D determines the maximum value for each channel of a sequential input. It is illustrated in Figure 8. [28]

Global Average Pooling 2D

Global average pooling 2D is the two dimensional equivalent of global average pooling 1D, used for input data with multiple channels. It calculates the average for each channel of the two dimensional input. [29]

Global Max Pooling 2D

Global max pooling 2D finds the highest value for each channel in a two dimensional input, such as an image with multiple color channels. [30]

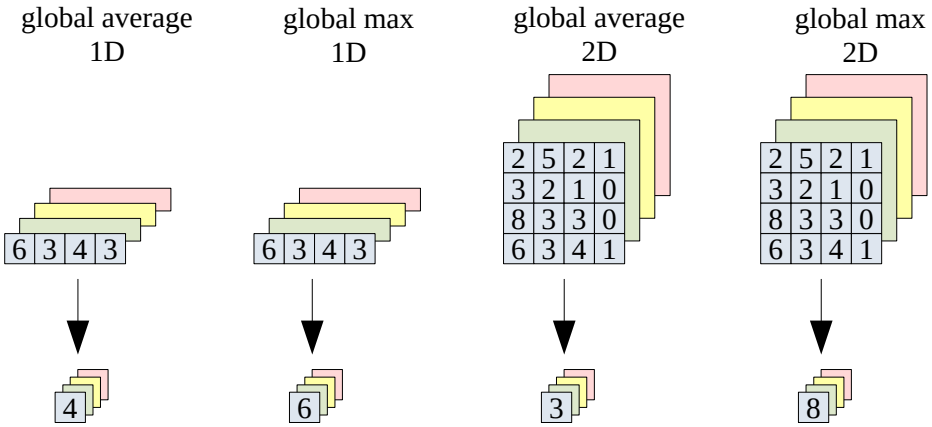

Figure 8: Global 1D and 2D poolings

2.5.3 Padding and Dilation

Padding for 1D Data

When processing sequences with a sliding window, the edges of the sequence pose a challenge due to the lack of neighboring elements for the window to operate on. To mitigate this, two padding strategies are commonly employed: valid padding and same padding.

Valid padding excludes the border elements from the operation, leading to a reduced output size. Same padding, in contrast, expands the input sequence by appending zeros to its edges. This method ensures that the output retains the same dimensions as the input. [31]

Figure 9 illustrates the application of same and valid padding with a three-element window. By adding zeros to the input sequence's edges as in the same padding case, the window's center (marked in red) aligns with the first element of the input.

Figure 9: Valid and same padding

Dilation for 1D Data

Dilation enlarges a kernel used in convolutions by inserting zeros between its elements. The right, one dimensional kernel in Figure 10 shows a dilation of two. This means a step-width of two is required to move from one of the kernels original elements to another. [23]

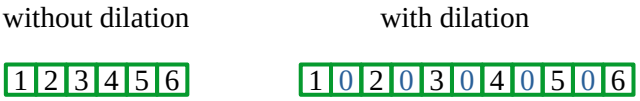

Figure 10: Dilation of 1D feature-maps

2.5.4 Concatenate

The concatenate operation merges n tensors into a single output tensor, provided they have the same shape except for the concatenation axis. Figure 11 illustrates an example of concatenation. In this example, three input tensors are concatenated on the second axis: the first tensor has 3×6 elements, the second has 3×7 elements, and the third has 3×2 elements. The resulting tensor has 3×15 elements. Concatenation is possible because all three tensors have the same number of elements on the first axis. [32]

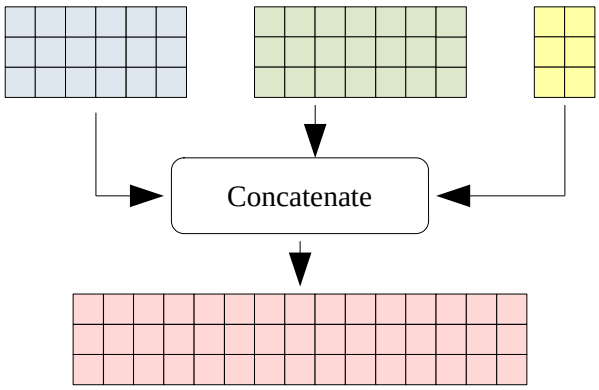

Figure 11: Concatenation illustration

2.5.5 Batch Normalization

Batch normalization is a technique used in neural networks to normalize selected feature-maps. Commonly batch normalization is used because it increases the training speed and reduces the chances for exploding gradients. It is inserted into the network as layer and uses the moving mean μ, the moving variance σ, the offset factor β, the scaling factor γ and the constant ϵ as parameters. [21]

The moving mean μ is the mean over time. It is initialized with zero.

$$\mu = \mu \cdot m + mean(batch) \cdot (1 - m) \tag{8}$$

Where $mean(batch)$ calculates the mean of the current input batch and the momentum m characterizes the resistance of the moving variance to change. A typical value for m is 0.99.

The moving variance σ is the variance over time. It is initialized with one.

$$\sigma = \sigma \cdot m + var(batch) \cdot (1 - m) \tag{9}$$

Where $var(batch)$ calculates the variance of the current input batch. The momentum m is the same value as used to calculate the moving mean.

Both, the moving mean and the moving variance are updated each time the layer is called during training.

The parameters β and γ are updated during the training using the gradient descent algorithm. The offset factor β is initialized with 0, γ with 1. ϵ is used as small configurable constant and is added for numerical stability. TensorFlow uses an ϵ of 0.001. [33, 34]

At inference time the batch normalization is calculated as shown in (10).

$$bn(batch) = \gamma \cdot \frac{batch - \mu}{\sqrt{\sigma + \epsilon}} + \beta \tag{10}$$

Example

To demonstrate the batch normalization an example is given. Assume a vector (which represents a single sample of a batch) with 30 random elements in the range [-1.5, 2.5]. The histogram in Figure 12 shows its distribution.

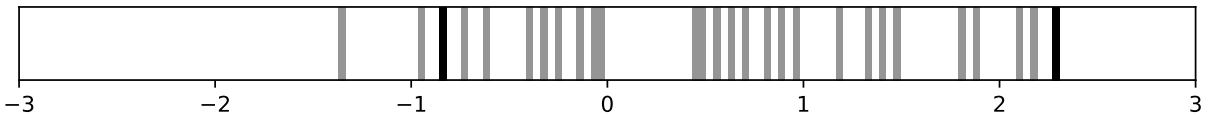

Figure 12: Histogram of a vector with floating-point elements

A batch normalization is now applied to the vector using different parameters. At first with $\beta = 0, \gamma = 1$ and $\epsilon = 0$.

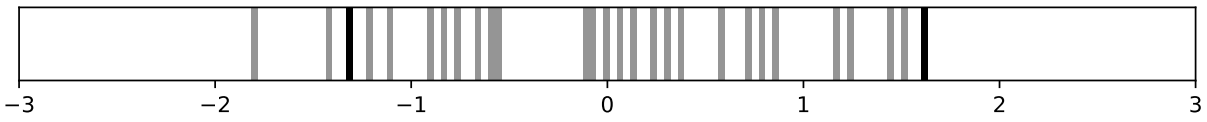

Figure 13: Batch normalization applied with β=0 and γ=1

In Figure 13 it is visible that the values are now distributed somewhere around 0 instead of 0.5. In Figure 14 the use of the scaling parameter γ is demonstrated by setting it to $\gamma = 0.58$.

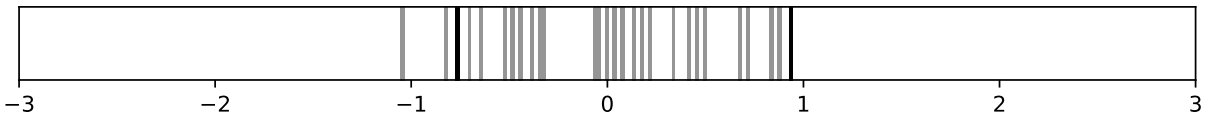

Figure 14: Batch normalization applied with β=0 and γ=0.58

The values are now scaled into a much smaller range. To offset the features for an alignment in between [-1,1], $\beta = 0.05$ is used.

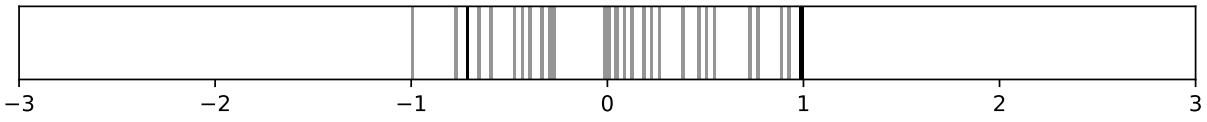

Figure 15: Batch normalization applied with β=0.05 and γ=0.58

By using the user parameter $\epsilon = 0.001$ the numerical stability of the batch normalization is improved. Since the variance is always positive, the value epsilon can be used to avoid a division by zero. The output of the batch normalization when using $\gamma = 0.58, \beta = 0.05$ and $\epsilon = 0.001$ is shown below (it should not be distinguishable from Figure 15 by a human eye).

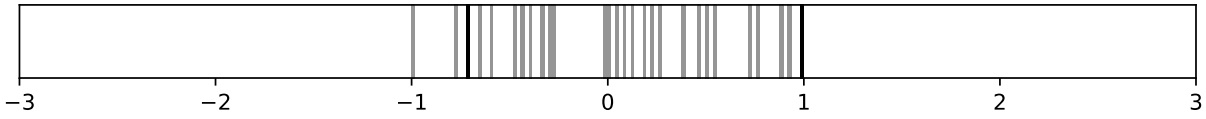

Figure 16: Batch normalization applied with β=0, ɣ=0.58 and ϵ=0.001

The Python script demonstrating batch normalization on a vector can be found in the appendix under *demos/batchnorm-demos.py*.

Folding

The folding of a batch normalization layer L consists of the removing of the batch normalization layer from the networks graph and in updating the models parameters to keep its predictive function unchanged. It is performed to improve the networks performance at inference time. The folding is executed after training the model.

Benoit Jacob et al. propose in their 2017 paper "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference" [35] an approach for folding a batch normalization layer into a neighboring expressive layer in backwards direction (see figure 17). These can be convolutional or dense layers. The batch normalization layer itself is completely removed. The approach can be denoted as shown in equation (11).

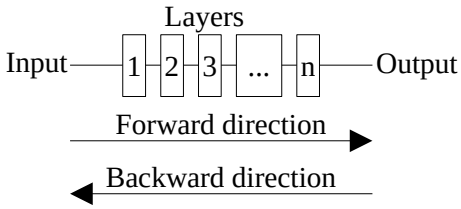

Figure 17: Directions in a sequential model

$$W_{foldB} = \gamma \frac{W}{\sigma + \epsilon} \tag{11}$$

Where W_{foldB} are the weights with folded batch normalization parameters, W the tensor of original weights, γ the scaling factor, σ the moving variance and ϵ for numerical stability.

With the BaN-OFF algorithm an improved folding was presented by Edouard Yvinec et al. in "To Fold or Not to Fold: a Necessary and Sufficient Condition on Batch-Normalization Layers Folding" [36]. It defines the weights folding for biases in backwards direction and adds definitions to fold weights and biases in forward direction. It proposes the folding of batch normalization layers even when no direct connection to an expressive layer exists.

The backwards folding of biases is done by:

$$b_{foldB} = \gamma \frac{b - \mu}{\sigma + \epsilon} + \beta \tag{12}$$

Where b_{foldB} are the backwards folded biases, b the original biases, μ is the moving mean and β the offset.

The forward folding of weighs and biases is described in (13) and (14).

$$W_{foldF} = W_{foldB} = \gamma \frac{W}{\sigma + \epsilon} \tag{13}$$

$$b_{foldF} = \gamma \frac{\mu}{\sigma + \epsilon} + \beta \cdot W + b \tag{14}$$

Note that the forward and backward folding of non-biased weights uses the same equation. To enable the folding even when no direct connection to an expressive layer exists, the BaN-OFF algorithm gathers all neighboring expressive layers in two sets: one for inputs and one for outputs of L, each starting with the batch normalization layer L itself. If a neighboring non-expressive layer is found, all neighboring expressive layers of this layer are added. This process is repeated until no new neighboring non-expressive layers are found. The batch normalization layer L is foldable, if at least one set:

1. Of gathered layers is not limited to L
2. All the leaves of the set are expressive layers [36]

If the set of input layers satisfies the second requirement, the gathered layers on the input side of L are updated using (11) and (12). The other ones negate the previous update using:

$$W = \frac{W}{\gamma}(\sigma + \epsilon) \tag{15}$$

$$b = b + \gamma \frac{W \mu}{\sigma + \epsilon} - \beta \cdot W \tag{16}$$

If not, the gathered layers on the input side of L are updated using (13) and (14) while other ones negate the previous update using:

$$W = \frac{W}{\gamma}(\sigma + \epsilon) \tag{17}$$

$$b = \frac{b}{\gamma}(\sigma + \epsilon) + \gamma \frac{\mu}{\sigma + \epsilon} - \beta \tag{18}$$

To fold batch normalization layers as described above, the Python package “tensorflow-batchnorm-folding” can be used. It is developed by the main author of the paper [36]. Since it is limited to folding dense and convolutional 2D layers, and requires the no longer supported TensorFlow version 2.9, it must be updated to be usable with the EmmiTranslator, which only supports TensorFlow version 2.11 or higher. [37]

2.6 Quantized Neural Networks

2.6.1 Introducing Quantized Neural Networks

Neural networks commonly use floating-point values to represent inputs, weights, biases, feature-maps, and outputs. However, when deploying on integer-only hardware, these values can be handled by either using soft-float on the destination device or by converting the model to a quantized integer model. In addition to being beneficial for integer-only hardware, quantization can also improve the performance on floating-point hardware by reducing storage requirements and increasing the execution speed of the neural network.

Quantization is the process of converting continuous values into a discrete representation. It involves converting arrays of floating-point numbers into arrays of integer values. In the context of neural

networks, this process is applied to all inputs, weights, biases, and feature-maps. A size reduction of up to four times can be archived when converting a 32-bit floating-point network to an 8-bit integer network.

Quantization techniques for neural networks can be categorized: post training quantization techniques or techniques using quantization aware training, static or dynamic quantization and true or fake integer quantizations.

True Integer Quantization

True integer quantized neural networks perform all operations using integer values, meaning that floating-point values are not used. To prevent overflows, techniques such as rescale operations must be employed since the multiplication of two integer values of bit-widths m and n results in an integer value of bit-width $m \cdot n$. These rescale operations use a scaling factor to reduce the bit-width of the feature-maps. Figure 18 illustrates a dense layer that employs true integer quantization and rescaling. [38]

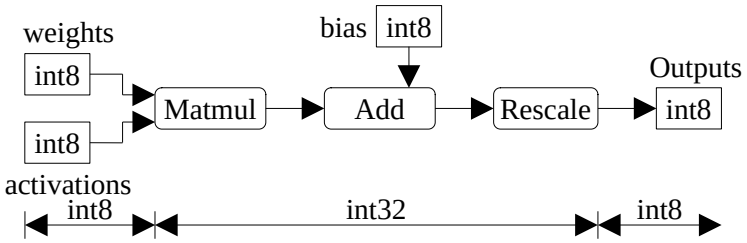

Figure 18: Inference using true integer quantization

Fake Integer Quantization

Fake quantized neural networks use floating-point arithmetic for some or all of their operations. Fake quantization involves storing weights as integers and converting them to floating-point values during inference, which can significantly slow down the process. The primary purpose of fake quantization is to reduce the storage requirements of neural networks. Figure 19 illustrates a dense layers inference using fake quantization with linear activation. [38]

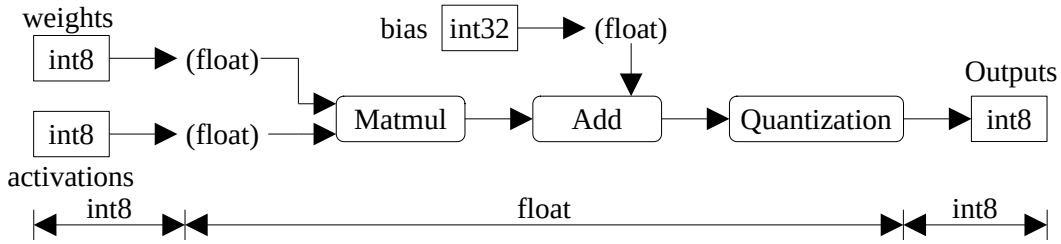

Figure 19: Inference using fake quantization

Dynamic Quantization

When utilizing dynamic quantization, only the quantization parameters for weights and biases are established during model conversion. The quantization parameters for the feature-maps are established at runtime. The rescaling process dynamically determines the scaling factor used to rescale the feature-maps. [38]

Static Quantization

In statically quantized neural networks, all quantization parameters are fixed and already known after the model has been converted. This applies to weights and feature-maps. They are determined by analyzing a representative dataset during the conversion phase, which is crucial for determining the quantization parameters for the network's feature-maps. This approach eliminates the need for dynamic rescaling, as a static rescaling factor is applied after each layer, improving the performance of the rescaling process applied during inferences. [38]

Post Training Quantization

Post training quantization is applied to a neural network after it has been trained. This converts its weights and biases from floating-point to quantized lower precision representations. The process involves quantizing the model parameters after training, which may result in a slight decrease in accuracy due to the quantization errors not being accounted for during training. [38]

Quantization Aware Training

Quantization aware training involves applying quantization during the model's training process, rather than only quantizing afterwards, as it is the case with post training quantization. During quantized aware training, the quantization of the model is simulated, often by quantizing and dequantizing the model after each epoch. This adds the quantization error to the models weights, which enables the model to fit its weights to the used quantization. Quantization aware training is considered to be beneficial to the accuracy of a quantized neural network. [39]

2.6.2 DYINQ

The Dynamic Inference Quantization is the post training quantization technique implemented in Emmi. It quantizes a neural network when converting a TensorFlow model to Emmi using the EmmiTranslator. DYINQ is a dynamic post training quantization, with support for true integer quantization. [7]

Quantizing Floats

DYINQ is designed to quantize vectors of floating-point values, using the same scale and zero offset. Single elements of float are quantized by multiplying them with a scale and then adding a zero offset that is used to align the quantized values in the range of a signed integer type.

$$q_i = round(y_i s + z) \tag{19}$$

Where y_i is an element of the floating-point vector y and q_i the resulting quantized value.

The scale s is calculated by dividing the quantization steps through the range of floating-point values.

$$s = round\left(\frac{Q_{steps}}{max(y) - min(y)}\right) \tag{20}$$

The zero offset z is calculated from the smallest possible value within the quantized space, the minimum value of the floating-point vector and the applied scale.

$$z = Q_{min} - min(Y)s \tag{21}$$

The number of quantization steps Q_{steps} depends in the chosen bit-width b of the integer value:

$$Q_{steps} = Q_{max} - Q_{min} = (2^{b-1} - 1) - (-2^{b-1}) = 2^b - 1 \tag{22}$$

Matrix Multiplication

The matrix multiplication within the quantized space using DYINQ is very similar to a standard matrix multiplication. It differs by subtracting the zero offsets and recalculating the scale in the second step.

$$q_3^{(i,j)} = \sum_{k=1}^{N} (q_1^{(i,k)} - z_1)(q_2^{(k,j)} - z_2) \tag{23}$$

Where $q_3^{(ij)}$ is an element in the resulting matrix at position i, j.

The output scale is calculated by multiplying the scales of the input matrices, the zero offset is set to zero.

$$s_3 = s_1 s_2 \tag{24}$$

Elementwise Addition

When performing an elementwise addition, both input vectors Q_1 and Q_2 are brought to the same scale. Afterwards the elementwise addition is performed.

$$Q_1' = (Q_1 - z_1) \cdot s_2 \tag{25}$$

$$Q_2' = (Q_2 - z_2) \cdot s_1 \tag{26}$$

$$Q_3 = Q_1' + Q_2' \tag{27}$$

The scale of Q_3 is determined by (24), the resulting zero offset is zero.

Elementwise Multiplication

Elementwise multiplication is performed by removing the zero offsets and multiplying the elements. The resulting zero offset is zero, the scale is determined by (24).

$$Q_3 = (Q_1 - z_1) \circ (Q_2 - z_2) \tag{28}$$

Where Q_1 and Q_2 are the factors and Q_3 is the product.

Division

When performing an elementwise division (denoted as ./), several steps are required. At first both input vectors are brought to the same scale.

$$Q_1' = (Q_1 - z_1) \cdot s_2 \tag{29}$$

$$Q_2' = (Q_2 - z_2) \cdot s_1 \tag{30}$$

Where Q_1 is the dividend and Q_2 the divisor.

In the next step the dividend is shifted until all bits of a register are used. This is done to prevent that all elements in quotient are zero since an integer multiplication is used. Afterwards the division is performed in (32) and the scale is updated in (33). The resulting zero offset is zero. On a system with p bits:

$$b_{shift} = (p - 1) - ceil(\log_2(max(|Q_1'|))) \tag{31}$$

$$Q_3 = shift_L(Q_1', b_{shift}) ./ Q_2' \tag{32}$$

$$s_3 = 2^{b_{shift}} \tag{33}$$

Where b_{shift} is the number of bits to shift, Q_3 the quotient and s_3 the quotients scale.

Rescale

The rescale operation implemented in DYNQ is used within a neural network to dynamically rescale feature-maps without providing a predetermined rescale factor. This factor is determined by the rescale method itself.

To determine the rescale factor, at first the zero offset is corrected by calculating a new, optimal zero offset z_2 and correcting the values within the matrix to adjust for the new zero offset.

$$Q_{min} = min(Q_1) - z_1 \tag{34}$$

$$Q_{max} = max(Q_1) - z_1 \tag{35}$$

$$z_2 = \frac{Q_{min} - Q_{max}}{2} - Q_{min} \tag{36}$$

$$z_{adj} = z_2 - z_1$$

$$Q_1' = Q_1 + z_{adj}$$

Where Q_1 is the matrix to rescale, z_1 the zero offset of the matrix, Q_{max} the maximum without zero offset, and Q_{min} the minimum without zero offset. Q_1' is the matrix to rescale with the new zero offset.

After correcting the zero offset, the absolute maximum of the tensor is calculated and the number of shifts required to match a given bit-width *b* is determined.

$$Q_{amax} = max(|Q_1'|) \tag{37}$$

$$b_{shift} = \begin{cases} ceil(\log_2(|Q_{amax} + 1|) - (b-1)) & if \quad \log_2(Q_{amax} - 1) \leq b - 1 \\ 0 & else \end{cases} \tag{38}$$

In the last step the shift is performed on all elements of Q_1' and its scale. The resulting matrix Q_2 is within the constraints of bit-width *b*.

$$Q_2 = shift_R(Q_1', b_{shift}) \tag{39}$$

$$s_2 = shift_R(s_1, b_{shift}) \tag{40}$$

2.6.3 Runtime Accuracy Improvements

To enhance the precision of quantized networks using DYINQ, the bit-width of the quantization can be increased, as demonstrated in [7]. This concept of increasing the bit-width can be applied to the feature-maps in between the networks layers. Rather than maintaining all feature-maps at the same bit-width as the quantization bit-width, a higher bit-width is chosen. The internal, higher bit-width has no impact on the size of the quantized network and can be adjusted after converting the network for the embedded AI framework.

One of the benefits of DYINQ is its support to use different bit-widths within a single operation. If a higher bit-width is used for the output feature-maps of a layer, the subsequent layer can process this higher bit-width as its input, even if the weights of the subsequent layer are stored in a different, lower bit-width.

Details on the implementation of the feature-map bit-width can be found in section 4.4.3. A brief overview of possible gains in accuracy is given in the next subsection.

Accuracy Gains when Increasing Feature-Map Bit-Widths

To analyze how the size of internal feature-maps affects accuracy, an example is implemented using Python. The network used in this analysis has two dense layers, each with 32 neurons. The input vector has 32 elements. All weights and the input vector are randomly generated for 100 samples. Weights and biases are quantized with 8-bits.

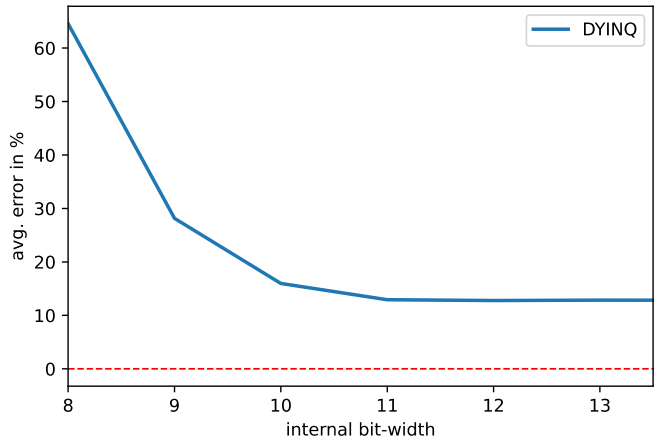

Figure 20: Average error when increasing internal feature-map bit-widths

Figure 20 shows that the average error decreases when increasing the bit-width of the internal feature-maps. The largest decrease in error is between 8- and 9-bits with 24%. For 11-bit or higher, the rate of error reduction slows down. The increase of the average error when using 13-bits is caused by the small number of samples. Detailed error values are shown in Table 3.

Table 3: Average error when increasing internal feature-map bit-widths

Bit-width	Average Error
8	41.17%
9	17.98%
10	9.03%
11	8.03%
12	7.63%
13	7.69%
14	7.5%

The Python script demonstrating the internal bit-width can be found in the appendix under *src/EmmiTranslator/demos/demo_internal_qbits.py*.

2.6.4 Quantization against Soft-float

Quantization has many disadvantages, such as loss of accuracy, especially after running a rescale operation, which is required after each multiplication to avoid overflow. But why quantize at all, if floating-point support can be emulated with soft-float? The reason is the low performance of soft-float compared to integer-only quantization, as shown in Figure 21.

It compares the performance of a quantized convolutional neural network to a floating-point convolutional network. Both networks are executed on the same hardware, using the same input data and are both storing their data in 32-bit variables. The quantized network uses int32_t and the floating-point network 32-bit float, which is emulated in software. It is visible that the quantized neural network outperforms the floating-point network by factor 3.24. This is caused by the high overhead required by soft-float implementations.

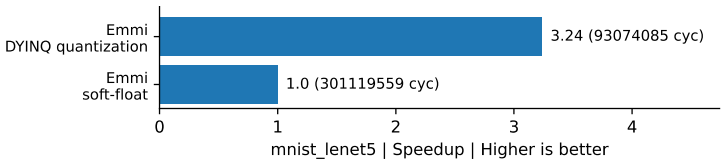

Figure 21: LeNet-5 in Emmi using DYINQ and soft-float

The example comparing the performance can be found in the appendix under *demos/emmi-LeNet5-demo*.

2.6.5 Fix-Point Quantization

An alternative to integer quantization is the fix-point quantization. During fix-point quantization a shared exponent is used between all values. This shared exponent is chosen based on the value distribution of a neural network. Figure 22 illustrates an 8-bit floating-point number, consisting of a sign-bit, a 4-bit exponent and a 3-bit mantissa, as well as an 8-bit fixed-point number, utilizing a shared exponent. [40]

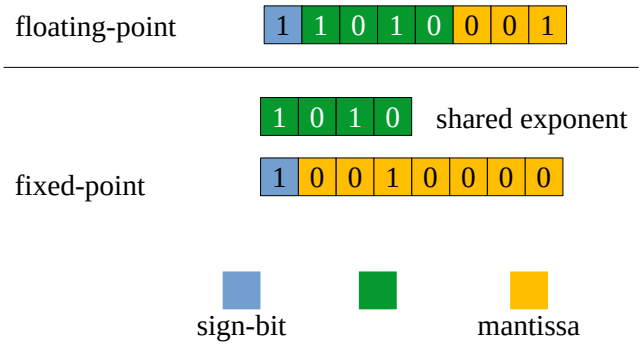

Figure 22: Representation of floating-point and fixed-point numbers

Currently fix-point quantization is neither supported by TensorFlow or PyTorch. For its support additional software, such as QPyTorch is required. Since this thesis deals with the embedded AI Framework Emmi, which focuses on integer quantization, fix-point quantization is not further investigated. [41, 42, 43]

3 Analysis and Design

This section defines the requirements for all software written within the scope of this thesis, analyzes architecture of the embedded AI Framework Emmi and presents the design decisions made to improve the framework.

3.1 Software Requirements

All software developed for the Emmi framework should meet the existing standards within the Emmi Framework, which can be divided into five categories:

TensorFlow Keras Compatibility

- All supported layers are compatible to their TensorFlow Keras equivalents
- Models with unsupported layers are rejected by the EmmiTranslator

Maintainability

- Software is divided into small modules that can be used by multiple components
- Code is clean, efficient and as simple as possible

Documentation

- C code is documented using doxygen compatible function descriptions [44]
- Python code is documented according to PEP 257 [45]

Testability

- Layers in the Emmi Core are testable using unit tests
- EmmiTranslator is testable in integration tests in combination with the EmmiCore and vmath

Embedded RISC-V

- The Emmi Core and its dependencies are executable on RISC-V based embedded systems
- Keep code size in mind: datatypes are as small as possible
- Layers Emmi Core are vectorizable for the RISC-V Zve32x extension using GCC's automatic vectorizer, details on GCC's automatic code vectorization are described in [46]

3.2 Analysis of Existing Components

As already described in section 2.4.1 Emmi is separated into three different modules. This subsection provides an overview of all components and their parts. A more detailed description of all components and parts can be found in [7].

3.2.1 EmmiTranslator

The EmmiTranslator is separated into several namespaces consisting of different Python modules. The root namespace contains the *model_decoder*, which decodes Keras models into an internal model representation, which is then used by the *translator* module to generate Emmi compatible models. The *misc* module contains small tools, for example for printing debug information. The *header_generator* and the *code_generator* are both responsible for generating C code.

The namespace *layer_representation* contains the modules responsible for the internal representation of the original Keras model. *Tools* contains additional functionalities to analyze neural networks. A Python implementation of the quantization technique DYINQ can be found in the namespace *quantization*, which also contains a module implementing the integer-only quantization by Benoit Jacob (abbreviated as JAQ) as well as a module containing buffers which can be used to store quantized tensors.

Unit tests for the modules *DYINQ* and *JAQ* can be found in the namespace *tests*. Unit tests for other modules of the EmmiTranslator are not implemented.

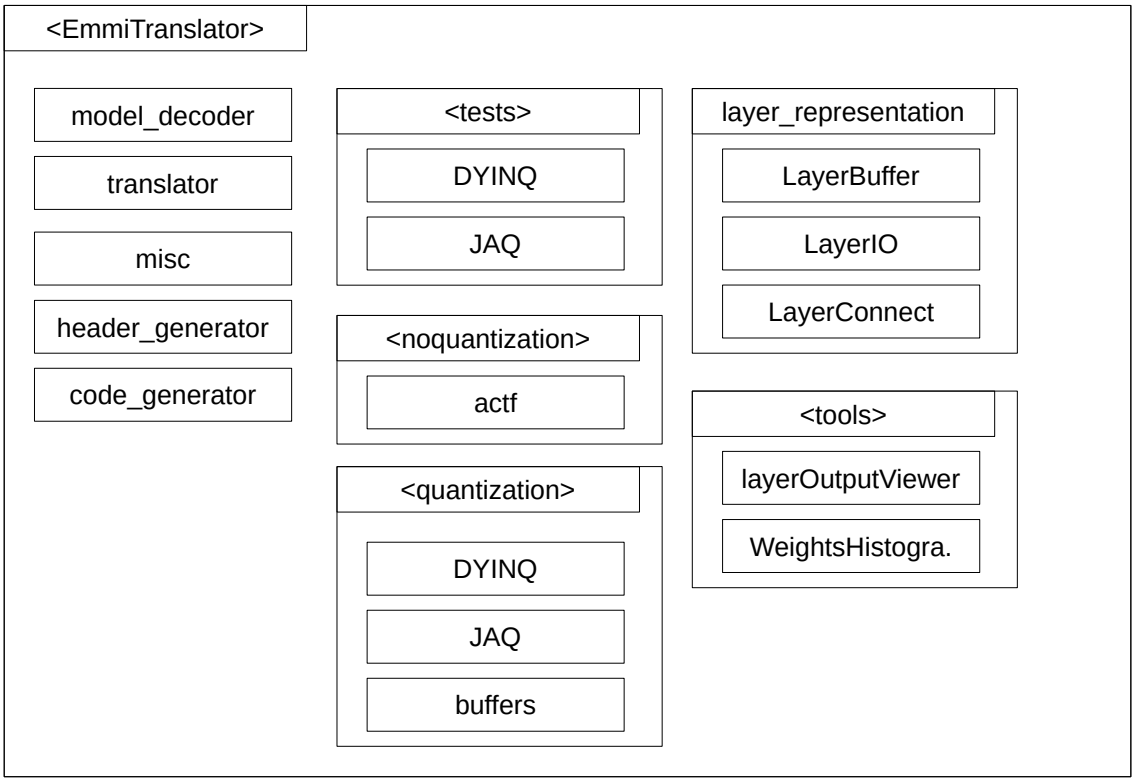

Figure 23: Modules and namespaces of EmmiTranslator

3.2.2 Emmi Core

The Emmi Core is separated into seven C modules. At first there are three modules dealing with the implemented activation functions. The module *actf_flt* implements activation functions for the type float, wheres the module *actf_q32* implements activation functions for 32-bit quantized types. *Actf_factors* stores the thresholds for hard activation functions.

Layers, such as the dense layer or the convolution are implemented in *layers_flt* and *layers_q32*. Small functionalities that are shared across different layers, for example the calculation of dilated tensors, are implemented in the *tools* module. *Analysis* contains static functions such as the mean square error.

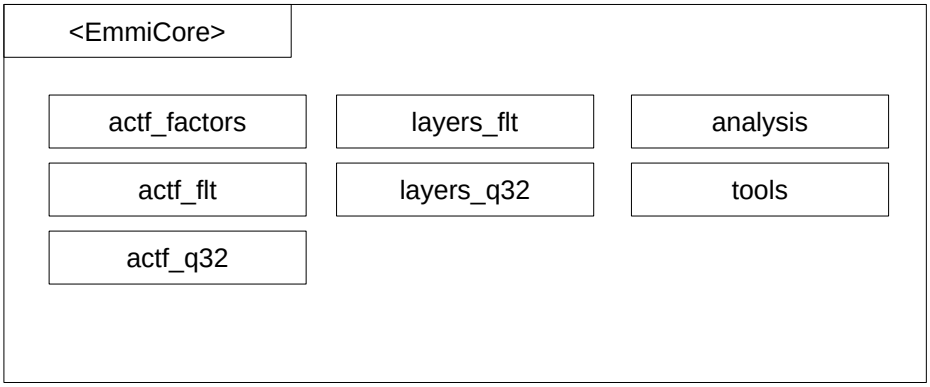

Figure 24: Modules of Emmi Core

3.2.3 Vmath

Vmath is a library for vector and matrix operations, supporting floating-point, integer, and quantized types. Besides vector and matrix operations, it implements approximations for the exponential function and functionalities for comparing and printing tensors. Since vmath is beyond the scope of this thesis, it will not be examined further.

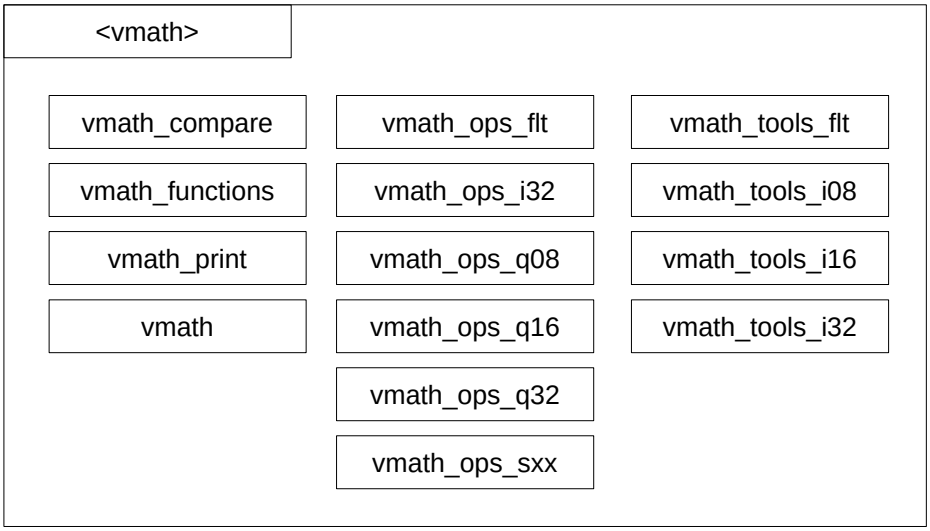

Figure 25: Modules of vmath

3.2.4 Opportunities for Improvements

Usage of lower bit data types

Quantized neural networks are consistently stored in 32-bit types, even when a lower bit-width is sufficient, such as with 8-bit quantization. To support native 8- and 16-bit types for quantized neural networks, it is necessary to add support for the already by vmath supported types Tensor_q16 and Tensor_q8 in Emmi. Additionally, the EmmiTranslator must be updated to select the most appropriate data type, which will reduce the size of quantized networks.

Accuracy Improvements

The feature-maps in quantized networks currently use the same bit-width as the weights, which is not always necessary since DYINQ supports operands with varying bit-widths. By increasing the bit-width of the feature-maps, accuracy can be improved without increasing the model size, which was shown in section 2.6.3.

3.3 Design Decisions

During the design phase, general design decisions for the batch normalization are made. It is also decided how the framework is restructured, which parameters the new build system should support and where function prototypes for the new features are placed and integrated into the framework.

3.3.1 Batch Normalization

Since a true quantized implementation of batch normalization at inference time would require a square root function within the quantized space, which has not yet been developed, a true quantized implementation of the batch normalization will not be realized within Emmi. Instead the batch normalization folding will be implemented by improving and using the package “tf-batchnorm-fold”.

3.3.2 Project Structure

Currently, both Emmi and vmath store all their files in a single project directory without a clear separation of source, header, and documentation files. This lack of organization hinders the delivery of pre-built binaries and headers to customers. Implementing a structured approach to file management, such as separating source, header, and documentation files, will streamline the project structure and facilitate easier distribution of necessary components to customers.

Approach

The project structure of Emmi and vmath are restructured to separate source code, headers, documentation, and build scripts. All source code is moved to the “src” directory, all public header files to the “include” directory and all documentation to the “doc” directory. The build scripts are kept alongside the readme in the repositories root directory.

Table 4: New Emmi root project structure

Directory / Filename	Description
build/	Contains object files and executable. Directory is automatically generated when executing the makefile.
doc/	Emmi documentation in markdown.
include/	Public header files.
src/	C source code files.
Makefile	Build rules and profiles
README.md	Repository description

Table 5: New vmath project structure

Directory / Filename	Description
Build/	Contains object files and executable. Directory is automatically generated when executing the makefile.
include/	Public header files.
src/	C source code files.
Makefile	Build rules and profiles
README.md	Repository description

Unit Tests for both repositories are still maintained in separate repositories. These so called testbenches contain besides the unit tests also integration tests and benchmarks for accuracy and performance. How the testbenches include the repositores of vmath and Emmi is shown in Figure 26. The source code of the unit and integration tests are stored in the subdirectories “e5aisuite-testbench/tests” for the Emmi testbench and “vmath-tests/tests” for the vmath testbench. Note that the testbench of Emmi still uses the old name “e5AISuite” for compatibility reasons.

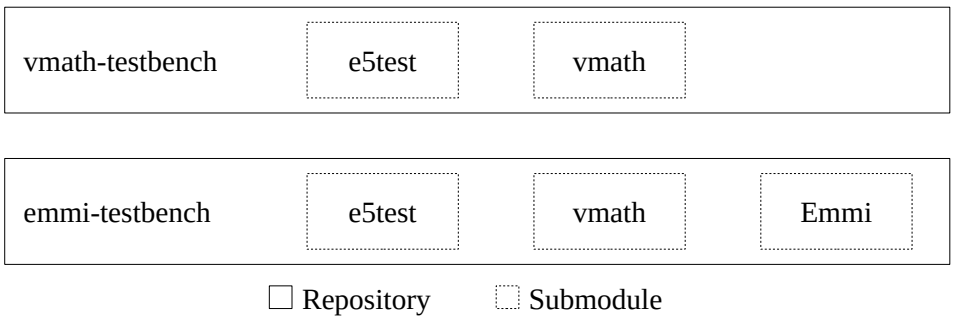

Figure 26: Repository structure of the testbenches

3.3.3 Build System

The Emmi Core, vmath and the testbenches each require individual makefiles for building for RISC-V and AMD64 platforms. These makefiles both employ their own build profiles, parameters and do not offer support for different GCC prefixes. A redesign of the build system to use a single makefile per

component for multiple platforms will reduce maintenance effort and simplify the build process. Support for different GCC prefixes and preconfigured targets is also added.

Parameters

To set the destination platform, the target parameter is introduced. Options are:

Table 6: Supported target platforms

Option	**Description**
target=amd64	Compile for AMD64
target=emsa5	Compile for RISC-V, link the EMSA5 HAL, use the EMSA5 memory map and startup script
target=spike	Compile for RISC-V, use spike memory map and startup script

To set the compiler optimizations and to control weather debug information should be included in the binary file, the profile parameter is introduced.

Table 7: Supported compilation profiles

Option	**Description**
profile=default	-O2, enable debug information
profile=debug	-O0, enable debug information
profile=vector	-O2, enable automatic vectorization (RISC-V only)
profile=release	-O2

3.3.4 EmmiTranslator

Within the EmmiTranslator the planned changes and improvements focus on the Python modules *model_decoder* and *translator*. Within the *model_decoder*, the in Figure 27 red marked functions and variables will be added during the implementation.

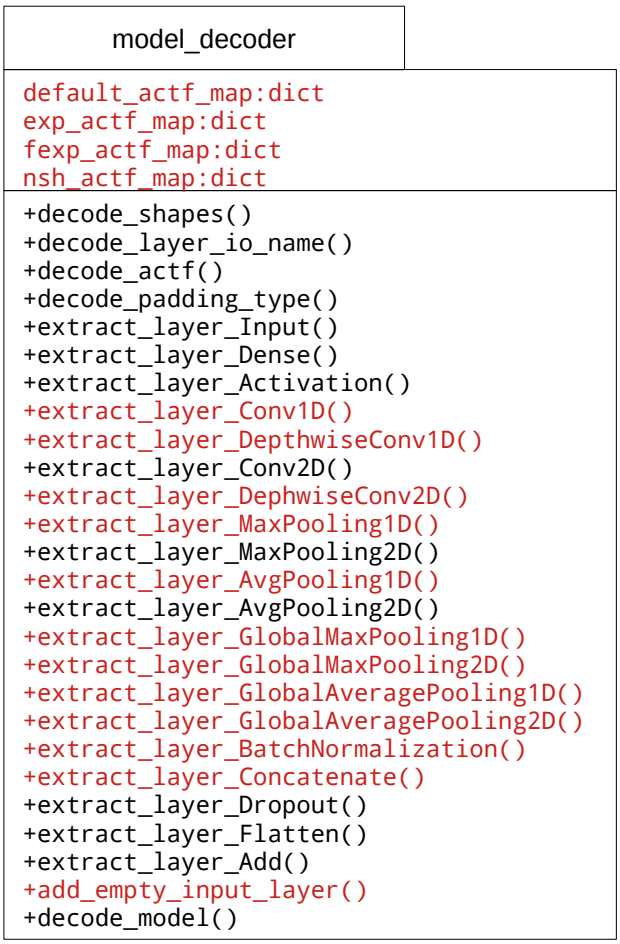

Figure 27: EmmiTranslator module model_decoder

The planned changes for the *translator* focus on the process when translating the model. The changes are described in section 4.3.3.

3.3.5 Emmi Core

Most of the changes to the Emmi Framework are made within the Emmi Core. Here, support for new quantization types is added, as well as support for new layers. Figure 28 shows modified modules in green. Newly implemented modules are shown in red.

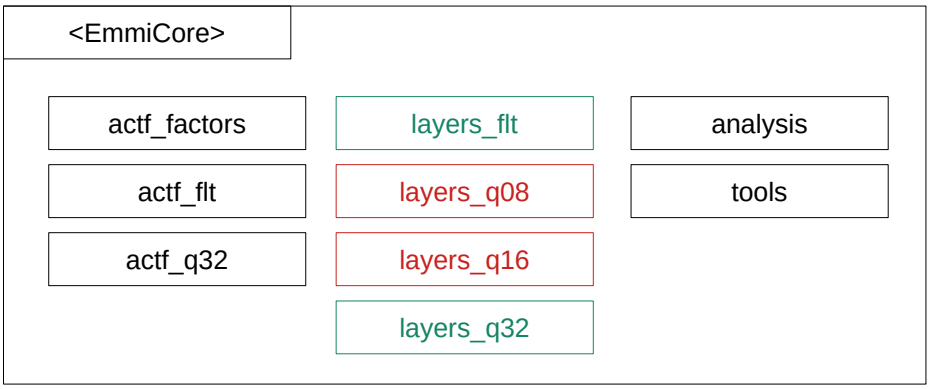

Figure 28: New Emmi Core modules

For the *layers_flt* and *layers_q32* modules and an overview of all implemented functions is presented in Figure 29. The modules *layers_q08* and *layers_q16* implement the same functionality as *layers_q32*, but with native 8- and 16-bit types.

Figure 29: Emmi Core modules layers_flt and layers_q32

4 Implementation

4.1 Convolutional Operations

4.1.1 One-Dimensional Convolution

The one-dimensional convolution is the first operation implemented during the course of this thesis. The implementation supports parameters such as stride, dilation and padding, wheres the implemented zero padding does not add zeros at the borders of the feature-map. Instead the indexing is modified in a way that no expensive resize of the feature-map is required.

The one-dimensional convolution is implemented for floating-point and for quantized neural networks. An overview of all major steps performed in the implementation is given below. Steps within [square brackets] are only performed when performing a quantized operation. Implementation details of each step may differ between the quantized and the floating-point implementation.

1. Check input shape
2. [Calculate output quantization parameters]
3. Calculate dilated kernel shape
4. Calculate output dimensions, start- and stop offset for indexing the input feature-map
5. Write bias into output tensor
6. Iterate over kernels, input channels and elements per channel
   6.1. Check padding, calculate kernel indexing offsets
   6.2. Iterate over current kernel
      6.2.1. Apply current kernel element to current input feature
   6.3. Write result for current input feature to output tensor
7. [Rescale operation]
8. Apply activation function

4.1.2 Depth-wise Convolution

The depth-wise convolution is implemented for one- and two-dimensional inputs with multiple channels. Both implementations support parameters stride, dilation and padding. The major difference in the implementation to the standard convolution is in the sixth step: Instead of iterating over input kernels (line 302 in Listing 1), input channels (line 303) and elements per channel (line 304), it is iterated over input channels (line 380), the channel specific kernels (line 381) and the elements per channel (line 384).

Listing 1: Compare implementation of depth-wise and standard convolution

```
                                                                Standard Convolution
302 for(int32_t q = 0; q < kernel->q; q++) {
303       for (int32_t p = 0; p < in->p; p++) {
304             size_t out_index = q*out->elements_mn;
305             for (int32_t n = -start_offset_n; n < (int32_t)
                              (in->n - stop_offset_n); n+=stride) {
306                   float kernel_out = 0;
                      ...

                                                              Depth-wise Convolution
380 for (int32_t p = 0; p < in->p; p++) {
381       for(int32_t q = 0; q < kernel->q; q++) {
382             size_t out_index = (p*kernel->q + q) * out->elements_mn;
383             for (int32_t n = -start_offset_n; n < (int32_t)
                            (in->n - stop_offset_n); n+=stride) {
384                   float kernel_out = 0;
                      ...
                                                Excerpts from file "emmi/src/layers_q32.c"
```

4.1.3 Floating-Point and Quantized Implementation

Discrepancies between the floating-point and quantized implementations of convolutional operations are, as a result of a portable implementation, confined to the following steps (excerpt from section 4.1.1):

2	[Calculate output quantization parameters]
5.	Write bias into output tensor
6.2.1.	Apply current kernel element to current input feature
6.3	Write result of current input feature to output tensor
7	[Rescale operation]

Calculate output quantization parameters

In the quantized implementation, the second step calculates the output quantization parameters. The zero offset is set to zero, since it is subtracted. The scale is calculated from the input scale, the scale of the kernel and the scale of the bias. An excerpt is shown in Listing 2.

Listing 2: Calculate quantization parameters in a 1D convolution

```
281 out->zero_offset = 0;
282 out->scale = in->scale * kernel->scale * bias->scale;
                                                Excerpts from file "emmi/src/layers_q32.c"
```

Write bias into output tensor

When writing the bias into the output tensor of the convolutional operation, the quantized and the floating-point implementation differ slightly. In the quantized implementation the zero offset of the bias must be removed from the values, which are afterwards brought to the output scale of the convolution, as shown in Listing 3.

Listing 3: Writing the bias into the ouput tensor of a 1D convolution

```
                                Floating-Point Implementation "emmi/src/layers_flt.c"
606 for (size_t p = 0; p < out->p; p++) {
607       for (size_t i = 0; i < out->elements_mn; i++) {
608             out->data[p*out->elements_mn + i] = bias->data[p];
609       }
610 }
                                     Quantized Implementation "emmi/src/layers_q32.c"
295 for (size_t p = 0; p < out->p; p++) {
296       for (size_t i = 0; i < out->elements_mn; i++) {
297             out->data[p*out->elements_mn + i] =
                  (bias->data[p] - bias->zero_offset) * in->scale * kernel->scale;
298       }
299 }
```

Apply current kernel element to current input feature

The implementation of the 1D convolution is using the outer loops to iterate through the input tensor and the inner loop, to iterate over the kernel (which is shown in Listing 4). When performing the multiplication of a kernel element and an input element, the quantized implementation requires the removal of the zero offset before multiplying, similar as described in section 2.6.2.

Listing 4: Inner loop of a 1D convolution

```
                                Floating-Point Implementation "emmi/src/layers_flt.c"
617 float kernel_out = 0;
... ...
629 for (int32_t kn = kn_start; kn < kn_stop; kn++) {
630       int32_t ind = tensor_representative_index(kernel, 0, kn, p, q);
631       float a = kernel->data[ind];
632       int32_t i_idx_n = n + kn * dilation;
633       int32_t b_ind = tensor_representative_index(in, 0, i_idx_n, p, 0);
634       float b = in->data[b_ind];
635       kernel_out +=  a * b;
636 }
637 out->data[out_index] += kernel_out;
                                     Quantized Implementation "emmi/src/layers_q32.c"
306 int32_t kernel_out = 0;
... ...
318 for (int32_t kn = kn_start; kn < kn_stop; kn++) {
319       int32_t ind = TENSOR_REPRESENTATIVE_INDEX_MACRO(kernel, 0, kn, p, q);
320       float a = kernel->data[ind];
321       int32_t i_idx_n = n + kn * dilation;
322       int32_t b_ind = TENSOR_REPRESENTATIVE_INDEX_MACRO(in, 0, i_idx_n, p, 0);
323       float b = in->data[b_ind];
324       kernel_out +=  (a - kernel->zero_offset) * (b - in->zero_offset);
325 }
326 out->data[out_index] += kernel_out * bias->scale;
```

Write result of current input feature to output tensor

After iterating though all kernel elements within the inner loop, the result is written on-top of the bias, which is already stored in the output tensor. Since an addition is performed, both parameters must use the same scale. For this, the kernel output is multiplied with the scale of the bias when using the quantized implementation, as shown in Listing 5.

Listing 5: Write result to output tensor in a 1D convolution

```
                                Floating-Point Implementation "emmi/src/layers_flt.c"
637 out->data[out_index] += kernel_out;
                                     Quantized Implementation "emmi/src/layers_q32.c"
326 out->data[out_index] += kernel_out * bias->scale;
```

Rescale operation

The rescale operation is only applied in the quantized implementation. It takes the tensor containing the output of the convolution and the bit-width to rescale to and rescales all elements in-place when passing the result tensor as input and output parameter.

Listing 6: Calling the rescale operation

```
331 tensor_rescale_q32_q32(out, out, nbits);
                                          Excerpts from file "emmi/src/layers_q32.c"
```

4.2 One-Dimensional Pooling Operations

4.2.1 Max Pooling

The implementation of max pooling 1D supports parameters such as stride, dilation and padding. Instead of iterating through the elements of the kernel, the iteration is performed on all elements of the pooling scope, with the aim of determining the maximum value of the current scope. Each step of the implementation is outlined below:

1. Check input shape
2. [copy quantization parameters to output tensor]
3. Calculate dilated kernel shape
4. Calculate output dimensions and start- and stop offset for indexing the input feature-map
5. Iterate over input

   5.1. Check padding, calculate pooling indexing offsets

      5.1.1. Iterate though pooling scope

         1. Find maximum value

   5.2. Write maximum value into output tensor

4.2.2 Average Pooling

Average pooling 1D supports, as max pooling 1D, the parameters stride, dilation and padding. It is iterated though the elements within the pooling score to determine its average. Each step performed within the implementation is outlined below:

1. Check input shape
2. [copy quantization parameters to output tensor]
3. Calculate dilated kernel shape

4. Calculate output dimensions and start- and stop offset for indexing the input feature-map
5. Iterate over input
    5.2. Check padding, calculate pooling indexing offsets
    5.3. Iterate through pooling scope
        5.3.3. Accumulate values within pooling scope
    5.4. Calculate average
    5.5. Write average into output tensor

4.2.3 Global Max Pooling

Global max pooling iterates though all elements in each channel and determines the maxima per channel. It does not support any parameters, an overview of the implemented steps is presented below:

1. Check input shape
2. [copy quantization parameters to output tensor]
3. Iterate through channels
    3.1. Iterate through all elements within a channel
        3.1.1. Find maximum value
    3.2. Write maximum value into output tensor

4.2.4 Global Average Pooling

The implementation of global average pooling only differs in a few steps from the implementation of global max pooling. As global max pooling, global average pooling does not support any parameters.

1. Check input shape
2. [copy quantization parameters to output tensor]
3. Iterate through channels
    3.2. Accumulate through all elements within a channel
    3.3. Calculate average of a channel
    3.4. Write average value into output tensor

4.2.5 Floating-Point and Quantized Implementation

The floating-point and quantized implementations for the pooling operations do not differ much. In quantized pooling no weights are involved, meaning only a single scale is used within the whole operation. This enables the quantized values to be handled in the same way as float values. No additional rescale operations are required.

4.3 Batch Normalization

For the batch normalization several implementations are provided. At first a layer using floating-point values is implemented in Emmi using C code. Afterwards this implementation is reused in an implementation for fake quantization. Meaning a layer that takes quantized values, performs the batch normalization as floating-point, and quantizes the result.

In the next step the support for folding the batch normalization parameters is added to Emmi. For this the Python package “tensorflow-batchnorm-folding” is analyzed and improved. Afterwards the package is added as dependency of the EmmiTranslator, which is modified to use the package to fold batch normalization layers into the current network.

4.3.1 Floating-Point Implementation

Batch normalization is implemented for all axes supported by emmi. The axis used is passed by the caller to the batch normalization function, which uses a switch case statement to perform the batch normalization on the selected axis.

An excerpt of the batch normalization function is shown in Listing 7. It shows how batch normalization is performed on axis m: It is iterated through all elements of the input tensor and performed using equation (10).

Listing 7: Batch normalization on axis m

```
1067 for (int q = 0; q < in->q; q++) {
1068   for (int p = 0; p < in->p; p++) {
1069     for (size_t m = 0; m < in->m; m++) {
1070       for (size_t n = 0; n < in->n; n++) {
1071         // Input and output have the same index.
1072         size_t i_in = tensor_representative_index(in, m, n, p, q);
1073 
1074         // Determine index for BN parameters, all BN parameters have the same
1075         // shape -> only a single index for all BN parameters is required.
1076         size_t i_bn_params = tensor_representative_index(moving_mean,m,0,0,0);
1077 
1078         // Calculate BN for a single element.
1079         out->data[i_in] = gamma->data[i_bn_params] * ((in->data[i_in] -
                 moving_mean->data[i_bn_params])
1080               / sqrt(moving_variance->data[i_bn_params] + epsilon)) +
                    beta->data[i_bn_params];
1081       }
1082     }
1083   }
1084 }
                                                Excerpt from file "emmi/src/layers_flt.c"
```

4.3.2 Fake Quantized Implementation

The implemented batch normalization for quantized neural networks is called with a quantized input tensor and floating-point batch normalization parameters. The input tensor is dequantized and passed to the floating-point implementation. Afterwards the output is quantized. The implementation is shown in Listing 8.

Listing 8: Fake quantized implementation of batch normalization

```
880 int32_t batchnormalization_q32(
      Tensor_q32 *in, Tensor_q32 *out, const Tensor_flt *moving_mean,
      const Tensor_flt *moving_variance, const Tensor_flt *beta,
      const Tensor_flt *gamma, const float epsilon, const enum Axis axis,
      const uint8_t nbits) {
881     float f_data[in->elements_mnpq];
882     Tensor_flt f;
883     f.type = FLOAT;
884     f.data = f_data;
885     f.elements_max = in->elements_mnpq;
886     int32_t ret = tensor_dequant_q32(in, &f);
887     ret |= batchnormalization_flt(
              &f, &f, moving_mean, moving_variance, beta, gamma, epsilon, axis);
888     ret |= tensor_quantization_q32(&f, false, 0, false, 0, nbits, out);
889     return ret;
890 }
                                          Excerpts from file "emmi/src/layers_q32.c"
```

The implemented approach has many drawbacks, particularly when running on integer-only devices. The use of soft-float to dequantize to floats on such devices results in a reduction of performance. Furthermore, the performance is negatively affected by first dequantizing and then quantizing the feature-maps.

A quantized implementation of the batch normalization is not employed, as the floating-point and the fake quantized implementation of batch normalization within Emmi only serve as a fallback in the event that folding during the model conversion is not possible.

4.3.3 Folding

To add support for batch normalization folding the Python Package “tensorflow-batchnorm-folding” is used. Its feature set and code base is analyzed, improved and integrated into the EmmiTranslator in the form of a Python dependency.

Analyzing the Python Package

At first the feature set and compatibility of the package are analyzed. It requires the no longer supported TensorFlow version 2.9 and does not provide support for newer versions. As result the EmmiTranslator is not compatible, since it requires TensorFlow 2.11 or newer. On the featureside the folding of batch normalization layers into dense and convolutional 2D is supported, but limited to functional models. TensorFlows sequential model architecture is not supported.

At last, the package structure is analyzed: Directory *src/batch_normalization_folding* contains two subdirectories, one containing the implementation, consisting of the decoding of the input TensorFlow model, the batch normalization folding and the rebuilding of the TensorFlow model. The other one containing the required functionalities for writing unit tests for comparing models with folded batch normalization layers with the original ones. The unit tests itself are implemented in a file called *folder.py* located in the *src/batch_normalization_folding* directory of the package. They test the folding of batch

normalization layers on several applications distributed with TensorFlow. The process of testing a model is illustrated in an activity diagram shown in Figure 30.

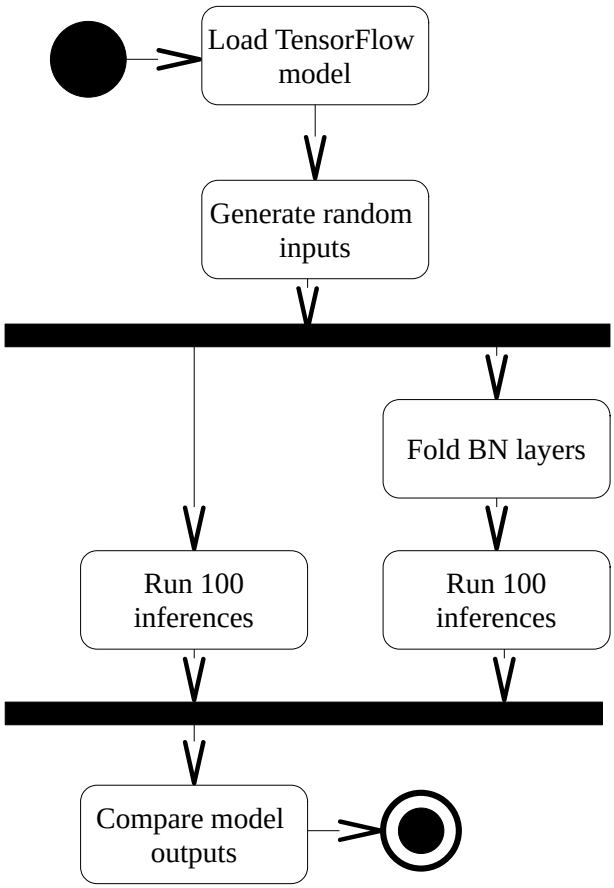

Figure 30: Process of testing a folded model

Adding Compatibility to TensorFlow 3.11

A single change is required to add compatibility with TensorFlow 3.11. The network decoding functionality is updated to use the layer.get_input_shape_at() function instead of the deprecated layer.input.shape property.

Add Support for Sequential Models

To support sequential models, the decoding of the original TensorFlow model is updated. As sequential TensorFlow models do not have a dedicated input layer, the search for input layers within the graph fails. To fix this, the model type is checked. If a sequential model is found, the first layer is marked as an input layer, allowing the model decoder to continue.

Add Support for 1D Convolutions

To add support for 1D convolutions, several files are modified. The file *src/batch_normalization_folding/TensorFlow/calculus.py* contains all implementations required for the folding of batch normalization layers into dense and 2D convolution layers. Added are the functions fold_root_backward_conv1D() and fold_root_forward_conv1D(), which perform the batch normalization folding as described by equations (11) and (13) for one dimensional convolutional layers.

Also modified is the file *src/batch_normalization_folding/TensorFlow/*add_biases.py. Its functions are used when rebuilding the network graph. Implemented is the function add_conv1D_bias(), which generates a new one dimensional, convolutional layer with the given parameters.

The file *src/batch_normalization_folding/TensorFlow/to_fold_or_not_to_fold.py* implements the functionality to decide weather a layer is folded or not. Here the one dimensional convolution is added to the list of supported operations.

Miscellaneous Improvements

Besides the improvements required for a compatibility to the EmmiTranslator, four minor improvements are implemented into the Python package tensorflow-batchnorm-folding. In a first step, batch normalization layers that have not been folded are counted and presented to the end user when enabling the verbosity of the package. Subsequently, the evaluation of the unit tests is improved. Now, instead of displaying a tick for each executed unit test, a pass or fail symbol is displayed depending on the actual result of the unit test. Also, an additional folding algorithm is implemented besides BaN-OFF. The folding algorithm, titled ‘simple’ only folds the batch normalization if it is foldable into a directly neighboring layer. At last, the projects metadata is updated as described in PEP 621. [47]

Fork and Merge Request

All changes made to the Python package tensorflow-batchnorm-folding are committed to a fork of the main project. Out of this fork, a merge request to the main project is created. It includes all changes and improvements and was integrated by the projects maintainer into version 1.0.9 of the package. [48]

Integration into EmmiTranslator

In the EmmiTranslator Python package, tensorflow-batchnorm-folding is added as dependency as shown in Listing 9.

Listing 9: Dependencies of Python package EmmiTranslator

```
19 dependencies = [
20   'numpy >= 1.24',
21   'Pillow >= 9.4',
22   'matplotlib >= 3.6.3',
23   'tensorflow >= 2.11',
24   'tensorflow-batchnorm-folding >= 1.0.9',
25   'pyserial >= 3.5',
26 ]
                                    Excerpt of "EmmiTranslator/pyproject.toml"
```

Within the translate_model() function, which is called by callers who want to translate a TensorFlow model for the embedded AI framework Emmi, the batch normalization folding is performed before decoding and converting the TensorFlow model for Emmi. An overview of the complete process when calling translate_model() is given in Figure 31.

The function translate_model() starts by checking if the output path exists and raises an exception if it does not. In the second step, the ctype, which is a string telling the converter the datatype to use when storing the weights of the model, is decoded. The global verbosity of the translator is configured and the desired model name is made compatible with the C code naming guidelines. It is checked if a TensorFlow model is passed, and if batch normalization folding should be enabled. The following check for the batch-normaization-package is good practice and hints users that are using the EmmiTranslator without a proper

installation by their package manager to install tensorflow-batchnorm-folding, how this is done in Python is shown in Listing 10. Afterwards the model is folded, decoded and converted to Emmi.

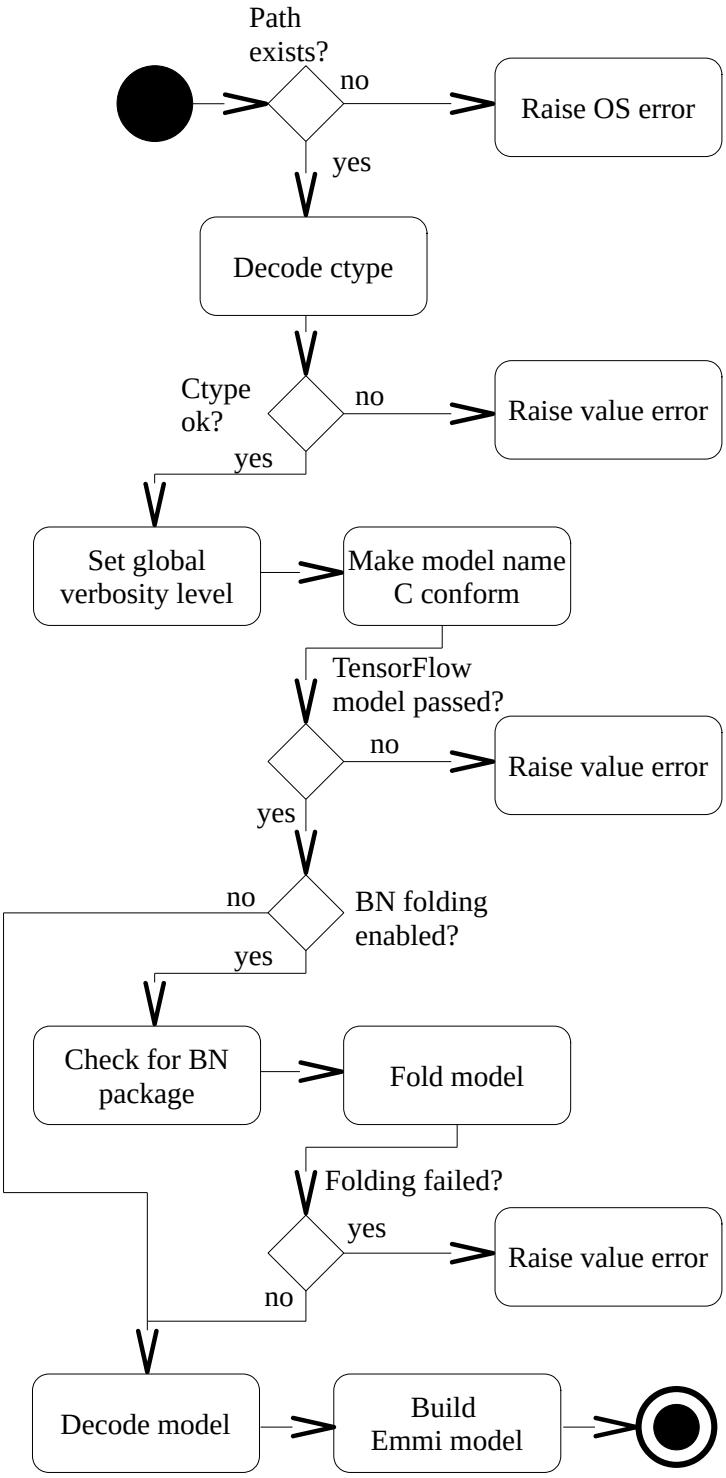

Figure 31: Activity diagram of function translate_model()

Listing 10: Check for tensorflow-batch-normalization package

```
120 if (batchnorm_fold==True):
121   try:
122     # Ensure to install batch_normalization_folding from
123     # https://gitlab.com/paspf/batch-normalization-folding
124     from batch_normalization_folding.folder import
            fold_batchnormalization_layers
125   except ModuleNotFoundError:
126     print_line(0)
127     print("Install batch_normalization_folding by using pip install
                  batch-normalization-folding"
128           "For the documentation of this package, visit:"
129           "https://gitlab.com/edouardyvinec/batch-normalization-folding")
130     print_line(0)
131     exit(1)
                          Excerpt of "EmmiTranslator/EmmiTranslator/translator.py"
```

4.4 General Improvements

Besides the addition of new features and functionality, general improvements to the projects build system, support for additional datatypes and a callback functionality are added to the embedded AI framework.

4.4.1 Build System

The makefiles of Emmi, vmath and the testbenches are completely rewritten to fulfill the functionality described in 3.3.3. At first support for building for specific targets is added. Later on, build profiles and the support for custom gcc prefixes is added.

4.4.2 Native 8- and 16-bit type support

Since its initial development, Emmi has only supported the structures Tensor_flt (for floating-point tensors) and Tensor_q32 (for quantized tensors, where int32_t is used to store their elements), which means that when 8-bit quantization is used, each weight and bias is stored in a 32-bit type, leaving 24-bits unused. To correct this, support for processing quantized data from Tensor_q8 and Tensor_q16 is implemented in Emmi by adding the functionality which is already present for Tensor_q32 also for the types Tensor_q8 and Tensor_q16. As described in 3.3.5 the implementations of all layer functionality for Tensor_q8 and Tensor_q16 are stored within the Emmi Core directories *src/layers_q08.c* and *src/layers_q16.c*.

To generate models for Tensor_q8 and Tensor_q16, the EmmiTranslator is modified. The desired type is selected using the "ctype" parameter, which is passed to the EmmiTranslator when the model is converted. By default, the parameter "ctype" is set to "auto". This means, that the type used is selected automatically based on the converter and quantization settings.

4.4.3 Feature-Map Bit-width

A method to improve the accuracy of the network is to increase the bit-width of the feature-maps, as shown in section 2.6.3. To implement this feature, the EmmiTranslator and the Emmi C code are modified.

The C code is modified so that each quantized layer and activation function requires an additional parameter: the bit-width of the output feature-map. Instead of rescaling the output feature-map to the bit-

width of the layer weights, it is rescaled to the bit-width passed as a parameter. Since the EmmiTranslator generates the calls to the layers and activation functions, it is modified to insert the configured bit-width as a new parameter. Since the DYINQ quantization technique can handle inputs and outputs with different bit widths, no additional modifications in the program logic are required.

4.4.4 Shared Bias Quantization

Another feature added to Emmi and the EmmiTranslator is the shared bias quantization. When enabled during model translation, the same quantization is selected for a layer's weights and biases. This means that the scale and zero offset of a layer's weights and biases are the same. At inference time, this results in a small performance improvement, as only two multiplications are required to bring a layer's input feature-maps, weights and biases to the same scale, rather than three.

4.4.5 Activation Function Selection

Within Emmi some activation functions, such as the sigmoid function, are implemented several times. One time using the exponential function from the c standard library, one time using an approximation of the exponential function, and one time using a hard, step-wise defined function. Currently the EmmiTranslator always inserts the default implementation of the activation function when translating the model. When another implementation shall be used, the generated C code must be edited. The new implementation introduces customizable maps of activation functions. They are passed to the EmmiTranslator and map activation functions to the actual Emmi implementation. Table 8 shows all preconfigured mappings.

Table 8: Preconfigured activation function mappings

	Function Map Name			
	default_actf_map	exp_actf_map	fexp_actf_map	nsh_actf_map
Description	Default function map applied by the EmmiTranslator.	Utilizes c standard library exponential function	Utilizes an approximated exponential function	Utilizes step-wise defined sigmoid function.
linear	linear	linear	linear	linear
Relu	relu	relu	relu	relu
sigmoid	sigmoid_fexp32	sigmoid_exp	sigmoid_fexp32	nsh_sigmoid
hard_sigmoid	hard_sigmoid_tf	hard_sigmoid_tf	hard_sigmoid_tf	hard_sigmoid_tf
softmax	softmax_fexp32	softmax_exp	softmax_fexp32	softmax_fexp32
tanh	tanh_fexp32	tanh_exp	tanh_fexp32	tanh_fexp32
swish	swish_fexp32	swish_exp	swish_fexp32	swish_fexp32
elu	elu_fexp32	elu_exp	elu_fexp32	elu_fexp32
selu	selu_fexp32	selu_exp	selu_fexp32	selu_fexp32

For more details on the implemented functions, see [56]. To customize a preconfigured mapping, the desired entries in the map are changed before passing it to the translator as shown in Listing 11.

Listing 11: Customize a function map

```
from Emmi.translator import translate_model
from Emmi.model_decoder_tf import ModelDecoderTF

# Load default mapping.
actf_map = ModelDecoderTF.default_actf_map.copy()

# Change mapping of softmax function.
actf_map["softmax"] = "softmax_exp"

translate_model(model=a_TF_model,
                dest_path="model_export",
                model_name="an_emmi_model",
                qbits=8,
                qbits_intern=12,
                ctype="int8_t",
                actf_mapping=actf_map,
                batchnorm_fold=True,
                shared_bias_quantization=True)
```

4.4.6 Callbacks and Debug Information

To enable developers to easily access the feature-maps within a model, to analyze the required runtime per layer (when executing on the EMSA5), or to place callback functions, which are executed after each layer, the EmmiTranslator model generation is improved.

Debug Information: Feature-Maps

When translating a model with the Option:

```
generate_debug_information='values'
```

The EmmiTranslator inserts prints of the feature-maps after each layer in the converted model. The automatically generated C code that prints the feature-map of a layer is shown in Listing 12. Not only the feature-map itself is printed, but also the name of the layer that outputs the feature-map.

Listing 12: Example for automatically generated C code printing feature-maps

```
maxpool2d_flt(&tmp_in, &tmp_out, 2, 2, 2, 2, 1, 1, valid_padding);
printf("Layer name: max_pooling2d | outputs:\n");
vmath_print(&tmp_out);
```

Debugging Information: Runtime Markers

Another option added to the EmmiTranslator is the insertion of runtime markers. These markers indicate the start and the end of the execution of a layer. They are printed to the outstream of the target system, and are analyzed by the Python module *EmmiTranslator.tools.runtimeAnalysis.RuntimeAnalysis*. When executing this module, it listens to a given COM port, and measures the time between the markers. When receiving the last marker, indicating the end of the models inference, a bar graph as shown in Figure 32 is generated. It shows the runtime time per layer.

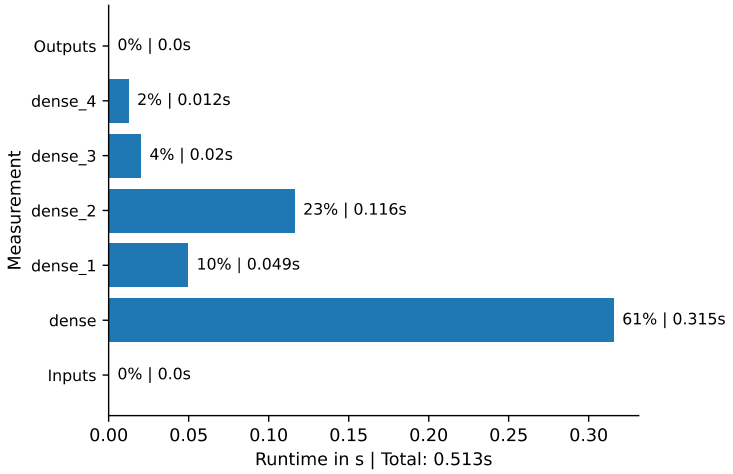

Figure 32: Bar chart created with the EmmiTranslator runtime tool

After converting a model with the option:

```
generate_debug_information='runtime_markers'
```

And running it on the target system, the runtime analysis is started as shown in Listing 13.

Listing 13: Using the Emmi Runtime Analysis

```
1 from EmmiTranslator.tools.runtimeAnalysis import RuntimeAnalysis
2
3 h = RuntimeAnalysis(timeout=20,
4                      serial_port="com6",
5                      figure_path="figures/out.svg")
6 h.start()
```

The Python module for analyzing the runtime per layer in a model can be found in the appendix under *src/EmmiTranslator/tools/runtimeAnalysis.py*.

Callbacks

The last option added to the EmmiTranslator is the placement of callback functions after each executed layer:

```
generate_debug_information='callback'
```

This enables the developer to pass a callback function, when calling the inference function of the model. On each callback, the ID of the executed layer, the name of the layer, the layers output feature-map, the layers return value and user-defined parameters are passed to the callee.

5 Evaluation

This chapter describes all testing and evaluation steps performed to verify and evaluate the implementations. It starts by presenting a brief overview of the implemented unit and integration tests, the used test scenarios and benchmarks. At the end it is presented how to executed the Emmi testbench in Spike, a RISC-V simulator.

5.1 Unit Tests

Unit tests are developed and implemented for all layers added to Emmi in this thesis (see Table 2). To write and integrate unit tests into the testbench, random tensors are generated and applied to the corresponding TensorFlow implementation. From these results the unit tests for Emmi are generated. The steps involved in generating a unit test are outlined below:

1. Define test case
2. Implement reference generation using TensorFlow
3. Generate unit test for Emmi using test case definitions and references from TensorFlow

During the course of this thesis, 555 unit tests were implemented, bringing the total number of unit tests implemented for the Emmi core to 806. All scripts that generate unit tests can be found in the appendix under *src/e5aisuite-testbench/scripts*. The actual unit test implementations can be found in *src/e5aisuite-testbench/e5aisuite-testbench/tests*.

5.2 Integration Tests

In addition to unit testing, integration testing is performed by translating models into Emmi, executing them within Emmi, and comparing the results of inference between TensorFlow and Emmi. The steps involved in implementing an integration test are outlined below:

1. Define test case
2. Generate data for test case
3. Train TensorFlow model
4. Run inferences using trained model and record outputs
5. Convert TensorFlow model to Emmi using EmmiTranslator
6. Write test cases running inferences in Emmi and comparing the outputs with the recorded results

In the course of this thesis, several integration tests are implemented for ten models, resulting in a total of 657 integration tests for Emmi. The models developed for the integration tests used in this thesis are described in section 5.2.1.

5.2.1 Models for Integration Tests

The integration tests developed during this thesis are focused on testing batch normalization layers (with and without folding), one-dimensional convolutions and poolings.

Scenario and Dataset

All integration tests are trained supervised using the same dataset. This dataset contains six different waveforms, including two sine waves with different periods, absolute sine, triangle, sawtooth, and pulse. The networks are tasked with detecting the form of the wave when provided with 35 input samples.

Figure 33 presents an excerpt of all signals in the dataset. The Python script generating the dataset can be found in the appendix under *e5aisuite-testbench/scripts/models/timeseries-datagen.py*.

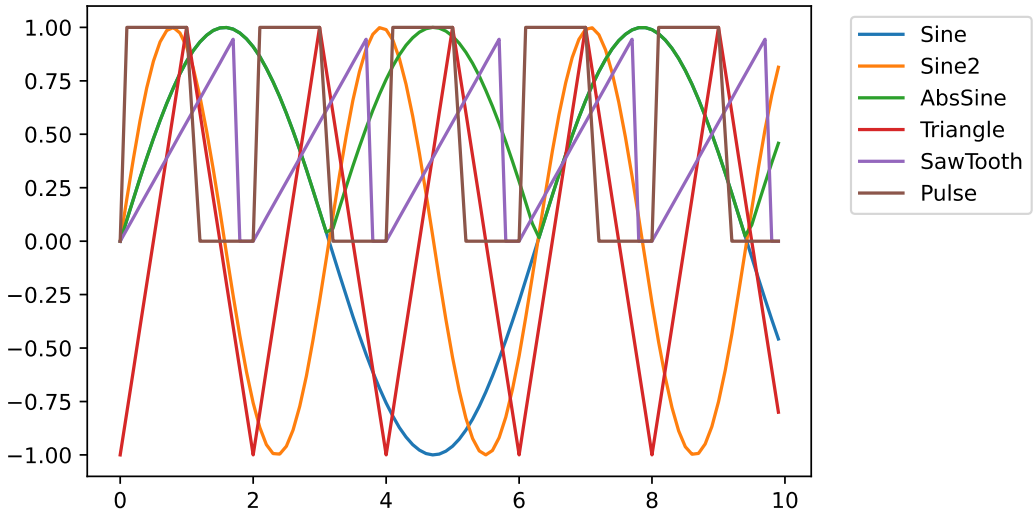

Figure 33: Signals within the timeseries dataset

Models

Overall ten models, differing in their architecture, are implemented, trained and exported to Emmi for the use as integration test. An overview of all implemented models and the layers utilized within the models is presented in Table 9.

Table 9: Overview of models used in integration tests

Model Name	**Used Layers**
network-timeseries-conv1D-category-concat	Conv1d, Concatenate, Flatten, Dense
network-timeseries-conv1D-category-fun	Conv1d, Flatten, Dense
network-timeseries-conv1D-category-fun-bn	Conv1d, BatchNorm, Flatten, Dense
network-timeseries-conv1D-category	Conv1d, Flatten, Dense
network-timeseries-conv1D-category-gap	Conv1d, MaxPool1d, GlobalAvgPool1d, Flatten, Dense
network-timeseries-conv1D-category-gmp	Conv1d, MaxPool1d, GlobalMaxPool1d, Flatten, Dense
network-timeseries-conv1D-category-maxpool	Conv1d, MaxPool1d, Flatten, Dense
network-timeseries-conv1D-category-maxpool-causal	Conv1d, MaxPool1d, Flatten, Dense
network-timeseries-conv1D-category-avgpool	Conv1d, AvgPool1d, Flatten, Dense
network-timeseries-depthwiseconv1d-category	DepthwiseConv1d. Conv1d, Flatten, Dense

The TensorFlow Keras implementation of all models listed in Table 9 can be found in the appendix in *e5aisuite-testbench\scripts\models*.

Test Cases

For each model, fifteen inferences are run and compared to the reference data. To export the reference data to the Emmi Testbench a Python script is written.

The Python script can be found in the appendix in *e5aisuite-testbench/scripts/models/timeseries-cexport.py*.

5.3 Benchmarks

5.3.1 Network Size and Execution Speed

To compare network size and execution speed when using different data types for weights and biases, a LeNet-5 is quantized and deployed to the EMSA5. The network size is read out of the EmmiTranslator, and the performance is measured by counting the number of clock cycles elapsed within a single prediction.

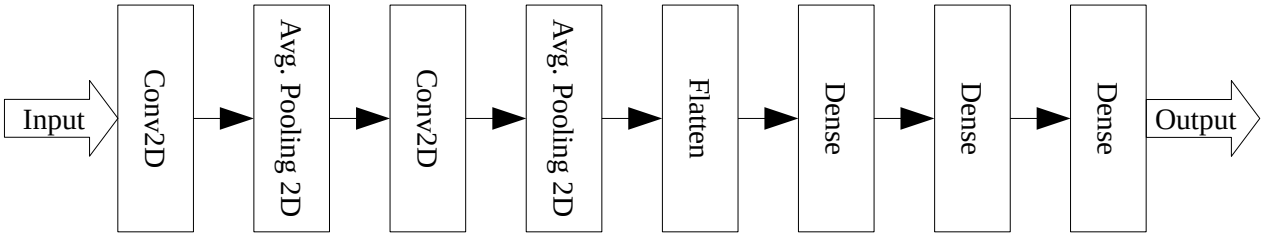

Figure 34: Layers of a LeNet-5

Table 10 shows the results. The speedup from int32_t to int8_t is 5%, while the reduction in required memory is 65%. Note that the size reduction from the model stored in 32-bit values to a model stored in 8-bit values is not a factor of four. This is because memory is also allocated for the internal feature-maps, which are always 32-bit.

Table 10: Network size and execution speed when using different datatypes for weights and biases

Quantization	Datatype	Size (Kilo Bytes)	Size (%)	Cycles	Speedup
None / Float	float	284.46KB	100%	296399159	0.12
8-bit	int32_t	284.46KB	100%	34284604	1.0
8-bit	int16_t	161.04KB	57%	33231619	1.03
8-bit	int8_t	99.32KB	35%	32659232	1.05

The example containing the code for the execution of the benchmark can be found in the appendix under *demos/emmi-LeNet5-demo*.

5.3.2 Batch Normalization Folding

The effect of batch normalization folding on the performance is tested using the model displayed in Figure 35. When activating batch normalization folding within the EmmiTranslator, the two Batch Normalization layers are folded in forward direction.

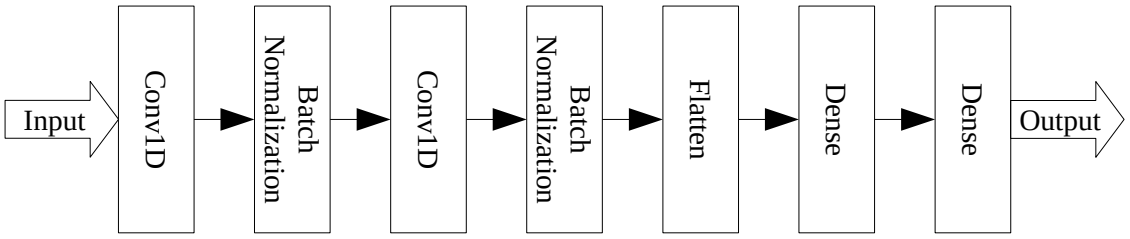

Figure 35: Layers of the model network-timeseries-conv1D-category-fun-bn

Figure 36 compares the models performance when using soft-float on the EMSA5. It is visible that the performance when using batch normalization folding increases by 22%.

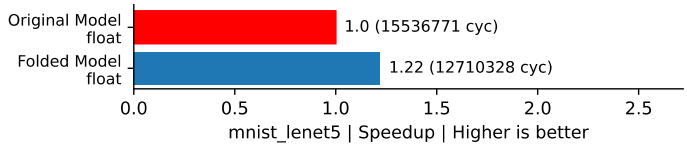

Figure 36: Speedup of a model using batch normalization folding (soft-float)

Figure 37 compares the models performances when using 8-bit quantization. When folding the models batch normalization layers, the performance increases by 33%.

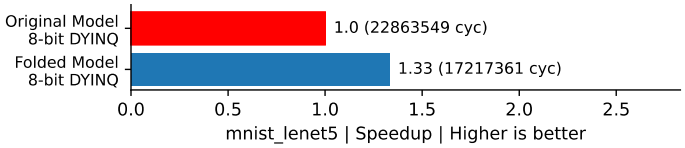

Figure 37: Speedup of a model using batch normalization folding (8-bit DYINQ)

The example containing the code for the execution of the benchmark can be found in the appendix under *demos/emmi-bn-fold-demo*.

5.3.3 MLPerf Tiny Image Classification

To evaluate the accuracy of a network using different internal feature map bit-widths, the MLPerf Tiny image classification benchmark is chosen. The goal of MLPerf Tiny is to "provide a representative set of deep neural networks and benchmarking code to compare performance between embedded devices" [49]. MLPerf Tiny can be used to evaluate devices in terms of accuracy, performance and efficiency. As the EMSA5 is implemented as an IP core on an FPGA board, only the accuracy benchmark is run, as all devices listed in the MLPerf Tiny database are in silicon. [50]

Table 11 lists the results of the first 1000 samples of the MLPerf Tiny Image Classification Benchmark, executed in Emmi on the EMSA5. It is visible that the accuracy increases with the bit-width. It also reaches the 85% required to participate in the MLPerf Tiny Image Classification Benchmark.

Table 11: Accuracy of different feature map bit-widths when running the MLPerf Tiny Image Classification benchmark in Emmi

Quantization bit-width	Feature map bit-width	Accuracy
8-bit	13-bit	88.1%
8-bit	12-bit	87.0%
8-bit	10-bit	86.2%
8-bit	8-bit	78.3%

5.4 Vectorized Testbench in Spike

To execute all unit and integration tests with the RISC-V Vector Extension Zve32x, a compiler with support for automatic vectorization and an ISA simulator with support for vector instructions is required. As compiler GCC14 is used, and as simulator Spike. Since Spike is only compatible with Linux platforms, both must be build for Linux.

5.4.1 Setup

GCC with Auto-Vector Support

At first the GNU C Compiler 14 is build on the destination platform. The build is configured for ilp32 (no hardware floating-point unit) and the use of multilib, which generates two different versions of the C standard library: One for platforms without vector support, and one for platforms with vector support.

```
git clone https://github.com/riscv-collab/riscv-gnu-toolchain.git –recursive
cd riscv-gnu-toolchain
./configure --prefix=/opt/riscv-gnu-toolchain-14 --with-arch=rv32gc --with-
abi=ilp32 –with-multilib-generator="rv32imc-ilp32--;rv32gcv-ilp32--"
make
```

Spike

The build process of spike, the RISC-V simulator, is uncomplicated. After cloning the sources from GitHub, the prefix of the installed RISC-V compiler is set and the build process is started. When running ‘make install’, the Spike binaries are copied into the directory of the RISC-V toolchain.

```
git clone https://github.com/riscv-software-src/riscv-isa-sim.git --recursive
cd riscv-isa-sim
```

```
mkdir build
cd build
../configure --prefix=/opt/riscv-gnu-toolchain-14
make
make install
```

PK

PK is a proxy kernel for spike. It allows the use of I/O functionality within spike by proxying I/O system related calls to the host computer. The build of PK must be compatible with the newlib built during the RISC-V GCC build. As the Emmi testbench is to be run with the vector extension, PK is configured for the vector extension. [51]

```
git clone https://github.com/riscv-software-src/riscv-pk.git --recursive
cd riscv-pk
mkdir build
cd build
../configure --prefix=/opt/riscv-gnu-toolchain-14 --host=riscv32-unknown-elf –with-
arch=rv32imcv_zicsr_zifencei
```

Because PK is built with vector extension and -O2, loops accessing I/O components may be vectorized, causing the system to fail. To prevent this, the compiler optimization is set to -O0 by manually editing the generated makefile. Then PK is built and installed.

```
make
make install
```

5.4.2 Running Testbench

To run the vectorized testbench, the testbench is compiled and afterwards spike is started with the options:

```
spike --isa=rv32imfcv_zicsr_zifencei --varch=vlen:128,elen:32 $RISCV/riscv32-
unknown-elf/bin/pk build/riscv32-unknown-elf-gcc_spike/emmi-testbench.elf
```

This will enable the full set of RISC-V vector extensions (Zve32x as standalone is not supported by Spike), set the size of the vector registers, enable the proxy kernel and load the compiled binary, which is run immediately after spike is started.

6 Embedded AI Application

This chapter presents an embedded AI application developed at the Fraunhofer IPMS. It introduces the scenario and the hardware used, as well as the tasks performed within the scope of this thesis.

6.1 Introduction

Conveyors are automated mechanical devices, consisting of belts stretched over pulleys. They facilitate the swift and efficient movement of goods and materials. They are versatile, capable of transporting everything from small items to large, heavy objects, and are essential in sectors ranging from manufacturing to retail.

For an optimal performance, the belt of the conveyor needs to beat an optimal tension. To determine if the belt of a conveyor must be tightened or loosened to reach the optimal tension, an accelerometer to measure lateral movements, an gyrometer the measure to measure angular movements and a magnetometer to measure differences in the magnetic field caused by the engine are used. An Embedded AI application should analyze the sensor data, and predict if the belt is too loose or too tight.

6.2 State of the Art

6.2.1 Conveyor

For the project a miniature conveyor is provided by a project partner. It has a length of approximately 20 centimeters and has the sensors already mounted. The magnetometer is mounted directly beside the engine, while the accelerometer and the gyrometer are both mounted beside the screw used to adjust the tension. Figure 38 displays a sketch of the setup.

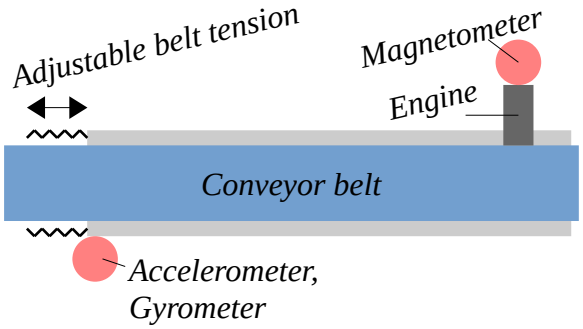

Figure 38: Sketch of the hardware setup

6.2.2 Dataset

The provided dataset contains six categories and 10000 elements per category. Category ‘0’ means loose, category ‘5’ means tight as shown in Figure 39. The optimal tension of the conveyor is reached in category ‘3’. Each entry contains 333 samples of all three sensors, in all their axes. Meaning for a single entry in the dataset, 2997 sensor values are provided.

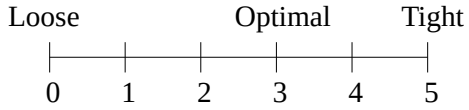

Figure 39: Overview of categories within the conveyor dataset

6.2.3 Task

The hardware including all the sensors, the embedded software to read out the sensors, and a dataset to train a neural network is already provided by project partners. The task is to design, train, convert and use a neural network to predict the tension of the belt on an embedded system using a minimalist version of the EMSA5, employing only 64KB ITCM, 64KB DTCM and support for RISC-V imc.

6.3 Embedded AI Workflow

In order bring a model to the EMSA5, the Embedded AI Workflow, described in section 2.3.1 was used. During the process, multiple models, with different architectures were designed until a model, executable on the minimalists version of the EMSA5, was found. The model (ID 3 in Table 12) was then converted using the EmmiTranslator and tested on the EMSA5.

Table 12: Overview of conveyor models

ID	Inputs	Samples per Input	Parameters	Float Accuracy	Quantized Size	Inference Time on EMSA5
1	Acc, Gyr, Mag	333	231396	98%	382 KB	33 seconds*
2	Acc, Gyr, Mag	100	74988	94%	136 KB	7 seconds*
3	Acc, Mag	100	18701	91%	26 KB	2 seconds
*Inference time measured on EMSA5 with 256MB memory						

For each model, a graphic displaying the detailed architecture can be found in the appendix under *models/**.

6.3.1 Test model using Live Data

The model is tested using live data recorded directly from the sensors connected to the EMSA5. After 100 samples from the accelerometer and the magnetometer are recorded, the data is preprocessed by scaling it into a range between zero and one. Subsequent to the preprocessing, the inference is performed.

6.3.2 Adding support for a Display

The final step is to add support for a display. The interface should show a loose or tight conveyor belt, depending on the current classification of the measurements. A seven inch NEXTION NX8048P070 is chosen as display. The interface shown by the display is controlled by sending UART commands. The interface itself is designed in the software provided by NEXTION. The interface is then stored on the controller of the display. Figure 40 shows the structure of the embedded AI application. The FPGA board that performs the inference is mounted under the display. The FPGA board on the right is placed for demonstration purposes and is not connected to the conveyor.

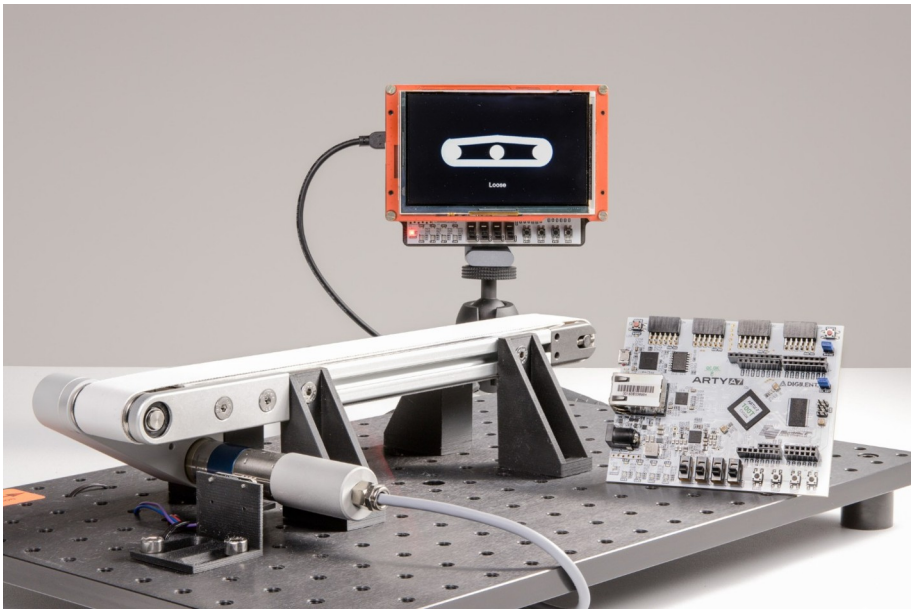

Figure 40: Demo setup of the conveyor application [52]

7 Conclusion

7.1 Summary

The aim of this master's thesis was to extend the functionality of the inference-only embedded AI framework Emmi. The foundational groundwork for this extension was laid in section two, which provided an overview of the EMSA5, Emmi, quantized neural networks and delved into the theory required to implement new functionality.

The following section three analyzed the architecture and feature set of the embedded AI framework Emmi. It stated opportunities for the frameworks improvement and presented an overview of the functions planned for development during the course of the thesis. Section four continued with the implementation of the convolutional and pooling operations, followed by the implementation of the a batch normalization layer and the integration of batch normalization folding into the EmmiTranslator. The section was closed by describing general improvements made to the framework in order to reduce the storage requirements of models, the build system, and the accuracy of quantized neural networks. These implemented functionalities were evaluated in section five. An overview of the implemented unit and integration tests were given and benchmarks on network size, accuracy and performance were executed. Section six presented an embedded AI application example: It predicts the tension of a belt.

7.2 Research Questions

Four research questions were stated in section 1.1.1. Question Q1, asking for the implementation of a batch normalization was answered in section 4.3. The question of why a folding of batch-normaization is considered was answered is 3.3.1, and details of the implementation were given in 4.3.3. Questions Q2 and Q3 were both explored though discussions in section 3.2.4, the resulting implementations are described in section 4.4. Question Q4 led to a RISC-V simulator called spike, which was used to run the Emmi testbench as described in 5.4.

In addition to the research questions, section 1.1.1 listed secondary points, two of which were fully investigated. S1 was addressed in section 6 and S2 in sections 2.5.1 and 4.1.2.

7.3 Final Outcome

Throughout this thesis, all research questions have been resolved, and the necessary extensions have been successfully implemented. In addition, two of the three secondary points were addressed, and several optimizations were made to the framework.

Significant improvements have been achieved in the Emmi Core. It now supports 1D convolutional layers, batch normalization, concatenation and smaller datatypes, effectively reducing the memory requirements of quantized models. The EmmiTranslator now supports the folding of batch normalization layers through the integration of the Python package tf-batchnorm-fold, which has been enhanced to work with the full feature-set of Emmi.

Finally, a practical example of embedded AI was presented to demonstrate the real-world applicability of Emmi.

7.4 Perspective

Emmi offers exciting possibilities in the world of embedded AI projects. It has already been used in two projects at the Fraunhofer IPMS, leveraging its wide range of functionality. With the upcoming support for Long short-term memory (LSTM) networks alongside a dedicated user interface for model conversion, the usability and feature set of the framework will be further extended.

Appendix

The appendix of this thesis is supplied as a digital archive, that includes:

- Demos
- Model illustrations
- Online references
- Source Code

References

[1] Tech Trends 2023, Deloitte, [online]. Available: https://www2.deloitte.com/content/dam/insights/articles/us175897_tech-trends-2023/DI_tech-trends-2023.pdf accessed: 2024-01-10

[2] Karen Hao, Here are 10 ways AI could help fight climate change, MIT Technology Review, [online]. Available: https://www.technologyreview.com/2019/06/20/134864/ai-climate-change-machine-learning/ accessed: 2024-01-10

[3] What is predictive maintenance?, IBM Services, [online]. Available: https://www.ibm.com/topics/predictive-maintenance accessed: 2024-01-10

[4] Rich Collins, How the RISC-V ISA Offers Greater Design Freedom and Flexibility, ChipEstimate, [online]. Available: https://www.chipestimate.com/How-the-RISC-V-ISA-Offers-Greater-Design-Freedom-and-Flexibility/Synopsys/Technical-Article/2024/03/05 accessed: 2024-04-18

[5] TensorFlow Lite for Microcontrollers, TensorFlow, [online]. Available: https://www.tensorflow.org/lite/microcontrollers accessed: 2024-01-10

[6] AIfES for Arduino®, README.md AIfES_for_Arduino, Fraunhofer IMS, [online]. Available: https://github.com/Fraunhofer-IMS/AIfES_for_Arduino accessed: 2024-01-10

[7] Pascal Pfeiffer, Development of a Machine Learning Framework for Quantized Neural Networks on Embedded RISC-V Systems, 2023

[8] RISC-V “V” Vector Extension Version 1.0, RISC-V International, [online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf accessed: 2024-03-11

[9] Sarah L. Harris, David Money Harris, Digital Design and Computer Architecture RISC-V Edition, Morgan Kaufmann, 2022

[10] RISC-V International, RISC-V International Members, [online]. Available: https://riscv.org/members/ accessed: 2024-04-17

[11] Andrew Waterman, Krste Asanovic, The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213, RISC-V International, Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf accessed: 2024-04-18

[12] RISC-V “V” Vector Extension Version 1.0, RISC-V International, Available: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf accessed: 2024-04-18

[13] TinyML Foundation, Homepage, [online]. Available: https://www.tinyml.org/ accessed: 2024-03-18

[14] H. -Y. Lin, Embedded Artificial Intelligence: Intelligence on Devices, Computer, vol. 56, no. 9, pp. 90-93, 2023, doi: 10.1109/MC.2023.3280397

[15] Getting started with X-CUBE-AI Expansion Package for Artificial Intelligence (AI), STMicroelectronics, UM2526, Rev 8, 2022

[16] Pascal Pfeiffer, Fraunhofer IPMS, Internal Resoruce

[17] Hideaki Abe, Embedded AI-Accelerator DRP-AI, Renesas Electronics Corporation, 2021

[18] Saheli Bhattacharjee, The MLPerf™ Tiny benchmark: Reproducing v1.0 results and providing new results for v1.1, Krai Ltd, 2023

[19] Let’s make your AI tiny and fast, Plumber AI, [online]. Available: https://plumerai.com/benchmark accessed: 2024-03-11

[20] Keras layers API, Keras API reference, [online]. Available: https://keras.io/api/layers/ accessed: 2024-03-19

[21] Francois Chollet, Deep Learning with Python, Second Edition, Manning Publications, 2021

[22] Jörg Frochte, Maschinelles Lernen, HANSER, 2019

[23] tf.keras.layers.Conv1D, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv1D accessed: 2024-03-30

[24] tf.keras.layers.DepthwiseConv1D, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/DepthwiseConv1D accessed: 2024-03-30

[25] tf.keras.layers.AveragePooling1D, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/AveragePooling1D accessed: 2024-03-30

[26] tf.keras.layers.MaxPool1D, TensorFlow [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool1D accessed: 2024-03-30

[27] tf.keras.layers.GlobalAveragePooling1D, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalAveragePooling1D accessed: 2024-03-30

[28] tf.keras.layers.GlobalMaxPool1D, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalMaxPool1D accessed: 2024-03-30

[29] tf.keras.layers.GlobalAveragePooling2D, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalAveragePooling2D accessed: 2024-03-30

[30]tf.keras.layers.GlobalMaxPool2D, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalMaxPool2D accessed: 2024-03-30
[31]Understanding masking & padding, TensorFlow, [online]. Available: https://www.tensorflow.org/guide/keras/understanding_masking_and_padding accessed: 2024-03-30
[32]tf.keras.layers.Concatenate, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Concatenate accessed: 2024-03-30
[33]tf.keras.layers.BatchNormalization, TensorFlow, [online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization accessed: 2024-03-30
[34]Sergey Ioffe, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Google Inc, 2015, [online]. Available: https://arxiv.org/abs/1502.03167 accessed: 2024-02-13
[35]Benoit Jacob, Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference, Google Inc., 2017, [online]. Available: https://arxiv.org/abs/1712.05877 accessed: 2024-02-13
[36]Edouard Yvinec, To Fold or Not to Fold: a Necessary and Sufficient Condition on Batch-Normalization Layers Folding, Sorbonne Université | CNRS, 2022, [online]. Available: https://arxiv.org/abs/2203.14646 accessed: 2024-02-13
[37]Edouard Yvinec, Tensorflow-batchnorm-folding 1.08, PyPI, [online]. Available: https://pypi.org/project/tensorflow-batchnorm-folding/1.0.8/ accessed: 2024-02-13
[38]Zhewei Yao, HAWQ-V3: Dyadic Neural Network Quantization, Proceedings of the 38 th International Conference on Machine Learning, PMLR 139, 2021, [online]. Available: https://arxiv.org/abs/2011.10680 accessed: 2024-04-09
[39]Quantization aware training, TensorFlow, [online]. Available: https://www.tensorflow.org/model_optimization/guide/quantization/training accessed: 2024-04-05
[40]Sungrae Kim, Zero-Centered Fixed-Point Quantization With Iterative Retraining for Deep Convolutional Neural Network-Based Object Detectors, IEEE Access, 2021
[41]Post-training quantization, TensorFlow, [online]. Available: https://www.tensorflow.org/lite/performance/post_training_quantization accessed: 2024-06-22
[42]Quantization. PyTorch, [online]. Available: https://pytorch.org/docs/stable/quantization.html accessed: 2024-06-22
[43]QPyTorch Functionality Overview, QPyTorch, [online]. Avaialbe: https://qpytorch.readthedocs.io/en/latest/examples/tutorial/Functionality_Overview.html accessed: 2024-06-22
[44]Documenting the code, Doxygen Developers, [online]. Available: https://www.doxygen.nl/manual/docblocks.html accessed: 2024-05-22
[45]PEP 257 – Docstring Conventions, Python Enhancement Proposals, [online]. Available: https://peps.python.org/pep-0257/ accessed: 2024-05-22
[46]Auto-vectorization in GCC, Free Software Foundation, [online]. Available: https://gcc.gnu.org/projects/tree-ssa/vectorization.html accessed: 2024-06-20
[47]PEP 621 – Storing project metadta in pyproject.toml, Python Enhancement Proposals, [online]. Available: https://peps.python.org/pep-0621/ accessed: 2024-05-07
[48]Pascal Pfeiffer, Update TF and fold Conv1D layers, [online]. Available: https://gitlab.com/edouardyvinec/batch-normalization-folding/-/merge_requests/4 accessed: 2024-05-08
[49]MLPerf™ Tiny Deep Learning Benchmarks for Embedded Devices, README.md, ML Commons, [online]. Available: https://github.com/mlcommons/tiny accessed: 2024-06-10
[50]MLPerf Inference: Tiny Benchmark Suite Results, ML Commons, [online]. Available: https://mlcommons.org/benchmarks/inference-tiny/ accessed: 2024-06-10
[51]RISC-V Proxy Kernel and Bootloader, README.md, RISC-V International, [online]. Available: https://github.com/riscv-software-src/riscv-pk accessed: 2024-06-05
[52]Sebastian Lassak, Predictive Maintenance Demonstrator for Industrial Equipment, Fraunhofer IPMS, 2024

-last page-