Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

On-Device Classification for the Geophone Dual Processor Platform

Semester Project

Francesco Spadafora
fspadafo@student.ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Matthias Meyer
Dr. Jan Beutel
Prof. Dr. Lothar Thiele

04.03.2019

Abstract

We present an implementation of a Convolutional Neural Network (CNN) for pipelined acoustic event classification. This system is cross-platform compatible, whereas the goal of this project is a future usage on the Geophone Dual Processor Platform (GDPP). On the GDPP it is necessary to make continuous predictions for longer seismic events. The benefits of a continuous classification on the edge node rather than the cloud are e.g. diminished data transmission costs, lower latency and reduced network congestion. For a longer event, it is crucial to classify it in real-time in order to trigger respective actions. We based our findings on the previous Work "Quantized Convolutional Neural Network for Embedded Platforms" [4], whose implementation was ported to a Free Real-Time Operating System (FreeRTOS) [6], modified for continuous classification and tested on a test environment. The test results confirm the functionality of continuous classification.

Acknowledgements

I would like to express my special appreciation and thanks to my supervisor Matthias Meyer for his exceptional support during this project. I am very grateful for the countless meeting and his tireless support, even when we were running out of time while still a lot of work had to be done. Under his supervision, I was able to challenge myself to improve my knowledge about machine learning and microcontroller programming. I would also like to thank Dr. Jan Beutel and Prof. Lothar Thiele for the opportunity of this project. A special thanks to Reto Da Forno for providing a framework for FreeRTOS.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 20

CNN Convolutional Neural Network. i, vii, 1–3, 6, 7, 10, 13, 16, 17, 20, 21, 35

CPU Central Processing Unit. 16, 17

DSP Digital Signal Processing. 17

FFT Fast Fourier Transformation. v, 3, 6, 11, 12, 24, 33

FPU Floating Point Unit. 17

FreeRTOS Free Real Time Operating System. i, ii, 17, 18, 29, 34, 39

GDPP Geophone Dual Processor Platform. i, 16, 34, 35

INQ Incremental Network Quantization. 7

MCU Microcontroller Unit. vi, 1, 2, 6, 16–21, 23, 24, 27–29, 32–35, 39

ReLU Rectifiying Linear Unit. 3, 19

RISC Reduced Instruction Set Computer. 16

SRAM Static Random Access Memory. 17

tdp Time-Distributed Processing. 1, 7, 13, 16, 24, 28, 34, 35

TF Tensorflow. vi, 20, 22, 23

UART Universal Asynchronous Receiver-Transmitter. 27, 39

WSN Wireless Sensor Network. vi, 1, 2, 6, 16, 17, 35

Chapter 1

Introduction

1.1 Context

In August 2017 multiple avalanches hit the small village of Bondo (GR) [17]. It was a tragedy with eight fatalities, all of them mountaineers. Even with a detection system that triggered an alarm,this tragedy was not avoided. This case example shows that further improvement is needed in order to protect humans from natural disasters.

The approach followed by the Networked Embedded Systems Group at ETH Zürich is to make detection systems smarter. The technology goes towards edge computing, with challenges in reliability and scalability. In order to physically detect acoustic events, Akos Pasztor in his Master Thesis proposed an event-based microseismic sensing platform with co-detection [10]. In the PermaSense Project [18], those platforms were used on the Matterhorn-Hörnligrat and Dirruhorn Rock Glacier and form a Wireless Sensor Network (WSN). In order to use the WSN for avoidance of natural disasters, Timo Farei-Campagna proposed in his Master Thesis "Quantized Convolutional Neural Networks For Embedded Systems" [4] a Convolutional Neural Network (CNN) able to classify acoustic events with a Time-Distributed Processing (tdp). Nevertheless, the proposal comes with different drawbacks. It does not allow continuous classification in real-time and the implementation is hardcoded to one specific Microcontroller Unit (MCU). In its actual form, it would not be suitable to run on a warning system in real-time.

1.2 Contribution and Organization

In this project, we push forward on the idea of a real-time alarm system. Our contribution is the proposal of a system able to continuously detect mountaineers, not hardcoded to a specific platform. A continuous classification requires the pipelining of the classification over the CNN. To reach the pipelining, the core challenge is the removal of the zero-padding between the classification of acoustic events. By removing the zero-padding, the influence of the intermediate buffers inside the tdp must be reevaluated. As a consequence, weights and bias of the kernel for the CNN must be retrained. This represents a new challenge on how to train the weights, such that the new receptive field is considered. In a final step, the accuracy of the implementation on a MCU must be verified. Ideally, the testing platform uses the same MCU as the one of the WSN on the Matterhorn-Hörnligrat and Dirruhorn Rock Glacier.

This documentation is going to explain how the introduced challenges were approached and verified. Chapter 2 will give an overview on the used theoretical concepts, such as tdp, CNN, pipelining and receptive field. Chapter 3.3 explains how the original implementation becomes cross-platform compatible. Chapter 4.3 will explain the influence of zero-padding on the receptive field, whereas chapter 4.4 explains the removal of it. In chapter 4.5, the methods used to pipeline the classification are explained, which then are verified in chapter 6.1. The accuracy testing and verification can be found in chapter 4.1 and 6.2. A final conclusion is given in chapter 7.

Chapter 2

Theory

This chapter will introduce the most important technologies, concepts and related work on which the project is based.

2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) is a class of deep neural networks. They mainly consist of convolution layers, but can include connected layers or other variations. Each layer consists of neurons, whereas a neuron consists of inputs weights and an activation function. The neuron translates these inputs into a single output, which can then be picked up as input for another layer of neurons later on. Compared to a fully connected layer, where each neuron is connected to all preceding neurons, neurons in convolution layers are only connected to subsets. A kernel (contains all weights and bias of the neurons) is convoluted over this subset. The result is passed as input into an activation function, which adds non-linearity to the system, such as e.g. Rectifying Linear Unit (ReLU). The set of neurons affecting a neuron in the subsequent layer is called the receptive field . From [9], we know that CNN's gained a lot of interest due to their advanced feature extraction and iterative optimization procedure. [9] described the kernel, weights, as well as the bias for each layer as trainable parameters, evaluated by a loss-function (e.g. cross-entropy for multiple output).

In [4], acoustic event are analyzed by first using aFast Fourier Transformation (FFT) and creating dependencies in time and frequency, also called a spectrogram. With CNN resulted advantageous in analyzing those time- and frequency dependencies. This is due to the fact that kernels are only connected to a subset of the neurons inside the previous layer, making them attractive for the analysis of spacial dependencies.

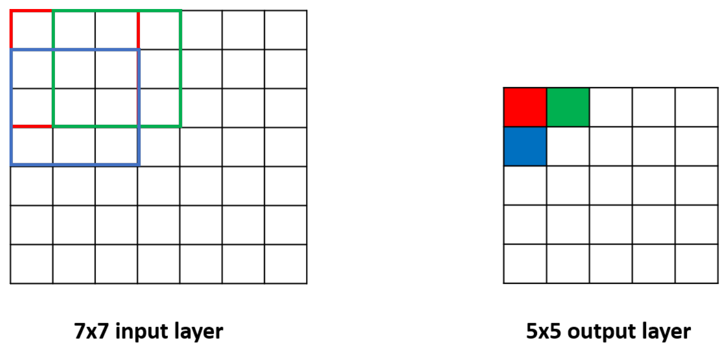

Figure 2.1: Two-dimensional example of a 3x3 convolution with a stride of 1.The output layer decreases to 5x5.

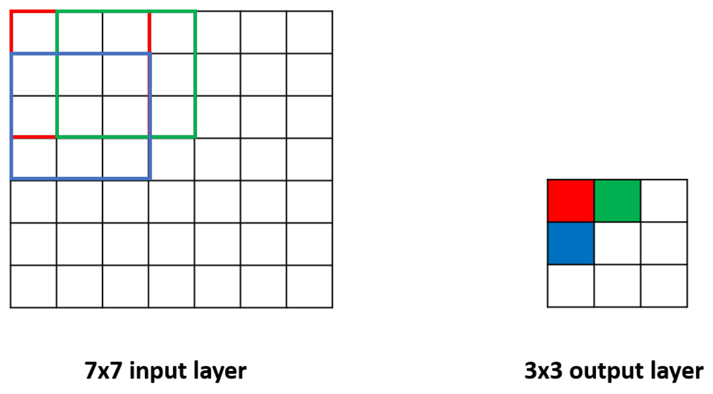

Figure 2.2: Two-dimensional example of a 3x3 convolution with a stride of 2. The output layer decreases to 3x3.

2.1.1 Zero-Padding

The kernels are convoluted over all input layer, resulting in a feature map. In a first case, one input layer with a 7x7 input size, convoluted with a 3x3 kernel, should be considered. 2.1 shows the process graphically. The kernel is convoluted over the whole input layer, resulting in an output layer of 5x5. In this case the stride is one. A stride is the amount by which the kernel is shifted. The output layer gets even smaller with a stride greater than one. Figure 2.2 shows how the original input layer of size 7x7 is reduced to a output layer of 3x3 because of the stride of two.

During a feedforward process, the output layer is reduced in size at every step. It is assumed that with the reduction in size also information gets lost. Especially the early layers of the network should preserve as much information as possible. To avoid those reductions, zero-padding is used. Figure 2.3 shows the idea behind zero-padding. The input layer is padded on the borders with

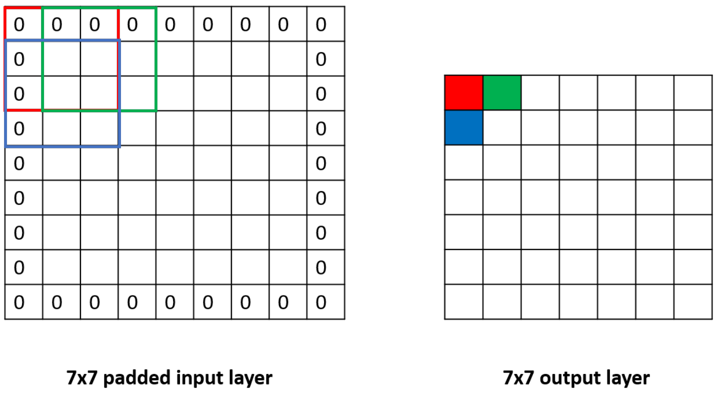

Figure 2.3: Two-dimensional example of a 3x3 convolution with zero-padding and a stride of 2. The output layer does not change in size.

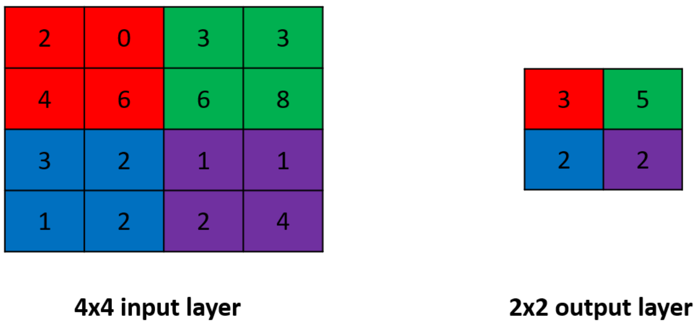

Figure 2.4: Example of an average pooling with stride of 2 and a kernel of 2x2. The size of the output layer decreases and the content is the average value.

zeros. However, it can be chosen to just pad zeros on one axis as well, depending what dimension should remain constant in size.

2.1.2 Average Pooling

In contrast to the zero-padding, the purpose of pooling layers is to reduce the layer sizes. In figure 2.4 an example of an average pooling layer with an input layer of size 4x4 and a stride of 2 is shown. As for convolutions, a kernel is slided over the input layer. But instead of computing convolutions, the average of all values inside of it is computed. In that way, information can be compressed and the computation complexity can be reduced. Another advantage of those layers are the control of overfitting, s.t. a good generalization of the problem can be achieved. Nevertheless, with reduction in size information is lost.

2.2 Edge Computing

In the actual implementation of the WSN on the Matterhorn-Hörnligrat and Dirruhorn Rock Galcier, the acquired data is collected from different actors. [4] made a first approach to shift the classification onto the nodes of the WSN. This project aims to further increase those detection capabilities. The term of pushing intelligence to the node is called Edge Computing. [11] defines Edge Computing as the deployment of data-handling activities or other network operations away from centralized and always-connected network segments, toward individual sources of data capture. Pushing signal and data processing towards the edge of a network can bring different advantages [14].

- Sensor fusion: Different off the shelf sensors can be combined to one synthetic sensor. Such systems could detect more complex events.
- Bandwidth: Neural Nets could be partitioned such that some layers are evaluated on the device and the rest in the cloud. With this approach, workload and latency could be balanced. To additionally reduce the payload size, additional feature selection could be applied in order to only transmit the necessary information.
- Transfer learning: The practice of modifying different parts of the network to perform different tasks is called transfer learning. If the neural network is splitted between the edge end the cloud, the MCU could be in charge of the main feature extraction. By changing the layers on the cloud, the same preprocessing could be used for different tasks. This approach would increase scalability and maintainability.

The biggest challenge of of shifting complexity to the edge, is how to fit those complexity on a MCU with different restrictions. In [4], the question was how to fit large neural networks on a MCU. The biggest restrictions are given by the size of the weights and network footprint, the energy consumption and the operating frequency. The solution approached of [4] are given in the next section.

2.3 CNN Architecture for classification

Given acoustic events, the application should be able to classify if a mountaineer passed by or not. The CNN for this application is taken from "Event-triggered Natural Hazard Monitoring with Convolutional Neural Networks on the Edge" [7] and shown in table 2.1. Before executing the CNN, the input data is preprocessed with a FFT. This creates a two-dimensional data with a time-axis and a frequency-axis. The two-dimensional data is then used as input of the CNN described in table 2.1. To reduce the footprint of the CNN, it is proposed to

Name	Type	Kernel	Stride	Input size
C0	Conv2D + ReLU	3x3	1	24x61x1
C1	Conv2D + ReLU	3x3	2	24x64x32
D0	Dropout	-	-	12x32x32
C2	Conv2D + ReLU	3x3	1	12x32x32
C3	Conv2D + ReLU	3x3	2	12x32x32
D1	Dropout	-	-	6x16x32
C4	Conv2D + ReLU	3x3	1	6x16x32
C5	Conv2D + ReLU	1x1	1	6x16x32
D2	Dropout	-	-	6x16x32
C6	Conv2D + ReLU	1x1	1	6x16x32
Af	Average (Frequency)	1x16	1	6x16x1
At	Average (Time)	6x1	1	6x1x1
C7	Conv2D + Sigmoid	1x1	1	1x1x1
O	Output	-	-	1

Table 2.1: Original architecture of CNN for classification of a mountaineer from the geodata. This table illustrates the original architecture with zero-padding on the time- and frequency-axis. With permission from [7].

implement it as buffered systems [7]. The concepts underneath the tdp were investigated by [4] and are explained in the next section.

2.4 Buffered implementation of the CNN architecture

The main challenge in [4] was to reduce the overall memory footprint. One method used was the Incremental Network Quantization (INQ). It allowed to reduce the weights of the CNN from the original 32-bit values to 4-bit values without accuracy loss [19]. Because the memory reduction of the INQ was not enough for his target platform, in addition the time-distributed buffering (tdp) was introduced and successfully implemented.

2.4.1 Concepts behind time-distributed processing (tdp)

One advantage of the tdp is the reduction of the utilized memory for calculation. The disadvantage is the increased latency. To explain the tdp strategy, we use examples taken from [4]. First consider the CNN example shown in figure 2.5. On the left side of the figure, a visualization of of the fully inference in case of a CNN with a 3x1 kernel. An alternative approach is shown on the right side of the image. It assumes that the input data is provided over a period of time. It

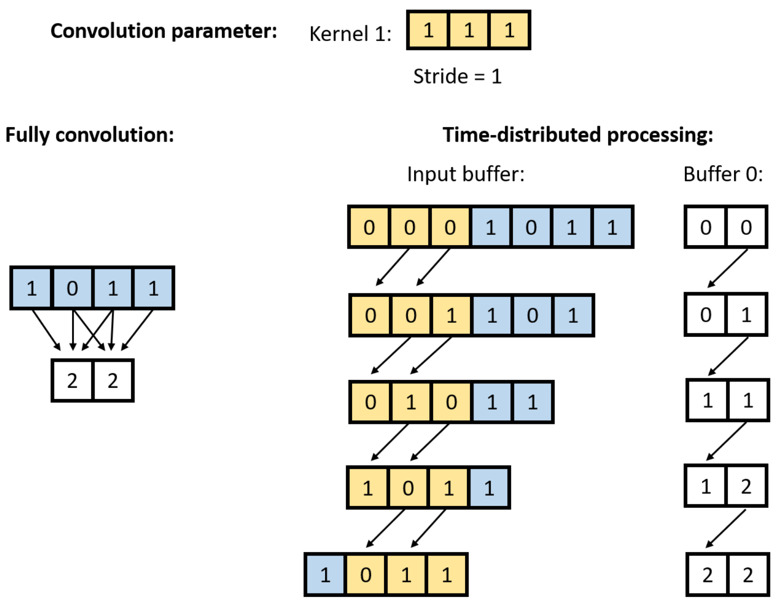

Figure 2.5: On the left, the classical convolution, computed in one step with matrix multiplication. On the right, the buffered system.

would be optimal to start the process already when only a fraction of the data is stored in the memory, instead of waiting for the complete input data to arrive. In the example of figure 2.5, the neuron has a size of three. At least three elements are needed for one step of the forward inference. Starting at the top right of the image, the input buffe, initialized with zeros, waits until the first elements of the input arrives. Then step by step the inference is computed. Those results are stored into the next buffer, in figure 2.5 called Buffer 0 and initialized with zeros as well. The input vector is provided over a certain time, so we can imagine the shifting into the input buffer as a certain update rate. In our case, the input is shifted elementwise through the input buffer. Note that figure 2.5 has a stride of one. The same principle applies for a stride of two of more. Depending on the stride sizes, the inference delay and steps must be adapted.

The number of sifted data inside a buffer depends on the previous layer. The subsequent buffer is shifted by the number of new generated values. This is emphasized 2.6. On the right side, the input buffer is convoluted with a 3x1 kernel, the subsequent Buffer 0 shifted by one and updated accordingly. The new introduced values are called processing window, whereas the shifted values are called buffer.

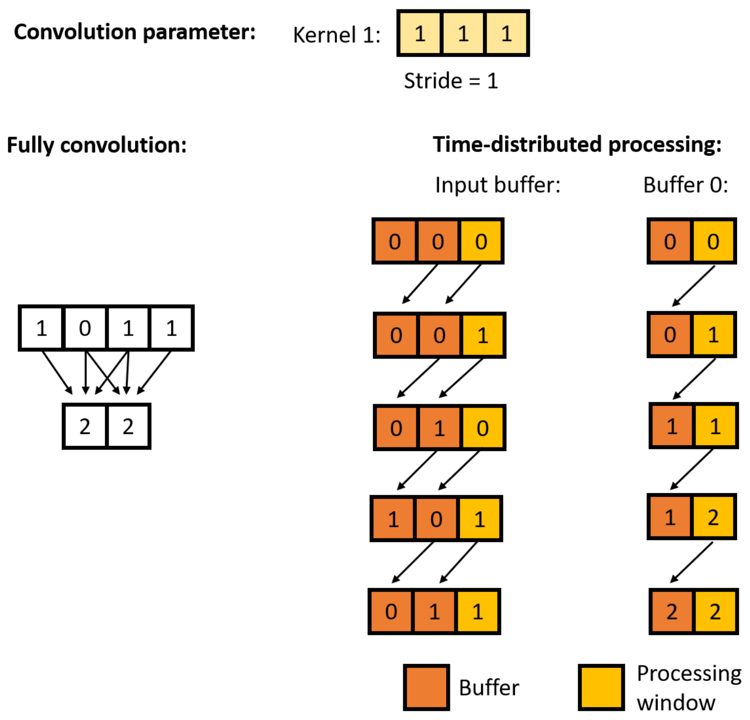

Figure 2.6: An example of a tdp implementation with two layers. The convolution is computed, the subsequent buffer is shifted and updated accordingly.

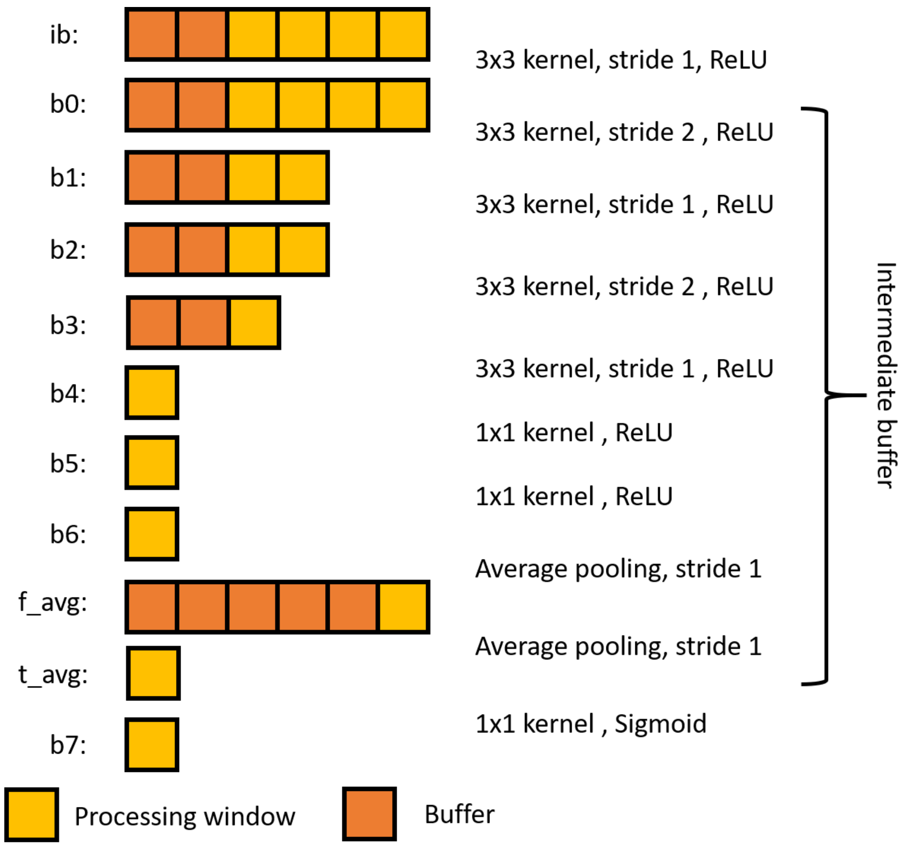
ib:
b0:
b1:
b2:
b3:
b4:
b5:
b6:
f_avg:
t_avg:
b7:
3x3 kernel, stride 1, ReLU
3x3 kernel, stride 2 , ReLU
3x3 kernel, stride 1 , ReLU
3x3 kernel, stride 2 , ReLU
3x3 kernel, stride 1 , ReLU
1x1 kernel , ReLU
1x1 kernel , ReLU
Average pooling, stride 1
Average pooling, stride 1
1x1 kernel , Sigmoid
Intermediate buffer
Processing window
Buffer

Figure 2.7: Time-distributed version of the CNN in table 2.1. For simplicity, the channels are omitted. Each box represents a time point with the associated 64 frequency points.

2.4.2 Time-distributed implementation of CNN for mountaineer detection

The tdp-implementation of the CNN in table 2.1 consists of eleven intermediate buffers. One for the input buffer (ib), eight for the convolution layers (b0 - b7) and two for the average pooling layers (t_avg and f_avg). Figure 2.7 shows the time-distributed version of the CNN in 2.1. For simplicity, this consideration omits the third dimension, which represents the number of channels. Each box represents a time point with the associated 64 frequency points.

When a new frame is forwarded to the pipeline or input buffer, the input buffer ib is shifted by the size of the processing windows. Then four time points (size 64x4) are forwarded into the processing window. The kernel is convoluted with the buffer content and the whole process repeats for every layer. It should be noted that between buffer b6 and f_avg, as well as between f_avg and t_avg, there is an average pooling and not a convolution.

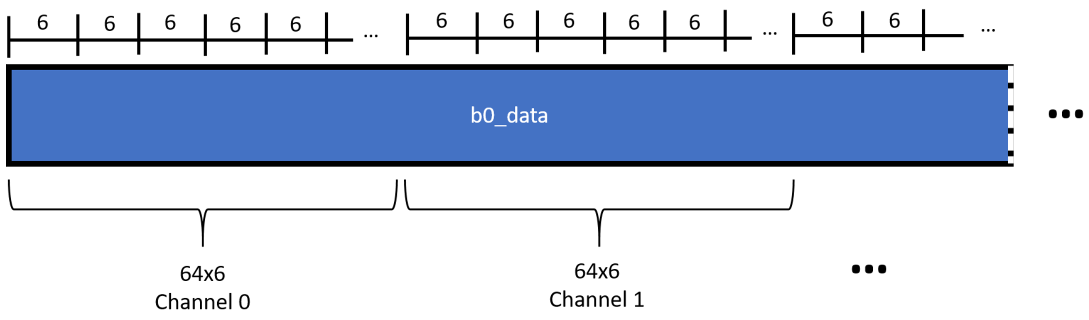

Figure 2.8: Example for the buffer b0. All values are stored inside a one-dimensional array. To access a value, the number of channel, as well as the position on the frequency-axis (max. 64 values) and time-axis (max. 6 values) must be considered.

2.4.3 Storage order in intermediate buffers

In the implementation, all buffers are one-dimensional arrays. Figure 2.8 shows how exactly the data inside the buffer b0 is stored. To represent the time-axis, the last six values are stored. To each point in time, 64 data points on the frequency-axis are assigned. In the one-dimensional array, the frequency information is stored successively. In fact the packets of size six in figure 2.8 represent the time information at a certain frequency. In addition, the number of channels must be considered. In order to read and write from those data, a read- and write-pointer is used. Figure 2.9 shows the case for one block with fixed frequency and channel number. The read-pointer points to the beginning of a time-axis, the following six values are the time information at a given frequency and channel. In contrast, the writing pointer is used to write the new value inside the buffer. This allows to distinguish between the buffer and the processing window. When new data has to be stored, first the last two values at position four and five are shifted into the buffer (orange in figure 2.9). Then starting from the write-pointer, the four new values are saved into the processing window. This is repeated for all frequencies and channels.

2.5 Classification sequence

This section gives an overview on how we get from an acoustic event to a classification. Figure 2.10 shows the whole process graphically. At the beginning, the geophone measures activity. Once a threshold is reached, the acoustic event is transformed with an FFT. The result is a spectrogram with time- and frequency-dependencies. For the following chapters, a spectrogram of size 64x24 will be called a frame. A frame is composed of six subframes of size 64x4. When for-

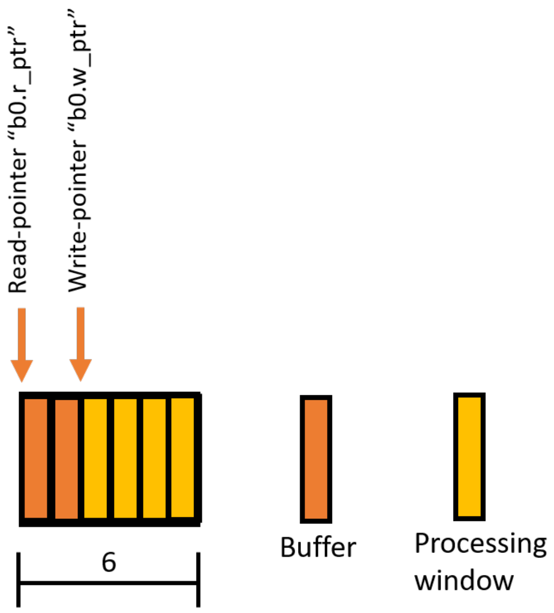

Figure 2.9: One timing block of size six with fixed frequency and channel number. The read- and writing-pointer are used to distinguish between buffer and processing window.

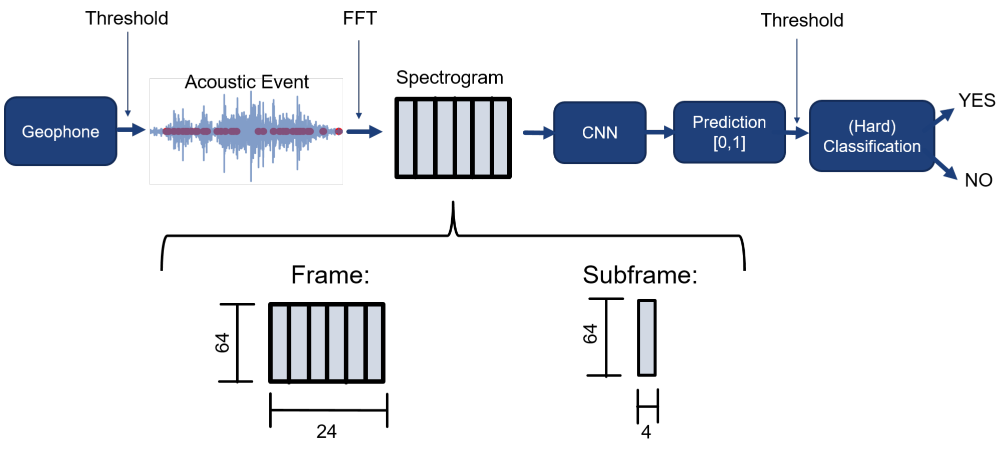

Figure 2.10: Sequence from an acoustic event to a classification. The waveform is transformed with an FFT, forwarded for prediction and classified.

warding a frame to the CNN, they are forwarded subframe by subframe. This will be used for the pipelining in chapter 2.6 and 4.5. After the complete frame was forwarded, the prediction is available. A prediction is between zero and one. Because a clear classification is needed if a mountaineer was detected or not, the next step is a hard classification. A hard classifier takes a threshold and classifies it to either “Mountaineer Detected”or “No Mountaineer Detected”, not considering any uncertainty or edge case.

2.6 Pipelining

To increase the prediction throughput, the concept of pipelining is used for this project. Pipelining is a technique where processing elements can be executed overlapping [16]. The pipeline is divided in stages, and each stage completes a part in parallel. The slowest stage determines the speed. This can be used for instructions, as well as for computing elements. Common characteristics to describe a pipelining are the following [5]:

- **Latency:** Time it takes until a PE us processed.
- **Throughput:** Number of predictions in a span of time.
- **Speed-up:** Determines how much faster it becomes with pipelining.

$$\text{Speed-up} = \frac{\text{Processing Time Without Pipelining}}{\text{Processing Time With Pipelining}}$$

Figure 2.11 shows an example with three frames. Without pipelining, the frames must be executed serially. With a pipelining of three stages, a new frame is loaded every 0.5ms. The whole pipeline can be executed with a speed-up of 3ms/2ms = 1.5. The latency remains equal with 1ms. The throughput is increased to one prediction every 0.5ms, once all three stages are filled.

2.6.1 Receptive field of pipelined tdp-systems

The receptive field of the tdp allows to analyze how values between layers affect each other. In [4], between classification of an acoustic event, zeros were padded. This leads to a receptive field of figure 2.12 on the left. Data fields between the classification were programmatically set to zero. As a result, values of preceding and successive frames do not influence the values inside the intermediate buffers.Pipelining the classification leads to the elimination of the zero-padding between frames, which changes the receptive field of the intermediate buffers. Figure 2.12 shows an example on the right side. Now the preceding data point are still inside the buffered systems, which influences all following convolutions.

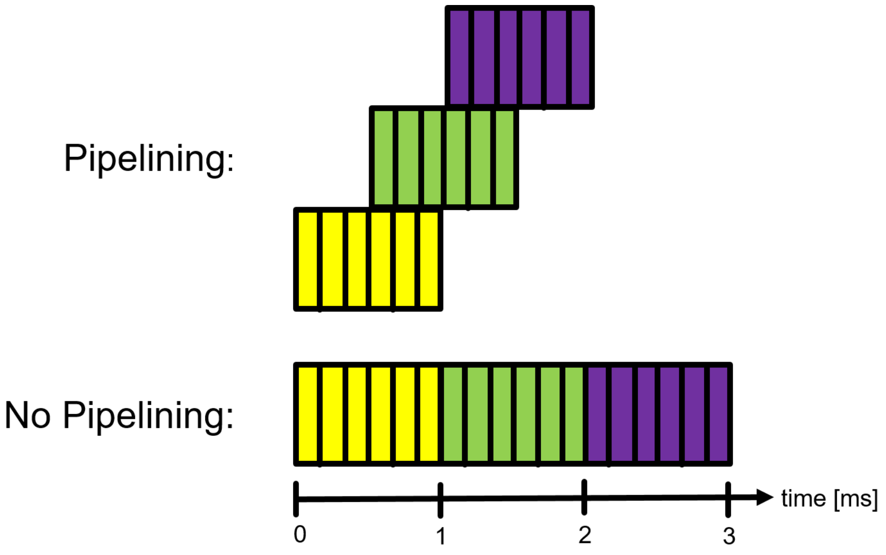

Figure 2.11: The concept of pipeliningof three frames. Above the pipelined version, below the non-pipelined version.

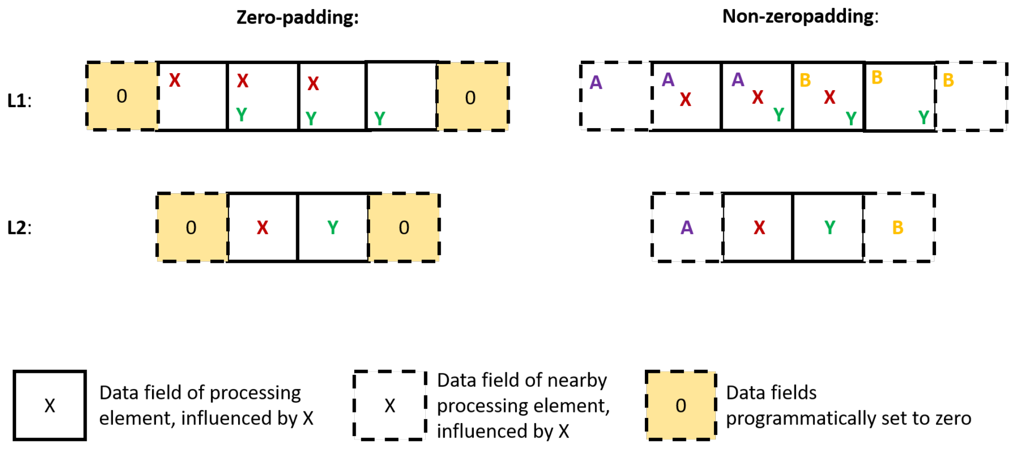
Zero-padding:
Non-zeropadding:
L1:
L2:
0
X
Y
A
B
Data field of processing element, influenced by X
Data field of nearby processing element, influenced by X
Data fields programmatically set to zero

Figure 2.12: Example for receptive field between layer L1 and L2. The difference lies in the influence of preceding data points for the calculation of the next layer.

As conclusion, pipelining a tdp-system can increase the throughput of classification while causing dependencies between the events to classify inside the pipeline. This represent an additional challenge, especially regarding the training of the weights.

Chapter 3

System overview

The main goal of this semester project is to implement CNN with tdp on a MCU, able to classify continuously. This system may then be adapted on the WSN on the Matterhorn-Hörnligrat or Dirruhorn Rock Glacier. This chapter describes the platforms used for deployment and testing, such as the sensor network and the used embedded MCU platform Nucleo L496ZG.

3.1 Wireless Sensor Network (WSN)

The PermaSense Consortium unites different research projects from Swiss universities and companies. The focus lies on environmental research of climate, geomorphdynamics, cryptosphere and the connection between those research areas [18]. In order to investigate those areas, multiple WSN with different hardware haven been deployed in different locations. All those WSN are optimized for low-latency, data management, low-power and often need customized sensors. The WSN we are interested in for this project is theGDPP. It was adapted in the Master Thesis "Event-based Geophone Platform with Co-detection" [10]. The (GDPP) with the STM32L49 is installed currently on two location, the Matterhorn-Hörnligrat and the Dirruhorn Rock Glacier [18]. Figure 3.1 shows the conceptual system design.

3.2 Microcontroller Unit (MCU)

For our investigation we use the same chipset used for the nodes of the WSN with the GDPP: STM32L49. The implementation changes were not directly tested on the GDPP, but on the NUCLEO-L496ZG. This is a development board of ST Microelectronics [12] with the STM32L49 on it. The family of the STM32L4 family is optimized for low-power embedded system applications with ultra-low power features. The used STM32L49 MCU features a 32-bit Cortex-M4 Reduced Instruction Set Computer (RISC) Central Processing Unit (CPU) and a Floating

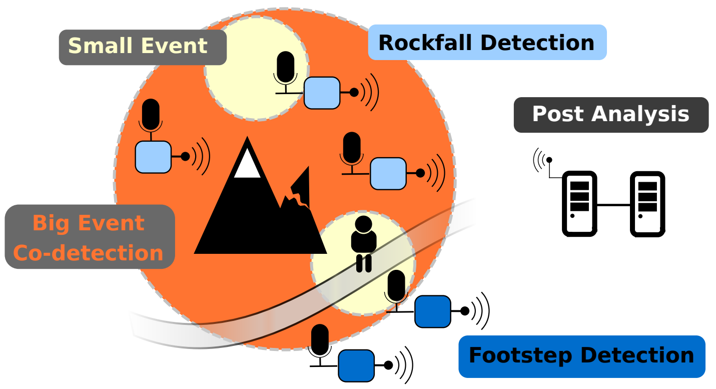

Figure 3.1: Conceptual overview of a WSN for natural hazard monitoring. Taken from [7].

Point Unit (FPU) with single precision. It also includes instructions for Digital Signal Processor(DSP). Furthermore, the MCU features up to 1MB Flash Memory and 320KB SRAM.The CPU clock ranges up to 80 MHz and an operating voltage between 1.71V and 3.6V. The general operating conditions report a power dissipation between 156 mW ($T_A = 125°C$) and 625 mmW ($T_A = 85°C$). All data taken from the official datasheet [13].

From the given specification, the question arises whether the CNN from table 2.1 can be implemented in buffered form with the given restrictions in memory footprint and CPU speed. The results can be found in chapter 6.

3.3 Real-Time Operating System

In order to create a cross-platform compatible implementation, it was ported to FreeRTOS. FreeRTOS is an online available and open-source Real Time Operating System [6]. In general, it can run applications with hardware and software requirements. Its scheduling algorithm can be interpreted as a fixed-position preemptive scheduling with time slicing [15]:

- **Fixed priority:** Assigned priority can not be changed during execution.
- **Preemptive:** Running task can be stopped if higher priority task arrives.
- **Time slicing:** If more than one task with same priority arrives, then each will run for the same amount of time.

The porting of the implementation to FreeRTOS was done by Reto Da Forno. In general, the operating system is available free of charge and can be downloaded from the homepage [6]. In order to make the FreeRTOS operating on a MCU, additional configuration files specific to the used chip must be added. Different reference manuals [1] explain how to merge the generated MCU code with the FreeRTOS.

Chapter 4

Methods

Chapter 2.5 introduced the overall classification sequence and notations. In [4], an acoustic event was classified frame by frame, as shown in figure 4.1. In order to reach the goal of continuous classification further algorithmic changes, added components and changed procedures are needed. This chapter explains the whole development process needed to reach continuous classification.

4.1 Testing model

The original implementation in [4] comes with a training code with Keras [3] and and an MCU implementation in C. A clear disadvantage of this implementation is the debugging of the tdp-processing on the target device. E.g. saving intermediate results in a file for postanalysis is impossible because of the restricted memory size and functionality of the target device. For scalability reasons, we decided to introduce an additional debugging step, called "Python-tdp". The original C-functions from [4] (e.g. 2D-convolution, ReLU, input shift and input update) are implemented in python, following the same flow and structure. By using python on a commodity laptop, the tdp-processing can be simulated and

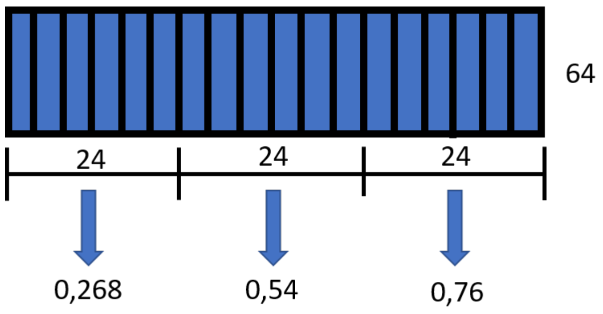

Figure 4.1: Example of a data input stream after FFT. The frames are classified frame by frame.

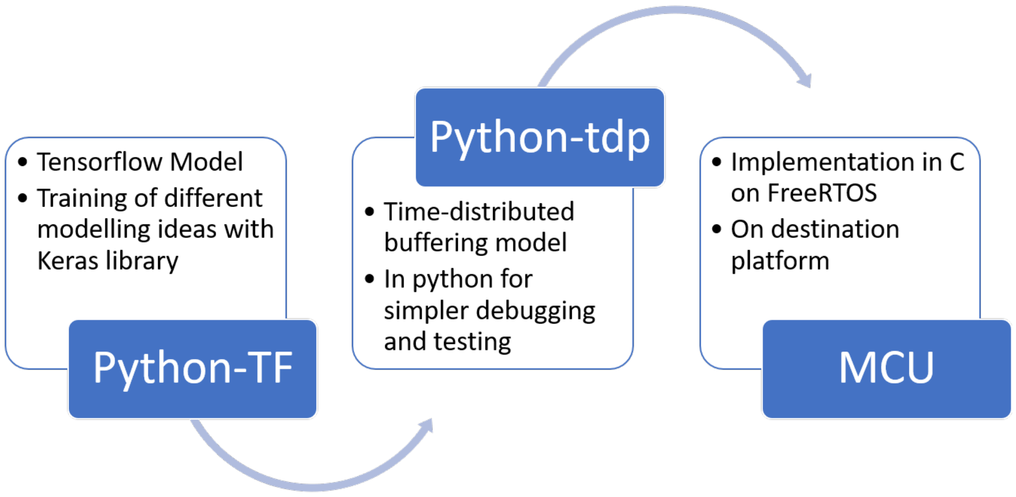
Tensorflow Model
Training of different modelling ideas with Keras library
Python-TF
Python-tdp
Time-distributed buffering model
In python for simpler debugging and testing
Implementation in C on FreeRTOS
On destination platform
MCU

Figure 4.2: Order of the testing models. With Keras, the TF model was trained. For debugging the implementation of time distributed processing was realized in python, which was then used for the MCU implementation.

improved. The final step is to implement the changes on the MCU in C. Figure 4.2 shows all the testing frameworks and their dependencies. For the code, see Appendix A.

4.1.1 Stage 1: Python-TF

The first stage is called Python-TF. This model uses the python library Keras [3], a high-level Application Programming Interface (API) for Tensorflow (TF). The CNN is trained and tested. The desired model with its structure and weights is exported as Keras model file. The model is reused one one hand as reference for the prediction and classification, on the other hand for debugging. Especially for debugging, it is useful to compare the stored values inside the intermediate buffers. The Keras model allows to analyze those values for any layer.

4.1.2 Stage 2: Python-tdp

Stage 2 contains the functions of the original MCU implementation, but in python 3.6. The functions (e.g. 2D-convolution, ReLU, input shift and input update) are implemented following the same programming flow. In that way, the functions are close to the final target functions, allowing meaningful debugging results. In this stage, the prediction errors between the Python-TF and Python-tdp must be minimized. Ideally the buffer contents and prediction value must be equal.

4.1.3 Stage 3: MCU implementation

The final step is to implement the tdp CNN on the target device. The same programming flow from the Python-tdp should be implemented in C on the target device. For this project we used the "Atollic TrueSTUDIO for ST32". This is the official toolchain supported by STMicroelectronics [2]. As reference values while debugging, the buffer contents and prediction values should be used. As explained in Python-TF, they can be generated from the Keras model file.

4.2 Optimization parameters

To evaluate and optimize the implementation or reduce the mismatch between the testing models, different parameters can be considered. A list of those is shown below:

- **Accuracy**: After a feedforward, the value inside buffer b7 represents the actual prediction, which thresholded gives the classification. In the end, all three models must predict the same value. During this project, we focused on the difference between those predictions and tried to minimize them as much as possible.
- **Program flow**: All models should follow the same program flow. To reduce mismatches between the models, the implementation should be as similar as possible. For python-tdp we introduced variables with the function of a pointer, in order to be as close as possible to the later MCU implementation.
- **Intermediate buffered values:** The results of a convolution are stored in the intermediate buffers. The size of them varies from buffer b7 (size of 1) to buffer b0 (6 columns with 64 row and 32 channel gives a buffer size of 12288). Especially buffer b0 to b5 are hard to debug because of their size. In the Python-tdp, we included subfunctions that stores the buffers to a file, aligns and separates them by channel. This allows to inspect the content of the intermediate buffers.

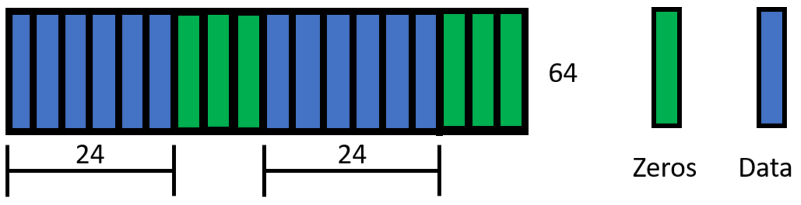

Figure 4.3: Classification with zero-padding. In [4], after the last subframe is forwarded to the input buffer, the processing window of the input buffer is set three times to zero.

4.3 Influence on receptive field

In [4], between frame predictions an equivalent of three subframes full with zeros are padded in between. Figure 4.3 shows the concept. As explained in chapter 2.6, in this case the frames to classify do not influence each other. By pipelining the predictions, the receptive field is extended and the frames influence each other. To approach this challenge, the extended receptive field was analyzed and the training model adapted accordingly.

4.3.1 Considerations of receptive field on training

The usage of zero-padding between frames allows a simpler training in TF, knowing exactly the intermediate values inside the buffers when the first and last subframe is forwarded. This leads to the case in figure 2.12 on the left. Between frames to classify, zeros are added or set to zero by the program.

By eliminating the zero-padding between frames, the values in the intermediate buffer depend of the convolution results of the preceding frame. The example on the right in figure 2.12 shows the extended receptive field between two layers. The calculation of the preceding frame affect the frame actually pushed into the prediction pipeline. Actually, the considered CNN in 2.1 has a total of eleven intermediate buffer of different sizes, as shown in figure 2.7. As a direct consequence of this sizes and convolution orders, the 20 data points of the preceding frame influence the prediction of the actual frame which is being classified. 20 data points are an equivalent of five subframes.

4.3.2 Consequence of receptive field consideration

The increased receptive field was considered by increasing the input size of the python-TF model. It was retrained with increased input size. Instead of using a size of 64x24 (regular frame size), it was increased to 64x44.

Figure 4.4 shows how the frame is selected. The first five subframes are

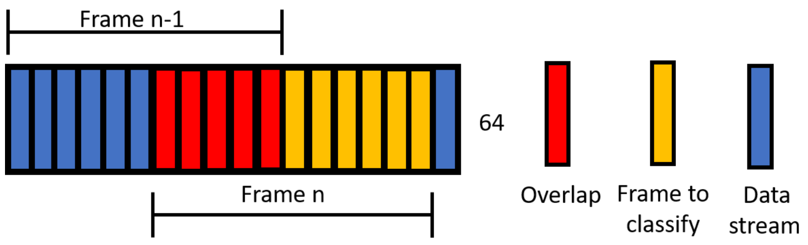

Figure 4.4: To consider the increased receptive field while training the TF model, the time-axis is increased of 20 data points (subframes). As conclusion, the additional 5 subframes to overlap with the preceding frame. The input size for the python-TF is 64x44.

only for the simulation of the already filled buffer. The next six subframes are the actual frame to be predicted. When considered all together, a clear overlapping with the previous frame can be noticed. This overlapping represents the previous processed values, which in the tdp-implementation are stored inside the buffers. This change in input size was implemented and the weights were retrained accordingly.

4.4 Eliminating zero-padding on time-axis

The model considers the preceding frame influence with the adapted weights. The next step is to change the implementation.

4.4.1 Changes in functions

To avoid zero-padding on the time-axis, the final adaption of the working MCU code needed a few changes in functions like input shift, buffer update, convolution and average pooling. Especially in the $CNN\_init()$ and the $conv2D\_3$. All changes in the C-code are shown in Appendix B. The most important changes are listed below:

- The size of the columns of buffer b0 was increased form 5 to 6. This allows data alignment.
- Because of the increased column size in buffer b0, the writing pointer "b0.w_ptr" starts at index 2 and not 1. In that way, the buffer was set to a size of two, the processing window to a size of four. As reference, see chapter 2.4.3.

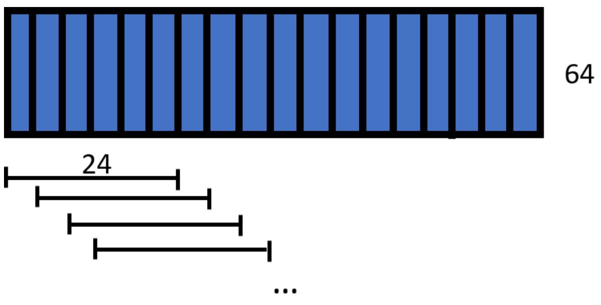

Figure 4.5: General idea of a pipelined classification system. A classification should be possible for every position of the classification window.

- In the convolution function with the 3x3 kernels conv2D_3(), values were hardcoded to zero inside buffers. Those changes were eliminated. Those changes are referenced in appendix B.

4.5 Pipelining the predictions

Once the influence of the extended receptive field is considered and the zero-padding on the time-axis eliminated, the next step was to implement the pipelining. This step enables continuous prediction like in figure 4.5.

4.5.1 Code flow

In order to pipeline the prediction, the program flow of the $do\_inference()$ function was redesigned. The overall architecture remains as stated in table 2.1. The zero-padding between frames of size 64x24 was eliminated, such that frames are forwarded to the input buffer one after another. The overall program flow is shown in figure 4.6. After the acquired data was preprocessed with an FFT and a subframe of size 64x4 is available, the $do\_inference()$ function is called and the data is forwarded to the input buffer. Between calls the values inside the intermediate buffers and the counter is stored. The first call is handled differently, because only the buffers affected of new values should be convoluted. Otherwise the bias weights introduce wrong values inside deeper layers, propagates through the buffer system and falsifies future values as well. The number of those different calls depend on the propagation delay of the tdp. In our case, the propagation delay is three. Once the first values have regularly reached the last layers, a complete feedforward over all buffers is computed for each of them.

Technically, each iteration can produce a valid prediction. In chapter 6.2, we want to associate the prediction of python-TF to the MCU prediction for accuracy measurement. In order to know which prediction should be compared with each other, figure 4.6 only updates every sixth prediction. In a final implementation, the return value can be set TRUE after every feedforward. In that case, the output would be updated after every new input of values, leading to a continuous prediction as figure 4.5 suggest.

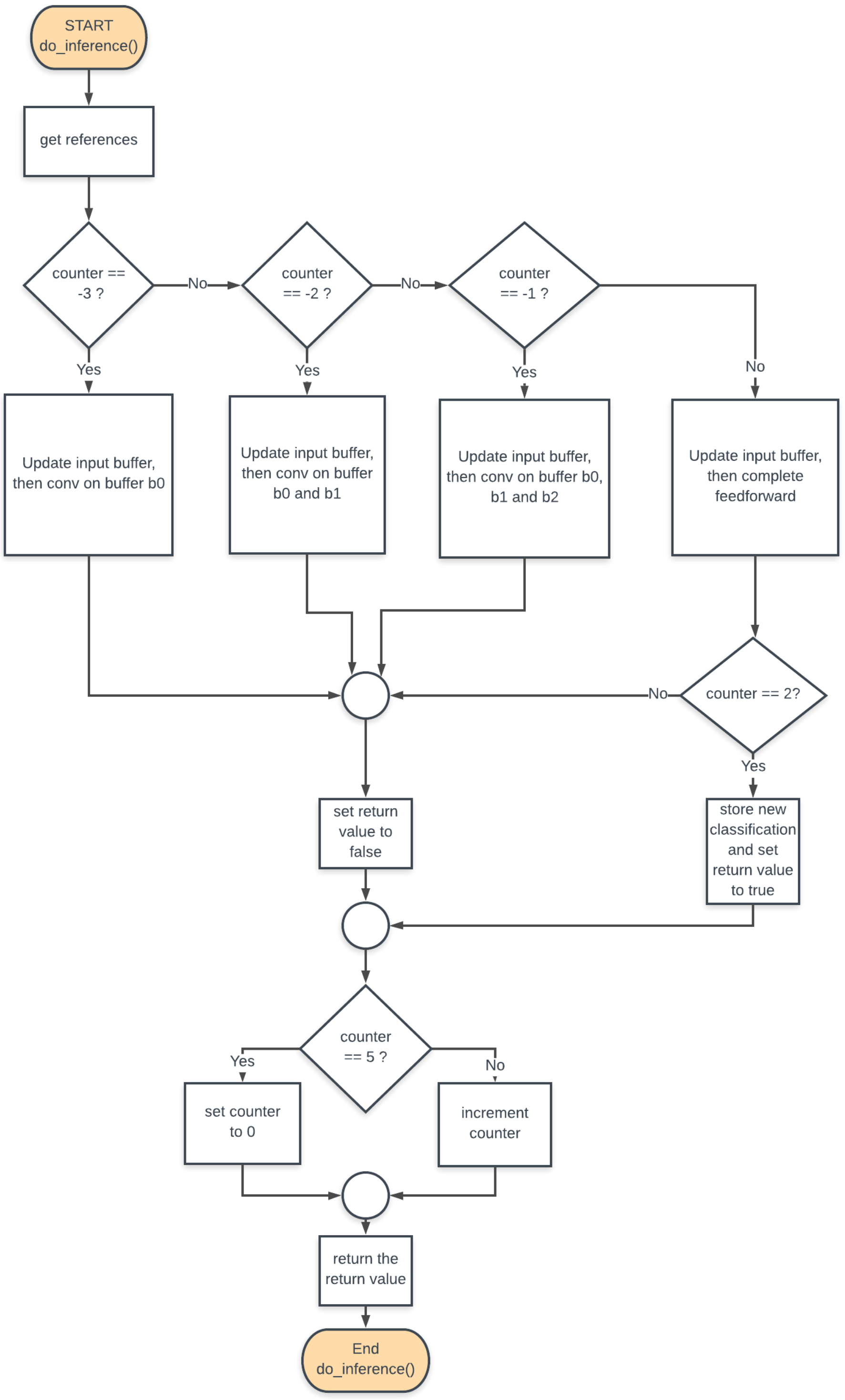
START do_inference()
get references
counter == -3 ?
No
counter == -2 ?
No
counter == -1 ?
Yes
Yes
Yes
No
Update input buffer, then conv on buffer b0
Update input buffer, then conv on buffer b0 and b1
Update input buffer, then conv on buffer b0, b1 and b2
Update input buffer, then complete feedforward
No
counter == 2?
Yes
set return value to false
store new classification and set return value to true
counter == 5 ?
Yes
No
set counter to 0
increment counter
return the return value
End do_inference()

Figure 4.6: Flow diagram of the pipelined $do\_inference()$ function. The counter goes to 5, then restarts at 0. At a counter value of 2, the classification is returned. Between calls, the buffer values remain stored in the buffer. With a system reset, the counter is initialized to -3. The very first iteration with zeros in all buffers is handled differently.

Chapter 5

Experiments

We want to verify that the system is pipelined and the classified values are coherent with the classification from the Python-TF. In order to run such experiments, the test framework explained in chapter 4.1 was used. In this chapter, the experiments regarding the pipelining and the accuracy measurement are explained. Note that the interpretation of the predictions is not part of this project, since this proof was part of the previous work from [4].

5.1 Verification of pipelining

As input values, acoustic waveforms were taken from [8]. A whole waveform has a duration of two minutes. In a first try, those waveforms were cut with different lengths and then flashed on the internal memory of the MCU. In that way, instead of simulating the acquisition of the acoustic data, the data was streamed directly form the internal FLASH memory. This allowed to exclude external errors on the acquisition. Forwarding one subframe is triggered by a push button. Each feedforward increments the counter, whereas only every sixth prediction should be displayed and later compared for accuracy. As a drawback, only the classification of a few waveforms could be tested in that way.

The implementation in FreeRTOS comes with the possibility to stream waveforms to the UART input of the NUCLEO-L496ZG. This option can be activated by setting the constant USE_PUSHBUTTON_TRIGGER for the MCU implementation in "config.h" to 1. See appendix A. The classification result is the same, independent if streamed from the internal FLASH memory or externally over the UART input. To debug we used the option to read from the FLASH memory. In order to test more than a few predictions, streaming over UART must be used. We used a test set from [8] that allowed to compare 194 predictions. The results are interpreted in chapter 6.

5.2 Accuracy measurement

As shown in figure 4.2, in a first step the difference between the python-TF and python-tdp is minimized. Then the difference between the MCU implementation and the python-tdp is minimized, ideally being equal to the python-TF prediction. To determine the accuracy of the implementation from the MCU, the accuracy is measured. When determining the accuracy, we measure the difference from tdp prediction on the MCU to the python-TF prediction. The smaller, the better. The error is expected to be close to zero, the results are interpreted in chapter 6.

The pipelined prediction can give a prediction in every time step. In order to align and compare the predictions to the python-TF predictions, only every sixth prediction leads to a comparable value. This is also shown in the flow diagram 4.6. Because we compare the classification after one whole fragment, the classification in between do not update the output. In a final implementation, the return value can be set TRUE after every feedforward, given a prediction in every update step.

Chapter 6

Results

In chapter 5, the experiments for verification were introduced. This section will discuss the results of the experiments regarding pipelining and accuracy.

6.1 Verification of pipelining

We were able to implement a time-distributed processing on the MCU. Figure 4.6 shows the program flow. As the original implementation in[4], the subframe is completely forwarded to the input buffer for classification. Compared to the implementation in [4], every time the input buffer is updated, a new prediction is is generated from the feedforward. To determine the functionality of this pipelining, 194 predictions were generated and tested. Following can be stated:

- Latency: Compared to the original implementation in [4], the latency remains equal. From the moment the first subframe of a frame enters the input buffer until the classification is available, six subframes must be forwarded to the input buffer.
- Speed-up: Over all generated prediction, the time for a valid prediction was measured and averaged. The speed-up of the pipelined version compared to the non-pipelined in [4] is 6.62 in average. It is more than six because of two reasons. One is that the non-pipelined version calls the feedforward six times, whereas the sixth call has more operations (estimated 1.5 times more) than the others, because of the zero-padding. The second reason is the function call. Each time the feedforward function returns a value, the FreeRTOS computes own system functions and prepossessing of subframes before calling the feedforward again. Both lead to a speedup higher than the throughput.
- Throughput: With pipelining, in each feedforward a classification is available. In the time the original implementation in [4] generated one prediction, the pipelined classification generates six predictions. The pipelined throughput is six times higher.

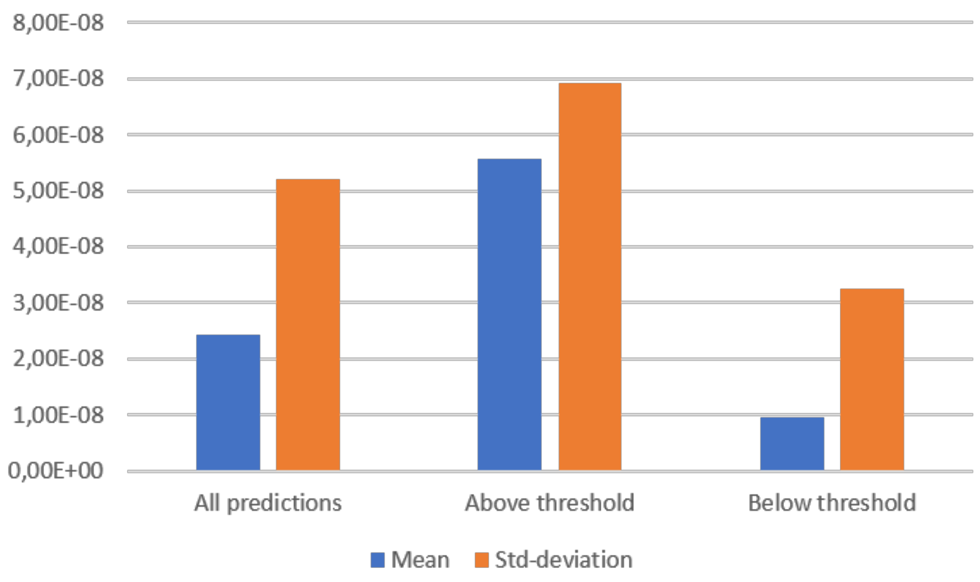

Figure 6.1: Mean and standard deviation of the measured prediction difference. In total 194 classification points are considered. 62 predictions are above the threshold of 0.6 and detected a mountaineer, 132 predictions detected no mountaineer.

From those results, we deduce that the system is pipelined regarding classification of mountaineers based on acoustic events.

6.2 Verification of accuracy

With the experiments from chapter 5.2, 194 predictions were generated and compared to the prediction from the python-TF model. The average error over all points is $2.427 \cdot 10^{-8}$, with a standard deviation of $5.242 \cdot 10^{-8}$. This is also plotted in figure 6.1 on the left. The standard deviation is bigger than the average error. This can be deduced from the measurement method and distribution of waveforms. On one hand, only the absolute error is considered. On the other hand, the test data contains more examples of no mountaineer than such with a mountaineer. From the 194 classifications, 62 contain a mountaineer and 132 do not contain a mountaineer. The distribution of the test waveforms is considered in figure 6.2. Each measured absolute error is associated to its prediction ground truth. The threshold of the pipelined classification is at 0.6. All predictions above are classified as “Mountaineer Detected”, whereas all below classify “No Mountaineer Detected”. It can be noticed that the absolute error of no the category “Mountaineer Detected” spreads less than the one of the category “No Mountaineer Detected”, even if there are more waveforms containing no moun-

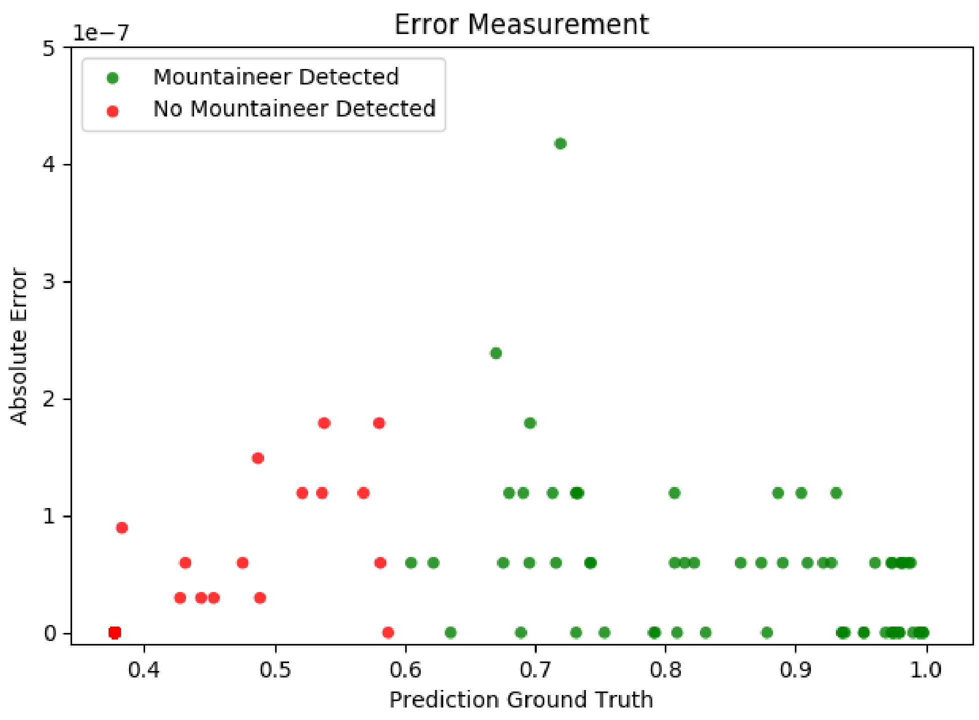

Figure 6.2: Overview of all 194 test predictions. The measured absolute error is associated to the prediction ground truth. Most absolute errors are zero. Threshold is at 0.6.

taineer. In fact, when considering only examples with a mountaineer detected, we get 62 examples with a mean error of $5.576 \cdot 10^{-8}$ and standard deviation of $6.927 \cdot 10^{-8}$. In contrast, the 132 examples with no mountaineer have a mean error of $0.948 \cdot 10^{-8}$ and standard deviation of $3.246 \cdot 10^{-8}$. We noticed that most predictions only forward a zero to the last layer, which ends up being added to the bias weights of the output layer (0.377 with the used Keras model). The weights are for both models the same, so the bias is the exactly same as well and ends up with an error of perfect zero. This reduces the mean error, as well as the standard deviation.

Overall, all differences are $\propto 10^{-8}$. From this analysis, we deduce that the error between the python-TF and the tdp on MCU can be neglected. As conclusion, the python-TF and tdp on MCU generate the same prediction.

Chapter 7

Conclusion

The small village of Bondo (GR) was hit by multiple avalanches in August 2017 [17], cause the fatality of eight mountaineers. Motivated to contribute to a real-time alarm system, we proposed a system able to continuously detect mountaineers on embedded platforms. Using [4] as reference, the main challenge was to extend it to continuous classification. To continuously classify, the prediction generation was pipelined. In order to reach pipelining, the zero-padding was removed by determining the influence of buffers on the receptive field. The weights and bias were retrained accordingly and the prediction accuracy on the target MCU was verified.

In chapter 4.3, we analyzed the influence of intermediate buffers on the repetitive field when eliminating the zero-padding between acoustic events. After FFT, the values of the preceding fragment are inside the intermediate buffers when a new frame is forwarded. We found that exactly 20 data points (five subframes) of the preceding frame influence a prediction. As a consequence, the input of the python-TF model was increased from 64x24 to 64x44. This in order to respect the extended receptive field. All weights, bias and the hard classification threshold were updated accordingly.

After the influence on the receptive field was analyzed, the next step was to eliminate the zero-padding on the time-axis. Chapter 4.4 explains all adaptions on the implementation needed. Those changes affected the size of the intermediate buffer b0, the processing window, as well as implicit zero-padding in the convolution function. The code changes are referenced in appendix B.

Once eliminated the zero-padding on the time-axis, the next challenge was to pipeline the predictions. In chapter 4.5 can be found the adapted code flow with respective diagram in figure 4.6. Chapter 6.1 formally verified the pipelining results. By pipelining we increased the throughput, resulting in a speed-up of 6.62 compared to the implementation in [4]. The affected code changes are referenced in appendix C.

The last step is the accuracy verification of all those steps. In order for being able to debug and test the waveforms, the testing model introduced in chapter

4.1 was created and used. It includes the training on Tensorflow, the testing on the python-tdp and the final implementation verification of the tdp on the MCU. In chapter 6.2 we explain the results of the accuracy verification. Over all test point a mean error of $2.427 \cdot 10^{-8}$ with a standard deviation of $5.212 \cdot 10^{-8}$ was measured. Both are $\propto 10^{-8}$, which can be neglected as error. We conclude that the final implementation of the tdp on MCU gives the correct predictions and the respective classification.

The reference implementation in [4] was hardcoded to one specific MCU. Chapter 3.3 gives an overview about FreeRTOS and how to port it to a target MCU. For the testing, the NUCLEO-L496ZG with the STM32L49 MCU was used. This because it is the same MCU used on the GDPP, makes a future testing on the GDPP simpler.

Overall, this project contributes successfully to the improvement of pipelined classification and event detection on embedded platforms. We have shown that the implementation can be adapted to a pipelined system. Our testing has shown that the final prediction with tdp on a MCU can reach the desired accuracy in classification. In summary, this system wold be feasible for the continuous prediction of mountaineers on the GDPP.

Chapter 8

Outlook

In this Semester Project we implemented a pipelined acoustic CNN and tested it on the STM32L49 MCU. This chapter presents further improvement ideas for future work.

During this project, we eliminated the missmatch between the prediction of the python-TF model and the tdp implementation on a MCU. We ignored the fact that removing zero-padding and pipelining the prediction might lead to a worse python-TF model. The scientific interpretation of the relative output values was not part of this project and may be analyzed in a future work.

In order to reach pipelining, the priority was to eliminate the zero-padding in the time-axis. Nevertheless, the final implementation still uses zero-padding on the frequency-axis. This zero-padding happens during the convolution functions with the 3x3 kernel. With the elimination of the zero-padding on the frequency-axis, the dimensions shrink from layer to layer. As result, the number of rows must be adapted as well. In addition the weights and threshold must be retrained as well.

The long-run goal is to create a system that protects humans from natural disasters. The implementation could be extended to classify more than one class, e.g. rockfalls. In addition to the used test waveforms from [8], an additional one for rockfalls could be used for train the CNN implementation on rockfall detection. The more classes an environment detection system can detect, the better the context understanding. In that way, more precise alarms could be triggered.

In his Master Thesis, Akos Pasztor [10] launched and supervised the mass-production of 50 event-based microseismic sensing platform [10], which were used as nodes of the WSN on the Matterhorn-Hörnligrat and Dirruhorn Rock Glacier. In our project, the STM32L49 was chosen as testing platform because it is the same MCU as its target platform (GDPP). In a next step, the tdp implementation of the CNN can be merged into the system of the WSN nodes. In that way, more knowledge from the operation under real conditions could be gained.

Bibliography

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

[2] Atollic. Atollic truestudio for stm32. `https://atollic.com/truestudio/`. Accessed: 2019-02-19.

[3] François Chollet et al. Keras. `https://keras.io`, 2015.

[4] Timo Pascal Farei-Campagna. Quantized convolutional neural networks for embedded platforms. Master's thesis, ETH Zürich D-ITET, March 2018.

[5] Prof. Dr. Anton Gunzinger. Applied computer architectures- chapter 5 - pipelining, oct 2019. Form Slides of class Applied Computer Archtecture, Fall 2019 ETH Zürich.

[6] Real Time Engineers Ltd. Freertos references. `https://www.freertos.org`. Accessed: 2018-12-12.

[7] Matthias Meyer, Jan Beutel, and Lothar Thiele. Event-triggered natural hazard monitoring with convolutional neural networks on the edge. *Submission 31 for IPSN19 in Canada*, 2019.

[8] Matthias Meyer, Samuel Weber, Jan Beutel, Stephan Gruber, Tonio Gsell, Andreas Hasler, and Andreas Vieli. Micro-seismic and image dataset acquired at Matterhorn Hörnligrat, Switzerland, July 2018.

[9] Matthias Meyer, Samuel Weber, Jan Beutel, and Lothar Thiele. Systematic identification of external influences in multi-year micro-seismic recordings using convolutional neural networks. *Earth Surf. Dynam. Discuss.*, 2018. in review, https://doi.org/10.5194/esurf-2018-60.

[10] Akos Pasztor. Event-based geophone platform with co-detection. Master thesis, ETH Zürich, may 2018. Available Online: https://pub.tik.ee.ethz.ch/students/2017-HS/MA-2017-25.pdf.

[11] Techopedia THE IT Education Site. Edge computing. Online Dictionary, December 2018. https://www.techopedia.com/definition/32472/edge-computing.

[12] STMicroelectronics. *STM32 Nucleo-144 boards.* Online available: https://www.st.com/en/evaluation-tools/nucleo-l496zg.html.

[13] STMicroelectronics. Stm32l496xx datasheet. Available Online: https://www.st.com/en/microcontrollers/stm32l496rg.html.

[14] Neil Tan. Why machine learning on the edge? Online, May 2018. https://towardsdatascience.com/why-machine-learning-on-the-edge-92fac32105e6.

[15] Lothar Thiele. Lecture notes in embedded systems, spring semester 2018, 2018.

[16] Prof. Dr. Lother Thiele. Computer engineering 1 - chapter 5 - processor pipelineimplementation. Class Slides Fall 2015, sep 2015. Slides from Class of Computer ENGINEERING 1 at ETH zürich, Chapter 5.

[17] Author unknown. Storm sparks another big landslide, bondo hit again. *thelocal.ch*, Sep 2017.

[18] UZH and ETH. The permasense consortium. `https://www.permasense.ch/en.html`. Accessed: 2018-12-16.

[19] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *CoRR*, abs/1702.03044, 2017.

Appendices

Appendix A

Code reference

All referenced code and framework is available under www.gitlab.ethz.ch as "SA - Spadafora - ClassificationGPP". The clone link over HTTPS is:

$$https : //gitlab.ethz.ch/tec/students/projects/2018/sa\_fspadafo.git$$

Under this link can be found:

- python code for the tdp-classification
- MCU implementations in C
- Documentation
- Implementation with FreeRTOS and possibility to stream acoustic waveform over UART

Appendix B

Changes on core function in MCU implementation

The C code listing below shows the relevant changes inside the file "cnn.h". For the fully pipelined classification, the constants *PROPAGATION_DELAY*, DELAY, and *NUMBER_FRAMES* were added. In addition, the global constant *B0_NUM_COLUMNS* was increased to 6 for the non-zero-padding case.

```c
 ...

#define TEMP_INPUT_NUM_CHANNELS 1
#define TEMP_INPUT_NUM_ROWS 4
#define TEMP_INPUT_NUM_COLUMNS 64
#define TEMP_IMG_SIZE TEMP_INPUT_NUM_CHANNELS*TEMP_INPUT_NUM_ROWS*
    TEMP_INPUT_NUM_COLUMNS

#define COMPLETE_T 24

//-------- additional parameters -----

#define PROPAGATION_DELAY 3
#define DELAY 3
#define GLOBAL_ZEROPADDING 0
#define NUMBER_FRAMES 6

// size of the buffers necessary for each layer and the input
#define B_INPUT_NUM_CHANNELS 1
#define B_INPUT_NUM_ROWS 64
#define B_INPUT_NUM_COLUMNS 6
#define B_INPUT_SIZE B_INPUT_NUM_CHANNELS*B_INPUT_NUM_ROWS*
    B_INPUT_NUM_COLUMNS
#define B_INPUT_SHIFT 4

#define B0_NUM_CHANNELS 32
#define B0_NUM_ROWS 64
#define B0_NUM_COLUMNS 6 //Increase size for non-zero padding
#define B0_SIZE B0_NUM_CHANNELS*B0_NUM_ROWS*B0_NUM_COLUMNS
#define B0_SHIFT 4
```

```

  ...
```

Additional changes in the functions for the neural network were also made inside "cnn.c". The most noticeable changes were made in the function for the convolution with a 3x3-kernel where the opcode is 1 (case for complete convolution). Here the branches for the different strides were eliminated. The different cases are handled by accordingly updating the pointers at the end.

```
void CNN_Init()
{
  ...

  // initialize the b0 buffer
  b0.r_ptr = &b0_data[0];
  if(GLOBAL_ZEROPADDING){
    b0.w_ptr = &b0_data[1];
  }else{
    b0.w_ptr = &b0_data[2];
  }
  b0.in_ptr = ib.r_ptr;
  b0.num_channels = B0_NUM_CHANNELS;
  b0.num_rows = B0_NUM_ROWS;
  b0.num_columns = B0_NUM_COLUMNS;

  ...


void conv2D_3(uint8_t opcode, data_buffer *inbuf, data_buffer *
    outbuf) {

  ...

  else if(opcode == 1) {  // compute all buffer columns
        // compute convolution
        do {  // loop over output channels
          out_ptr = out_channel_start_ptr;  // set the out_ptr to
    the correct place for the current output channel

            do {  // loop over rows of the current output channel
              uint32_t col = 0;
              do {
                out_ptr[col] = LuT_bias[*bias_ptr];
                col++;
              } while(col < outbuf->num_non_buffer_columns);

              out_ptr += outbuf->num_columns;
            } while(out_ptr != out_channel_limit_ptr);

          bias_ptr++; // increment the bias pointer to the next
    filter channel
          out_channel_limit_ptr += invariant0;  // increment the
    bias pointer's limiter to the end of the next output channel
```

```

        out_ptr = out_channel_start_ptr;  // reset output pointer for MAC operation

        do {  // loop over input channels
          in_row_limit = &in_ptr[invariant1]; // determine the row limit for the current input channel

          // store the current filter channel (in CPU registers)
          filter0 = LuT_kernel[kernel_ptr[0]];
          filter1 = LuT_kernel[kernel_ptr[1]];
          filter2 = LuT_kernel[kernel_ptr[2]];
          filter3 = LuT_kernel[kernel_ptr[3]];
          filter4 = LuT_kernel[kernel_ptr[4]];
          filter5 = LuT_kernel[kernel_ptr[5]];
          filter6 = LuT_kernel[kernel_ptr[6]];
          filter7 = LuT_kernel[kernel_ptr[7]];
          filter8 = LuT_kernel[kernel_ptr[8]];

          if(fd->stride == 1) {

             //simulate zero-padding on f-axis
            in_col_limit = &in_ptr[outbuf->num_non_buffer_columns*fd->stride];
            do { // loop over input columns of the first input row in the current input channel
              input0 = in_ptr[0];

              input1 = in_ptr[1];
              input2 = in_ptr[2];
              input3 = in_ptr[inbuf->num_columns];
              input4 = in_ptr[inbuf->num_columns + 1];
              input5 = in_ptr[inbuf->num_columns + 2];

              *out_ptr += input0*filter3;
              *out_ptr += input1*filter4;
              *out_ptr += input2*filter5;
              *out_ptr += input3*filter6;
              *out_ptr += input4*filter7;
              *out_ptr += input5*filter8;

              in_ptr = in_ptr + fd->stride;
              out_ptr++;

            } while(in_ptr != in_col_limit);

            out_ptr += (outbuf->num_columns - outbuf->num_non_buffer_columns); // to hop over the first actual buffer column!

            in_ptr -= outbuf->num_non_buffer_columns * fd->stride; // reset input pointer
          }

        do {  // loop over input rows in the current input channel
            in_col_limit = &in_ptr[invariant2]; // determine the column limit for the current input row

            float* in_ptr1 = &in_ptr[inbuf->num_columns];
            float* in_ptr2 = &in_ptr1[inbuf->num_columns];

            do {  // loop over input columns in the current input column
              // compute MAC operation
              input0 = in_ptr[0];
              input1 = in_ptr[1];
              input2 = in_ptr[2];
              input3 = in_ptr1[0];
              input4 = in_ptr1[1];
              input5 = in_ptr1[2];
              input6 = in_ptr2[0];
              input7 = in_ptr2[1];
              input8 = in_ptr2[2];

              *out_ptr += input0*filter0;
              *out_ptr += input1*filter1;
              *out_ptr += input2*filter2;
              *out_ptr += input3*filter3;
              *out_ptr += input4*filter4;
              *out_ptr += input5*filter5;
              *out_ptr += input6*filter6;
              *out_ptr += input7*filter7;
              *out_ptr += input8*filter8;

              // update input/output pointers
              out_ptr++;
              in_ptr += fd->stride;
              in_ptr1 += fd->stride;
              in_ptr2 += fd->stride;
            } while(in_ptr != in_col_limit);

            out_ptr += invariant3; // to hop over the actual buffer columns!

            in_ptr += invariant4; // increment input pointer to the next row
          } while(in_ptr != in_row_limit);

    // simulate zero-padding on f-axis
        in_col_limit = &in_ptr[outbuf->num_non_buffer_columns*fd->stride];
        do { // loop over input columns of the second last input row of the current input channel
          input0 = in_ptr[0];
          input1 = in_ptr[1];
          input2 = in_ptr[2];
          input3 = in_ptr[inbuf->num_columns];
```

```
                input4 = in_ptr[inbuf->num_columns + 1];
                input5 = in_ptr[inbuf->num_columns + 2];

                *out_ptr += input0*filter0;
                *out_ptr += input1*filter1;
                *out_ptr += input2*filter2;
                *out_ptr += input3*filter3;
                *out_ptr += input4*filter4;
                *out_ptr += input5*filter5;

                in_ptr += fd->stride;
                out_ptr++;

            } while(in_ptr != in_col_limit);

            out_ptr += (outbuf->num_columns - outbuf->num_non_buffer_columns); // to hop over the actual buffer columns !

            in_ptr += (inbuf->num_columns - outbuf->num_non_buffer_columns*fd->stride) + inbuf->num_columns;  // increment input pointer to the next input channel

            kernel_ptr += 9;  // increment kernel pointer to the next filter channel

            out_ptr = out_channel_start_ptr;  // reset output pointer for the next input channel
          } while(in_ptr != in_limit);

          in_ptr = &inbuf->r_ptr[0];  // reset input pointer for the next filter channel

          out_channel_start_ptr += outbuf->num_rows*outbuf->num_columns;  // update output channel start pointer
        } while(out_channel_start_ptr != out_limit);
    }

  ...
}
```

Appendix C

Fully pipelined classification: $do\_inference()$

Compared to the original $do\_inference()$ function from [4], the whole function was rewritten and adapted. In general, in every feedforward step new values are forwarded to the input buffer for classification.

```c
  ...

uint8_t do_inference(float *input_buffer, CNN_Output_t *cnn_output,
    int start_value)
{
  static int first_call = 0;
  static int img_fraction_counter = 0;
  static uint32_t return_value = 0;

  //set right delay of img_fraction_counter
  if(first_call == 0){
    img_fraction_counter = start_value;
    first_call = 1;
  }

  ib.in_ptr = input_buffer;

  if(img_fraction_counter == -DELAY){
    input_update(1);

    conv2D_3(1,&ib,&b0);
    relu(&b0);

  }else if(img_fraction_counter == -DELAY+1){
    input_shift();
    input_update(1);

    shift(&b0);
    conv2D_3(1,&ib,&b0);
    relu(&b0);

    shift(&b1);
    conv2D_3(1,&b0,&b1);
    relu(&b1);

  }else if(img_fraction_counter == -DELAY+2){
    input_shift();
    input_update(1);

    shift(&b0);
    conv2D_3(1,&ib,&b0);
    relu(&b0);

    shift(&b1);
    conv2D_3(1,&b0,&b1);
    relu(&b1);

    shift(&b2);
    conv2D_3(1,&b1,&b2);
    relu(&b2);

  }else if(img_fraction_counter >= 0){
    input_shift();
    input_update(1);

    shift(&b0);
    conv2D_3(1,&ib,&b0);
    relu(&b0);

    shift(&b1);{
    conv2D_3(1,&b0,&b1);
    relu(&b1);

    shift(&b2);
    conv2D_3(1,&b1,&b2);
    relu(&b2);

    shift(&b3);
    conv2D_3(1,&b2,&b3);
    relu(&b3);

    conv2D_3(1,&b3,&b4);
    relu(&b4);

    conv2D_1(&b4,&b5);
    relu(&b5);

    conv2D_1(&b5,&b6);
    relu(&b6);

    shift(&f_avg);
    f_avg_update(&b6, &f_avg);

    t_avg_update(&f_avg, &t_avg);

    conv2D_1(&t_avg, &b7);

    activation_sigmoid(&b7);

    if(!GLOBAL_ZEROPADDING && img_fraction_counter == 2){
      for(uint16_t ii=0;ii<NUM_LABELS;ii++){
        cnn_output->value[ii] = b7.r_ptr[ii];
          if(b7.r_ptr[ii] > NN_OUT_THRESHOLD){
            cnn_output->decision[ii] = 1;
          }else{
            cnn_output->decision[ii] = 0;
          }
        }
      return_value = 1;
    }else{
      return_value = 0;
    }
  }

  if(img_fraction_counter == (NUMBER_FRAMES -1)){
    img_fraction_counter = 0;
  }else{
    img_fraction_counter += 1;
  }
  return return_value;

}

  ...
```