Investigation of dynamic control ML algorithms on existing and future Arm microNPU systems

DAVID CORDESIUS & JOEL ÅHLUND
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Investigation of dynamic control ML algorithms on existing and future Arm microNPU systems

David Cordesius, Joel Åhlund
lan15dco@student.lu.se, jo7106ah-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisors:
Kevin Wohnrade (Arm)
Steffen Malkowsky (LTH)
Lucas Ferreira (LTH)

Examiner:
Erik Larsson

June 29, 2021

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

In this thesis, dynamically controlled machine learning algorithms running on state of the art Arm microNPUs, with an attached Cortex-M CPU, were investigated. The machine learning framework used was Tensorflow and different subsets of it, such as Tensorflow Lite micro. Compiling the network to run on the microNPU was done with the use of an open-source compiler called, *Vela*. In order to investigate the dynamic support - the algorithm, *MTCNN*, as proposed by K. Zhang et, al. [1] was implemented on the aforementioned hardware; MTCNN was chosen due to its dynamic properties, as the algorithm structure changes depending on the input it receives. Where some parts of the algorithm may not be executed as frequently, or at all, depending on the result of earlier stages.

Full dynamic support was possible on the CPU through custom kernel implementations. Several of the components needed for full use of dynamic control, on the microNPU, were unsupported throughout the toolchain. Because of this, MTCNN was divided into several parts (removing the need for dynamic support) to further investigate the performance gain by natively supporting the dynamic control operators on the microNPU. The results obtained clearly outlines a general performance gain by adding dynamic control support and the ability to run and schedule an algorithm, as a single ML model. However, for MTCNN in particular it was concluded that a major speedup was achieved by executing the static parts of the algorithm on the microNPU. The dynamic parts, with regards to *MTCNN*, amounts for a small percentage of the total run-time.

Popular Science Summary

When it comes to computer science in general; dynamic control flow, such as conditional statements and loops, are a common tool for algorithm development. However, the same can not be said for machine learning applications, even though there is a deep connection between the two. The world is becoming increasingly connected, with intelligent devices present in everyday products ranging from security cameras to LED lights. The need for smarter edge devices is growing and more demand is put on said devices - a combination of these two topics are touched upon in this thesis.
The possibility of implementing machine learning algorithms that are in nature, dynamic, on embedded devices was investigated. This was done via implementing an algorithm called *MTCNN* on a system containing a general purpose computer with an attached accelerator. The accelerator is specialized to perform tasks often found in machine learning algorithms. The process of running an algorithm on the accelerator included extensive use of different tools.

The fundamental approach to this investigation was reminiscent of the famous phrase "Divide and conquer", however, "Divide and analyze" is a better description; *MTCNN* was deconstructed into smaller parts to analyze what was, and what was not, supported by the accelerator.
This resulted in the ability of running performance estimates for the complete algorithm. A resounding increase to performance was achieved when larger parts of the algorithm were able to run on the accelerator, however, the dynamic parts had to be executed on the general purpose computer. The reason for this was the lack of support for dynamic control flow in the different tools and hardware; Due to this, the possibility of running algorithms like *MTCNN* completely on the accelerator is currently not possible.

Acknowledgements

First of all, we would like to thank Arm Sweden for the opportunity of conducting this master thesis in collaboration with them.

We would like to express our gratitude towards our supervisor Kevin Wohnrade, who has provided useful insight and guided us throughout this thesis. A special thank you also goes out to Arm and several Arm employees, especially Patrik, Axel, Niclas and Tomas, who have provided answers to more technical questions and showed great interest in our work.

We would also like to thank our supervisors at LTH: Steffen Malkowsky and Lucas Ferreira, for their great support and valuable input during this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DC	Dynamic Control
DS	Dynamic Shape
FNN	Feed-forward Neural Network
FPS	Frames Per Second
FVP	Fixed Virtual Platform
HW	Hardware
IoU	Intersect over Union
IR	Intermediate representation
IREE	Intermediate Representation Execution Environment
ML	Machine Learning
MLIR	Multi-Level Intermediate Representation
MTCNN	Multi-task Cascaded Convolutional Neural Networks
NMS	Non-Maximum Suppression
NN	Neural Network
NPU	Neural Processing Unit
O-Net	Output Network
P-Net	Proposal Network
R-Net	Refine Network
RNN	Recurrent Neural Networks
SoC	System-on-Chip
TF	Tensorflow
TFLite	Tensorflow Lite
TFL micro	Tensorflow Lite Micro
TOSA	Tensor Operator Set Architecture

Chapter 1
Introduction

The use of machine learning (ML) has surged in popularity in recent years due to the expanding capability of the underlying hardware; The convenience and usefulness of machine learning has made it a popular tool in many industries across the world.

An often used machine learning variant is a neural network (NN). The NN is a defined system of layers containing nodes and the layers are connected via edges. There are typically weights on the edges between nodes and layers, which change during the learning process of the network and alters the output. Despite this, many NNs are still static in the sense that the number of layers, nodes and edges remain the same during program execution.

If one wishes to use conditional and/or loop statements to determine - during runtime, the outcome or architecture of a NN, larger requirements would be put on the software chain and underlying hardware; Thus, investigating the requirements and possibilities of implementing a NN with these properties on Arm's state of the art NN hardware accelerators [7] is of interest and the topic of this master thesis. A network with these properties is called a dynamically controlled NN. Even though many traditional networks are static, there is a high interest in dynamic control implementations according to Yuan, et al: "We analyzed more than 11.7 million unique graphs for machine learning jobs at Google over the past year, and found that approximately 65% contain some kind of conditional computation, and approximately 5% contain one or more loops". [8] These dynamic control properties are typically used either to efficiently implement ML algorithms, or to extend NN based algorithms with extra functionality during training and inference. [9]

Arm's neural processing units (NPUs) are hardware accelerators for common operations in NN algorithms. Therefore several well-known NN algorithms and custom implementations are possible to be executed on the NPUs. [7] Due to the extensible configurations of NNs, compability with the Arm NPU system is not guaranteed for all NN variants. The NPUs are implemented in a system that includes a central processing unit (CPU) capable of generic instructions. The operations that can not be performed by the NPU will be executed on the CPU instead. [10]

In order to investigate what would be required of the underlying hardware, the proposed algorithm *MTCNN* by K. Zhang et al [1] will be used; The algorithm is based on three dynamically controlled convolutional neural networks and is used

for facial recognition and alignment. This algorithm is of particular interest to investigate - due to its inherent dynamical properties.

1.1 Related Work

The subject of dynamic control in machine learning is being actively researched in several projects; With new technologies and frameworks developed as a response. This subsection will briefly introduce a few of these. Due to the early stages of these projects, they were not used in producing results for the thesis; However, they will be discussed later in the report as part of future work.

TOSA TOSA is an open-source tensor operator set architecture (early in development), comparable to the instruction sets in general purpose computers but with focus on NN operations. [6] TOSA is covered in more depth in Section 2.6.

MLIR MLIR is a compiler technology which simplifies the representation of dynamic control in the compiler stack. It also allows for easy lowering between different intermediate representations. [11]

IREE IREE is an end to end compiler solution for machine learning frameworks (Tensorflow, etc.) that aims to support dynamic shapes and dynamic control. IREE is being developed by a team at Google and is, at the time of writing, in a very early stage of development (implemented in MLIR). [12]

1.2 Master Thesis Contribution

As mentioned above, work and effort is being put into solving/easing the use of dynamically controlled NN algorithms. This is the motivation for investigating the topic - and implementation effort, of a dynamically controlled NN algorithm; The NPU itself is not the only point of interest, but the entire NPU ecosystem: Software toolchain and other components, such as the CPU. The questions in the following section will be the main point of evaluation for this thesis:

1.2.1 Research questions

- Is it possible to implement a dynamically controlled ML algorithm on the current Arm NPU product line?
- What is required of future/existing NPU architectures to make dynamic control possible?
- Does it give a performance increase when executing these algorithms on the NPU?
- What ML areas, and/or NN algorithms, benefit from dynamic control?
- Can offloading parts of the algorithm to the CPU be done efficiently?

- How does the NPU software toolchain compare to TOSA, IREE or other related frameworks?

1.3 Disposition

The disposition of this thesis is as follows: Chapter 2 will introduce the theory used to create results and discussion; This presents different machine learning frameworks, hardware used, description of neural networks and MTCNN.
Chapter 3 familiarizes the reader with the methods used to produce the results - presented in Chapter 4. These results are then discussed in chapter 5.
Chapter 6 will shortly present conclusions made for the research questions posed in Section 1.2.1.

Chapter 2
Theory

This chapter will familiarize the reader with the background theory required to draw conclusions from the results of this thesis. It will introduce the tools used in this thesis as well as theory about artificial neural networks, MTCNN and dynamic control.

2.1 Tensorflow

Before Tensorflow (TF) became public, it was an in-house tool at Google, named DistBelief, and provided ways of expressing ML algorithms and executing them. TF is now an open source venture (the next iteration of DistBelief) and a platform for machine learning. It is at the time of writing, actively being developed and worked upon by companies and individuals around the world. TF is interfaced via its libraries for different programming languages - C++ and Python. This enables modeling of NN algorithms and structures for different systems.

TF at its core is built around the concept of dataflow graphs, where the nodes are operations and the data crossing the edges are N-dimensional arrays of a defined type (called tensor) - an operation represents a computation on the given input tensors. The desired operation can then be implemented on a specific device, this is referred to as a kernel - the graph operators are unaware as to how they should transform the data. This enables TF to run efficiently on a wide variety of devices by the use of device specific kernels. [13] A TF graph can then be exported and executed on said devices using the kernel implementations; TF provides kernels for many operations [14]. An example graph can be seen in Figure 2.1, this graph perform an Add operation; Adding two tensors, a and b, as seen in Listing 2.1.

Figure 2.1 can be constructed with the following code:

```
1 def simple_graph(a, b):
2   return tf.add(a, b)
```

Listing 2.1: Simple example code of a tensorflow layer performing the add opertion.

Tensorflow also allows for eager execution - executing the operations in the language interpreter: For example the Python interpreter. This enables the full

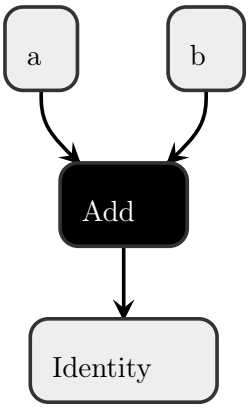


Figure 2.1: Simple TF graph

support of Dynamic Control (DC) that Python offers. This, however, means that portability to different systems not supporting the Python interpreter, will not work with eager execution. [15]

2.1.1 Tensorflow Lite

Tensorflow Lite (TFLite) is a collection of tools (see subparagraphs below) for TF. TFLite contains optimized operators for resource constrained devices; such as mobile phones and microcontrollers. At the time of writing, only a select subset of TF operations are supported by the TFLite converter. [16]

TFLite converter The TFLite converter optimizes, defines types, includes supported operations and then converts a TF graph into a flatbuffer. The flatbuffer is then easily transferred to other systems/configurations. [17]

Flatbuffer Flatbuffers are serialized objects (in our case - a TF graph with its operations, weights and connections). [18]

TFLite quantizer The TFLite quantizer performs quantization on a given TF graphs weights/activations and operation/tensor types. Meaning that the datatypes are changed according to the TFLite quantization scheme. Separate quantization layers can also be added to the model, which bridges the gap between different datatypes. [19] A quantized graph enables easier arithmetics and lower memory requirements on the targeted device - e.g TFLite can quantize the tensor types from *float32*, to *int8*, removing the need for floating point operations and alleviate memory requirements.

The decrease in tensor resolution post-quantization will give rise to a quantization error, which can negatively impact model performance. For int8 conversion, the theoretical maximum error is:

$$\frac{1}{2^{8+1}} = 0.19\% \tag{2.1}$$

The TFLite quantizer require a representative dataset to base the quantization on. This will be used to determine the value range and distribution of the model output, and perform the quantization to represent this value range. [19]

TFLite interpreter The TFLite interpreter will match kernels to operators and execute the flatbuffer produced by the TFLite converter. [20]

2.1.2 Tensorflow Lite Micro

TFLite micro is a subset of TFLite and used to run inference on microcontrollers. TFLite micro further constraints the operators used in the model that is being converted, since TFLite micro only supports a subset of the TFLite operators (TF kernels). [21] Graphs converted with TFLite containing only TFLite micro supported operators allows conversion to a C byte array that can be compiled on the intended micro controller. [22]

2.1.3 MLIR

At the time of writing, TF is being converted to adopt Multi Task Intermediate Representation (MLIR) as an intermediate representation (IR) in the compilation process. MLIR is a high level IR that enables, among many things - custom types, Dynamic shapes (DS) and DC. [23, 24] MLIR is analogous to a programming language and MLIR dialects are analogous to applications/programs written in said programming language. Dialects can then be used to lower the high level ML representations (e.g graphs in TF) to a compilation stage closer to the hardware. The features of MLIR (DS and DC) can thus be represented for a larger part of the compilation process; Later stages would then need to handle the aforementioned MLIR features.

2.2 Artificial neural networks

Artificial neural networks are inspired by neuroscience, and consists of layers containing artificial neurons. These are similar to biological neurons as each artificial neuron take one or more inputs and gives an independent activation value as output. This activation value is based on a combination of the weighted input value and a internal bias term, described in further detail in Equation 2.2. These weights and biases are adjusted during training of the network, often by using backpropagation. [25] [26, pp. 165, 200-207] A complete ANN typically consist of an input layer, output layer and one or more hidden layers. The hidden layers are the ones containing artificial neurons and they are called hidden layers as their input or output parameters are not directly accessed during training. [26, pp. 164-165] Each layer is connected by edges to the previous and following layer. This is illustrated in Figure 2.2. An additional neuron is usually introduced, with unary input and weight corresponding to the bias term, for the internal bias parameter of each layer. Meaning that the bias is commonly included when referring to the *weights* of a NN layer. [27]

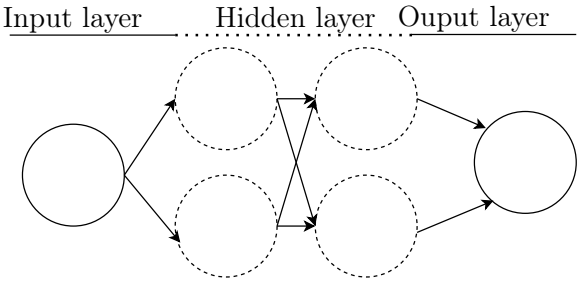


Figure 2.2: Simple illustration of a fully connected ANN

Two main ANN versions exist: Feed-forward neural networks (FNNs) and recurrent neural networks (RNNs).

FNN The name feed-forward is based on the network structure, where information flows through the network by sequential computations. Meaning that the output data from one layer is fed as input to the next layer. An example of this is displayed in Figure 2.2.

Each layer in an FNN is described according to:

$$\mathbf{h}(\mathbf{x}) = g(\mathbf{W}^T\mathbf{x} + b) \tag{2.2}$$

Where $\mathbf{x}$ is the input to the layer, $\mathbf{W}$ are the weights, b bias, g is an activation function and $\mathbf{h}$ is the output of the layer. [26, pp. 192-194, 207-208] There are several possible activation functions for a layer to use, the ones implemented in this thesis are presented under paragraph *activation functions* in Section 2.2.1.

A common subtype to FNNs is the convolutional neural network (CNN), which include layers with convolutional operations.[26, pp. 326] CNNs are commonly used for visual tasks such as: Image recognition and classification, as they have historically provided great results on this type of applications. [28, 29]

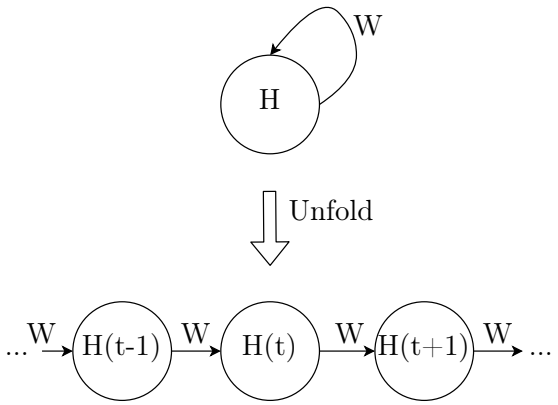


Figure 2.3: Example of unfolding a recurrent node.

RNN The difference between a RNN and a FNN is that the RNN includes feedback from later to earlier layers. RNNs can be described as operating on sequences of input data. Where the maximum sequence length is the limiting factor on how far back in time recurrence relations has to occur. For a finite sequence length, the RNN can be unrolled in the temporal domain and represented as an FNN. An example of unfolding, also called unrolling, a RNN layer is shown in Figure 2.3. The parameter t denote the timestep the current node is present in, going from 1 to τ. With τ being the maximum sequence length. H is an arbitrary node and W are the weights.

A generalization of the hidden RNN layers can be defined as:

$$\mathbf{h}^t = f(\mathbf{h}^{(t-1)} \cdot \mathbf{x}^t) \tag{2.3}$$

Where t is the current timestep, h is the output of the hidden layer and f is an arbitrary function. This clearly show the recurrence relation, as $\mathbf{h}^t$ reference to itself back in time: $\mathbf{h}^{(t-1)}$. [26, pp. 367-371]

2.2.1 ANN layers

This section gives a background to a selection of common ANN layers, which will be used in the later stages of this thesis.

Convolutional layer The convolutional layer is by definition present in all CNNs. It is primarily defined by its filters, or kernels, and kernel stride. The filters are matrices containing trainable weights, this allows for different results to be obtained by swapping filter between convolutional operations on the same input data. The input, output, stride, filter values and filter sizes are fixed for each layer.

$$S(t) = (P * K)(t) = \int P(a)K(t-a) \tag{2.4}$$

A single-dimensional, continuous convolutional operation is performed according to Equation 2.4. Where K is the kernel, P is an input parameter and S is the output of the operation. Where all input and output parameters are one-dimensional.

The input parameters are usually tensors (multi-dimensional arrays) in ML applications, making a multi-dimensional convolution the operation of choice. [26, pp. 328-329] As the ML application will operate on a finite set of data, discrete convolution is also desired. The convolutional operation, in Equation 2.4, have to be generalized according to these requirements before being adopted in a CNN.

$$S(i,j) = \sum_n \sum_m P(i-m, j-n)K(m,n) \tag{2.5}$$

A discrete 2D-convolution is described in Equation 2.5, which is the type of convolution that will be referred to later in this report. In Equation 2.5: K is the kernel, P is the input matrix, S is the output of the operation and i and j are the start row and column indices within the input matrix. [26, pp 336-339]

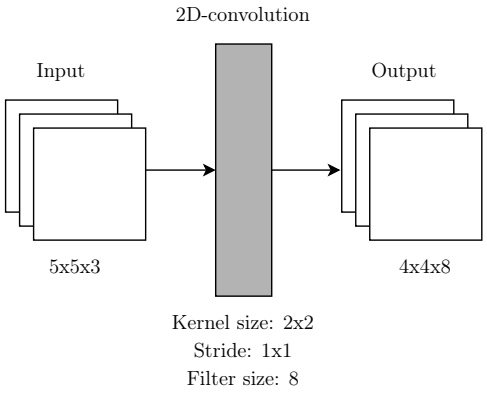


Figure 2.4: High-level overview of an example 2D-convolutional layer.

An example 2D-convolutional operation is shown in Figure 2.5 and an overview of the entire convolutional layer is shown in Figure 2.4. Where the input to the convolutional layer is a $5x5x3$ matrix, with a sliding kernel of size $2x2$ and stride $1x1$ is used. The stride of $1x1$ means that the sliding kernel moves one step between each convolutional operation. The filter size is 8, which can be observed in the output dimension as well.

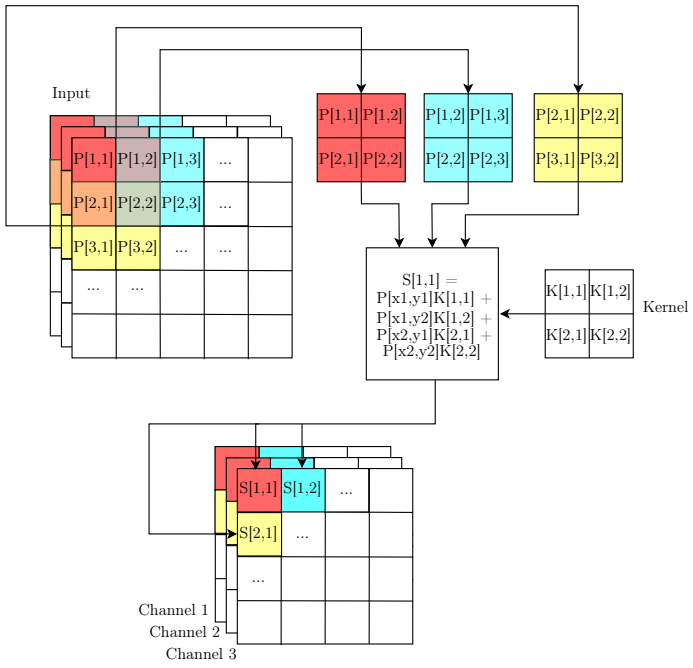


Figure 2.5: Example of a 2D-convolutional operation.

Referring to Equation 2.5, each output value can be derived as the sum of the multiplications between the kernel and input matrix. With the kernel sliding across the entire input matrix, giving one of the output values per operation. An example of this is shown in Figure 2.5, where K denotes the kernel weights, P input and S output values. With each kernel, or filter, having its own weights. Figure 2.5 shows the computation for three values in the output matrix. The remaining values are calculated in a similar manner by sliding the kernel position by 1 across the input matrix between each computation. [26, pp. 328-331] Each color on the input data in Figure 2.5 represent different kernel positions.

Note that the input is three-dimensional while the operation itself is 2D-convolution, meaning that the same 2D convolution is done separately for each of the three 2D-tensors on the input, also referred to as *channels*.

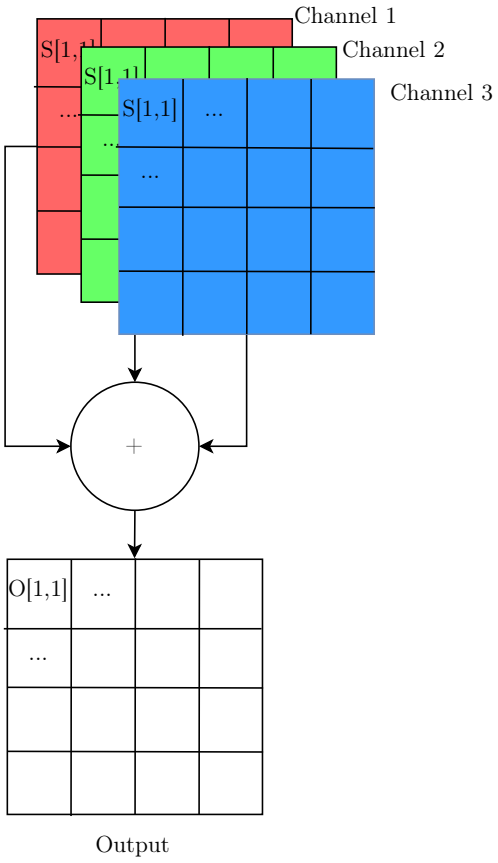


Figure 2.6: Accumulation of output of three-channel 2D-convolutional operation.

The resulting three separate output channels, from Figure 2.5, are summed up to form one single output tensor, illustrated by Figure 2.6. Where the red, green and blue matrices are the output of the 2D-convolutional operation on each

separate channel, and the bottom is the final output of the convolutional layer. Resulting in a single $4x4x1$ output matrix. [30] This procedure, presented in Figure 2.5 and 2.6, is then repeated for all different filters. Producing the final $4x4x8$ output shown in Figure 2.4.

Pooling layer A pooling layer is commonly used to make the output of another CNN layer invariant to small changes. It determines the value of each output node based that of its neighboring nodes, referred to as the *neighborhood*. Some variants of this is: Max Pooling layer and Average Pooling. Max Pooling sets the output values to the maximum within the neighborhood. This is implemented as a layer with a set stride and kernel size. Where the kernel is the size of the neighborhood and stride defines the movement of the kernel. Average Pooling functions the same way as the Max Pooling, except that it calculates the average of the neighborhood, instead of the maximum value.[26, pp. 335-339].

Fully connected layer The fully connected, or dense, layer connects all of the neurons between two layers. See Figure 2.2 for a simple illustration.

Activation functions Nonlinear activation functions are commonly used after any of the earlier mentioned layers, as described in Equation 2.2. These may or may not contain trainable weights. This report will cover ReLU, PReLU and Softmax activation layers: ReLU (Rectified Linear Unit) is defined according to Equation 2.6. [28]

$$f(x_i) = max(0, x_i) \tag{2.6}$$

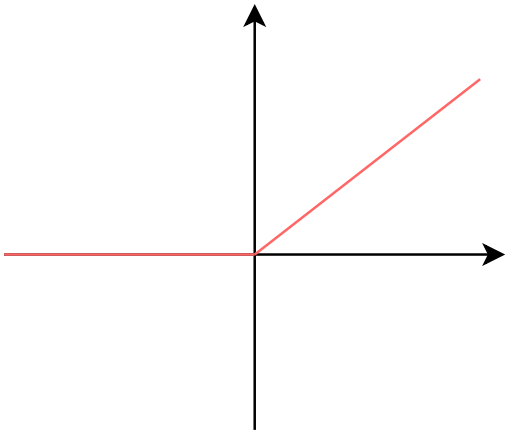
PReLU (Parametric Rectified Linear Unit) is a generalization of ReLU and is defined according to the equation:

$$f(x_i) = max(0, x_i) + a_i \cdot min(0, x_i) \tag{2.7}$$

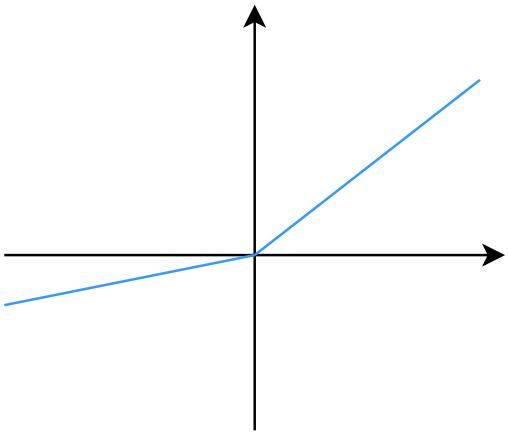
Where a is a vector of weights which are learned during training of the network. [31] The example PReLU presented in Figure 2.7b has the slope of a on input values below 0. Softmax is another nonlinear activation function, and is defined according to Equation 2.8.

$$f(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2.8}$$

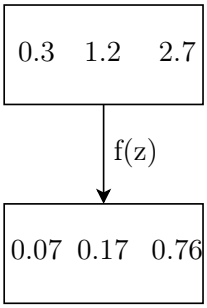
Where z_i denotes an element in the input vector z. The sum in the denominator in Equation 2.8 is a sum of all values in the input vector z. Softmax binds the output between 0 and 1. This property makes softmax common to use in NNs where the desired output is a probability distribution. [26, pp. 180-183] The three activation functions described above can be seen in Figure 2.7. With an example of a softmax operation in Figure 2.7c.



(a) ReLU graph.



(b) PReLU graph.



(c) Softmax illustration.

Figure 2.7: ReLU, PReLU and softmax activation functions.

2.3 Dynamic control

Dynamic control flow, or just *dynamic control*, means that the ordering of instructions in an algorithm can change dynamically during execution, depending on input data or memory states.

The fundamental dynamic control operators are related to ordering of operations, also called decision making, which are represented by conditional and non-conditional branch instructions on an assembly programming level. [32] These instructions then act as building blocks for other control flow operations present in high level programming languages. [33] There is also an important difference if DC is dependent on input data or not. If it is not dependent on input data, then the dynamic behavior can be resolved to static at execution time. For example by resolving branches and/or unroll loops at compile time. [32] On top of this: Operations that depend on input data does not have to be dynamic control operations.

It is only dynamic control if the result may alter the execution order of succeeding instructions.

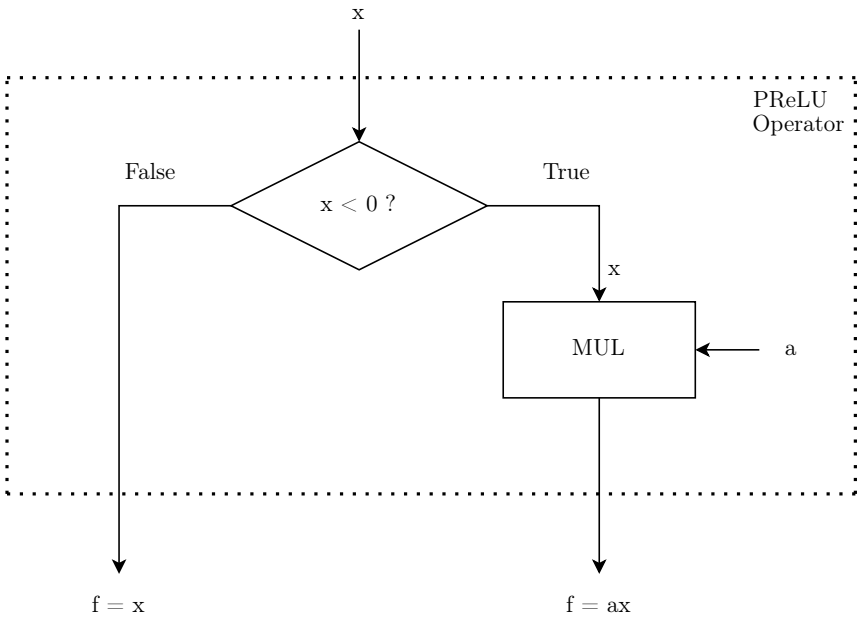


Figure 2.8: Example of a PReLU activation function implementation, using a conditional operator.

An example of a dynamic control in ML applications is the implementation in Figure 2.8. This is a dataflow graph describing a PReLU activation function, introduced in Section 2.2.1, with x input, f output and a being the weight parameter. As described by Equation 2.7. This is clearly dynamic control, as a decision is made based on input data, which determine the following instruction; MUL by a, or not.

If a variable has DS its datatype is not set at compile time, but is one of a set of options. This follows the earlier presented definition of DC, as doing the same operation but with different datatypes would utilize different instructions on an assembly programming level. [34] This differentiation between DS and other types of DC is made to easier categorize issues in higher level programming.

2.4 MTCNN

The algorithm presented by Zhang, et al. in: "Joint Face Detection and Alignment using Multi-task Cascaded Neural Networks" [1] is commonly known as *MTCNN* and is a ML algorithm used for face detection and alignment. As the name suggests, it is based on cascaded neural networks. The networks are dynamically controlled based on their input, which is a result of previous computations.

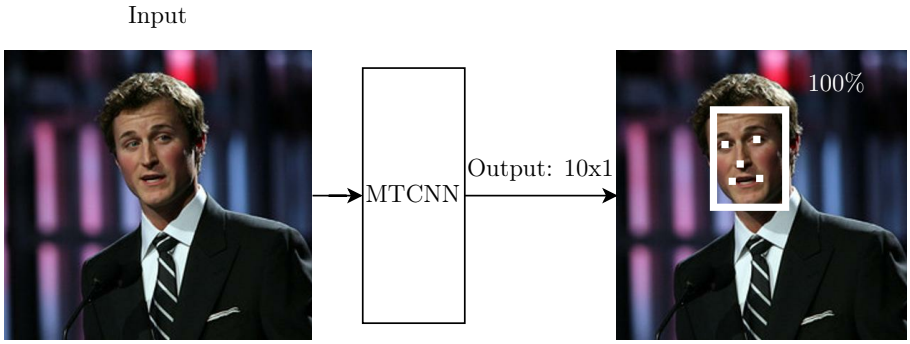


Figure 2.9: MTCNN example output. Input image is taken from the WIDER FACE dataset. [2]

An example execution of the MTCNN algorithm is presented in Figure 2.9. The input is an image and the outputs are the coordinates of the bounding box, facial landmarks and probability of the box containing a face, for each detected face in the input image. The bounding box, facial landmarks and probability are plotted on top of the input image in Figure 2.9, which is done post-MTCNN.

The structure of MTCNN is described in Figure 2.10. It consists of three CNNs: Proposal network (P-Net), refine network (R-Net) and output network (O-Net). Where P-Net is visible in the image pyramid layer, described in greater detail in Figure 2.11. These CNNs are designed and trained to perform different tasks. Additional pre- and post-processing have to be performed between each CNN to make the full MTCNN algorithm possible, each of these stages are visible in Figure 2.10. The purpose of these stages are typically: Removing false positive candidates, reshaping and calibrating the data to its proper format.
An in-depth description of each stage is provided below:

Normalize The input image consists of three arrays (R-, G-, B-channels) of pixel values between: 0 and 255. To simplify calculations in the upcoming stages, the pixel values are normalized to a value between: 1.0 and -1.0, ($[0, 255] \mapsto [-1, 1]$).

Image pyramid The full image is resized to form an image pyramid. This is required since the input size to P-Net is fixed; To find faces that are bigger than the input size of P-net, the image has to be resized to a smaller size. Making the entire face possible to fit in a single input array to P-Net. This stage is dynamically controlled since it should work for any image size, where the image size is the factor which decides both how many pyramid images are needed and how large each of the pyramid images should be. The image sizes in the pyramid can be described by the pseudo code in Listing 2.2. The overall structure of the image pyramid can be seen in Figure 2.11

Moreover, DC is required in the image pyramid to provide the correct amount of 12x12 image patches to P-Net. Also as a direct result of the image size.

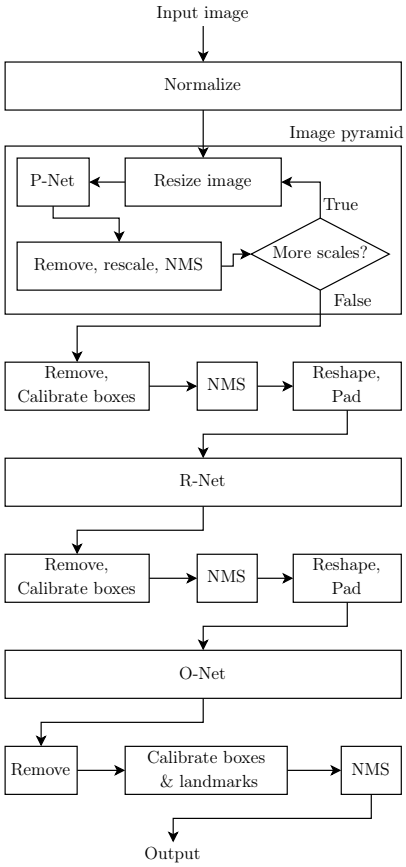


Figure 2.10: Detailed MTCNN algorithm structure.

Proposal Network P-Net is a shallow CNN which takes 12x12x3 patches of the images in the image pyramid, meaning that each patch is a 12x12 pixels wide part of the image on each of the RGB-channels. A stride of two pixels between each sample is used by Zhang, et al. [1] A stride of two instead of one greatly reduces the amount of computations that has to be done on each input image - the computational benefit can be seen in Equation 2.9.

$$N = (\frac{X - K}{S} + 1) \cdot (\frac{Y - K}{S} + 1) \tag{2.9}$$

Where N is the number of patches from an image of size X*Y with kernel size K and stride S.

The task of this CNN is to produce candidate bounding boxes which are likely to contain a face. For each input sample it outputs coordinates for a bounding box within the input, as well as the probability of it containing a face or not. The probability of the candidate containing a face is given as a 1x2 vector: With the probability of both containing a face and not containing a face, respectively.

```
i = 0
x = x_original
y = y_original
scales = []
while(min(x,y) >= 12){
   scale = 0.6 * power(0.709, i)
   i++
   x = scale * x_original
   y = scale * y_original
   scales.append(
      rescale(image_original, (x,y)))
}
```

Listing 2.2: Pseudo-code for scales in the image pyramid.

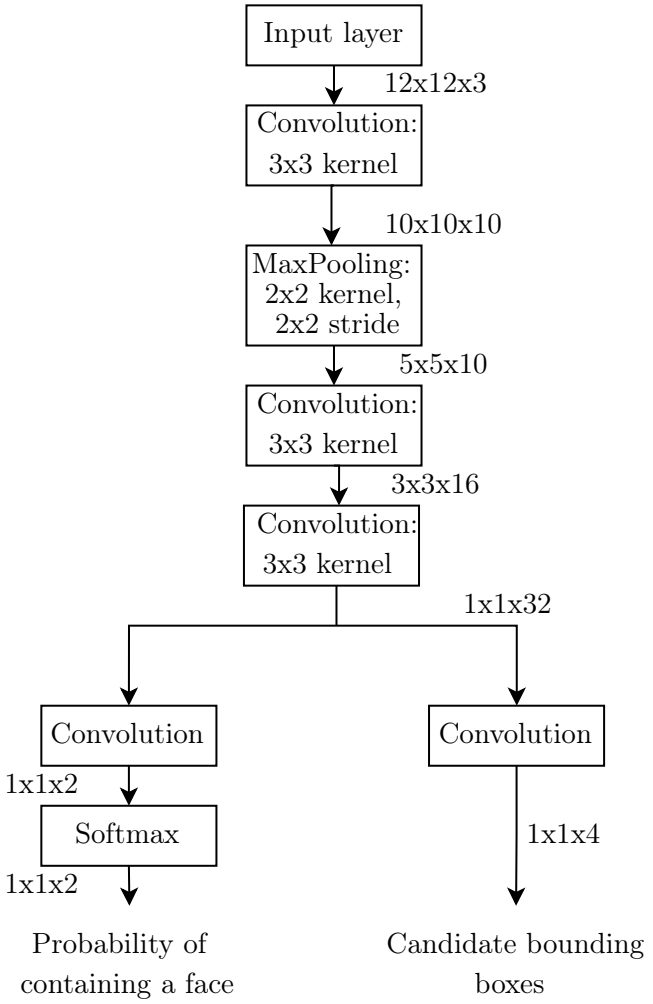


Figure 2.12: Proposal network structure.

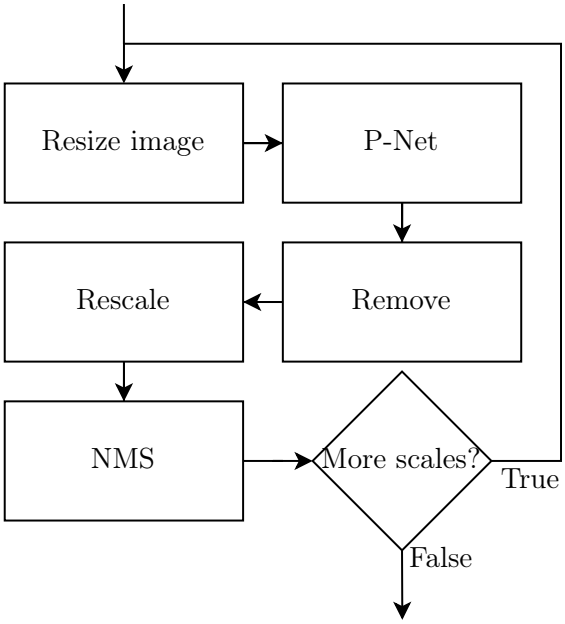


Figure 2.11: High-level algorithm description of the image pyramid.

Figure 2.12 show which neural network layers P-Net is constructed of as well as their input and output sizes. The stride and kernel size is 1 unless otherwise specified. Each convolution that is not immediately preceding the output layer is followed by a PReLU activation function, which is omitted from Figure 2.12.

Remove To avoid excessive computations, the boxes which are deemed by P-Net to have a low probability of containing a face are removed.

Calibrate boxes P-, R- and O-Net outputs the coordinates for each candidate bounding box within the input image it has received. These coordinates have to be calibrated so that they represent the position of the candidate within the original image. This can be done by keeping track of where each input patch sent to the CNNs is originated in the original image, and then summing up the input patch and its internal bounding box coordinates.

Non-maximum suppression (NMS) NMS is a technique used to remove highly overlapping candidate bounding boxes. This is needed since P-Net has a lot of overlapping input samples and the final bounding boxes should contain a unique face each. NMS is performed by comparing the IoU (Intersect over Union) between two boxes. IoU is calculated according to:

$$IoU = \frac{A_i}{A_u} \tag{2.10}$$

Where A_i is the intersection area and A_u is the union area between the two boxes. If the IoU is higher than a given threshold then the bounding box with the lowest probability of containing a face is removed. This is done incrementally, starting from the box with highest probability of containing a face, until all boxes have been compared to each other. The boxes are sorted based on their probability of containing a face to do this efficiently, since only two boxes can be compared at once.

Reshape After removing and merging candidates, the remaining bounding boxes has to be reshaped to a square with sides of equal length; In order to avoid losing parts of the image, the shortest pair of sides should always be extended to match the longest side pair.

Pad Because of the reshaping, some candidate bounding boxes may be extended to reach outside of the original image. These candidate will be padded, to still provide valid pixel values inside the entire bounding box. The padded pixels will have a value of 0.

Resize The resulting bounding boxes will be of varying sizes: As they come from different parts of the image pyramid and are altered by P-Net. Therefore the bounding boxes have to be resized to a size of 24x24 pixels before being fed to R-Net. With inter- or extrapolation if required. The resize stage use both the coordinates from the bounding boxes as well as the corresponding data from the stored input image. This is since R-Net require the pixel values of the facial candidates as input.

Refine Network R-Net is a deeper CNN compared to P-Net, with the main task of rejecting a lot of the false candidates it receives as input. It takes a bigger resolution (24x24) of the candidate determined by P-Net. Which results in R-Net being able to determine if the candidate contains a face or not with higher accuracy, but it also has a longer computation time for each input candidate, compared to P-Net. The outputs are candidate bounding boxes, within the provided 24x24 input, and their corresponding probability of containing a face.

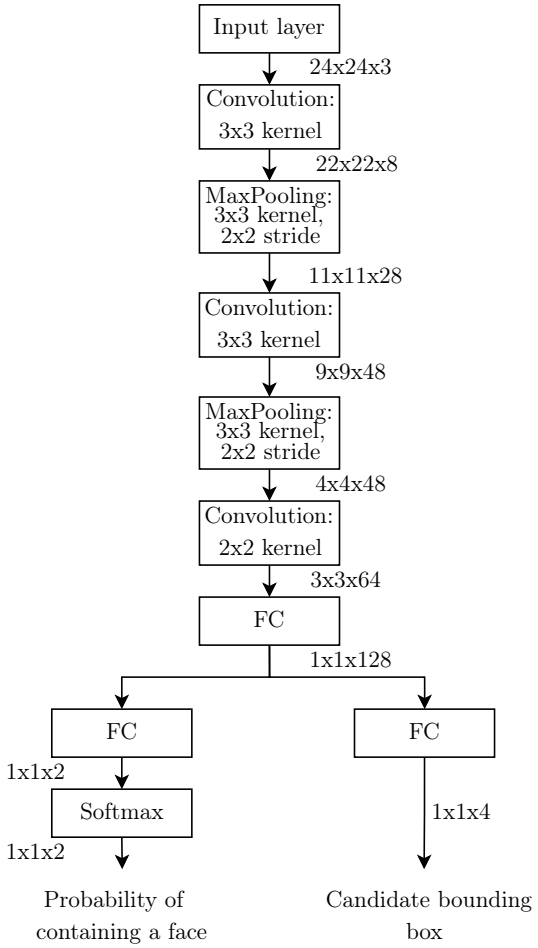


Figure 2.13: Refine network structure.

Figure 2.13 show the internal layers in R-Net, with the input and output sizes specified between each layer. There is a PReLU operation, omitted from the figure, after each convolutional or fully connected (FC) layer. With an exception for the output layers. All kernel and stride sizes are 1 if not stated otherwise. The output from R-Net is handled similarly to that of P-Net: The remove, calibrate, NMS, reshape, pad and resize operations are performed. Resize is computed in the same way as pre-R-Net, but with a size of 48x48 pixels instead of 24x24 this time. After that, the correct input can be provided to O-Net.

Output network O-Net is the deepest CNN in the MTCNN algorithm. Its main task is to produce the final output: Setting the final bounding boxes, their probabilities of containing a face and giving five facial landmark locations, for each candidate it receives at the input.

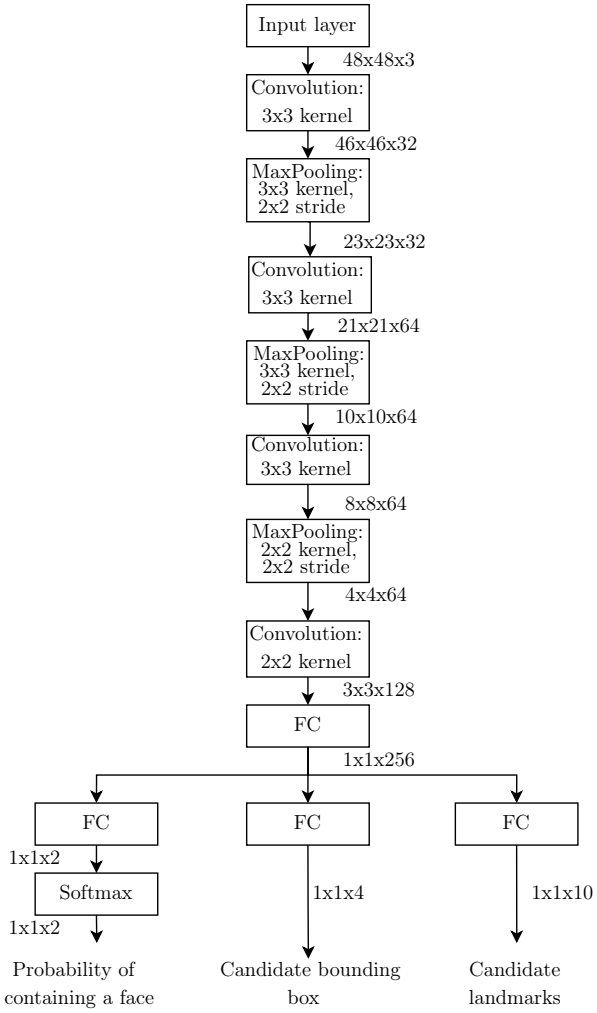


Figure 2.14: Output network structure.

The internal structure of O-Net is shown in Figure 2.14, where the input and output sizes of each layer is shown between the layers. Each fully connected and convolutional operation, except on the output layer, is followed by a PReLU activation function. All kernel and stride sizes are 1 where not stated otherwise. The bounding boxes produced by O-Net are calibrated and NMS is applied to remove any overlapping candidates. Candidates with low probability are also removed. Unlike the output from R-Net and P-Net, there is no need to reshape or pad any of the remaining candidate bounding boxes.

Calibrate landmarks The five facial landmarks provided as output from O-Net have to be calibrated, in a similar manner as the candidate bounding boxes. Each landmark is represented by two coordinates which represent their position within the corresponding bounding box. The bounding box coordinates are then

used to calibrate the facial landmark coordinates to where they are located in the original image, instead of in the bounding box.

2.4.1 Dynamic control in MTCNN

One of the strengths of MTCNN is that unnecessary computations are avoided by removing false candidates in the early stages of the algorithm. This happens in the remove/NMS stages, and means that it must be possible to decrease the number of candidates from the input batch between each stage. Another implication of this is that the number of iterations multiple stages have to run differ depending on the input image.

If zero potential candidate faces are detected, additional conditional statements must be inserted. This is due to the fact that *null* tensors causes runtime errors to be thrown, equivalent to division by zero; Instead, the algorithm is finished and should terminate. This can only happen after one of the remove stages, since all other stages with one or more boxes provided as input will result in at least one box on its output.

The size of the input image will dynamically control the normalize, image pyramid, calibrate, reshape and pad stages, in the sense that the number of operations that has to be performed are increased with an increasing image size. The depth of the image pyramid also depends on the image size. This means that the only static parts of the algorithm are the P-, R- and O-Nets. All other stages are dynamically controlled by one or more factors.

To summarize, dynamic control in MTCNN is related to the following:

- The amount of iterations, per stage, depends on the input image and the resulting amount of facial candidates produced.
- Candidate bounding boxes must be possible to remove.
- The algorithm should terminate if all facial candidates are removed.
- Input image size is variable and impacts multiple stages.

2.5 microNPU system

The hardware that will be evaluated is based on the Ethos-U Neural Processing Unit (NPU) from Arm, also referred to as *microNPU*. [35] It comes in two main variations: Ethos-U55 and -65, both with customer configurable parameters and implementation targets such as: Number of computational units (MACs), clock speed and memory size. Which allows for a wide range of future customer implementations with varying computational throughput, power consumption and usability. The microNPU has to be deployed in a larger system featuring a Cortex-M co-processor, referred to as CPU. [3]

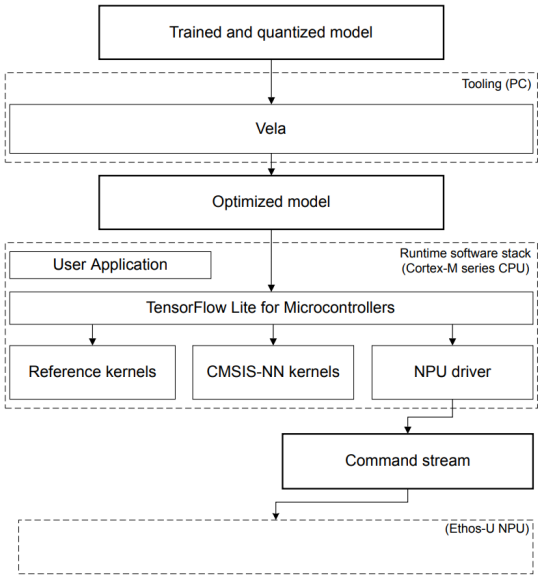


Figure 2.15: Software stack for the microNPU system. [3]

The software stack of the microNPU system is described in Figure 2.15. It is divided into two parts: Tooling and runtime software stack. Where the tooling part is required for making the higher level ML model executable on the NPU system and the software runtime is for running inference on the device. [3]

2.5.1 Tooling

The trained and quantized ML model provided as input in Figure 2.15 is a TFLite file, quantized to use 8-bit integer computations where possible. This is processed by the Arm compiler tool Vela before being deployable, as an optimized model, on the microNPU system.

2.5.1.1 Vela

Vela is an open-source tool to make the TFLite model executable on the microNPU. It takes a quantized TFLite file as input, performs optimization and scheduling and outputs a flatbuffer of commands to be executed on the microNPU and/or CPU. [36] Vela will schedule the operators supported by the microNPU [37] to be executed on the microNPU, with the possibility for unsupported operators to be scheduled on the CPU instead. [10]

2.5.2 Runtime software stack

The software runtime consist of kernels to run inference using Arm's CMSIS-NN libraries, TFLite micro kernels, possibly other user defined kernels and a user application, all running on the CPU. The user application can be any embedded

application capable of running on the CPU. The ML model, now optimized by Vela, is executed on the microNPU as a command stream. If only parts of the model can be accelerated by the microNPU, then the remaining parts will fall back to the CPU. [10]

The full process of making an ML algorithm executable on the microNPU will be discussed in greater detail in Section 3.

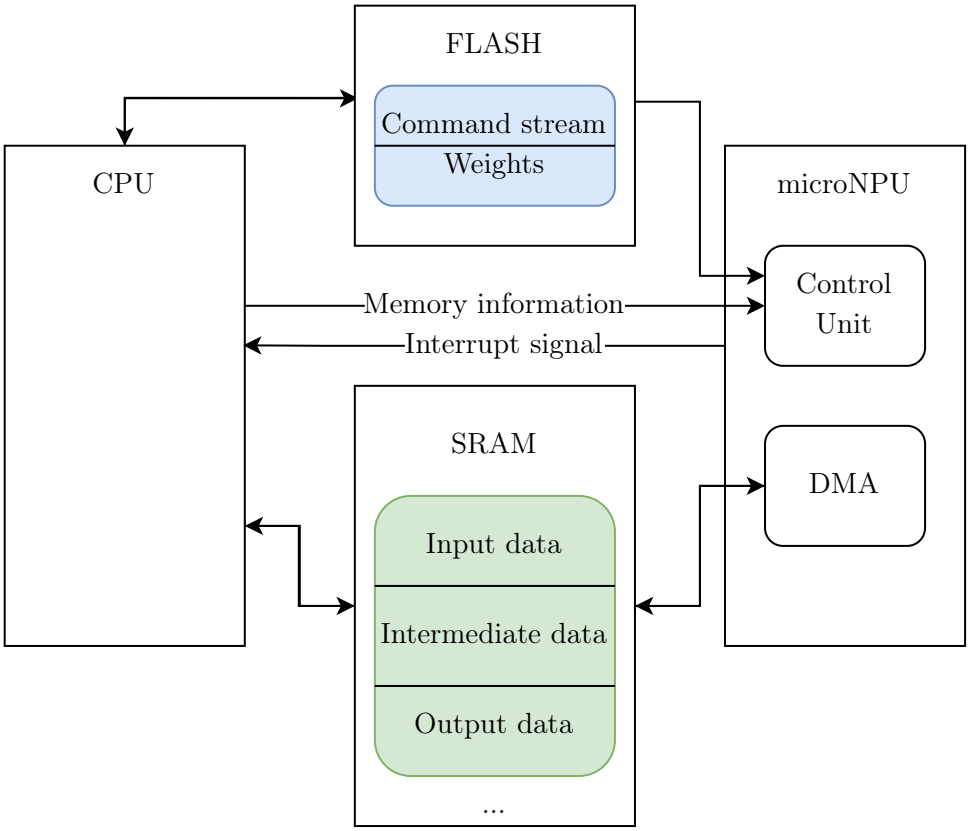


Figure 2.16: Overview of the communication between microNPU and CPU during inference.

Running an application on the microNPU could be done in multiple ways, for example by using different memory configurations. This thesis make use of the following flow:

The pre-compiled command stream and compressed weights are put in system Flash memory. Input data is put in SRAM, with space allocated for output data and storage of intermediate data. The CPU sends the memory locations of the command stream in Flash memory and allocated SRAM regions to the NPU and signals it to start execution. The microNPU then executes the command stream, storing the output and temporary data in the mentioned pre-defined space in SRAM, and sends an interrupt to the CPU when finished. This is illustrated in

Figure 2.16, with the DMA-unit of the microNPU accessing the SRAM regions (green) and the Control unit reading data (blue) from Flash based on the information sent from the CPU. [10]

This exerts hard requirements on the available SRAM and Flash memory, which has to be taken into account when developing these type of embedded applications. This will be further discussed in Section 5.3.3.

2.6 TOSA

Tensor Operator Instruction Set (TOSA) is an open-source operator set architecture, yet not finalized. It aims to standardize the primitive tensor operations used in machine learning applications, with focus on operations often used/found in neural networks. TOSA describes the low level operators in an algorithm and is separated from the implementing framework, e.g TF and Pytorch (Pytorch is another widely used ML framework). TOSA also provides the required type specification, data layout (tensor shapes), quantization and reference implementation for the different operators, this makes it possible for different entities to implement hardware that can execute the operators with the same outcome - enabling the possibility of TOSA compliant models/hardware. TOSA contains different profiles depending on the use case. If training is required, an extended version of TOSA would be necessary (the microNPU does not support training) - at the time of writing this extension is not included. The profiles also mainly target different systems depending on their type compatibility (integer and float) during inference. [6]

Table 2.1: Overview of TOSA profiles [6]

Profile	Purpose
BI	Integer inference
MI	Integer and float inference
MT	MI with training

In Table 2.1 the type support and descriptions of the different TOSA profiles can be seen; BI targets inference on smaller devices, such as a micro controller. MI targets inference on regular devices with support for floating number representation. MT is an extenstion on MI with support for training. [6]

Different frameworks handles quantization differently, e.g TF passes along the needed quantization data on the tensors. Quantization in TOSA is executed before/after the operator in a separate RESCALE operator. This allows for different quantization representations in the frameworks to be unified when lowering to TOSA and the same quantization can be expected across all TOSA supported models/hardware.[6]

TOSA supports *if* statements and *while* loops in terms of DC. However, no support for dynamically shaped tensors are described in the specification.

TOSA MLIR TOSA has been implemented as a dialect in MLIR. It is a standalone (separated from higher/lower compiler stages) IR that implements the tensor and quantization specification of TOSA. The input to TOSA MLIR would suitably be an IR from a framework, e.g Tensorflow lite.

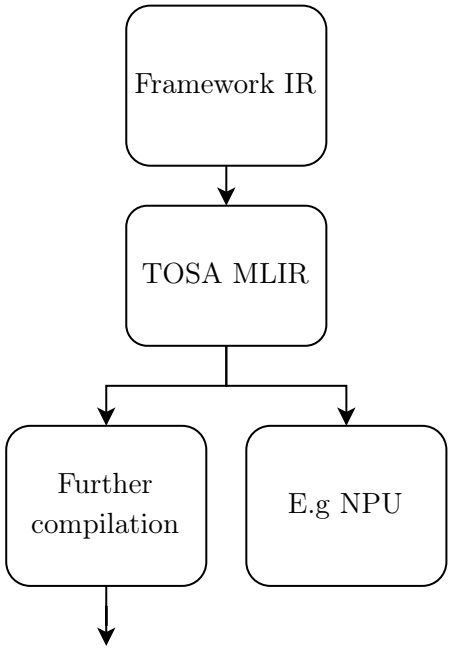


Figure 2.17: Example use of TOSA IR.

The output of TOSA MLIR can then be further compiled to different targets or directly target specialized hardware. [38]

Chapter 3
Method

To make the MTCNN algorithm executable on the microNPU, the following workflow was adopted: Dividing MTCNN into smaller layers/sections, convert each layer/section into a TF graph (using only native TF operators), quantize the supported graphs for int8 support and finally convert the quantized layers using Vela to a command stream for the microNPU; unsupported operators on the microNPU will fallback to the Cortex-M55 CPU. The aim of the workflow was to localize unsupported parts of MTCNN. The sublayers of MTCNN contained various operators, both supported and unsupported, in different steps of the software/hardware chain.

Verification and profiling could be performed at different steps in the implementation process.

3.1 Implementation workflow

In this section the workflow for model implementation is described in more detail.

Figure 3.1 show the workflow, and tooling, to go from an ML algorithm to executable program on the NPU. It is an extension based on the requirements for deploying and executing an ML application on the NPU system, presented in Figure 2.15. Figure 3.1 includes verification, which will be discussed in Section 3.2. It illustrates how the verification can be done at different stages in the implementation process. The verification then acts as base for decision making if the implementation is good enough or has to be changed in an earlier stage.

3.1.1 MTCNN model

The implementation of MTCNN, as described in the paper by Zhang et al. [1], as well as a python implementation [5] was studied and reimplemented anew in python with different structuring. The new structuring allowed for dividing MTCNN into smaller subsections and putting necessary constraints on MTCNN. These constraints are further discussed in the discussion section. The three networks in MTCNN: P-, R- and O-net, described in Section 2.4 and paper [1], were given weights [39] provided by the python implementation. [5] Thus removing the need for training the network using the method described in [1].

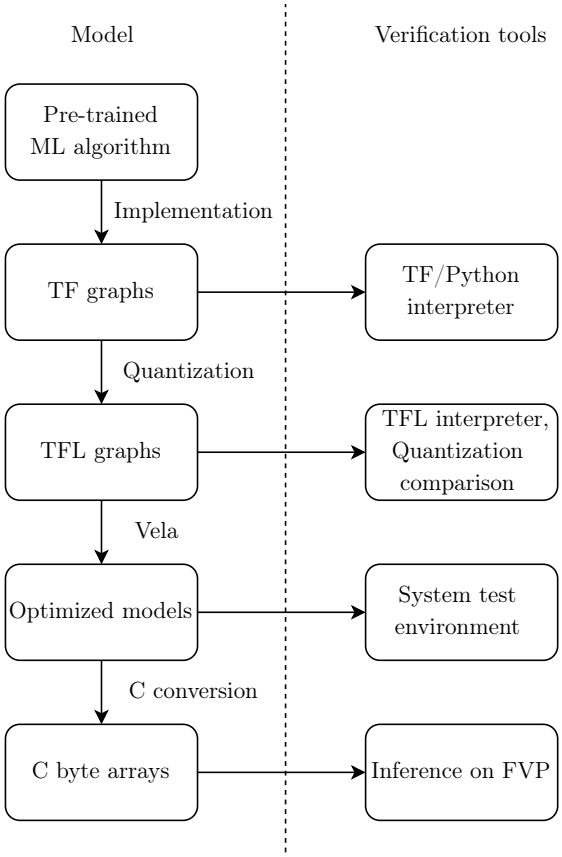


Figure 3.1: Overview of model implementation and verification workflow.

3.1.2 TF graph

The subsections could then be converted to TF graphs using the TFLite converter, described in Section 2.1.1. An example of this is presented in Appendix A. Which shows the TF graphs of the different layers described in Section 2.4.

3.1.3 Quantization

The TF graphs could be quantized using the TFLite toolset, described in Section 2.1.1, to use int8 types throughout the layer execution. Quantizing a graph exerts extra requirements on the graph and its operators, such as not allowing subgraphs. These requirements are presented and further discussed in Section 4.1. The graphs were implemented to use only TFLite supported operators. Operators with support for int8 data types were prioritized, since the NPU mainly support 8-bit integer operators [37], with fallback to use float32 data types where this was not possible.

3.1.4 Vela

The quantized TFLite graphs were then compiled using Vela. Producing an optimized TFLite model as output.

3.1.5 NPU inference

The optimized TFLite model could be converted to a C byte array using C conversion tools. [40] This makes the model possible to load into device memory and be called from the main application. Manual memory mapping was done for the memory regions that incorporate TFLite micro stages.

A C++ program was written to act as the main application for executing the MTCNN algorithm. This application calls the individual MTCNN stages using the TFLite interpreter, handles intermediate data between each stage and execute the operations that cannot be executed on the NPU. All according to Section 2.5.2.

3.2 Verification and profiling

Each step of the implementation process, described in the section above, can result in drawbacks in model performance. This makes verification an important part of model implementation.

3.2.1 Full model in TF

The reimplementation of MTCNN was built using TF operators and 32-bit floating point data types. After being converted to a TFLite file, with fallback to standard TF operators if needed, inference could be run using the TFLite interpreter [20] with fallback to the Python interpreter for TF native operators. The result was evaluated based on images from the Wider Face dataset [2], it was also compared to the python MTCNN implementation [5] by feeding both networks with the same input image and comparing the output difference. This was done on the full MTCNN model, described in Section 2.4, with the constraints presented in Section 5.2.

3.2.2 Quantization

After quantization from float32 to int8, evaluation was performed by inspecting the quantization error of each stage. This quantization error was derived by subtracting the output from the quantized model with that of the non-quantized model, when running inference on the same input. To do this comparison, the input and output data types must be the same. So the quantized model was extended with a quantization layer from float32 to int8 before the input layer, and a dequantization layer from int8 to float32 after the output layer. The quantization and dequantization layers could be automatically generated using the TFLite quantizer, described in Section 2.1.1.

3.2.3 Vela

Vela produces performance estimates such as memory consumption and inference time. It also gives detailed information about the scheduling made: Operator utilization on the microNPU and CPU.

Arm provides models that can be used for verification and profiling of applications running on an Ethos-U system. [7] Similar models also exist for Arm CPUs, such as the Cortex-M55 [41] and Cortex-M7. [42] These models can act as building blocks to create a fixed virtual platform (FVP), which is a model of a complete platform based on Arm processors, memory and other peripherals [43]: such as the Corstone-300 FVP, which is based on the Cortex-M55 CPU and Ethos-U55 NPU. [44]

A system test environment, provided by Arm, was used to verify performance of the individual stages. The test environment receives a quantized TFLite file as input, build the necessary files using Vela and other tools and run inference with the TFLite micro interpreter on the NPU/CPU models. The TFLite micro inference results are then compared against the results of the original TFLite file running on the TFLite interpreter: With the result being deemed as successful if the TFLite and TFLite micro results are bit-exact to each other.

It also generates profiling information such as memory consumption, inference time; including latency and stalling, and operator usage.

3.2.4 NPU inference

The Corstone-300 FVP was used to verify performance of the stages not possible to be accelerated on the microNPU. This was done by executing the C++ main application, with input generated by the TF-model, on the FVP and comparing the output to the expected. Profiling could be done with regard to inference time and to see that enough memory could be allocated for the individual MTCNN stages to run on the FVP.

Inference was run on selected stages accelerated by the microNPU as described in Section 3.1.5. This was used as a proof-of-concept and not as verification for executing these stages, as they were already verified by the system test environment.

Chapter 4
Results

This chapter will present the results obtained from the investigation made in this thesis. It will cover the implementation results, performance measured for different hardware configurations as well as validation results from our implementation of *MTCNN*.

4.1 Implementation results

Tensorflow supports dynamic shapes and control in the form of operators, such as the TF *while* operator. When lowering the graph to TFLite with the supported operators that accompanies this dialect of the compilation, more restrictions are put on the availible operator set. TFLite also supports subgraphs, these are created by conditional statements and blocks. See Figure 4.1, where subgraphs are created by the body of the loop and the conditional statement of the while operator.

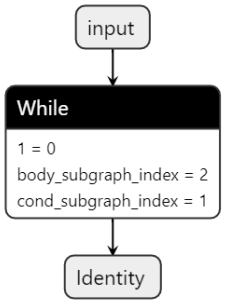


Figure 4.1: TF *while* loop, displayed using the visualization tool: Netron. [4]

When quantizing the TFLite converted TF graph, further constraints are put on the graph. Subgraphs are no longer supported and thus dynamic control in the form of loops are not supported either. Although, a work-around to this is possible by unrolling the loop, this is only valid if at compilation time the iteration count can be calculated. This is demonstrated in Figure 4.2.

Unrolling the loop removes the need for subgraphs (body and conditional) and

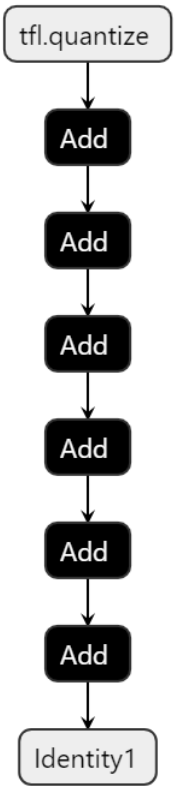


Figure 4.2: Quantized *while* loop unrolled 6 times, shown in Netron.

if the iteration count is unknown at compile time, an error is thrown by the TFLite quantizer and quantization of the graph fails due to the lack of dynamic support.

Converting the model to TFLite micro then further constrains the available operators. Operations on unknown dimensions of a tensor is unsupported and does not work, as this would require changing the tensor shapes during interpretation of the graph. By default, the shape of an unknown tensor becomes 1. A graph instantiated with unknown input shapes with defaulted tensors to (1, 1) can be seen in Figure 4.3. Running inference on a model with unknown shapes using the TFLite interpreter requires a resize of the input tensors to fit the input data.

This also registers an error when compiling the quantized TFLite converted graph with Vela as it detects the wrong input/output shapes. No work-around to this was found and thus the entire MTCNN model could not be executed on the microNPU as a single graph. The individual sub-layers could still be evaluated, with fallback execution on the CPU where required, and the following section will include results for this.

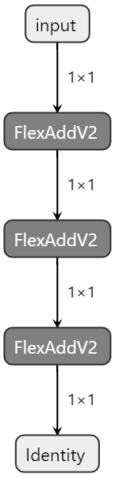


Figure 4.3: Quantized while loop unrolled 3 times with unknown input shapes, shown in Netron.

Table 4.1: Individual stages of the MTCNN implementation.

MTCNN stage	CPU	NPU
Normalize		X
Image pyramid	X	
P-Net		X
Remove	X	
NMS	X	
Scales		X
Calibrate boxes		X
Reshape		X
Resize	X	
Pad	X	
R-Net		X
O-Net		X
Calibrate landmarks		X

The sub-layers, or stages, the MTCNN algorithm had to be divided into are presented in Table 4.1, where it is also marked if the stage could be accelerated by the NPU, or has to be fully executed on the CPU. The individual stages are the same as those the MTCNN algorithm was divided into in Section 2.4.

4.2 Performance

4.2.1 Individual MTCNN sub-stages

Table 4.2: Hardware configurations used throughout this section.

HW configuration	MAC units	Clock frequency
HW1	32	500 MHz
HW2	128	500 MHz
HW3	512	1 GHz

The different sublayers of our MTCNN implementation was evaluated on the HW configurations presented in Table 4.2, which will be used throughout this section. All configurations include a Cortex-M55 as co-processor, referred to as *CPU*, with different microNPU configurations.

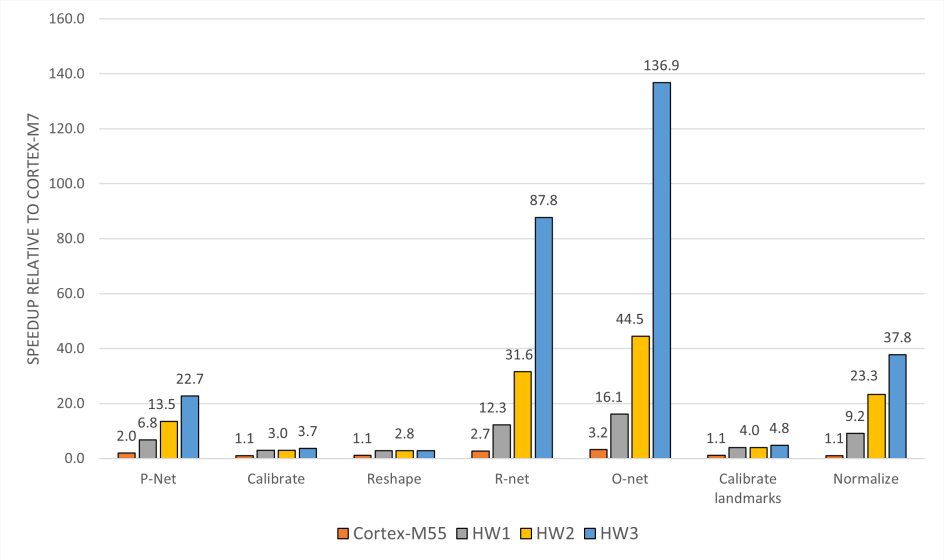


Figure 4.4: Relative speedup (cycles only) on different HW configurations for selected MTCNN stages.

Figure 4.4 presents the speedup of running inference for selected MTCNN stages, on the different HW configurations in Table 4.2 and the Cortex-M55 CPU. Only the stages possible to implement on the microNPU has been included in this figure. The speedup is calculated by dividing the inference time from Cortex-M7 with the inference time for the target HW. Meaning that the Cortex-M7 is used as baseline with the relative performance being shown for the other HW configurations. The inference time is measured in clock cycles only. Different clock frequencies between the processors would affect the runtime and will be further mentioned in Section 5.

Table 4.3: Memory footprint and operator utilization for selected MTCNN stages running on an Arm microNPU system.

Stage	SRAM	Flash	CPU/NPU OPS
Normalize	3.1 KB	0.4 KB	0/9
P-Net	5.3 KB	13.5 KB	0/81
Calibrate boxes	0.52 KB	0 B	0/31
Reshape	0.49 KB	1.3 KB	0/23
R-Net	47 KB	136 KB	0/91
O-Net	204 KB	463 KB	0/104
Calibrate landmarks	0.62 KB	3.2 KB	0/67

The NPU HW configurations used in the benchmarking is shown in Table 4.2. All are used alongside a Cortex-M55 co-processor, called *CPU* in Table 4.3. The memory footprint for on-chip SRAM, Flash memory and NPU/CPU operator utilization, for each stage that could be accelerated by the microNPU, is shown in Table 4.3. These performance metrics are *uncorrelated* to which HW configuration is being used, given that a microNPU is present. The Flash memory utilization metrics are only estimates, while the SRAM usage and operator utilization were derived during model verification.

Table 4.4: Profiling of the normalize stage in the MTCNN algorithm, running on the target HW from Table 4.2.

Target HW	NPU cycles	Memory access
HW1	1	0.47
HW2	0.81	0.75
HW3	0.69	1

To obtain more information about the speedup measurements in Table 4.4, further profiling was performed on selected stages and can be seen in Table 4.4 for the normalize stage. The NPU execution and memory access results are presented as their proportion of the total cycles when running the algorithm. These can run in parallel, which is the reason why their sum is greater than 100 %. The NPU execution and memory access values are only estimates, the actual utilization when running the model may differ.

4.2.2 Increasing performance

4.2.2.1 Custom PReLU operator

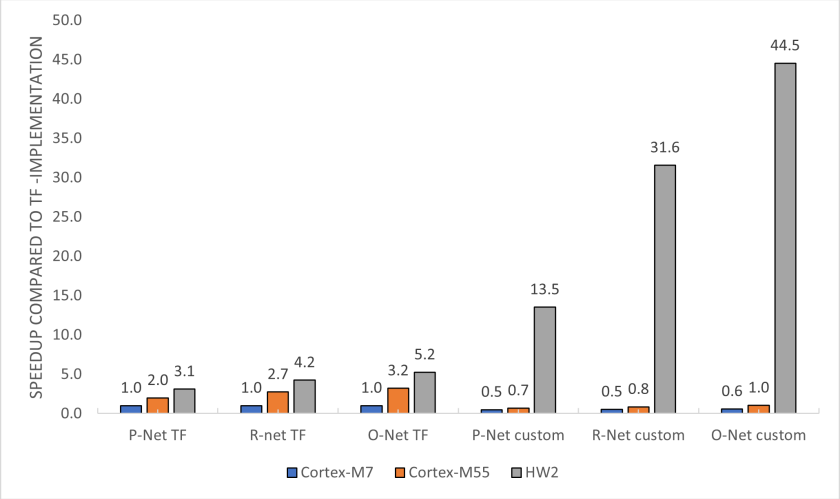


Figure 4.5: Speedup of custom and default (TF) PReLU implementation.

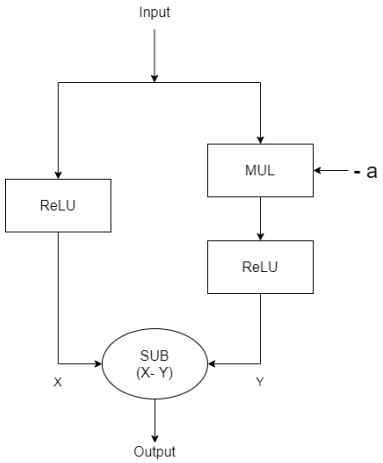


Figure 4.6: Custom PReLU operator implementation.

Figure 4.6 show the dataflow-graph of the custom PReLU operation, where a is the weight. It conforms to Equation 2.7, presented in Section 2.2.1. Since multiplying by $-a$ will result in a positive value if the input is negative, and a value at 0 or below otherwise. This implementation is only utilizing: MUL, SUB and ReLU operators. All these operators are supported by the NPU. [37] The code and

Table 4.6: Iterations per stage in the MTCNN algorithm.

# Faces	Normalize	P-Net	Calibrate boxes	Reshape	R-Net	O-Net	Calibrate landmarks
0-9	2186	126726	655	650	628	21.8	4.5
10-19	2186	126726	937	922	861.4	60.7	14.5
20-29	2186	126726	1191	1166	1068	97.7	24.5
30-39	2186	126726	1305	1270	1149	121	34.5
40-49	2186	126726	1535	1490	1345	145	44.5

TFLite graph for this implementation can be seen in Figure A.7, in Appendix A.

Table 4.5: Custom and TF PReLU operator placement and memory usage.

Stage	PReLU	CPU/NPU OPS	Memory footprint
P-Net	TF	3/80	3.0 KB
R-Net	TF	4/89	28 KB
O-Net	TF	5/101	137 KB
P-Net	custom	0/81	5.3 KB
R-Net	custom	0/91	47 KB
O-Net	custom	0/104	204 KB

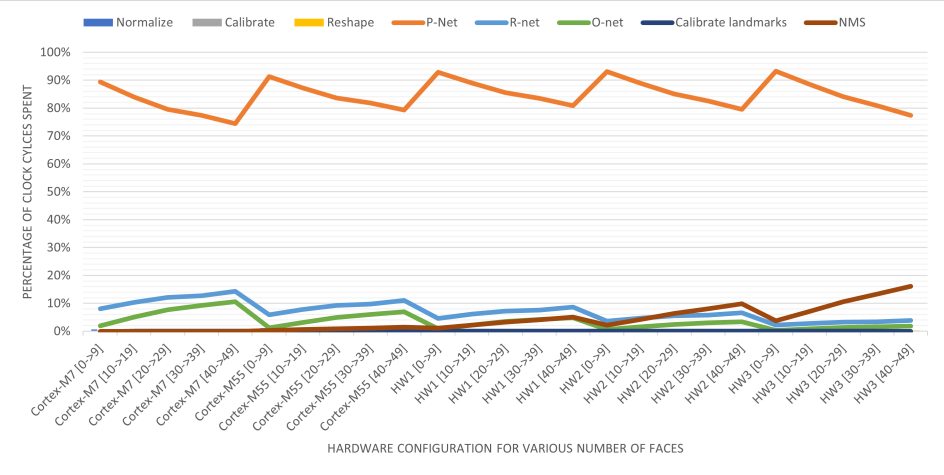
The performance of the MTCNN stages using PReLU: P-, R- and O-Net, is shown in Table 4.5 and Figure 4.5. All values in Table 4.5 are taken using HW configuration HW2 from Table 4.2. *TF* in the PReLU column refers to the default tensorflow PReLU-operator [45], *custom* means that our custom PReLU implementation is used. The memory usage refers to required SRAM.
The speedup values in Figure 4.5 are normalized by dividing each value with that of the same stage (P-, R- or O-Net) utilizing the TF-PReLU operator being executed on the Cortex-M7. The other results in the report, such as Figure 4.4, utilize the fastest available PReLU implementation for its underlying HW: P-, R- and O-Net with custom PReLU when executing on the NPU, and TF-PReLU when executing on the Cortex-M7 or -M55.

4.2.3 Execution time

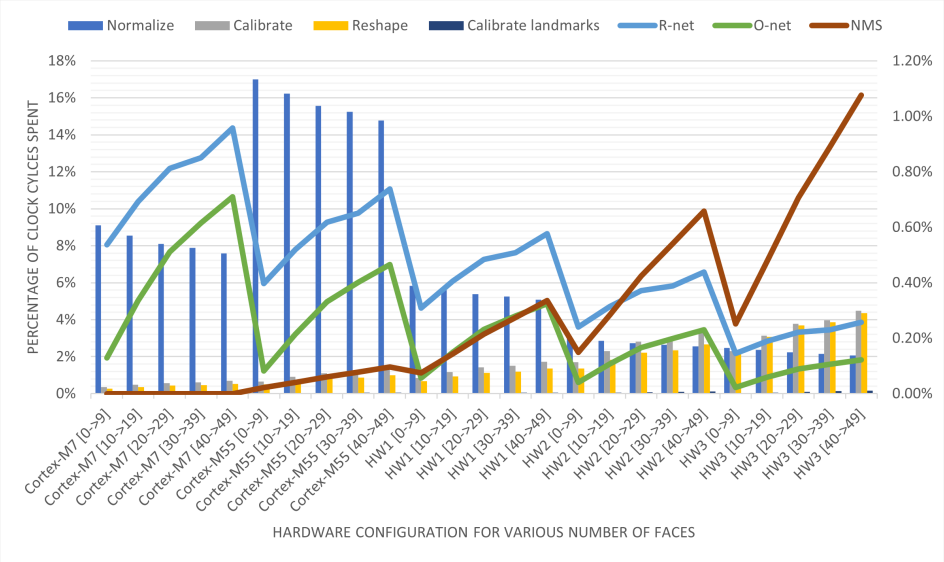
It was investigated if there is a correlation between the number of faces in the input image and the total MTCNN execution time, and the result is presented in Table 4.6; The average amount of iterations for each stage in the MTCNN algorithm for different amount of faces in the target image is shown. Only the stages accelerated by the NPU are included in this table.

These values were derived using 2500 test-images from the WIDER FACE dataset [2]. Since different stages have different execution times depending on the target HW it is executed on: The proportion of the total execution time for each stage is shown in Figure 4.7a. These values are based on the iterations per stage presented in Figure 4.6 and the cycle count for each stage when executed on different HW configurations, seen in Figure 4.2.

The average throughput of the MTCNN algorithm is presented in Figure 4.8 and Table 4.7, in units of frames per second (FPS). These values are based on



(a) With P-net



(b) Without P-net

Figure 4.7: Relative time spent by each layer, on different HW configurations with a varying amount of faces detected. The bars correspond to the right axis and the lines correspond to the left axis in Figure 4.7b.

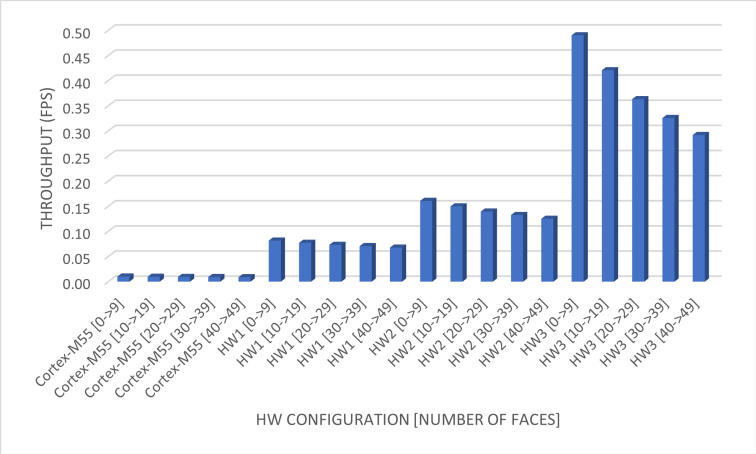


Figure 4.8: Throughput (FPS) of the MTCNN algorithm.

Table 4.7: Overview of throughput for MTCNN and other algorithm implementations.

Implementation	Algorithm	Throughput (FPS)	# Faces	Resolution	HW	Power requirements
This thesis.	MTCNN	0.49	0-10	561x561x3	HW3	10-100 mW
Centeno. [5]	MTCNN	4.5	1	561x561x3	Intel i7-3612QM @ 2.1 GHz	35 W [46]
Centeno. [5]	MTCNN	3.4	10	736x348x3	Intel i7-3612QM @ 2.1 GHz	35 W [46]
Zhang, et al. [1]	MTCNN	99	-	640x480x3	NVIDIA Titan	600 W Power supply [47]
This thesis.	MobilenetV2	467	-	224x224x3	HW3	10-100 mW
This thesis.	MTCNN	11.15	0-5	128x128x3	HW3	10-100 mW
STMicroelectronics. [48]	Person detect	6.8	0-1	128x128x3	STM32H747I @ 400 MHz	~600 mW [49]
Capotondi, Rusci. [50]	Image classification	6.1	-	160x160x3	STM32H743ZI @ 400 MHz	~600 mW [49]

execution using the HW configurations presented in Table 4.2, with a Cortex-M55 CPU. For a varying number of faces (0-49, in steps of 10 between each data point) in the input images. The CPU clock frequency is assumed to be 216 MHz for all HW configurations, and the microNPU clock frequency are presented in Table 4.2.

All clock frequencies are assumptions and can differ for HW based on the same settings. The throughput results are derived by accumulating inference results for all individual stages of the MTCNN algorithm, including the ones falling back to the CPU. The implications of this will be further discussed in Section 5.3.6.

Table 4.7 display the throughput and related metrics, such as resolution and HW platform, for different implementations of MTCNN and other NN based algorithms. With *resolution* and *#faces* meaning the image resolution and number of faces in the image. The person detect model is based on MobileNetV2, and the image classification is based on MobileNetV1, which is discussed further in Section 5.6.2. Details about the HW configuration for our MTCNN implementations can be seen in Table 4.2. The power requirements of the different implementations

in Table 4.7 is also to be noted. As the system investigated in this thesis is an embedded system, with requirements of lower power consumption compared to that of a desktop GPU, for example; The column stating the power requirements are estimates of the total power consumption for the different hardware configurations and should be seen as a way to compare the efficiency of these. The power consumption of the system investigated in this thesis are based on internal investigations made at Arm. The power requirements for the STM32H7 microcontrollers in Table 4.7 are derived based on typical current consumption at 400 MHz; With all peripherals enabled and code execution from Flash memory with cache enabled, multiplied by the typical operating voltage of the system.

4.3 Validation

Validation of the implemented model (only TF operators, running eager execution) compared to a python MTCNN implementation by I. Centeno [5], was done by running inference on the same images and comparing the outputs. As described in Section 3.2. Figure 4.9 portrays the absolute difference in bounding boxes coordinates over 500 images from the WIDER FACE dataset [2].

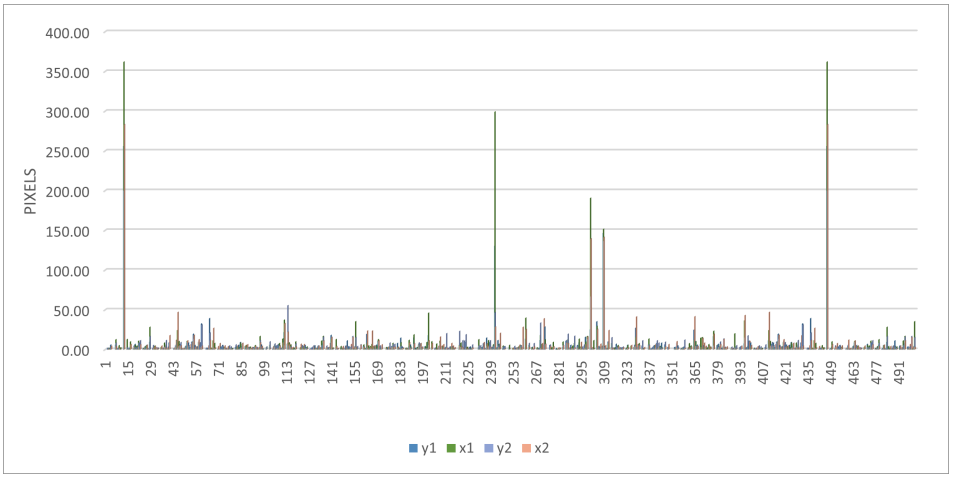


Figure 4.9: The absolute error over 500 images from the WIDER FACE dataset. [2]

Figure 4.9 displays five outliers in the data and these will be discussed further in Section 5.4. The average absolute error, in number of pixels, for the four corners of every bounding box is seen in Figure 4.10.

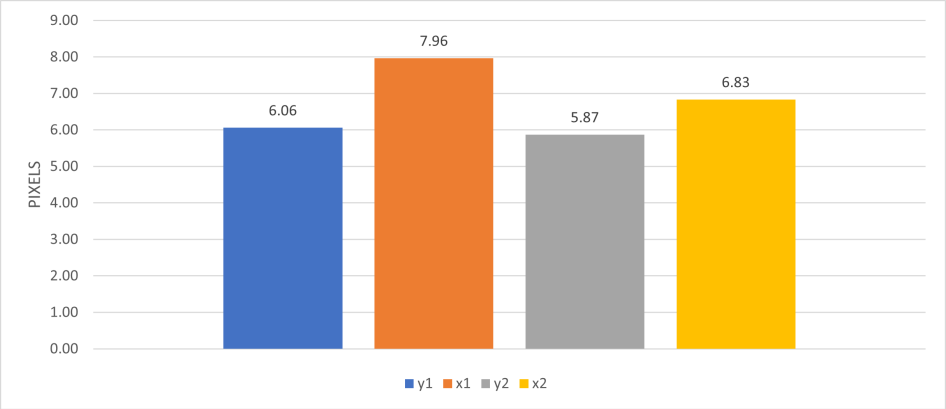


Figure 4.10: The average absolute error for Figure 4.9.

The average value was calculated across the absolute error (output delta) for the 500 images. Removing the five outliers (absolute error above 60) resulted in Figure 4.11.

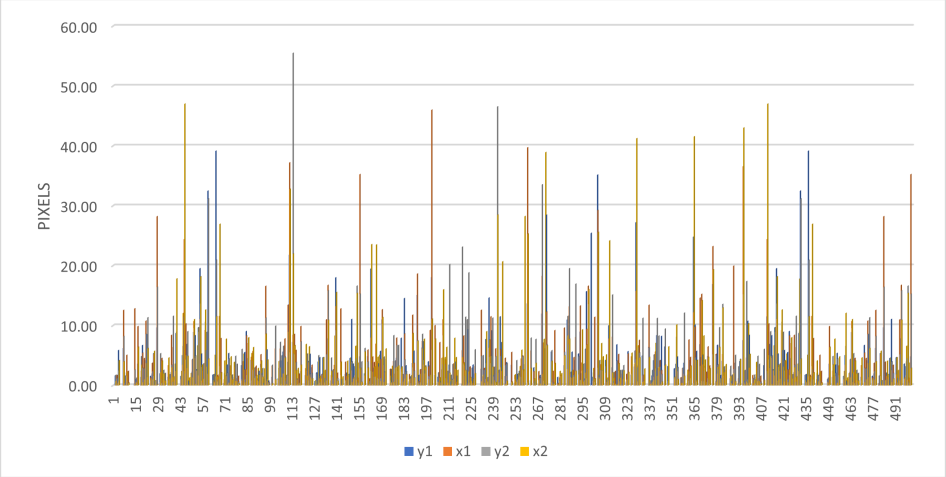


Figure 4.11: Figure 4.9 without the outliers.

The average absolute error, in number of pixels, without the outliers can be seen in Figure 4.12. With the outliers classed as failed detections; 69 out of the 500 images had too few faces detected and 29 had too many. Resulting in an accurate detection rate of 80.4% for the custom model compared to the implementation by I. Centeno [5] in terms of detected faces. The average error in confidence was: 0.89 %.

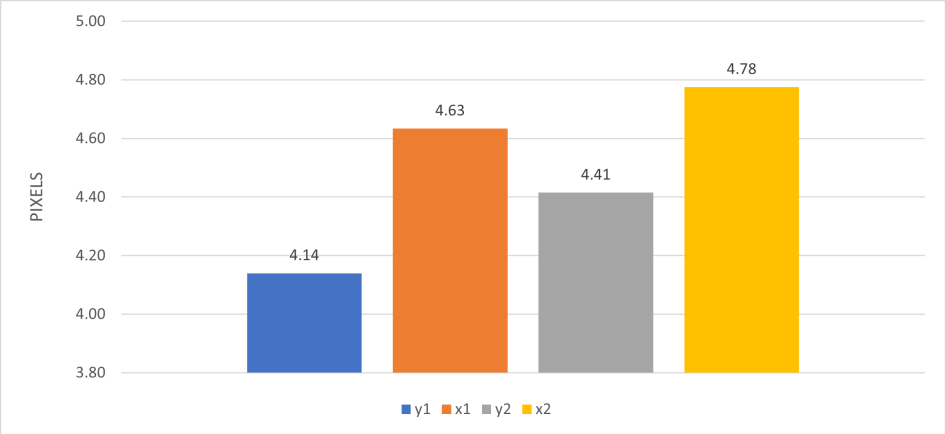


Figure 4.12: The average absolute error for Figure 4.11.

Figure 4.13 display the difference between the detected bounding box by our implementation (red), and the python MTCNN by I. Centeno [5] (blue), for an example image from the WIDER FACE dataset. [2] The image resolution is 561x561, which conforms to both algorithms. The average difference between each endpoint of these two boxes are 4.6 pixels, which is close to the average difference presented in Figure 4.12.

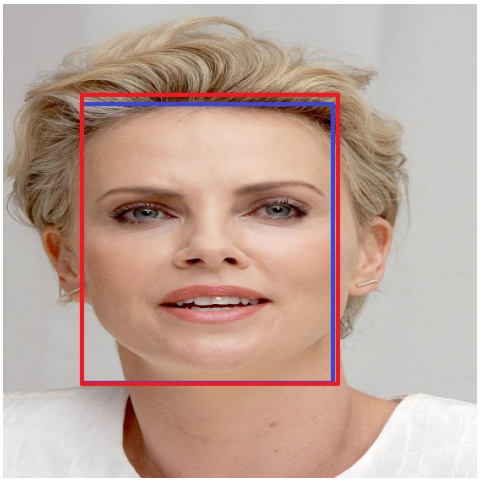


Figure 4.13: Bounding boxes from our MTCNN model (red) and a reference implementation [5] (blue), on a sample image from the WIDER FACE dataset. [2]

4.3.1 Quantization

In order to find the impact of quantizing the different layers, to use int8 weights and tensor types, the quantization error was computed; using the method described

in Section 3.2.2. The magnitude of this error represents the information loss via quantization.

Table 4.8: The average difference for quantized stages. P-, R- and O-net displays the difference for the coordinates and the confidence.

Layer	Tensor range	Average Difference
Normalize	[-1,1]	0.0039
P-net	[0,1]	0.0156, 0.0082
Calibrate boxes	[0,561]	2.39
Reshape	[0,561]	1.09
R-net	[0,1]	0.011, 0.015
O-net	[0,1]	0.0068, 0.017
Calibrate landmarks	[0, 561]	2.21

Table 4.8 presents the average difference between the quantized inference output and the regular output. The layers were quantized using 25 images from the WIDER FACE dataset [2]; An average difference was then also calculated on 25 different images from the WIDER FACE dataset [2].

The error presented in Table 4.8 was calculated for each layer, independent of other layers; The total error for one inference would then be the summation of all errors across the different layers - this would change if merging of layers were possible, as the need for dequantization would be reduced.

Chapter 5
Discussion

5.1 Limitations

Due to the requirement of an 8-bit quantized model for the microNPU, certain limits were enforced on the model in the different stages, when lowering from TF to the barebone NPU. The limitations put on the model will be discussed in this subsection.

5.1.1 Tensorflow

The limitations that was put on MTCNN with regards to implementing it in TF could be viewed across the different steps (TF → TFLite → TFLite micro).

TF The complete divided and modified model (only TF native operations) were able to run in TF using graph execution (not eagerly executed) on an ordinary x86_64 cpu in python. This meant that no fallbacks to the python interpreter took place when executing the graph - DC and DS is thus far supported.

TFLite The TFLite converter only supports a subset (albeit many) of the TF operators used by our implementation of MTCNN. The unsupported operators gets converted to what is called a *flex* operator. A flex operator is instead executed using the core TF runtime [16]; Since the microNPU leverage the TFLite micro runtime, flex operators are unsupported. The operators that were converted to their flex counterpart in MTCNN were: *tf.image.non_max_suppression*, *tf.image.extract_patches* and *tf.image.crop_and_resize*. [14] The necessity for the extract patches operation can be discussed, as this essentially performs the cropping of the images with a certain stride - equivalent to the initial part of a CNN. However, it impacted the: NMS, image pyramid, resize and pad stages, as all were using one of these operators. The implications of this will be further discussed in Section 5.3.4.4.

Solely converting a model to TFLite does not suffice for running it on the microNPU; Converting the model to use 8-bit integers is also necessary. The quantizer put constraints on the model as well, even though *while* and *if* is supported in TFLite, quantization does not support the use of sub graphs - in the case where the iteration count and condtional statements are unknown at compile-time

- true DS and DC.

TFLite micro The TFLite micro runtime supports a subset of the TFLite operations. Dynamic control (*while* and *if*) is not supported. DS in the graph is also not supported and defaults to a size of 1.

5.1.2 Vela

The overall structure of Vela can be summarized as; A TFLite file is transformed, optimized and scheduled to run on an Arm microNPU system using the TFLite micro interpreter. Due to the limitations in TFLite and the TFLite quantizer, Vela is unable to output said optimized TFLite file if the inclusion of DS and DC is present.

5.1.3 NPU

It is not possible to use any DC or DS operators on the microNPU, as these are not listed in the supported operators Vela can schedule on the microNPU. [37] Since this is an upstream limitation in the toolchain, it is extpected that the microNPU show no support for this.

5.2 Model Alterations

Because of the aforementioned limitations of TF and Vela, a complete replica of MTCNN was not possible to make entirely using only TFLite micro operators. To circumvent the problems of dynamic control and shapes, alterations were made and constraints put on the model. One important constraint was that the input image size to the model would be constant; This allowed for many of the shapes and iteration counts to be calculated at compile-time - especially the number of iterations in the image pyramid and the amount of sub images generated at each iteration. This is a reasonable constraint to be set on an edge device as the use-cases for these are often accompanied by sensors (cameras, etc) that does not change the format of its output. This also removed multiple previously required dynamic shapes.

Conditional operators could not be used, meaning that the model could not be terminated early if no facial candidates remain. This could be resolved in two ways: Letting the CPU handle the conditional for zero candidates, or including an extra low-confidence candidate which is only removed at the final stage of the algorithm. The latter approach was adopted; This made sure that all stages were executed at least once and avoided errors by executing a stage without a valid input. Having circumvented the issues at hand, performance could be measured to evaluate the gain of running these kinds of models on the microNPU.

5.3 Performance evaluation

This section will go summarize the individual stages performance in terms of inference speed, memory footprint and what could be done to increase performance; The full MTCNN model will be evaluated, based on execution time and accuracy.

5.3.1 HW constraints

The clock frequencies for the Cortex-M series processors as well as the microNPUs are decided by the system-on-chip (SoC) manufacturer. Assumptions had to be made on these to get the FPS-based results presented in Section 4.2.3. These clock speeds are presented in Table 4.2 for the different NPU configurations, and was chosen as 216 MHz for the CPU.

The assumption of microNPU clock speed was based on executing Vela [36] with the settings in Table 4.2 as input parameters, and noting the clock speed presented in the output profiling data. The CPU clock frequency was assumed to be the same as that of the STMicroelectronics mid-range Cortex-M7 based microcontrollers. [51] This was assumed since the Cortex-M7 has proven to be capable of executing TFLite micro NN based applications. [48] However, as the microNPU is able to accelerate ML performance in combination with the Cortex-M55, a mid-range implementation of the CPU might be enough - which is why the clock frequency of the mid-range (STM32F7), and not the high-performance (STM32H7) series was selected as assumption.

These clock speeds were only used when unavoidable, due to the high uncertainty of how it will relate to future SoCs, all other performance evaluations were based on clock cycles only. These assumptions and the fact that models and not physical HW was used to obtain the benchmarks, implies that all performance figures stated should be treated as **estimates** only.

5.3.2 NPU acceleration

The P-, R- and O-Net stages shows a clear speedup of approximately 2-3 times when running on the Cortex-M55 compared to the Cortex-M7. While for the other stages the speedup is around 10 % between these hardware configurations. The higher speedup is expected for the P-, R- and O-Nets as they are ANNs utilizing common ML operators, which the Cortex-M55 was made to accelerate. [52]

The three NPU settings; HW1, HW2 and HW3 presented in Table 4.4, all gives a speedup relative to the Cortex-M7 for all individual MTCNN stages. A similar pattern as for the Cortex-M55 results is visible, where the P-, R- and O-Nets achieve higher speedup compared to most of the other stages.

Combining the Cortex-M55 with a microNPU also accelerated the stages not containing an ANN: With the *normalize* stage achieving a comparable speedup to the CNN-based stages. This was not the case with only the Cortex-M55 CPU present. This shows that significant performance increases can be achieved even for **non-ANN** applications, when executed on the NPU. The *calibrate*, *reshape* and *calibrate landmarks* stages did not achieve a performance increase as impressive; This is due to the way the layers were implemented - seen in the Appendix A, the graphs can only execute on one bounding box at a time. These layers are

therefore unable to leverage the added performance increase of the parallelism provided by higher-end hardware configurations - without further enhancements to the software-stack. The same reasoning could explain why the *P-Net* and *normalize* stages did not achieve as great speedup as, for example, *O-Net* on the most powerful HW configuration (HW3); Since these stages contain fewer operations, and thus benefit less of increased parallelism.

In Figure 4.7 the relative run-time spent on each layer for one inference can be viewed; The most computationally demanding layers are accelerated by the NPU - where the *normalize*, *calibrate*, *reshape* and *calibrate landmarks* stages amount for a combined execution time of 1.2% to 0.2% depending on the hardware configuration.

The layers containing *ANNs* are by far the most demanding on the hardware (in MTCNN) and little performance would be gained by getting the four aforementioned layers accelerated by the NPU - Even though the *normalize* layer achieved a significant relative performance gain, the total gain by accelerating this layer is perceivably non-existant in the grand scheme of run-time for MTCNN.

5.3.2.1 NPU execution vs. memory access

Table 4.4 show memory access versus NPU utilization for the normalize stage on different hardware configurations. This explain the speedup between the different hardware configurations HW1, HW2 and HW3 in Figure 4.4, for the normalize stage: The NPU utilization relative to the total execution time decreased when HW with more MAC units were used. The memory accesses are assumed to be the same across the different configurations. Meaning that the NPU execution time became more effective due to the increase in MAC units, with the possibility to run more computations in parallel. Similarly, the hardware configuration with the most MAC units on the microNPU is limited by memory access operations, thus adding even more MAC units should not result in further speedup in this case.

5.3.3 Memory requirements

The memory required to be reserved for each stage is shown in Table 4.3. The maximum SRAM value is the most important, as the stages will not run in parallel and are therefore not required to occupy space in SRAM memory simultaneously. The memory can instead be allocated right before model execution. No hard limits to memory was considered, as this will depend on the physical SoC where the network is deployed; It does not exist any commercially available products incorporating an Arm microNPU yet. Instead, comparisons with current high end embedded microcontrollers based on a Cortex-M CPU was made.

O-Net exerts the largest memory requirement of 204 KB. This is deemed as acceptable since several high end Cortex-M based SoC has 256-1024 KB of SRAM. [53, 54, 51, 55] If 700 KB or more is available as on-chip flash, which is the case for multiple cortex-M based SoCs [53, 55, 54], the entire network can fit into the target SoC. The flash memory usage can also be placed off-chip, if required, allowing for bigger NNs when the SoC flash size is limited.

5.3.4 Increasing performance

5.3.4.1 P-Net

As shown in Figure 4.7a, the overall performance of the algorithm depends on an efficient implementation of P-Net. The system saw a bigger increase to performance with an increase in complexity for the network; As P-Net is relatively simple, and only runs for a $12x12$ section of the image, a big potential for performance increases can be made. The sections of the image analyzed by the P-Net can run independent of other sections and thus can benefit greatly from parallel execution by the P-Net. Another performance increase could be made by designing a custom P-Net for the system, the custom P-Net could be more complex in the sense that it could process larger parts of the image at once; It would be expected to see a performance increase similar to the one for O-Net, seen in Figure 4.4.

5.3.4.2 Custom PReLU operator

The entire P-,R- and O-Nets of the MTCNN algorithm could initially not be executed on the NPU, as seen in Table 4.5. This was because of the PReLU operator, which had to fall-back to the CPU and was used multiple times for each stage. This resulted in crosstalk between the CPU and NPU. To see what effect this had on the performance itself a custom PReLU operator was implemented and tested. A big speedup and increased memory usage was registered as a result. The memory increase was expected as more operators are being executed on the NPU.

This shows that it can be worthwhile the effort to create custom operators to obtain performance increases. While at the same time the potential drawback of increased memory usage is shown.

5.3.4.3 Decrease crosstalk

Another method to decrease crosstalk between CPU and NPU is merging stages in the MTCNN algorithm. This allows for the NPU to operate longer without requiring to communicate with the CPU. This would also reduce the times the TFLite micro interpreter has to be invoked and lead to less data handling, between each stage, in the main application. This makes the algorithm easier to deploy from a developers perspective. However, it may require more space in the SRAM as larger TFLite graphs are being run at once.

The major obstacle for merging more MTCNN stages are the operations that cannot be executed on the NPU. These are interleaved with the other stages in ways which makes merging impossible at several parts of the algorithm. The main problem relates to the NMS and remove stages, as these are executed after each of the three CNNs.

The reshape and calibrate layer was merged to investigate further performance enhancements through merging layers. A decrease of about 25% could be seen to the total run-time - this decrease is due to reduced crosstalk between the CPU and NPU as well as more efficient parallel computation; Vela has the possibility of scheduling the operations more efficiently, the same concept as link-time

optimization in compiler technology.

5.3.4.4 Issues with remaining stages

The stages including operators defaulting to *flex-ops*, mentioned in Section 5.1.1, was investigated further to localize the issues with executing these on the NPU:

Image pyramid Major parts of the image pyramid could - with the constraint of a fixed image size, be calculated at compile time; The size of the image pyramid and coordinate computations are, in this case, independent from the input data and not dynamical. The remaining issue was extracting the 12x12x3 patches for P-net to analyze. This resulted in the TF graph having a very high iteration count or a too large size, and it was deemed to be a more suitable operation to put on the CPU instead (note that this is the same operation performed in a regular CNN, extracting patches with a specific kernel and stride). This is not because the CPU gives a performance uplift, but due to the limited number of operators available in the toolchain. Since the operation mainly relates to forwarding input data, as most computations are finished at compile time, there was no expected performance uplift if this stage would be possible to be executed on the NPU. Deploying it on the NPU is not expected to result in a significant reduction in crosstalk between CPU and microNPU either, as this stage is only executed once in the MTCNN algorithm.

Remove The remove stage require removing parts of the input with a low probability of containing a face, decided by the preceding stages, and providing the remaining boxes as output. This trivial operation was not possible in TFLite as it require DS on the input and output. Contrary to the image pyramid, this would be expected to reduce the total crosstalk in the system, resulting in a performance uplift; The *remove* operation/layer has to be done previous to every *NMS*.

NMS NMS could not be accelerated by the microNPU as it require DS, on both input and output parameters, and DC in the algorithm.
The amount of bounding boxes NMS receives depend on the input image; With NMS having to sort and iterate over each box and therefore must know the total number of boxes. The NMS algorithm require control flow operators: *if* and *while* - as comparisons are made, and the input data has to be iterated over multiple times. The amount of comparisons and iterations on each NMS run depends on how often the candidate bounding boxes can be merged, and how many candidates there are.

The NMS algorithm was possible to implement on the Cortex-M55, as this supports DC and DS. So the entirety of the algorithm is executed on the CPU, with no possible acceleration from the NPU. NMS is a computationally intensive algorithm, making an acceleration by the NPU desirable not just for reducing crosstalk but also achieving a performance increase during NMS inference. As the number of faces in an image increases, so does the computational requirements of NMS; This can be seen in Figure 4.7b where it surpasses both R- and O-net in

computational requirements - when R- and O-net are accelerated on more powerful configurations of the microNPU.

Pad The padding stage was problematic to implement without any conditional DC operators. Since each bounding box should be padded for all coordinates that exceed the input image, conditional control flow is required. Another solution would be to run inference on pre-padded images and in turn, increase the runtime.

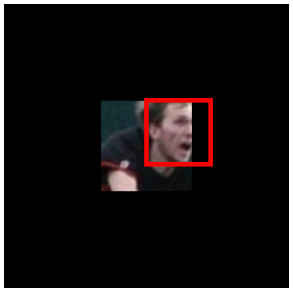


Figure 5.1: Example pre-padding. Middle image is taken from the WIDER FACE dataset. [2]

Another approach would be to create a pre-padded image, which has the input image in the middle and padding on the sides, allowing for bounding boxes to extend outside of the image. An example of this is shown in Figure 5.1, with black denoting padding. The red bounding box is an example candidate that could have been provided from the reshape stage.

This comes with two drawbacks: Pessimistic padding has to be made, as the theoretical maximum padding has to be possible even if none of the boxes has to be padded that much, or at all, in practice. The other issue is that the entire image, with padding, then has to be stored in device SRAM to be used by a TFLite micro program. Since the original image is 561x561x3 pixels, and the padded one will be even larger, this approach was deemed as too memory intensive to be deployable on the NPU.

Resize Bilinear resizing is supported by the NPU [37], but DS is also required to make the resize stage possible. This is because the input sizes differs between each input box, which has to be defined to perform the operation. This made the resize stage unsupported on the NPU.

The resize stage is always preceeding the pad stage. Meaning that for crosstalk to be reduced, both has to be implementable on the NPU instead of CPU. The resize stage was still expected to give an uplift in performance if executed on the NPU, as the main operator: Resize bilinear, is supported.

5.3.5 Impact of input data with varying number of faces

MTCNN differ from traditional ANNs, such as CNNs, in the sense that the input image will have a large effect on the execution time. This must be taken into

account to obtain proper MTCNN benchmarks.

With the constraint of a fixed image size, or *resolution*, P-Net will execute for the same number of times for each input image. The proceeding stages will however have different iteration counts depending on the P-Net output, which depends on the input image. Table 4.6 show a clear correlation between an increased number of faces and a higher iteration count for the later MTCNN stages. This was somewhat expected: As the iteration count for the later stages depends on the amount of candidates with a high probability of containing a face. The iteration count can still differ a lot for images with the same numbers of faces, as there will be false positives in the P-Net and R-Net output. This is why average values over a large set of images, 2500 from the WIDER FACE dataset [2], was used to obtain the data in Table 4.6; This also highlights the number of times P-net has to run compared to the other stages in the MTCNN algorithm - as P-net is not the individually most demanding layer on the hardware.

The non-NMS stages produce one output candidate per input, meaning that the execution time is proportional to the iteration count. The NMS stage however, is only executed once but with a variable number of input candidate bounding boxes. It was assumed to have a nonlinear time complexity relative to the number of input candidates, due to sorting the boxes and having to perform a calculations for each box depending on the remaining number of boxes, described in Section 2.4. The benchmarking of the NMS algorithm was performed similarly to the other stages, but by taking the median number of input boxes for each NMS instead of an iteration count, for images with 0-50 faces. The NMS execution time will also depend on the box coordinates, as more boxes being merged will lead to slightly less total comparisons in the end. This was however disregarded when obtaining the execution time benchmarks, as the amount of boxes was assumed to have a much greater impact than the coordinates. Since different box coordinates mainly affect the final comparisons, which are the less expensive than earlier comparisons, and the sorting only depends on the number of boxes.

The NMS operation was done on sample data from running the full MTCNN algorithm. This was not done for the other stages as their execution time only depends on the iteration count, not the input data itself.

5.3.6 Execution time

A clear speedup is visible between the different HW configurations in Figure 4.8. This speedup mainly comes from the speedup of P-Net; which dominate the total MTCNN execution time, as displayed in Figure 4.7a. However the clock frequencies used, shown in Table 4.2, also make a major contribution between the resulting throughput for HW1 and HW2 compared to HW3.

Compared to the implementations by Centeno [5] and Zhang, et al. [1] in Table 4.7, our MTCNN implementation achieves a substantially lower throughput when executed with otherwise similar settings: Resolution and number of faces in the input image. This was expected as the Centeno and Zhang, et al. benchmarks utilize more powerful HW. However that comes at the cost of a higher power consumption, which is also visualized in Table 4.7.

A person detect [48] and image classification [50] implementation was included

to show how our MTCNN model differs to other NN algorithms executing on similar HW. The functional differences between the person detect, image classification and MTCNN is discussed in Section 5.6.2. Our MTCNN algorithm was re-built and executed using an input image resolution of 128x128x3 to be as comparable to the other ML algorithms as possible. This reduces the computational complexity, which explain the major increase in throughput, for the lower-resolution MTCNN implementation.

Table 4.7 clearly displays the higher complexity of *MTCNN* compared to algorithms like MobilenetV2, running on identical HW. As MTCNN receives a significantly lower throughput compared to the mobilenetV2 benchmark, executed on HW3.

The Cortex-M55 combined with a microNPU is not the recommended choice for real time face detection, but instead the more powerful Ethos-N combined with an Cortex-A CPU. [10] This coincides with our results: Even though real-time performance can possibly be achieved, it is at the expense of reducing the resolution of the input image. Which can be seen as a downgrade to the MTCNN algorithm, as the input information is reduced. It is good that our performance estimate of MTCNN can achieve similar performance to NNs with more basic tasks, running on similar HW and other settings, but it might not make as much sense for MTCNN to run on this low resolution as, for example, an object classification or detection NN.

5.4 Validation

The accuracy of the model was only measured in terms of reassuring/validating functionality for the custom implementation of the different layers.

This was done, as described in Section 3.2; By comparing the output of our model to that of a python MTCNN implementation [5], on a set of images from the WIDER FACE dataset. [2] This approach was deemed as the most fitting to guarantee functionality, as it avoids manually having to look at a large set of images to determine if the algorithm works or not. The WIDER FACE dataset could have been used on its own, but that does neither take the minimum face size (which MTCNN is trained to detect) into account nor the input resolution of our model. Meaning that a lot of the information stated in the dataset would be undetectable; due to input degradation when resizing and MTCNN minimum face size requirement.

By using the python implementation as reference, many undetectable boxes due to these limitations are immediately discarded. Some issues do remain however, such as boxes below the minimum face size being detected even though the algorithm was never trained for that. These boxes were still included in the results in Figure 4.11 and 4.12, as both implementations has a chance to detect them, but it adds another level of uncertainty in the results. The outliers removed between Figure 4.11 and 4.12 were deemed as being too far from the detected face, and instead treated as a missed box by our implementation.

It is difficult to determine the coordinates of a face with pixel-perfect precision: This was illustrated by Figure 4.13. Thus having some pixel variations was

determined as acceptable, with a low error in detection rate and confidence being of higher importance.

Our implementation was deemed as functional. Based on the average pixel error of around 4.5 pixels, displayed in Figure 4.12, alongside a low confidence error of below 1% (using the same verification as for the bounding boxes). When comparing the two implementations. As the implementations are done in a similar manner, use the same weights and are based on the same algorithm; ideally the outputs would have been identical. However, this result is still seen as good enough for a valid MTCNN implementation.

5.4.1 Quantization

Verifying the model after quantization to 8-bit integer is important since this can result in a performance degradation, as discussed in Section 2.1.1. Two approaches were considered for this: Testing the full quantized model and verify its performance, or comparing the quantization error of each individual stage in the MTCNN algorithm. The latter approach was chosen, as described in Section 3.2. The first approach was turned down as it would not give any meaningful output to which stage had a faulty quantization. It would only produce information with regards to if the entire quantized model achieved a good enough result.

The results presented in Table 4.8 show the quantization error of the individual stages. This was seen as reasonable, since all values are close to the theoretical minimum quantization error, presented in Equation 2.1. This deviation, of a few pixels or percent in confidence, is deemed as having low enough impact on the final output of the layer. Mainly due to the previous argument; that it is difficult to determine a face with pixel-perfect precision.

5.5 DC in other ML algorithms

MTCNN is not the only algorithm that incorporates dynamic control, which have been addressed by Yuan, et al: "[M]odels based on recurrent neural networks and on reinforcement learning depend on recurrence relations, data-dependent conditional execution, and other features that call for dynamic control flow". [8]

RNNs contain recurrence relations that are commonly modeled as loops on implementations using a high level programming language. [56] RNNs with finite sequence lengths can be represented without dynamic control at execution time if the loops are unrolled, according to Section 2.2, but infinite sequence length RNNs cannot be unrolled and thus require dynamic control operators to implement. Using control flow operators can also reduce the memory footprint, as an unrolled loop can result in more instructions placed in device memory [34], which is important for memory constrained devices. Such as embedded microcontrollers. This implies that the available memory, in which the unrolled RNN will be placed, constrains the sequence length of the RNN itself. [8]

Even though RNNs and reinforced learning are prime examples of algorithms that would benefit of dynamic control, Google has also emphasized that dynamic control will enable future ML algorithms: "Dynamic shapes, dynamic flow control, dynamic multi-model dispatch, streaming models, tree-based search algorithms,

and other are all good examples of exciting ML evolution". [12]

Another benefit of DC operators is ease of use and flexibility when implementing and training ML algorithms. Where more traditional, static NN based algorithms could be extended with DC operators to gain extra functionality. Both during training of the algorithm and during inference. [8, 9]

5.6 Similar purpose ML algorithms

MTCNN was selected due to its unique architecture and dynamic properties. Other algorithms that perform similar tasks will be presented in this section with a comparison to MTCNN. This is to highlight algorithms deployable on similar hardware as the NPU system, but which are different to MTCNN.

5.6.1 MobileNetV1

MobileNetV1 is a NN architecture which target mobile and resource constrained devices. It can be trained to perform different tasks, such as geolocalization and image classification. [57] The image classification algorithm presented in Table 4.7 is a MobileNetV1 trained for image classification. [50] This makes it quite different to MTCNN, as it does not give any bounding boxes or landmarks within the image, but can be trained to classify the image as containing a person or not.

5.6.2 MobileNetV2

MobileNetV2 is a continuation of V1 with the same purpose; inference on resource constrained devices such as mobile platforms and microcontrollers. [58] MobileNetV2 implementations fully executable on an microNPU system exist [59] but does not, as opposed to MTCNN, consist of dynamic control.

MobileNetV2 can just like V1 be trained to perform a multitude of different tasks, such as object detection, image classification and semantic segmentation. [58] With person detection being closest to the face detection and alignment task of MTCNN. The person detection algorithm presented in Table 4.7 receives an input picture and produce a binary output (yes or no) depending on if the image contain a person or not. [48] MTCNN give more elaborate results: Capability of detecting multiple faces, each with its own probability, facial landmarks and bounding box.

5.7 DC support in emerging ML frameworks

The TFLite micro based toolchain described in Section 2.5 presents the official approach to deploying and executing an application on the current microNPU product line. This was later extended by us to form a workflow, displayed in Figure 3.1, which presents a complete toolchain for implementing a ML model on the NPU system.

The NPU HW, when taped-out on a physical SoC, is inflexible. The toolchain is not. The limitations of the toolchain, presented in Section 5.1, could potentially

be bypassed by using other frameworks. However, as the NPU is designed to support this specific workflow, it is possible to have limitations tied to those in the toolchain as well. Meaning that solving these issues might require changes to the HW as well.

This makes other frameworks for ML applications important to consider: As they can be a replacement, or amendment, to the toolchain for the current or future NPU versions. DS and DC support in modern ML frameworks is actively being developed and new algorithms are emerging that can take advantage of this additional DC; Following the framework, support for DS and DC in devices (hardware accelerators, GPUs, etc) and compilation stack.

The following subsections will go into detail on these frameworks and what they would mean for MTCNN and other DC based ML algorithms.

5.7.1 Pytorch

Pytorch is a framework that delivers the same functionality as TF (they both provide graphs) but they differ in the way they build graphs. TF, as mentioned throughout this report builds static graphs that are not meant to change in a dynamic way; Dynamic graphs are one of the features of Pytorch that differs from TF. This would presumably not change the outcome however, as the rest of the software toolchain would also have to account for this; Compilation/interpretation closer to the hardware would have to support a runtime that has this feature. [60]

5.7.2 IREE

IREE is, as mentioned in Section 1.1, still in early stages. This project is focused on providing the full compilation stack between the framework (TF, Pytorch etc) and hardware. One of its main features is to efficiently support DC and DS. [12] The goal with this project strongly coincides with the investigation in this thesis and can perhaps, in a future version, solve many of the issues we faced.

5.7.3 TOSA

TOSA, as mentioned in Section 2.6, aims to standardize the most commonly used primitive operations of popular neural networks. In TOSA, some DC is supported and in the current revision, *while* and *conditional* statements are supported. [6] However, TOSA is not specific with regards to DS for the operators being supported or not - as TOSA reference implementations are often implemented using the pseudo code *"for_ each(index in shape)"* when performing the calculation on the dimensions; This implies DS support/usage but is never explicitly mentioned. This leaves the TOSA-compliant hardware/software implementer interpreting the use/solution of DS.

Viewing TOSA from the perspective of MTCNN raises the question, would MTCNN be TOSA-compliant? MTCNN consists of regular arithmetic, sorting (NMS), 2D convolutions and activation functions (softmax and PRELU), as explained in Section 2.4; The arithmetical and convolutional part of MTCNN is TOSA compliant.

The NMS used by the original MTCNN [1] sorts the bounding boxes by confidence and uses the corresponding indices for the rest of the computation; Assuming DS is supported, this could potentially be done by merging TOSA operators (e.g, WHILE, ARGMAX and GATHER). The rest of the NMS algorithm essentially run arithmetics, until no more boxes can be merged, all of which are TOSA-compliant.

The activation functions used in the original MTCNN paper [1] are softmax and PRELU. RELU, NEGATE, ABS and ADD is in the TOSA specification and can be used to create PRELU. Softmax is also part of the different CNNs and is described in the section MTCNN, using Equation 2.8. TOSA provides support for elementwise exponential calculations, division and reduction to sum (EXP, DIV and REDUCE_SUM), all needed to perform the softmax operation. In short, MTCNN is TOSA-compliant assuming *DS are supported*.

Chapter 6
Conclusion

Current issues with implementing DC based ML algorithms on Arm NPU systems was investigated in this thesis. With native dynamic control on the microNPU not being achievable; Current microNPUs requires DC to be scheduled on the CPU. It was also evaluated on which sections of the implementation process and software toolchain DC support was lacking, with multiple limitations present at different sections. This means that support would have to be implemented at different steps of the toolchain, with changes possibly made to the microNPU HW architecture as well - in order to run sufficiently dynamic algorithms strictly on the microNPU.

The implementation and evaluation conducted revolved around implementing the algorithm *MTCNN*. Despite its DC properties, an implementation for a majority of the algorithm on the microNPU was possible by exerting certain constraints on the model. The sections of the algorithm that was unable to execute on the microNPU was successfully offloaded to the CPU. A heavy speedup was achieved for this implementation, compared to implementations on other Cortex-M CPUs. It was both displayed that non-ML heavy parts of the algorithm could achieve a significant speedup when executed on the microNPU; This speedup, however, had a minimal impact on the total MTCNN execution time. The results demonstrate that offloading unsupported operations, to the CPU, could be done efficiently for this algorithm.

The need for DC support in current ML frameworks was also addressed. With RNNs being a prime example of NN algorithms that benefits from DC support. It would also make room for future DC based algorithms, and ease training and other features related to deploying NNs.

Finally, emerging frameworks for deploying ML algorithms were evaluated. Both their capability of supporting DC in general, and MTCNN in particular. This highlighted the differences in what these frameworks plan to support in the future and the limitations of the current toolchain. TOSA presented support to fundamental DC operations such as conditionals and loops, while potentially lacking other aspects such as support of dynamic shapes. With IREEE and Pytorch also taking DC into account.

6.1 Future work

This thesis has displayed the current limitations of DC support on Arm NPU system, and also hinted on what future frameworks have to offer and why DC is relevant for ML based applications. This can act as a foundation for several topics of future work:

Extending the Arm NPU models to support DC. This could be used to investigate what performance increase, if any, would be possible from running pure DC operations on the NPU, compared to offloading tasks to the CPU. Further investigations can be made into what impact this would have on *MTCNN* and similar algorithms.

Another topic would be to evaluate the related ML frameworks presented in this thesis (TOSA, IREE) closer to their release. To see how well they fit with the, at the time, available microNPU product line; And investigate if they solve the dynamic properties problems which have been investigated in this thesis.

More complex DC based ML algorithms coming in the future could also be investigated. To see if the basic DC operators presented are still sufficient to handle these networks, and if other architectural changes provide a performance uplift. A final topic would be to investigate how ML architectures containing weights handles a dynamic change in structure; This would be closely related to subgraphs and solutions as such.

Bibliography

[1] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint face detection and alignment using multitask cascaded convolutional networks," *IEEE Signal Processing Letters*, vol. 23, pp. 1499–1503, Oct 2016.

[2] S. Yang, P. Luo, C. C. Loy, and X. Tang, "WIDER FACE: A Face Detection Benchmark," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[3] Arm Ltd., "Arm ethos-u npu application development overview." `https://developer.arm.com/documentation/101888/0500/Introduction/Ethos-U-system`, Oct 2020. Version 5. Accessed: May 11, 2021.

[4] L. Roeder, "Netron." `https://netron.app/`, May 2021. Version: 4.9.4. Accessed: May 27, 2021.

[5] I. Centeno de Paz, "Multi-task cascaded convolutional neural networks for face detection, based on tensorflow." `https://pypi.org/project/mtcnn/`, Nov 2019. Version: 0.1.0. Accessed: May 11, 2021.

[6] Arm Ltd, et al. (Open source), "Tosa." `https://developer.mlplatform.org/w/tosa/`. Accessed: May 11,2021.

[7] Arm Ltd., "Arm ethos-u55 micronpu product brief." `https://www.arm.com/products/silicon-ip-cpu/ethos/ethos-u55`, Feb 2020. Retrieved: Apr. 14, 2021.

[8] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins, and et al., "Dynamic control flow in large-scale machine learning," *Proceedings of the Thirteenth EuroSys Conference*, Apr 2018.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," 2016.

[10] A. Skillman and T. Edso, "A technical overview of cortex-m55 and ethos-u55: Arm's most capable processors for endpoint ai," *2020 IEEE Hot Chips 32 Symposium (HCS)*, pp. 10, 12–14, 17, 2020.

[11] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021.

[12] Google, inc., "Iree (intermediate representation execution environment)." `https://google.github.io/iree/`. Accessed: May 12, 2021.

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[14] Tensorflow, "Tensorflow source code." `https://github.com/tensorflow/tensorflow`. Retrieved: 5/4-2021.

[15] Tensorflow, "Tensorflow: Introduction to graphs and tf.function." `https://www.tensorflow.org/guide/intro_to_graphs`. Retrieved: 8/5-2021.

[16] Tensorflow, "Tensorflow lite. operator compatibility." `https://www.tensorflow.org/lite/guide/ops_compatibility`. Retrieved: Apr. 5, 2021.

[17] Tensorflow, "Tensorflow lite. converter." `https://www.tensorflow.org/lite/convert`. Retrieved: Apr. 5, 2021.

[18] Fun Propulsion Lab, Google, "Google, github: Flatbuffer landing page." `https://google.github.io/flatbuffers/`. Retrieved: May. 4, 2021.

[19] Tensorflow, "Tensorflow lite. quantization." `https://www.tensorflow.org/lite/performance/post_training_quantization`. Retrieved May. 6, 2021.

[20] Tensorflow, "Tensorflow lite. interpreter." `https://www.tensorflow.org/lite/guide/inference`. Retrieved: Apr. 5, 2021.

[21] Tensorflow, "Tensorflow lite micro. build and convert models. operation support." `https://www.tensorflow.org/lite/microcontrollers/build_convert#operation_support`. Retrieved Apr. 18, 2021.

[22] Tensorflow, "Tensorflow lite micro." `https://www.tensorflow.org/lite/microcontrollers/`. Retrieved Jan. 18, 2021.

[23] LLVM Project, "Llvm, multi-level intermediate representation." `https://mlir.llvm.org/`. Retrieved: May. 4, 2021.

[24] The TensorFlow MLIR Team., "Mlir: A new intermediate representation and compiler framework." `https://mlir.llvm.org/`. Retrieved: May. 6,2021.

[25] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[26] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, pp. 164–165, 180–183, 192–194, 200–208, 326–331, 335–339, 367–371. MIT Press, 2016. http://www.deeplearningbook.org.

[27] Keras, "Making new layers and models via subclassing." https://keras.io/guides/making_new_layers_and_models_via_subclassing/. Retrieved: Mar. 30, 2021.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, May 2017.

[29] S. Lawrence, C. Giles, A. C. Tsoi, and A. Back, "Face recognition: a convolutional neural-network approach," *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 98–113, 1997.

[30] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2018.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015.

[32] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, pp. 90–96, 338–339. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2013.

[33] Python Software Foundation., "Python, compound statements." https://docs.python.org/3.9/reference/compound_stmts.html. Retrieved: May. 7, 2021.

[34] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, pp. 157–161, Appendix A.5. Elsevier Science, 2011.

[35] Arm Ltd., "Arm ethos-u series processors.." https://developer.arm.com/ip-products/processors/machine-learning/arm-ethos-u. Accessed: May 11, 2021.

[36] Arm Ltd., "Ethos-u-vela 2.1.1.." https://pypi.org/project/ethos-u-vela/. Accessed: Apr. 14, 2021.

[37] Arm Ltd., "Supported ops." https://review.mlplatform.org/plugins/gitiles/ml/ethos-u/ethos-u-vela/+/refs/tags/2.1.1/SUPPORTED_OPS.md. Vela 2.1.0. Accessed: Apr. 14, 2021.

[38] LLVM, et al., "Tosa dialect in llvm." https://mlir.llvm.org/docs/Dialects/TOSA/. Accessed: May 11, 2021.

[39] I. Centeno de Paz, "Mtcnn weights." https://github.com/ipazc/mtcnn/blob/master/mtcnn/data/mtcnn_weights.npy, Nov 2019. Numpy dataset. Accessed: May 10, 2021.

[40] Tensorflow, "Build and convert models. model conversion. convert to c array." https://www.tensorflow.org/lite/microcontrollers/build_convert#convert_to_a_c_array. Accessed: May. 11, 2021.

[41] Arm Ltd., "Arm cortex-m55 processor product brief." https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m55. Retrieved: May 11, 2021.

[42] Arm Ltd., "Arm developer. fast models. fast models library." https://developer.arm.com/tools-and-software/simulation-models/fast-models. Retrieved: May. 11, 2021.

[43] Arm Ltd., "Fast models user guide." https://developer.arm.com/documentation/100965/latest, Mar 2021. Version: 11.14. Retrieved: May 11, 2021.

[44] Arm Ltd., "Arm corstone sse-300 example subsystem. technical reference manual." https://developer.arm.com/ip-products/subsystem/corstone/corstone-300, Oct 2020. Issue: 0.2. Retrieved: May. 2, 2021.

[45] Keras, "Prelu layer." https://keras.io/api/layers/activation_layers/prelu/. Accessed: May 10, 2021.

[46] "Intel i7-3612qm product specification." https://ark.intel.com/content/www/us/en/ark/products/64901/intel-core-i7-3612qm-processor-6m-cache-up-to-3-10-ghz-bga.html. Accessed: June 7, 2021.

[47] "Gtx titan x user guide." https://www.nvidia.com/content/geforce-gtx/GTX_TITAN_X_User_Guide.pdf. Accessed: June 7, 2021.

[48] STMicroelectronics, "Artificial intelligence (ai) and computer vision function pack for stm32h7 microcontrollers. user manual (um2611)." https://www.st.com/en/embedded-software/fp-ai-vision1.html#documentation, Apr 2021. Revision: 4. Accessed: May 13, 2021.

[49] STMicroelectronics, "STM32H742xI/G, STM32H743xI/G, 32-bit Arm Cortex-M7 480MHz MCUs, up to 2MB Flash, up to 1MB RAM, 46 com. and analog interfaces." https://www.st.com/resource/en/datasheet/stm32h743vi.pdf, Apr 2021. Revision: 8. Accessed: Jun 8, 2021.

[50] A. Capotondi and M. Rusci, "Mobilenet V1 for STM32 over CMSIS-NN." https://github.com/EEESlab/mobilenet_v1_stm32_cmsis_nn. Accessed: May 18, 2021.

[51] STMicroelectronics, "Stm32f7 series of very high-performance mcus with arm cortex-m7 core." https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html. Accessed: May 10, 2021.

[52] Arm Ltd., "Arm cortex-m55 processor datasheet." https://developer.arm.com/ip-products/processors/cortex-m/cortex-m55, Feb 2020. Retrieved: May 14, 2021.

[53] Nordic Semiconductor, "nrf52840 product specification." https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.1.pdf, Feb 2019. Version 1.1. Accessed: May 10, 2021.

[54] Microchip Technology Inc., "Sam e70/s70/v70/v71 family." https://ww1.microchip.com/downloads/en/DeviceDoc/SAM-E70-S70-V70-V71-Family-Data-Sheet-DS60001527E.pdf, Dec 2020. Revision: E. Accessed: Jun 1, 2021.

[55] NXP Semiconductors, "i.mx rt1064 crossover processors data sheet for consumer products." https://www.nxp.com/docs/en/data-sheet/IMXRT1064CEC.pdf, Mar 2021. Revision: 3. Accessed: May 17, 2021.

[56] Tensorflow., "Recurrent neural networks (rnn) with keras." https://www.tensorflow.org/guide/keras/rnn. Retrieved: Mar. 30, 2021.

[57] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," Apr 2017.

[58] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," Mar 2019. Version: 4.

[59] Arm, Ltd., "ARM software. ML-zoo. MobileNet v2 1.0 224 UINT8." https://github.com/ARM-software/ML-zoo/tree/master/models/image_classification/mobilenet_v2_1.0_224/tflite_uint8. Accessed: May 18, 2021.

[60] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," Dec 2019.

Appendix A

Custom MTCNN model implementation

This chapter will present the python code and TF graphs for some of the layers presented in this thesis. The work presented in this chapter is inspired by the paper, "Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks" and the derived work "Multi-task Cascaded Convolutional Neural Networks for Face Detection, based on TensorFlow". [1, 5]

Model

```
def custom_mtcnn():
    input = tf.keras.Input(
        shape=(settings.IMG_Y, settings.IMG_X, settings.IMG_Z
    ))

    normalize = custom_layers.Normalize()
    normalized_input = normalize(input)

    sub_images_layer = custom_layers.Pre_p_net_sub_images()
    sub_images = sub_images_layer(
        normalized_input, 1)

    [p_out1, p_out2] = custom_layers.p_net(sub_images)
    p_out1 = tf.squeeze(p_out1)
    p_out2 = tf.squeeze(p_out2)

    post_p_1 = custom_layers.Post_p_net_scale()
    bboxes = post_p_1(settings.y_position_divided[0],
    settings.x_position_divided[0], settings.
    scales_extended_divided[0])

    post_p_2 = custom_layers.Post_p_net_inter_nms()
    bboxes = post_p_2(bboxes, p_out1, p_out2)

    for i, scale in enumerate(settings.scales_i):

        sub_images_pyramid = sub_images_layer(
    normalized_input, scale)
```

```
25          sub_images = tf.keras.layers.Concatenate(
26              axis=0)([sub_images, sub_images_pyramid])
27
28          [p_out1, p_out2] = custom_layers.p_net(
    sub_images_pyramid)
29          p_out1 = tf.squeeze(p_out1)
30          p_out2 = tf.squeeze(p_out2)
31
32          post_p_1 = custom_layers.Post_p_net_scale()
33          bboxes_scale = post_p_1(settings.y_position_divided[i
     + 1], settings.x_position_divided[i + 1], settings.
    scales_extended_divided[i + 1])
34
35          post_p_2 = custom_layers.Post_p_net_inter_nms()
36          bboxes_scale = post_p_2(bboxes_scale, tf.reshape(
    p_out1, shape=(-1, 2)), tf.reshape(p_out2, shape=(-1, 4)))
37
38          bboxes = tf.keras.layers.Concatenate(
39              axis=0)([bboxes,bboxes_scale])
40
41      post_p_2 = custom_layers.Post_p_net_nms()
42      bboxes = post_p_2(bboxes[:, 0:4],bboxes[:, 3:5],bboxes[:,
         5:])
43
44      post_p_3 = custom_layers.Calibrate_Rerec()
45      bboxes = post_p_3(bboxes)
46
47      s = custom_layers.Subimages_24x24()
48      subimages_24x24 = s(normalized_input, bboxes)
49      [r_out1, r_out2] = custom_layers.r_net(subimages_24x24)
50
51      r = custom_layers.Post_r_net_nms()
52      bboxes = r(bboxes, r_out1, r_out2)
53      post_r_2 = custom_layers.CalibratePlusOne_Rerec()
54      bboxes = post_r_2(bboxes)
55
56      s = custom_layers.Subimages_48x48()
57      subimages_48x48 = s(normalized_input, bboxes)
58      [o_conf, o_reg, o_landmarks] = custom_layers.o_net(
    subimages_48x48)
59
60      post_o_1 = custom_layers.Calibrate_plus_one()
61      bboxes = post_o_1(tf.concat([bboxes, o_conf[:, 1:2],
    o_reg], axis=1))
62
63      post_o_2 = custom_layers.Post_o_net_nms()
64      landmarks, bboxes, conf = post_o_2(bboxes, o_conf, o_reg,
         o_landmarks)
65
66      post_o_3 = custom_layers.Post_o_net_calibrate_landmarks()
67      ret_val = post_o_3(bboxes, landmarks)
```

```
68
69      model = tf.keras.Model(input, [landmarks, bboxes, conf])
70
71      return model
```

Listing A.1: Code for custom MTCNN implementation

Normalize

```
1 class Normalize(tf.keras.layers.Layer):
2     def __init__(self, **kwargs):
3         super(Normalize, self).__init__(**kwargs)
4     @tf.function(input_signature=[tf.TensorSpec(shape=(1,
      settings.IMG_Y, settings.IMG_X, settings.IMG_Z), dtype=tf.
      float32)])
5     def call(self, input_image):
6         return (input_image - 127.5) * 0.007812
```

Listing A.2: Code for Normalize layer

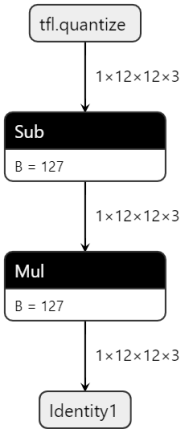


Figure A.1: Graph of normalize layer.

P-Net

```
1 def p_net(sub_images):
2     p_inp = tf.keras.layers.Input(shape=(12, 12, 3))
3     p_layer= tf.keras.layers.Conv2D(10, kernel_size=(3, 3),
      strides=(
4         1, 1), padding="valid")(p_inp)
5     p_layer = custom_PRELU(shared_axes=[1, 2])(p_layer)
6     p_layer = tf.keras.layers.MaxPooling2D(pool_size=(
```

```
7           2, 2), strides=(2, 2), padding="same")(p_layer)
8
9       p_layer = tf.keras.layers.Conv2D(16, kernel_size=(
10          3, 3), strides=(1, 1), padding="valid")(p_layer)
11      p_layer = custom_PRELU(shared_axes=[1, 2])(p_layer)
12
13      p_layer = tf.keras.layers.Conv2D(32, kernel_size=(
14          3, 3), strides=(1, 1), padding="valid")(p_layer)
15      p_layer = custom_PRELU(shared_axes=[1, 2])(p_layer)
16
17      p_layer_out1 = tf.keras.layers.Conv2D(
18          2, kernel_size=(1, 1), strides=(1, 1))(p_layer)
19      p_layer_out1 = tf.keras.layers.Softmax(axis=3)(
        p_layer_out1)
20
21      p_layer_out2 = tf.keras.layers.Conv2D(
22          4, kernel_size=(1, 1), strides=(1, 1))(p_layer)
23      model = tf.keras.Model(inputs=p_inp, outputs=[
        p_layer_out1, p_layer_out2])
24      model = set_weights_for_p_net(model)
25      return model(sub_images)
```

Listing A.3: Code for P-Net implementation

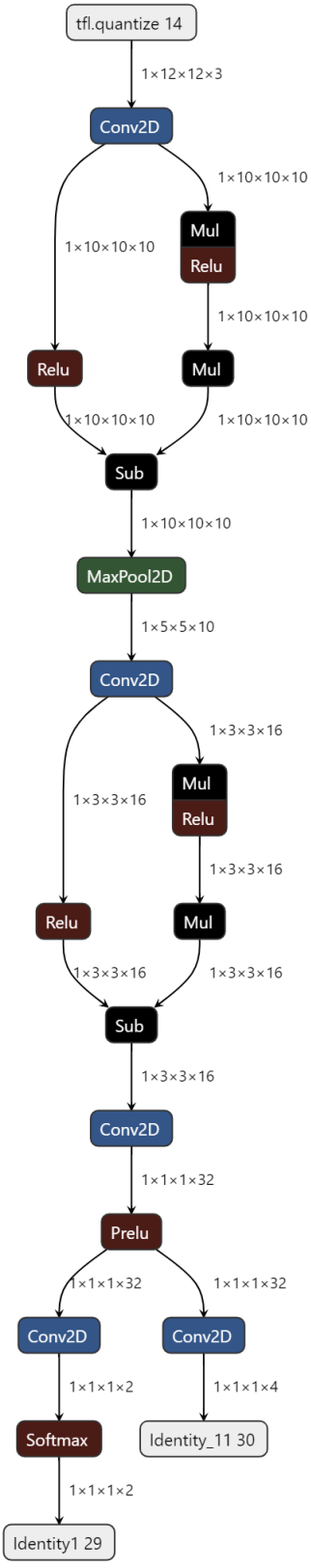


Figure A.2: Graph of P-Net.

R-Net

```
def r_net(sub_images):
    r_inp = tf.keras.layers.Input(shape=(24, 24, 3))
    r_layer = tf.keras.layers.Conv2D(28, kernel_size=(3, 3),
    strides=(
        1, 1), padding="valid", input_shape=())(r_inp)
    r_layer = custom_PRELU(shared_axes=[1, 2])(r_layer)
    r_layer = tf.keras.layers.MaxPooling2D(pool_size=(
        3, 3), strides=(2, 2), padding="same")(r_layer)

    r_layer = tf.keras.layers.Conv2D(48, kernel_size=(
        3, 3), strides=(1, 1), padding="valid")(r_layer)
    r_layer = custom_PRELU(shared_axes=[1, 2])(r_layer)
    r_layer = tf.keras.layers.MaxPooling2D(pool_size=(
        3, 3), strides=(2, 2), padding="valid")(r_layer)

    r_layer = tf.keras.layers.Conv2D(64, kernel_size=(
        2, 2), strides=(1, 1), padding="valid")(r_layer)
    r_layer = custom_PRELU(shared_axes=[1, 2])(r_layer)
    r_layer = tf.keras.layers.Flatten()(r_layer)
    r_layer = tf.keras.layers.Dense(128)(r_layer)
    r_layer = custom_PRELU()(r_layer)

    r_layer_out1 = tf.keras.layers.Dense(2)(r_layer)
    r_layer_out1 = tf.keras.layers.Softmax(axis=1)(
    r_layer_out1)

    r_layer_out2 = tf.keras.layers.Dense(4)(r_layer)
    model = tf.keras.Model(inputs=r_inp, outputs=[
    r_layer_out1, r_layer_out2])
    model = set_weights_for_r_net(model)
    return model(sub_images)
```

Listing A.4: Code for R-Net implementation

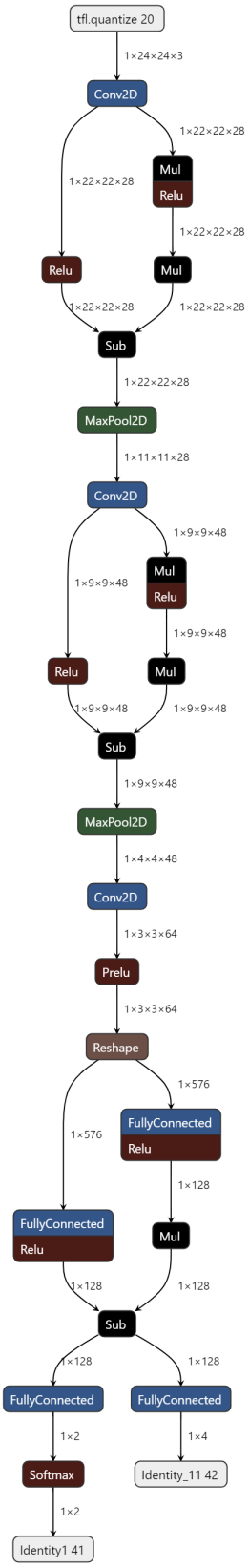


Figure A.3: Graph of R-Net.

O-Net

```
def o_net(sub_images):
    o_inp = tf.keras.layers.Input(shape=(48, 48, 3))

    o_layer = tf.keras.layers.Conv2D(32, kernel_size=(
        3, 3), strides=(1, 1), padding="valid")(o_inp)
    o_layer = custom_PRELU(shared_axes=[1, 2])(o_layer)
    o_layer = tf.keras.layers.MaxPooling2D(pool_size=(
        3, 3), strides=(2, 2), padding="same")(o_layer)

    o_layer = tf.keras.layers.Conv2D(64, kernel_size=(
        3, 3), strides=(1, 1), padding="valid")(o_layer)
    o_layer = custom_PRELU(shared_axes=[1, 2])(o_layer)
    o_layer = tf.keras.layers.MaxPooling2D(pool_size=(
        3, 3), strides=(2, 2), padding="valid")(o_layer)

    o_layer = tf.keras.layers.Conv2D(64, kernel_size=(
        3, 3), strides=(1, 1), padding="valid")(o_layer)
    o_layer = custom_PRELU(shared_axes=[1, 2])(o_layer)
    o_layer = tf.keras.layers.MaxPooling2D(pool_size=(
        2, 2), strides=(2, 2), padding="same")(o_layer)

    o_layer = tf.keras.layers.Conv2D(128, kernel_size=(
        2, 2), strides=(1, 1), padding="valid")(o_layer)
    o_layer = custom_PRELU(shared_axes=[1, 2])(o_layer)

    o_layer = tf.keras.layers.Flatten()(o_layer)
    o_layer = tf.keras.layers.Dense(256)(o_layer)
    o_layer = custom_PRELU()(o_layer)

    o_layer_out1 = tf.keras.layers.Dense(2)(o_layer)
    o_layer_out1 = tf.keras.layers.Softmax(axis=1)(
    o_layer_out1)
    o_layer_out2 = tf.keras.layers.Dense(4)(o_layer)
    o_layer_out3 = tf.keras.layers.Dense(10)(o_layer)
    model = tf.keras.Model(inputs=o_inp, outputs=[
    o_layer_out1, o_layer_out2, o_layer_out3])
    model = set_weights_for_o_net(model)
    return model(sub_images)
```

Listing A.5: Code for O-Net implementation

Scales

```
class Post_p_net_scale(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(Post_p_net_scale, self).__init__(**kwargs)

```

```
5       @tf.function(input_signature=[tf.TensorSpec(shape=(None,
        1), dtype=tf.float32), tf.TensorSpec(shape=(None, 1),
        dtype=tf.float32), tf.TensorSpec(shape=(None, 1), dtype=tf
        .float32)])
6       def call(self, y_position, x_position, scales_extended):
7           x1 = x_position[:, 0:1]
8           y1 = y_position[:, 0:1]
9           x2 = tf.math.add(x1, tf.math.multiply(
10              12.0, tf.math.reciprocal(scales_extended[:, 0:1])
        ))
11          y2 = tf.math.add(y1, tf.math.multiply(
12              12.0, tf.math.reciprocal(scales_extended[:, 0:1])
        ))
13          x1 = tf.math.floor(x1)
14          x2 = tf.math.floor(x2)
15          y1 = tf.math.floor(y1)
16          y2 = tf.math.floor(y2)
17          return tf.concat(
18              [x1, y1, x2, y2], axis=1)
```

Listing A.6: Code for scale layer

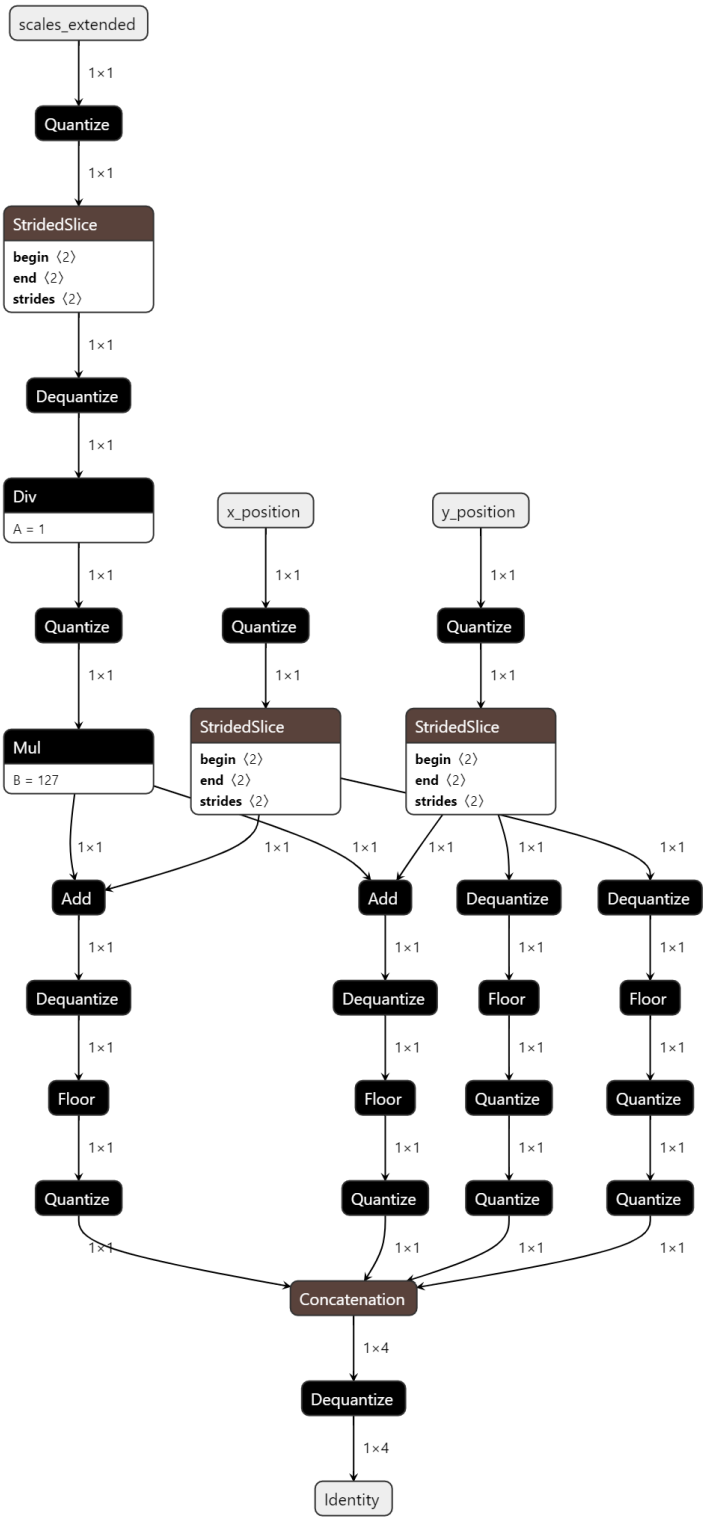


Figure A.4: Graph of scales layer.

Calibrate

```
class Calibrate(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(Calibrate, self).__init__(**kwargs)
    @tf.function(input_signature=[tf.TensorSpec(shape=(None,
    9), dtype=tf.float32)])
    def call(self, bboxes):
        w = tf.math.subtract(
            bboxes[:, 2:3], bboxes[:, 0:1])
        h = tf.math.subtract(
            bboxes[:, 3:4], bboxes[:, 1:2])
        qq1 = tf.math.multiply(bboxes[:, 5:6], w)
        qq1 = tf.math.add(bboxes[:, 0:1], qq1)
        qq2 = tf.math.multiply(bboxes[:, 6:7], h)
        qq2 = tf.math.add(bboxes[:, 1:2], qq2)
        qq3 = tf.math.multiply(bboxes[:, 7:8], w)
        qq3 = tf.math.add(bboxes[:, 2:3], qq3)
        qq4 = tf.math.multiply(bboxes[:, 8:9], h)
        qq4 = tf.math.add(bboxes[:, 3:4], qq4)
        return tf.concat([qq1, qq2, qq3, qq4], axis=1)
```

Listing A.7: Code for Calibrate layer

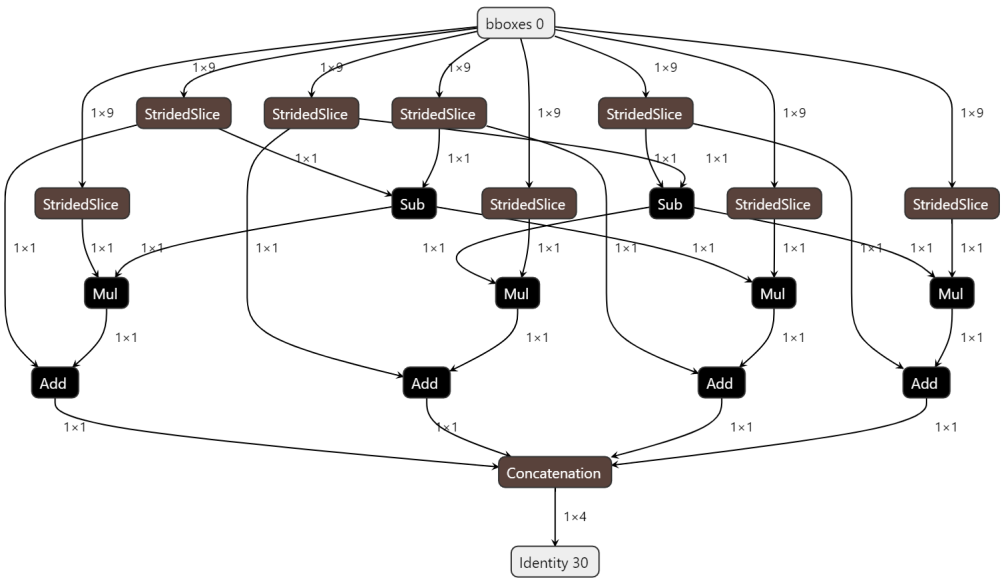


Figure A.5: Graph of calibrate layer.

Reshape

```
class Rerec(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(Rerec, self).__init__(**kwargs)
    @tf.function(input_signature=[tf.TensorSpec(shape=(None,
    4), dtype=tf.float32)])
    def call(self, bbox):
        heigth = tf.math.subtract(bbox[:, 3:4], bbox[:, 1:2])
        width = tf.math.subtract(bbox[:, 2:3], bbox[:, 0:1])
        max_side_length = tf.maximum(width, heigth)
        new_x1 = tf.math.multiply(
            tf.math.subtract(width, max_side_length), 0.5)
        new_x1 = tf.math.add(bbox[:, 0:1], new_x1)
        new_y1 = tf.math.multiply(
            tf.math.subtract(heigth, max_side_length), 0.5)
        new_y1 = tf.math.add(bbox[:, 1:2], new_y1)
        new_x2 = tf.math.add(bbox[:, 0:1], max_side_length)
        new_y2 = tf.math.add(bbox[:, 1:2], max_side_length)
        return tf.concat([new_x1, new_y1, new_x2, new_y2],
    axis=1)
```

Listing A.8: Code for Reshape layer

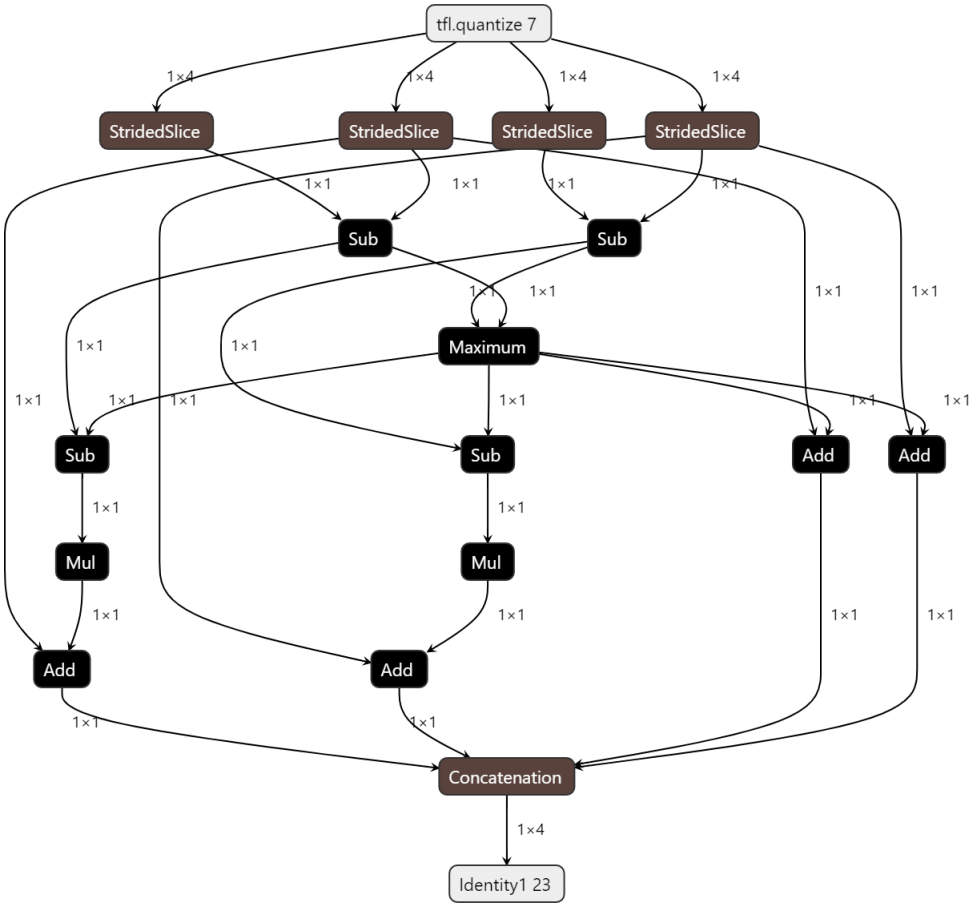


Figure A.6: Graph of reshape layer.

Custom Prelu

```
class custom_PRELU(tf.keras.layers.Layer):

    def __init__(self, alpha_initializer='zeros',
    alpha_regularizer=None, alpha_constraint=None, shared_axes
    =None, **kwargs):
        super(custom_PRELU, self).__init__(**kwargs)
        self.supports_masking = True
        self.alpha_initializer = initializers.get(
    alpha_initializer)
        self.alpha_regularizer = regularizers.get(
    alpha_regularizer)
```

```
8           self.alpha_constraint = constraints.get(
    alpha_constraint)
9           if shared_axes is None:
10              self.shared_axes = None
11          elif not isinstance(shared_axes, (list, tuple)):
12              self.shared_axes = [shared_axes]
13          else:
14              self.shared_axes = list(shared_axes)
15
16      def build(self, input_shape):
17          param_shape = list(input_shape[1:])
18          if self.shared_axes is not None:
19              for i in self.shared_axes:
20                  param_shape[i - 1] = 1
21          self.alpha = self.add_weight(
22              shape=param_shape,
23              name='alpha',
24              initializer=self.alpha_initializer,
25              regularizer=self.alpha_regularizer,
26              constraint=self.alpha_constraint)
27          axes = {}
28          if self.shared_axes:
29              for i in range(1, len(input_shape)):
30                  if i not in self.shared_axes:
31                      axes[i] = input_shape[i]
32          self.input_spec = InputSpec(ndim=len(input_shape),
    axes=axes)
33          self.built = True
34
35      @tf.function(input_signature=[tf.TensorSpec(shape=(None),
     dtype=tf.float32)])
36      def call(self, inputs):
37          # positive branch
38          pos = tf.nn.relu(inputs)
39          # negative branch
40          neg = tf.math.multiply(inputs, -1)
41          neg = tf.nn.relu(neg)
42          neg = tf.math.multiply(neg, self.alpha)
43          return tf.math.subtract(pos, neg)
```

Listing A.9: Code for custom PReLU layer

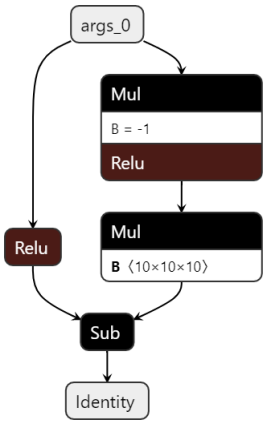


Figure A.7: Graph of custom prelu operation.

Printed by Tryckeriet i E-huset, Lund 2021

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2021-835
http://www.eit.lth.se