Concurrent Deep Learning Inference with Multiple Models on Single Neural Processing Units

Project Thesis SS2022

Master of Science in Engineering
Electrical Engineering

Author
Michael Schmid
Supervisor
Prof. Dr. Andreas Breitenmoser, IMES
Project Partner
Dr. Sebastian Stenzel, Sonova

OST - Ostschweizer Fachhochschule, Campus Rapperswil
July 1, 2022

This document is created with LaTeX and Ti*k*Z

The layout is based on Prof. Dr. Andreas Müller's *Partial Differential Equations* lecture notes

Abstract

Introduction Deep learning applications are slowly but surely making the transition from the stage of fundamental research to its practical applications. Everyday implementations cannot rely on large computer systems with Graphics Processing Units (GPUs), thus Deep Neural Networks (DNNs) are increasingly computed on space-saving and low-cost embedded devices. Embedded devices are subject to various limitations in terms of performance, power consumption and memory requirements. Today, complex systems with advanced sensor fusion are expected to do more than just compute inferences from a single DNN. A system with an embedded device that computes multiple DNN is demanded. The goal of this project is to investigate the issue of parallel execution of multiple DNN, using the Arm *Ethos-U* Neural Processing Unit (NPU). The *Ethos-U* NPU is a special class of deep learning processors for which supported DNNs are fused into an optimized operator which is then directly computed on the deep learning Application-Specific Integrated Circuit (ASIC). This process is not preemptive, so it cannot be interrupted. In order not to simply run several DNN sequentially, a way must be found to split the calculation of one DNN into several parts.

Approach First, the development environment around the *Ethos-U* was set up to compute self-trained DNN on the *Corestone-300* simulation platform. Then, different ways to compute a DNN in several parts were shown. The networks were modified in different hierarchy levels of the typical deep learning workflow. After all the functionalities of the modified networks have been tested, a system was built to execute them in parallel with a Real Time Operating System (RTOS). An appropriate task scheduling is carried out for the execution of DNNs with different inference rates as well as different priorities.

Conclusion The basic components of such a system are presented and many important tasks could be pointed out. The modification of the DNNs for parallel execution could all be demonstrated successfully and the functionality of the DNNs is still given after the modification. However, the implementation with the RTOS in combination with the NPU proved to be non-trivial. The correct interaction of the task scheduler and the deep learning implementation of TensorFlow Lite for Microcontrollers (TFLM) could not be applied and guaranteed correctly in all cases. The approach seems to be the right one, but more time needs to be invested for the correct implementation.

Acknowledgement

I would like to express my great appreciation to my supervisor, Prof. Dr. Andreas Breitenmoser, for his valuable and constructive support. A special thanks also goes to Dr. Sebastian Stenzel, the co-examinator of this project. They have helped me in many ways throughout the project with their extensive experience. I thank *Sonova* for providing the idea of this project.

Contents

Abbreviations

AI Artificial Intelligence

DNN Deep Neural Network

ANN Artificial Neural Network

CNN Convolutional Neural Network

RNN Recurrent Neural Network

ReLU Rectified Linear Unit

PReLU Parametric Rectified Linear Unit

DPU Deep-Learning Processor Unit

GPU Graphics Processing Unit

CPU Central Processing Unit

FPGA Field Programmable Gate Array

DSP Digital Signal Processor

VPU Vector Processing Unit

NPU Neural Processing Unit

ASIC Application Specific Integrated Circuit

IFM Input Feature Map

OFM Output Feature Map

DMA Direct Memory Access

SRAM Static Random-Access Memory

MAC Multiply Accumulate

RTOS Real Time Operating System

TFLM TensorFlow Lite for Microcontrollers

TFL TensorFlow Lite

TF TensorFlow

ASIC Application-Specific Integrated Circuit

AXI Advanced eXtensible Interface

LSTM Long Short-Term Memory

Chapter 1

Introduction

Motivation Machine learning and particularly deep learning techniques have shown to be exceptionally powerful for certain applications. As a consequence, more and more deep learning algorithms are also deployed on embedded devices. However, embedded systems come with their very own challenges - resources like processing power, memory or power supply are highly limited while requirements like responsiveness or real-time constraints are rather demanding.

An embedded system is often dedicated to one specific application. In the case of deep learning, it would run inference using a single NPU model that was trained for this exact application. Additionally, the system may be supported by coprocessors or hardware accelerators to boost the performance of the use case.

More complex embedded systems, such as hearing aids, may be tasked with more than one application and must serve multiple use cases in parallel. For example, hearing aids sense their surroundings constantly to classify the sound scene, receive and deliver enhanced audio, or listen to the user's inputs, such as voice commands. The realization of two or more such use cases through deep learning techniques now raises the question as to how the different NPU models can be executed on given hardware concurrently.

The goal of this project is to investigate this question of running multiple DNNs use cases in parallel, making use of the Arm *Ethos-U55* NPU in combination with a *Cortex M-55* or *M-33* microcontroller unit, and possibly further NPUs [1].

Approach The task was divided into several parts. First, the development environment around the NPU is set up. After that, a profound knowledge base about the world of Artificial Intelligence (AI) in the *Arm* development environment must be acquired. After that, the workflow will be familiarized and own DNNs will be successfully computed on the simulated NPU on the *Corstone-300* fixed virtual platform. Once everything is ready, the first attempts at a concurrent system can be made and various possibilities are to be tested. First with two sequential DNNs, then with networks which have a different repetition frequency and computational load. The use of a RTOS has to be continuously reviewed and incorporated into the project as appropriate.

Content Chapter 2 presents fundamentals about deep learning with a focus on the embedded implementations. Also, the principles about AI with *arm* and various DNN splitting techniques are presented. In chapter 3, the methods on how to implement a system for running concurrent DNNs on the *Ethos-U* NPU are shown. Chapter 4 states the results and chapter 5 presents the conclusion of the project.

Chapter 2

Fundamentals

2.1 Deep Learning

Many software systems of any kind are based on hard-coded parameters set by the engineer developing the system. AI can be used to generate knowledge from data. The generic term AI can be divided into three subcategories [2].

The essential part of this section has already been presented in [3] by the same author as in this paper.

- **Machine learning** uses various techniques to analyze data. A typical example would be classifying data into two subsets. A machine learning algorithm has raw data as input, in order for such algorithms to work properly, the data must be represented appropriately.
- **Representation Learning** is used to learn a good representation itself. Hand engineered representation can be very laborious and needs quite some guesswork sometimes. Machine learning algorithms can result in much better results and can be adapted quickly to a new set of data.
- **Deep Learning** provides a solution which includes the machine learning and the representation learning part. The term *deep* comes from the multilayer structure of the presented networks in deep learning

2.1.1 Neuron

The perceptron or neuron presented by Rosenblatt [4], was the first neuron with learnable parameters. The neuron is inspired by the human brain, hence the name. Such a neuron is defined by

$$f(\mathbf{x}) = \hat{y}_1 = \sigma(\mathbf{w} \cdot \mathbf{x} + b), \tag{2.1}$$

where

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{n} w_i x_i, \tag{2.2}$$

is the definition of the scalar product, $\sigma(\cdot)$ is an activation function and n is the number of inputs. The visualization of the neuron can be seen in Fig. 2.1.

Such a neuron is a binary classifier for supervised learning, where the weights $\mathbf{w}$ and the bias term b are learned.

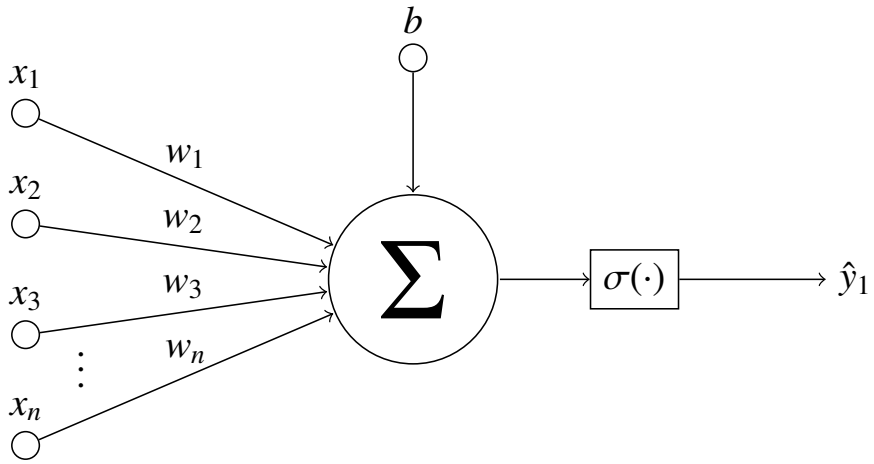


Figure 2.1: Graph of a neuron with n-inputs

2.1.2 Artificial Neural Network

This section about the Artificial Neural Network (ANN) is heavily inspired by [5]. Fundamentals about the training of ANNs are presented in Appendix C.1. An ANN is composed of many neurons, which are connected layer-wise with one another. In Fig. 2.2 the structure of such a network can be seen.

2.1.2.1 Forward Pass

The output $a_i(l)$ of the neuron i in layer l is illustrated in Fig. 2.3 and can be numerically computed with

$$a_i(l) = \sigma\left(\sum_{j=1}^{n_{l-1}} w_{ij}(l) a_j(l-1)\right) + b_i(l), \tag{2.3}$$

where n_{l-1} are the number of nodes of the previous layer. This can be written in matrix notation, with

$$\mathbf{z}(l) = \mathbf{W}(l)\,\mathbf{a}(l-1) + \mathbf{b}(l) \qquad \text{where } l = 2, 3, \ldots, L, \tag{2.4}$$

where the weights are arranged as follows

$$\mathbf{W}(l) = \begin{bmatrix} W_{11}(l) & W_{12}(l) & \cdots & W_{1n_{l-1}}(l) \\ W_{21}(l) & W_{22}(l) & \cdots & W_{2n_{l-1}}(l) \\ \vdots & \vdots & \ddots & \vdots \\ W_{n_l 1}(l) & W_{n_l 2}(l) & \cdots & W_{n_l n_{l-1}}(l) \end{bmatrix} \tag{2.5}$$

The activation function is applied to each neuron separately.

$$\mathbf{a}(l) = \sigma[\mathbf{z}(l)] = \begin{bmatrix} \sigma(z_1(l)) \\ \sigma(z_2(l)) \\ \vdots \\ \sigma(z_{n_l}(l)) \end{bmatrix}. \tag{2.6}$$

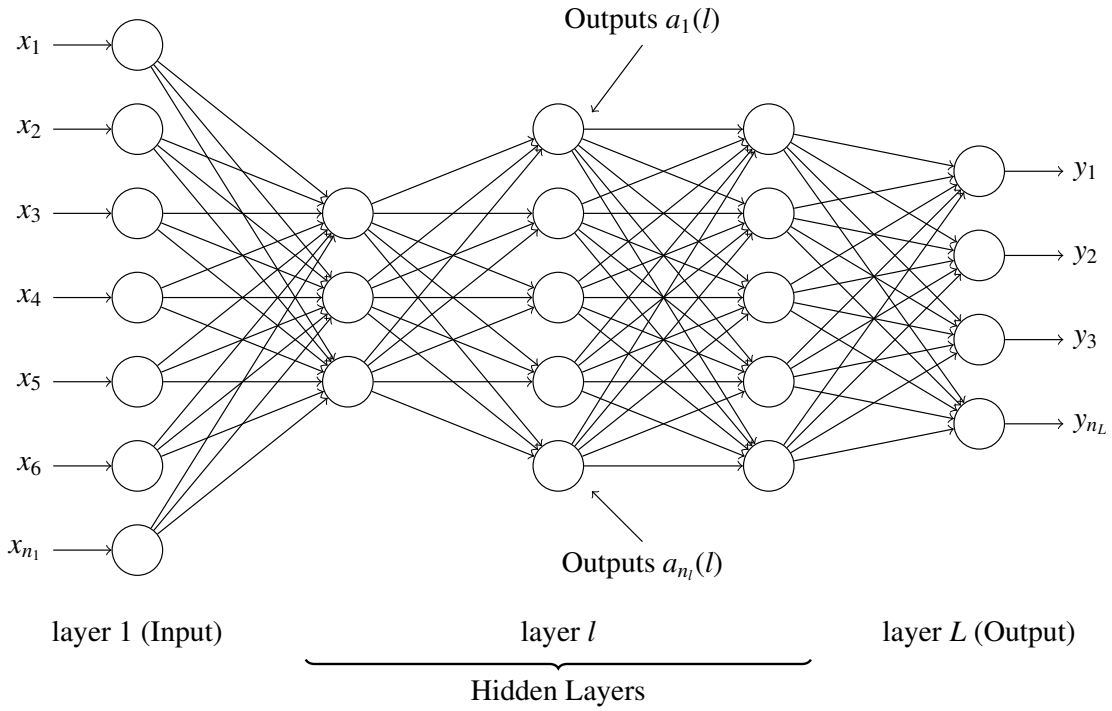


Figure 2.2: Graph of a ANN with L layers, which is also called *Fully Connected Neural Network* because each output of a neuron is connected with every neuron of the next layer. The first layer with n_1 nodes is connected to the hidden layers with n_l nodes. The last layer with n_L nodes provides the output.

2.1.3 Convolutional Neural Network

Through an ANN, each input node is connected to every other input node. Let's imagine an image of a dog where the network must correctly classify the animal. The top right pixel and the bottom left pixel, in combination, contain little to no information about the animal in the image. Much more information is contained in the area around a pixel. Convolution with a learned filter extracts the information from a surrounding area. Also, convolution is a translation invariant operation, which is a very useful property since it does not matter where the dog is located in the image.

In Fig. 2.4 a typical Convolutional Neural Network (CNN) architecture with a 2D input image is shown.

The output of a CNN can be a convolutional layer or a ANN (mostly for classification).

2.1.3.1 Forward pass in a CNN

With the same notation as in section 2.1.2.1, the convolution with the kernel $w(l)$ from layer l with the output $a(l-1)$ of the previous layer

$$\begin{aligned} z_{x,y}(l) &= \sum_l \sum_k w_{l,k}(l) a_{x-l,y-k}(l-1) + b(l) \\ &= w(l) * a_{x,y}(l-1) + b(l), \end{aligned} \tag{2.7}$$

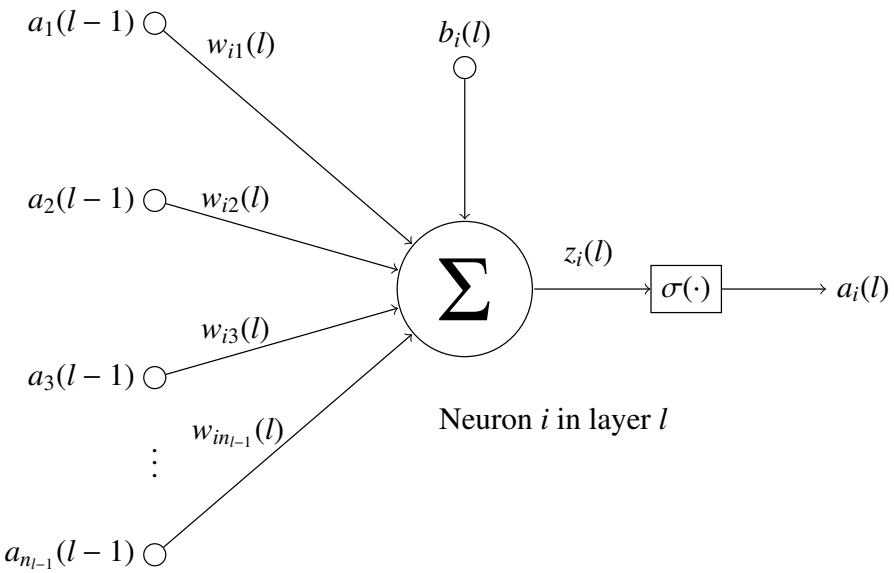


Figure 2.3: Graph of the i^{th} neuron in layer l of an ANN. The inputs $a_i(l-1)$ are from the output from the previous layer $l-1$, the weights w and the bias b are corresponding to the layer l, where $\sigma(\cdot)$ is an activation function

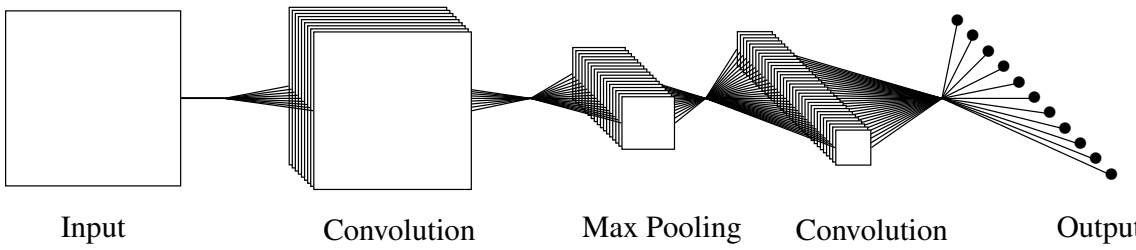


Figure 2.4: Graph of a CNN

where l and k span the the dimension of the kernel.

The convolution can be written with the $*$ operator. The output of a convolutional layer with the activation function

$$a_{x,y}(l) = \sigma\left(z_{x,y}(l)\right). \tag{2.8}$$

With this convolution the output is smaller than the input. Sometimes it is desired to have the same output shape as the input shape, in which case the input is padded with zeros. In Fig. 2.5 the convolution of one input feature and three filters (kernel) is illustrated.

In Fig. 2.6 the convolution of four input features and three filters (kernel) is illustrated.

2.1.3.2 Layers

There are many layers which are used in CNNs. Here, the layers relevant to this work are briefly introduced.

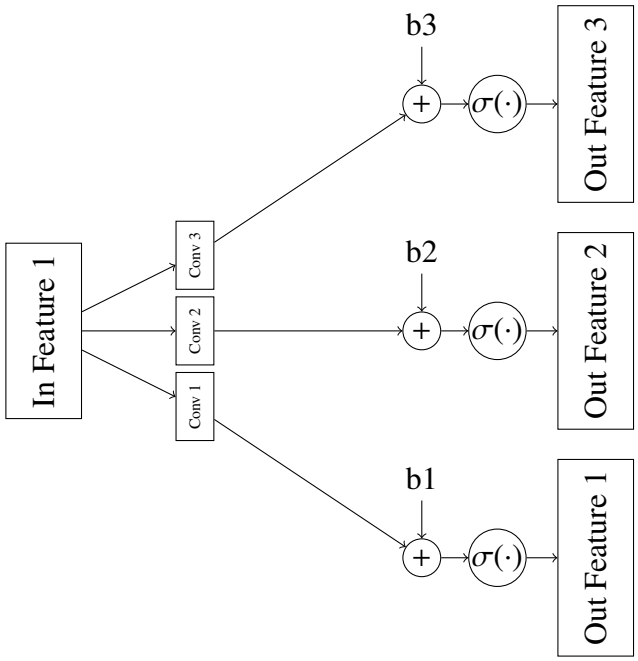


Figure 2.5: Graph of a convolutional layer with one input feature map and 3 filters. The input feature is convoled with three different kernels

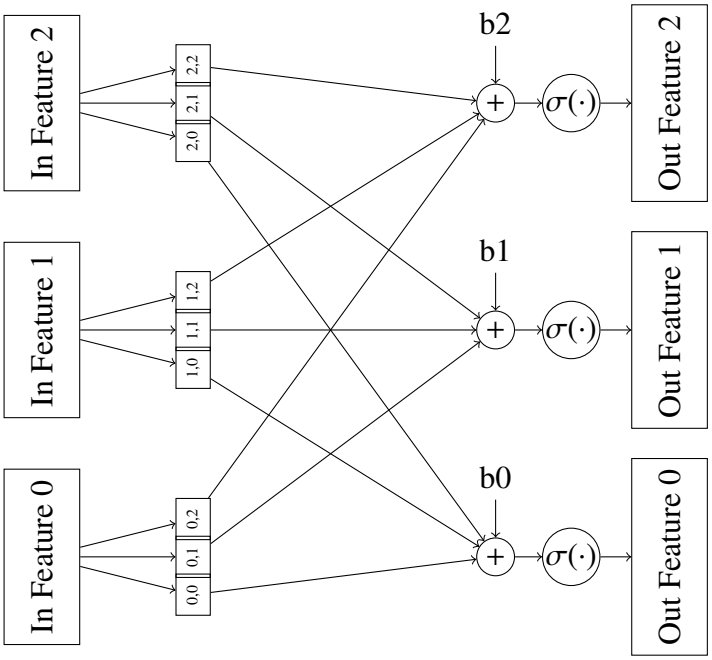


Figure 2.6: Graph of a convolutional layer with 3 input feature maps and 3 filters. Every input feature is convolved with four filters, in total there are $n_{convs} = n_{input} \cdot n_{filters}$ in this case $3 \cdot 3 = 9$ convolutions

Pooling For dimension reduction a pooling layer is used. As an example: Max pooling picks the max value of a certain neighborhood.

A 2×2 max-pooling layer would result in

$$\begin{bmatrix} 6 & -34 & 34 & -45 \\ 7 & -4 & 3 & 56 \\ 45 & 3 & 78 & 34 \\ -5 & 34 & -73 & 34 \end{bmatrix} \Rightarrow \begin{bmatrix} 7 & 56 \\ 45 & 78 \end{bmatrix}. \tag{2.9}$$

A 2×2 average-pooling layer would result in

$$\begin{bmatrix} 6 & -34 & 34 & -45 \\ 7 & -4 & 3 & 56 \\ 45 & 3 & 78 & 34 \\ -5 & 34 & -73 & 34 \end{bmatrix} \Rightarrow \begin{bmatrix} -6.25 & 12 \\ 19.25 & 18.25 \end{bmatrix}. \tag{2.10}$$

Up Sampling This layer repeats its value for n times in two dimensions. An example with $n = 2$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}. \tag{2.11}$$

Concatenate *"This layer takes as input a list of tensors, all the same shape except for the concatenation axis, and returns a single tensor that is the concatenation of all inputs"* [6]

2.1.4 Activation Function

The activation function at the output of a neuron can be of various shapes. The main idea is to bring non-linearity into the network. A neuron by itself is a linear operation. Following the commonly used functions and some which are relevant to this work.

Softmax The Softmax function

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad \text{for } i = 1, \ldots, k \tag{2.12}$$

normalizes input to a probability distribution. The softmax activation is mostly used for multi-dimensional classification in the last layer of a network.

Sigmoid For binary classification the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{x}} \tag{2.13}$$

can be used. It also converts the input into a probability

ReLU The Rectified Linear Unit (ReLU) function (Fig. 2.8)

$$\sigma(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases} \tag{2.14}$$

is widely used and can be implemented on hardware with ease.

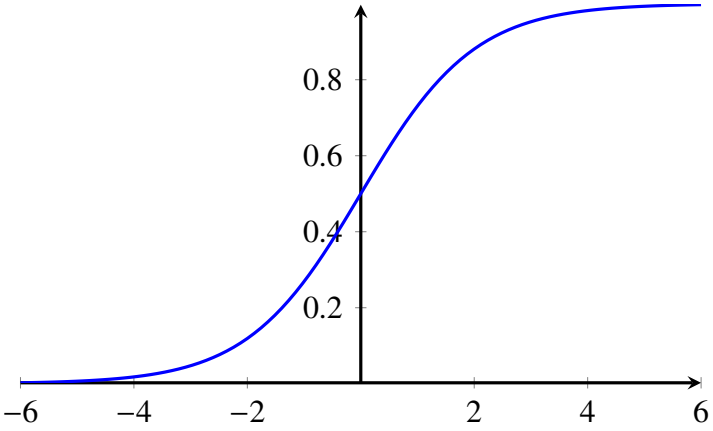


Figure 2.7: Sigmoid function

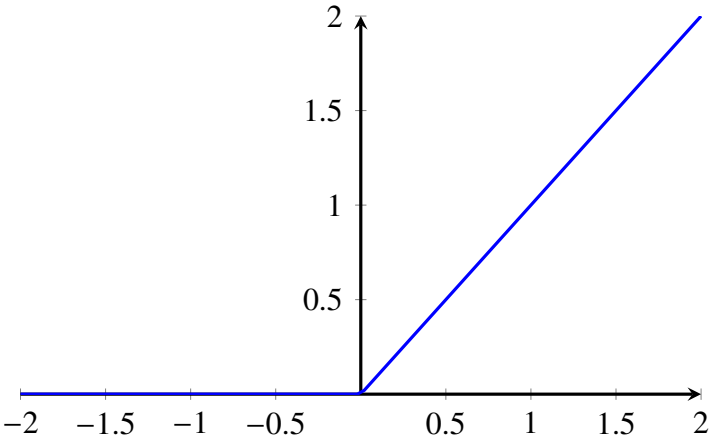


Figure 2.8: The ReLU function

LeakyReLU The LeakyReLU (Fig. 2.9)

$$\sigma(z) = \begin{cases} \alpha z, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases} \tag{2.15}$$

is very similar to the ReLU function, but with a predefined leak term α.

PReLU The Parametric ReLU is very similar to the LeakyReLU but the leak term α is made into a learnable parameter.

Swish The Swish function (Fig. 2.10)

$$\sigma(z) = \frac{z}{1 + e^{-z}} \tag{2.16}$$

looks similar to the ReLU but is derivable for higher order derivatives [7].

Hard-Swish The Hard-Swish function (Fig. 2.11)

$$\sigma(z) = x\frac{\text{ReLU6}(x + 3)}{6} \tag{2.17}$$

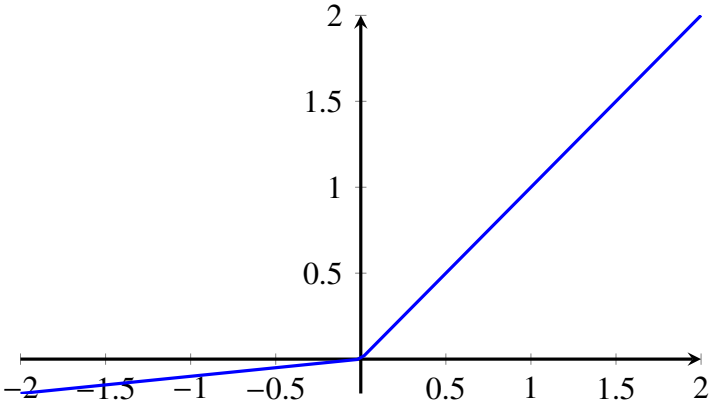


Figure 2.9: The LeakyReLU function

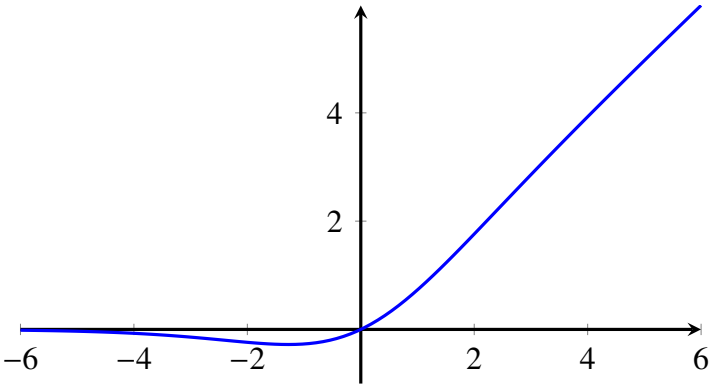


Figure 2.10: The Swish function

$$\text{ReLU6}(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{if } z \leq z \leq 6 \\ 0, & \text{if } 6 \leq z \end{cases} \tag{2.18}$$

replaces the computationally expensive sigmoid with linear operations [8].

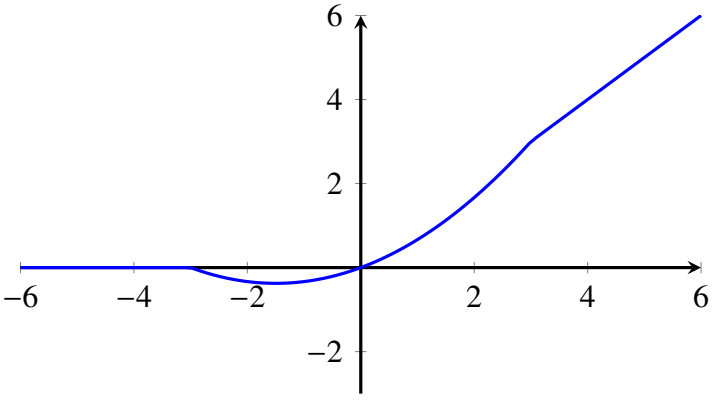


Figure 2.11: The Hard-Swish function

Elu The Elu function (Fig. 2.12)

$$\text{elu}(z) = \begin{cases} \sigma(z) = \alpha\left(e^z - 1\right), & \text{if} z < 0 \\ z, & \text{if } z \geq 0 \end{cases} \tag{2.19}$$

looks similar to the ReLU but has negative values. [9].

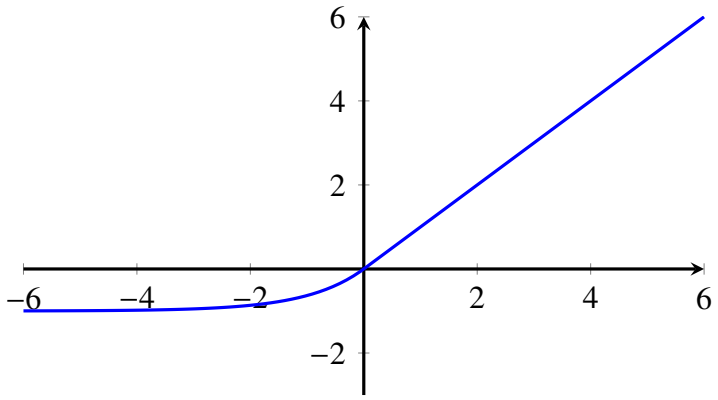


Figure 2.12: The Elu function

Tanh The Tanh function (Fig. 2.13)

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.20}$$

.

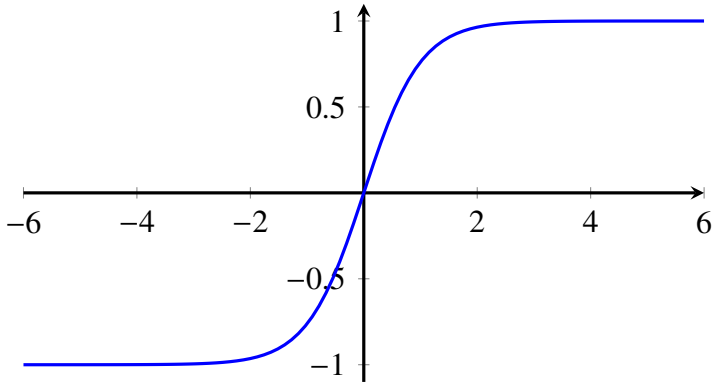


Figure 2.13: The Tanh function

2.1.5 Model Optimization

There are several methods to optimize a neural network for inference efficiency. Computationally efficient inference is of particular interest for embedded devices. Not only the number of operations per inference is crucial but also the effective required memory for storing the weights and biases.

Two possibilities:

- Reduce parameter count with pruning and structured pruning.
- Reduce representational precision with quantization.

2.1.5.1 Pruning

Weight pruning is the process of setting insignificant weights to zero to make the model sparser. Models with many weights set to zero are easier to compress. In sophisticated deep learning frameworks, weight pruning can also improve inference latency (promised for a future version of TensorFlow Lite (TFL)) [10]. Fig. 2.14 shows synapses pruning and Fig. 2.15 shows neuron pruning.

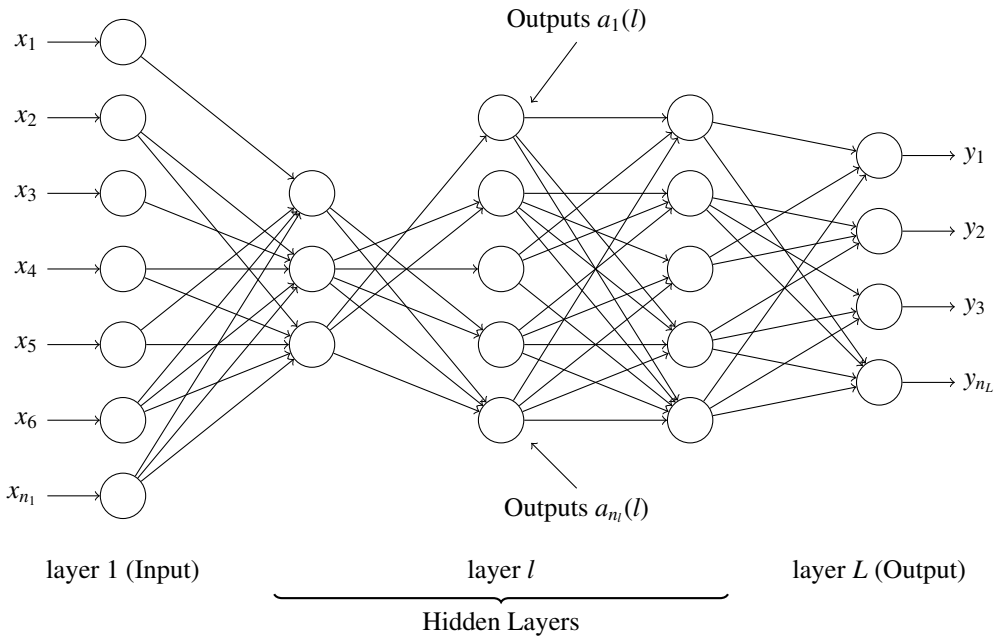


Figure 2.14: Fully connected neural network with applied synapses pruning, see the randomly missing connections between nodes

2.1.5.2 Weight Clustering

With the weight clustering technique, weights of a layer are shared within a finite group of clusters. The weights of these clusters are set to the clusters centroid value. This leads to a reduction of the memory usage of a DNN. Experiments have shown a five times improvement in model compression and with no significant loss in accuracy [11]. Fig. 2.16 visualizes the weight clustering.

2.1.5.3 Quantization

Quantization converts a model, which is typically trained in 32-bit float accuracy to a lower precision such as 8-bit integer or 16-bit float. For general computing architectures the time it takes to compute for example 8-bit integer multiplication is significantly lower as a 32-bit

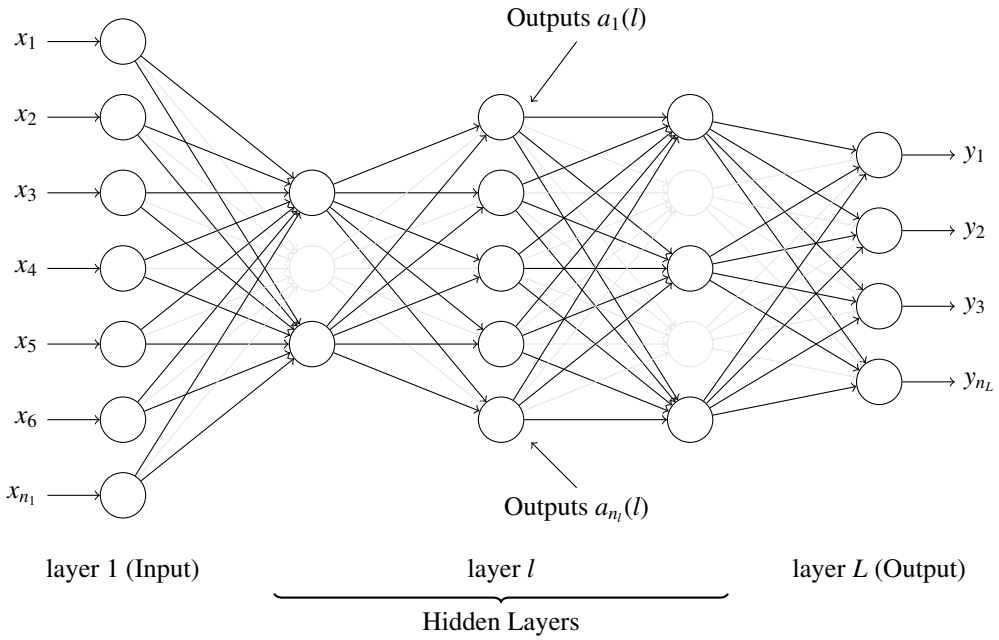


Figure 2.15: Fully connected neural network with applied neuron pruning

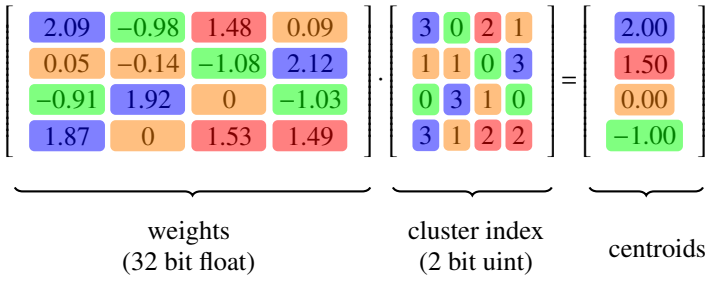


Figure 2.16: Weights clustering by scalar quantization [12]

floating point multiplication. Therefore, quantized networks usually have much lower latency. The quantization applied during inference with its different datatype is shown in Fig. 2.17 [13].

Post-Training Quantization The most common 8-bit integer post-training quantization uses a small subset of validation data to generate the numerical distribution of the model. With this distribution the quantization is computed.

Quantization Aware-Training Quantization aware training runs inference-time quantization during the training phase and uses that information to train the model. This leads to better accuracy after the final quantization.

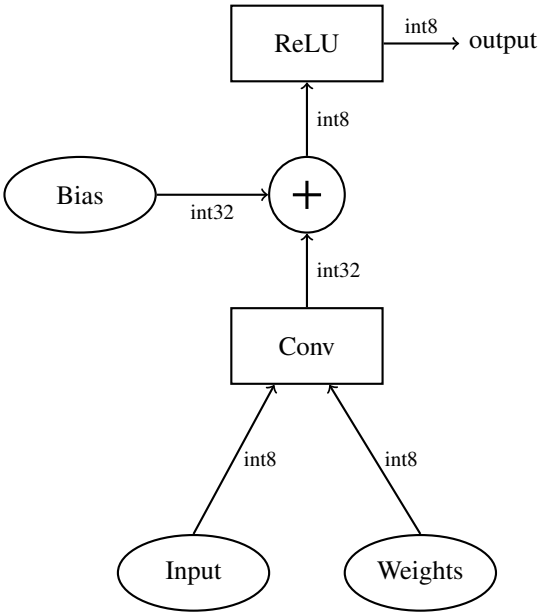


Figure 2.17: Integer-arithmetic-only inference

2.2 Artificial Intelligence with *Arm*

Like many semiconductor and software companies, *Arm* is pushing into the AI market these days. In recent years, they have offered several products in association with AI.

2.2.1 Cortex-M55

The *Cortex-M55* is *Arm's* most performant AI processor and the first one with a Vector Processing Unit (VPU). The VPU delivers power-efficient Digital Signal Processor (DSP) and AI performance. It is designed for 8-bit, 16-bit and 32-bit fixed point arithmetic, in Table 2.1 the throughput per cycle for a Multiply Accumulate (MAC) operation is shown.

Datatype	8-bit integer	16-bit integer	32-bit integer	Half-precision floating point	Single-precision floating point
MACs/cycle	8	4	2	4	2

Table 2.1: Throughput per cycle that can be achieved when executing a multiply-accumulate operation with the VPU [14]

Deep learning networks can be run directly on the *Cortex-M55* with the *CMSIS-NN* library and TFLM: "*CMSIS NN software library is a collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks on Cortex-M processors*" [15].

2.2.2 Ethos-NPU

The *Ethos-U* NPU is an ASIC specifically designed to accelerate deep learning inference for embedded devices. The *Ethos-U* NPU combined with the AI-capable *Cortex-M55* processor provides an uplift in performance over existing *Cortex-M* based systems [16].

Its key points:

- Energy Efficient
- Network support
  - Supports most common neural networks such as CNNs and Recurrent Neural Networks (RNNs)
- Future-Proof Operator Coverage
  - Common computationally expensive operators run NPU such as convolution, Long Short-Term Memory (LSTM), RNN, pooling, activation functions and so on. Non supported kernels run automatically on the Cortex-M55 using *CMSIS-NN*
- Offline Optimization
  - Offline compilation and optimization of neural networks, performing operator, and layer fusion as well as layer reordering
- Mixed Precision
  - Supports 8-bit integer and 16-bit integer

From the *Ethos-U* technical manual a brief description of the NPU: "*A neural network must be compiled offline using the open-source compiler to produce a command stream. The application invokes the driver, which communicates with the NPU to tell it where the command stream is and initiates the network traversal (see Fig. 2.18). The command stream describes the steps necessary for the NPU to execute the operators compiled into the command stream autonomously. When complete, the NPU raises an interrupt request to the driver. The NPU includes a Direct Memory Access (DMA) controller that can read and write to external memory (see Fig. 2.19). When the NPU performs inferences, the DMA controller reads the neural network description. This description contains:*

- *The command stream*
- *Network weights*
- *Bias information*
- *Scale information*

The DMA controller also transfers the Input Feature Maps (IFMs) and Output Feature Maps (OFMs) and NPU-private intermediate data that is also held in system memory (see Fig. 2.20). During runtime, TensorFlow Lite loads the `flatbuf` *file, in which the offline Compiler has created an Ethos-U command stream for each custom operator. The driver gives a pointer to this command stream so that the NPU hardware can execute it. This means that the*

entire network can be a single operator that is run fully on the Ethos-U. The NPU reads the data (weights, commands, IFMs, OFMs, bias and scale) autonomously using the DMA. The NPU uses a working buffer in Static Random-Access Memory (SRAM) for IFMs and OFMs in flight. The offline Compiler decides the scheduling of this buffer and codes it into the command stream. The NPU uses the DMA to read and write autonomously to this work buffer. The location of the buffer is set at runtime through registers, meaning the coding in the command stream is relative, not absolute" [17].

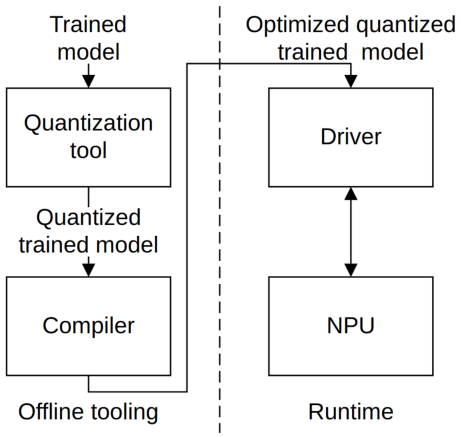


Figure 2.18:

- Offline tooling
  - Quantization of the trained model with its weights to 8-bit and activations to 8-bit or 16-bit values
  - The quantized model is then compiled. This optimizes the model for the NPU and outputs an optimized model that contains a command stream for the NPU
- Runtime tooling
  - The model is placed in system memory, which is accessible by the NPU
  - During runtime, the TFL tool reads the model and configures the operators
  - The NPU reads the model and runs the command stream. The processor runs any operation of the neural network that the NPU cannot execute
  - When inference is complete, the result is placed in a specific memory location [17]

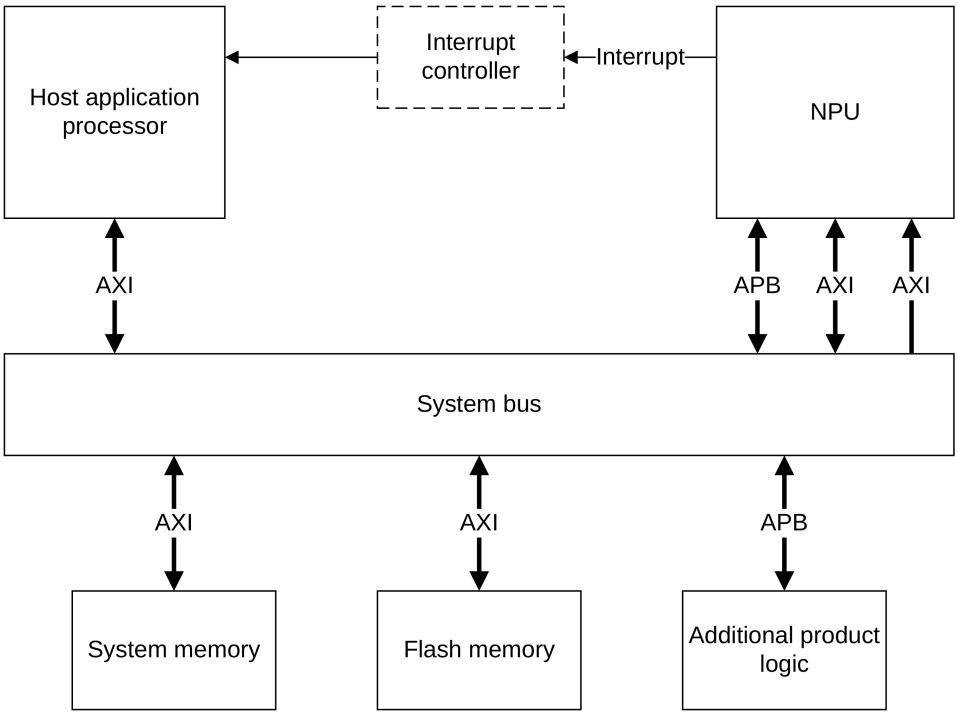


Figure 2.19: The block diagram of the NPU with the application processor. All function blocks are connected to the same system bus, which is responsible for data transfer [17]

2.2.2.1 Functional Blocks

The NPU itself consists several functional blocks (Fig. 2.21).

- The central control is the control unit of the NPU, it controls processing flow of the DNNs and manages the data transfer over the DMA bus.
- The DMA controller handles the data flow over the Advanced eXtensible Interface (AXI) bus
- The weight decoder reads the weights from the DMA and processed them to be read by the MAC unit.
- The MAC unit has an IFM unit, dot products units and an adder array.
  - The IFM unit reads the feature maps from the memory and feds them into the multipliers in the dot product units. It also zero-padding and upscaling if required
  - The dot product units perform the MAC operations for the convolutions, and it contains also a max operator for max pooling
  - The adder array reads the data from the accumulators and updates them with the current accumulations from the dot product units
- The output unit reads the finished data from the accumulators and scales them from the OFM. It also adds the bias and applies the activation function. Such as ReLU LeakyReLU, *tanh*, sigmoid, configurable lookup tables and more (see in section 2.1.4).

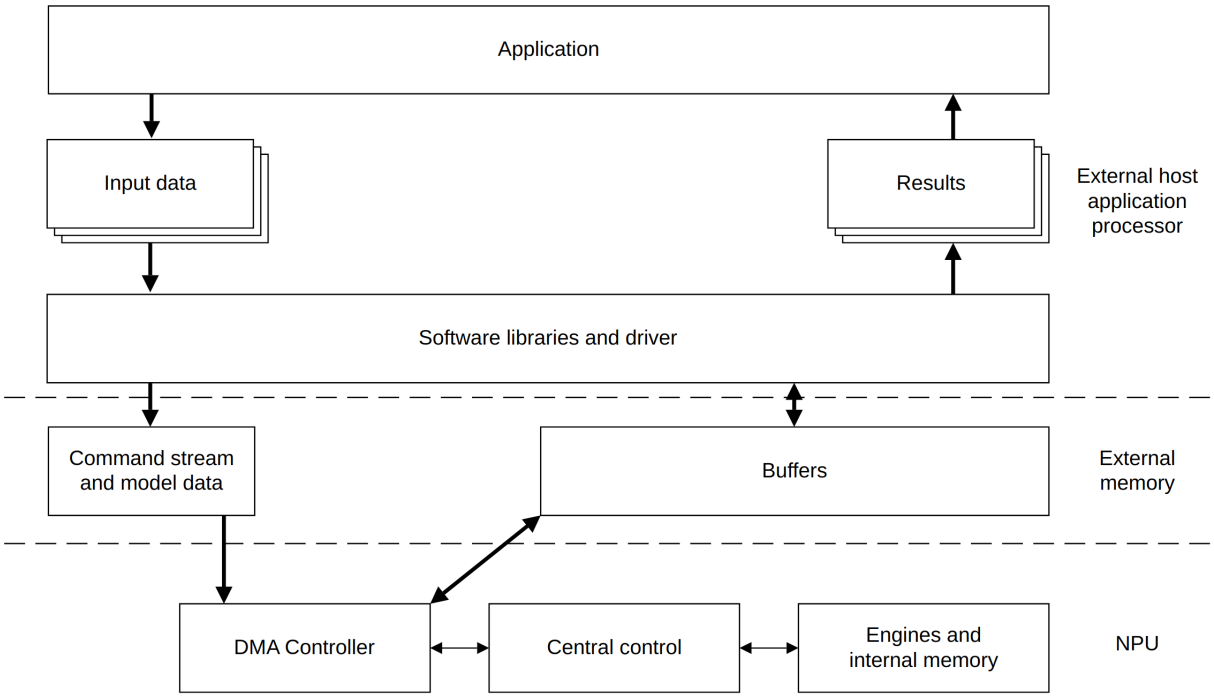


Figure 2.20: The control and data flow shows how everything is connected. The input data and results are processed by the host application which writes or reads from the external memory. The external memory is connected to the DMA controller from which the NPU is reading and writing [17]

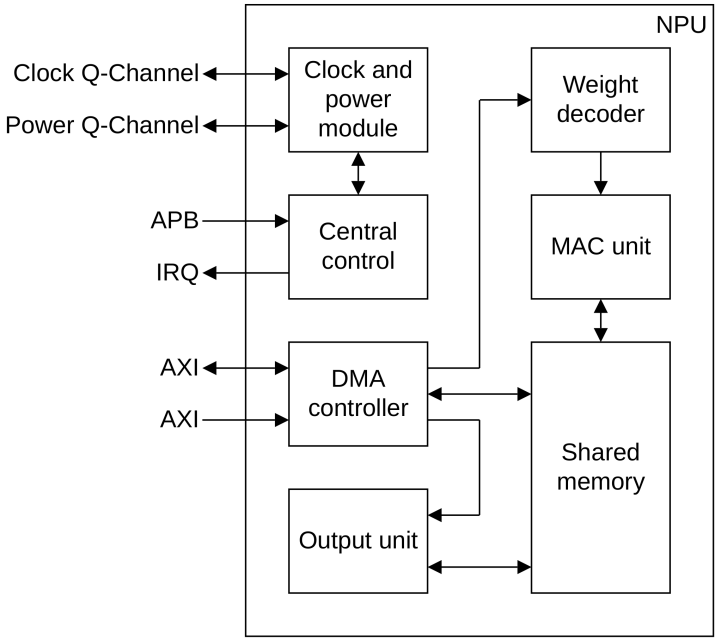


Figure 2.21: The NPU is composed of the central control, a DMA controller, a MAC unit, an output unit, and the interconnect fabric [17]

2.2.3 Vela

Vela is an open-source `Python` tool which compiles a DNN model to be run on an embedded system including an *Ethos-U* NPU. The optimized model contains a TFL custom operators for the NPU, for all by the *Ethos-U* supported layers. If some portion of the network cannot be accelerated by the NPU (unsupported layers), they are left unchanged and will instead run on the *Cortex-M55* [18].

However, a supported layer cannot be configured arbitrarily, they also still have specific requirements. Following is a list of constraints that for example the CONV_2D operator must satisfy in order to be scheduled on the NPU.

- Stride values for both width and height must be integer types
- Dilation factor values for both width and height must be integer types
- Stride values for both width and height must be in the range [1, 3]
- Dilation factor values for both width and height must be in the range [1, 2]
- Dilated kernel height must be in the range [1, 64]
- Product of dilated kernel width and height must be in the range [1, 4096]
- Weight tensor must be 8-bit
- Weight tensor must be constant
- The sum of the weights cannot exceed 8323072
- Optional Bias tensor must be of type: int32, int64
- Optional Bias tensor values must fit within 40-bits
- IFM Tensor batch size must be 1 [19]

Memory optimization As stated in [20] several memory optimizations are performed: *"The Vela compiler also performs various memory optimizations to reduce both the permanent (for example flash) and runtime (for example SRAM) memory requirements. One such technique for permanent storage is the compression of all the weights in the model. Cascading reduces the maximum memory requirement by splitting the feature maps of a group of consecutively supported operators into stripes. A stripe can be either the full or partial width of the feature maps. And it can be the full or partial height of the feature maps. Each stripe in turn is then run through all the operators in the group. The parts of the model that can be optimized and accelerated are grouped and converted into TensorFlow Lite custom operators. The operators are then compiled into a command stream that can be executed by the Ethos-U NPU.*

Finally, the optimized model is written out as a TensorFlow Lite model and a Performance Estimation report is generated that provides statistics, such as memory usage and inference time. The compiler includes numerous configuration options that allow you to specify various aspects of the embedded system configuration (for example the Ethos-U NPU configuration, memory types, and memory sizes). There are also options to control the types of optimization that are performed during the compilation process" [20].

2.2.4 Corstone-300

The virtual testing platform *Corstone-300* is a reference package to optimize the development flow for *Cortex-M55* based devices. It allows to simulate the *Cortex-M55* and the integration of the Ethos-U NPU [21].

2.3 Network Splitting

The main idea of this project is to run two neural networks quasi-parallel on a single NPU. In Fig. 2.22 the principle of how to run two models is shown. In general there is a large (*Model 1* in Fig. 2.22) and a small (*Model 2* in Fig. 2.22) neural network that have to be run. Both models are to be run with different inference rates.

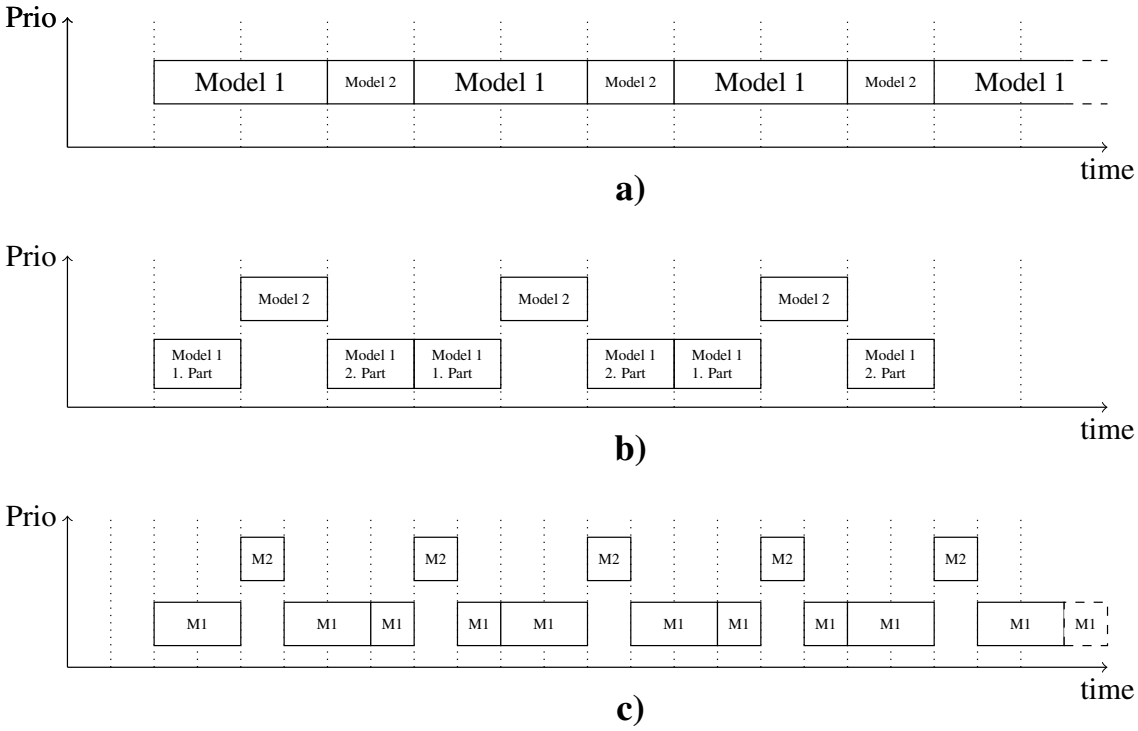
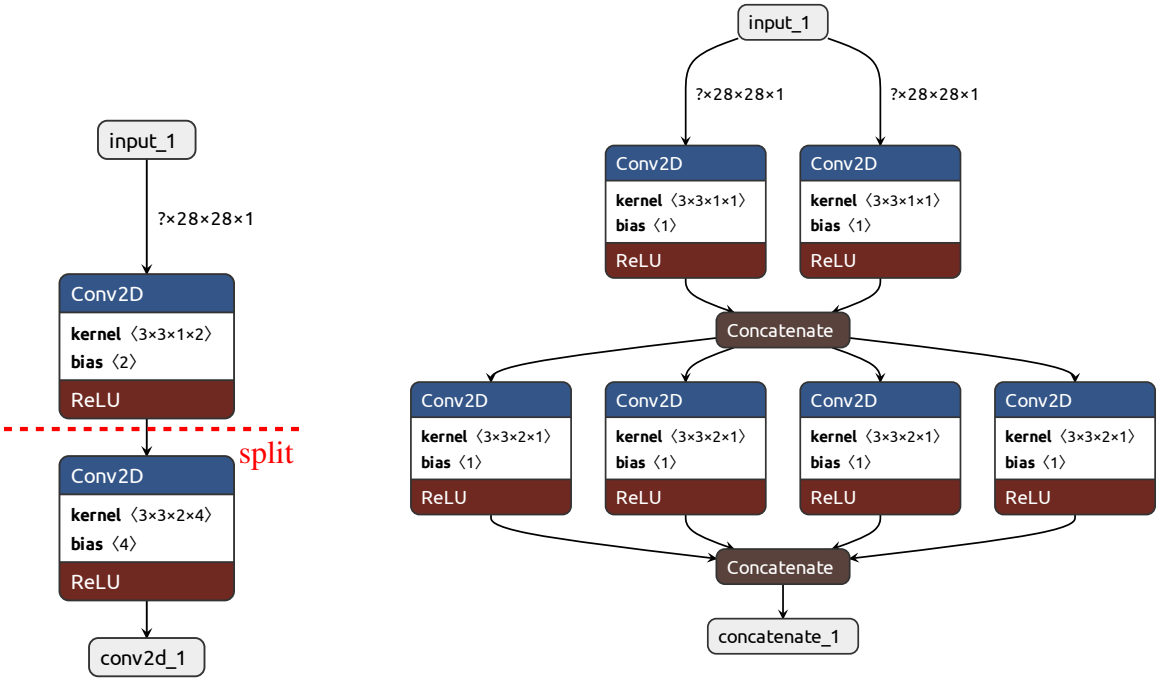


Figure 2.22: **a)** A simple sequential inference execution of the two models *model 1* and *model 2*. **b)** To execute the two models quasi-parallel, the system must interrupt the model with the lower priority. During the interruption, the model with the higher priority is executed, and then the inference of the other model finishes computing. **c)** The execution frequency of the high priority model must be selectable. *Model 1* should be interrupted only when necessary

In order to not only run two nets one after the other, the system needs a way to interrupt a model. Therefore, a neural network has to be divided into separate parts. Depending on the timing requirements of the two models, these parts can be run sequentially, or the other network can run inference in between if needed. The timing requirements of the system also determines how many parts the large network will be divided into.

2.3.1 Splitting Methods

The simplest way to split a neural network is after a layer. After each layer, the other net could theoretically be executed, Fig. 2.23a shows the basic idea. In addition to splitting after each layer, some layers can also be split further. For example, a Conv2D layer could be split after each filter. With this idea, each individual Conv2D operator could be executed separately. Fig. 2.23b shows the separated Conv2D operators and their concatenation.



(a) The DNN with a split after a layer. No modifications to the network architecture is needed

(b) The same network, but the Conv2D layers are divided by filter and reunited with a concatenation layer. A split after the concatenation layer is still an option

Figure 2.23: The two simple possibilities to split a DNN

Another method to further subdivide DNNs lies in the input of a neural network or, depending on the application, in the time dimension. In the previous methods, the input tensor was always given as a complete data package. Whereas in various applications this may not be necessarily the case, especially in speech processing with its continuous sampling over time. But let's stick to an example in computer vision, where the input is normally processed as a complete image. For a meaningful calculation of a convolution layer, not all data must be available at once. Theoretically, a pixel block of the size of the convolution filter would have to be available to complete all MAC operation of an output pixel. These blocks could be processed individually to allow even more subdivided calculation.

Chapter 3

Methods

3.1 Workflow with TensorFlow, TensorFlow Lite and Vela

The *MNIST*-dataset, as described in [22] is used as a reference classification problem, for the conceptual phase of this project. In Fig. 3.1a and Fig. 3.1b two simple CNN architectures to classify the *MNIST* dataset are shown.

This project uses the *TensorFlow (TF)* resp. *Keras* deep learning framework. Quantization is performed by the *TFL* library. The performance of the neural network is not of great interest, therefore no special techniques are used to train the networks. The *python* library *keras_flops* computes the number of floating point operations for a given DNN [23].

3.1.1 Compilation with Vela

The `vela` compiler takes the quantized *TFL* file and compiles it for the *Ethos-U* NPU. Fig. 3.2a shows the DNN after the quantization and Fig. 3.2b after the *vela* compilation. Because all operators in the quantized network are supported by the NPU the *vela* compiler fuses all operations into one single *Ethos-U* operator. A single *Ethos-U* operator cannot be interrupted and has to be run as a whole.

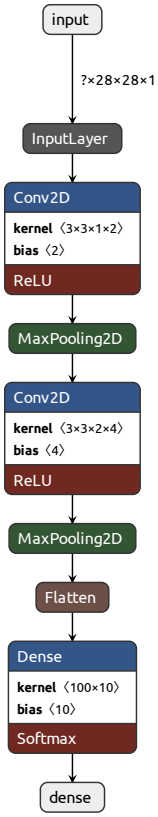
In Fig. 3.3a and Fig. 3.3b an example is shown where *TFL* converts the *swish* function (section 2.1.4) into known operators. *Vela* can therefore fuse all operator into one operators. In Fig. 3.3c and Fig. 3.3d an example is shown where *TFL* does not support the function, in this case the *elu* activation function (section 2.1.4). Firstly it cannot quantize the function and *vela* cannot convert it into a single *Ethos-U* operator.

3.2 Network Splitting

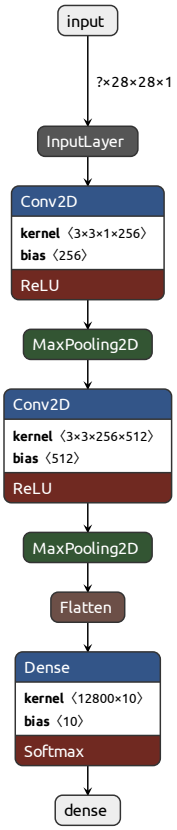
As described in section 2.3 there are several methods to split a neural network. To show the principles the simplest method has been implemented with the splitting after a layer. The neural network as shown in Fig. 3.1a is split after the last convolutional layer.

3.2.1 Splitting in Keras

The first approach was to split the network in the highest abstraction layer, in *Keras* itself. This is easily done as showed in Listing 3.1, where two different models are generated and the trained



(a) Small CNN with 2 filters in the first layer and 4 filters in the second layer. In total 1106 parameters and 47 kFlops per inference. An accuracy of 97 % can be achieved with the float model and 96 % with the int8 quantized model



(b) Big CNN with 256 filters in the first layer and 512 filters in the second layer. In total 1,310,730 parameters and 289 MFlops per inference. An accuracy of 99 % can be achieved with the float model and 99 % with the int8 quantized model

Figure 3.1: Simple CNN architecture with two convolutional layers, a Max-Pooling layer respectively and a Fully-Connected layer with a softmax activation for the classification at the output

weights of the original model are assigned to the two new separate models. As a result one gets two separate *TFL* files which can be run sequential or another model can be run in between. The disadvantage of this method is that for the inference of a DNN several *TFLM* interpreters have to be instantiated. This is not suitable in an application with a RTOS.

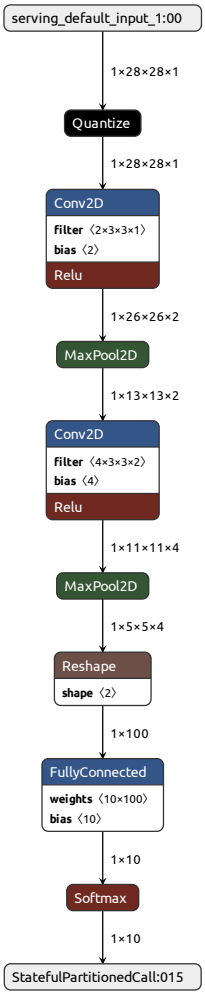
Fig. 3.4 shows the resulting neural networks and the output after the *vela* compiler

3.2.2 Dummy Layer

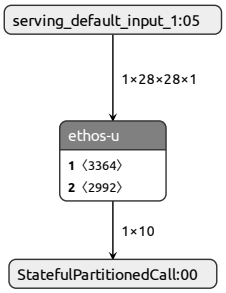
In order not to get two separate *tflite*-files like with the method shown in section 3.2.1 a method is needed to interrupt the inference of a neural network. If we remember that an *Ethos-U* operator cannot be interrupted once it has been started. Therefore, a special layer can be inserted into the network which the *vela* compiler cannot map to the *Ethos-U*. This special layer has only the purpose of splitting the *Ethos-U* operator into two parts and is not supposed to do anything else, hence let's call this layer *dummy layer*. During the execution of the *dummy layer*, the

```
1  def split_model(self):
2    model = tf.keras.Sequential(
3        [
4          tf.keras.Input(shape=self.input_shape),
5          tf.keras.layers.Conv2D(filters=2,
6                                 kernel_size=(3, 3),
7                                 activation="relu"),
8          tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
9          tf.keras.layers.Conv2D(filters=4,
10                                kernel_size=(3, 3),
11                                activation="relu"),
12         tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
13         tf.keras.layers.Flatten(),
14         tf.keras.layers.Dense(self.num_classes, activation="softmax"),
15       ]
16     )
17
18   sub_model_1 = tf.keras.Sequential(
19       [
20         tf.keras.Input(shape=self.input_shape),
21         tf.keras.layers.Conv2D(filters=2,
22                                kernel_size=(3, 3),
23                                activation="relu"),
24         tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
25         tf.keras.layers.Conv2D(filters=4,
26                                kernel_size=(3, 3),
27                                activation="relu"),
28         tf.keras.layers.MaxPooling2D(pool_size=(2, 2))
29       ]
30     )
31
32   sub_model_2 = tf.keras.Sequential(
33       [
34         tf.keras.Input(shape=(5,5,4)),
35         tf.keras.layers.Flatten(),
36         tf.keras.layers.Dense(self.num_classes, activation="softmax"),
37       ]
38     )
39
40   model.load_weights('weights.h5')
41   weights = model.get_weights()
42   sub_model_1.set_weights(weights[:4])
43   sub_model_2.set_weights(weights[4:6])
44
45 return sub_model_1, sub_model_2
```

Listing 3.1: The first model shows the original configuration, whereas in `sub_model_1` and `sub_model_2` the network is split into two parts. The trained weights are loaded into the original model and then with lines 41-43 loaded into the sub-models



(a) The small network converted and quantized with *TFLM*



(b) The small network compiled with the *vela* compiler, all operators are fused into one ethos-u operator

Figure 3.2: The networks during the general work flow with *TFL* and *vela*

inference of the network can be paused and another model can be executed.

3.2.2.1 Dummy Layer in Keras

There is a well known method to build custom layers in *Keras* [24]. Listing 3.2 shows a possible implementation of the *dummy layer*. The problem with this method is that *TFL* analyses the custom layer and tries to rewrite it with known operators as shown in Fig. 3.3a. Because our *dummy layer* is not supposed to do something meaningful, *TFL* removes the *dummy layer* automatically. Therefore, this method can not be used for this purpose.

```
1 class DummyLayer(tf.keras.layers.Layer):
2   def __init__(self):
3     super(DummyLayer, self).__init__()
4
5   @tf.function
6   def call(self, x: tf.int8):
7     return x
```

Listing 3.2: Custom layer in *Keras* where the input of the layer is directly mapped to the output

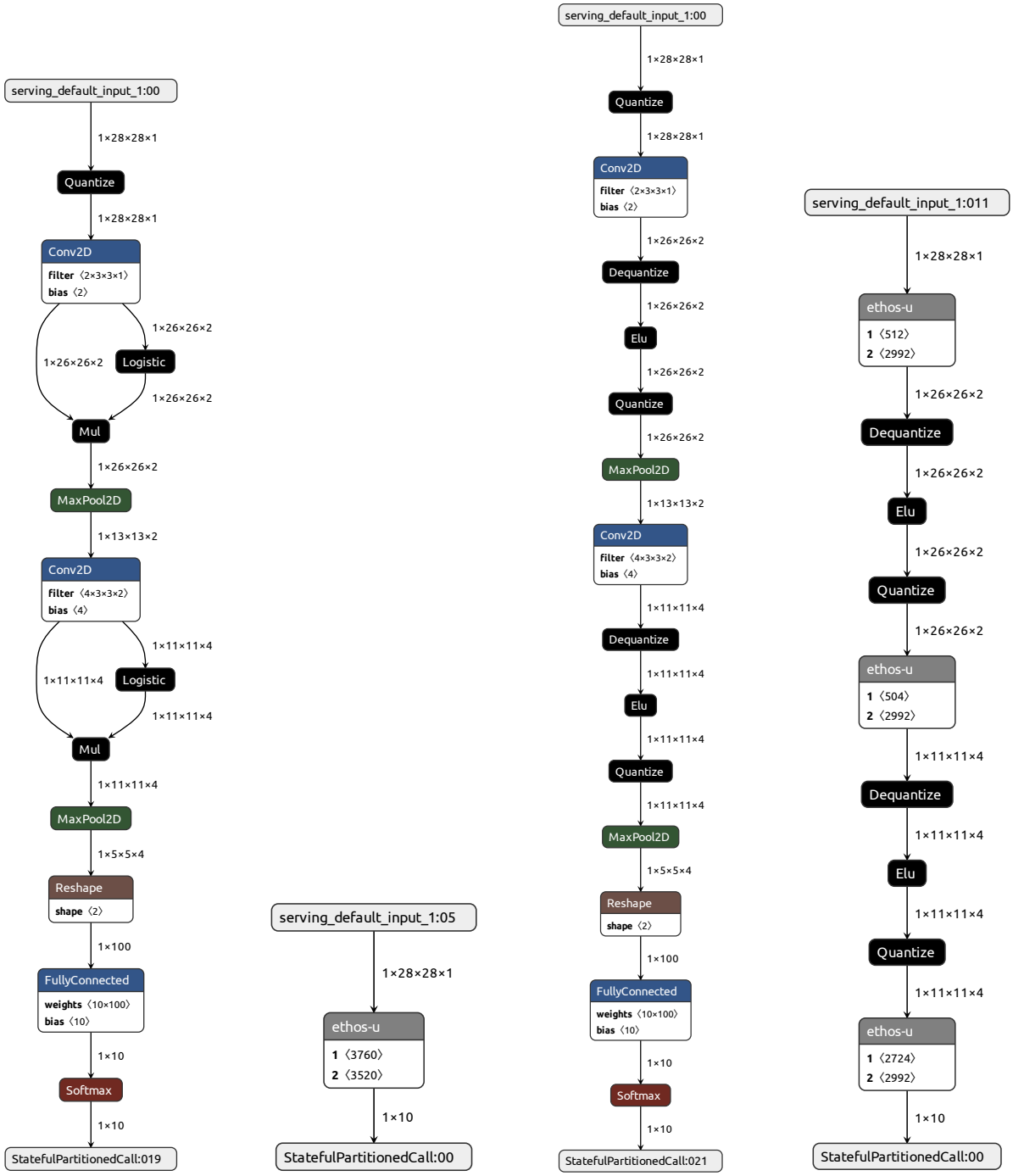
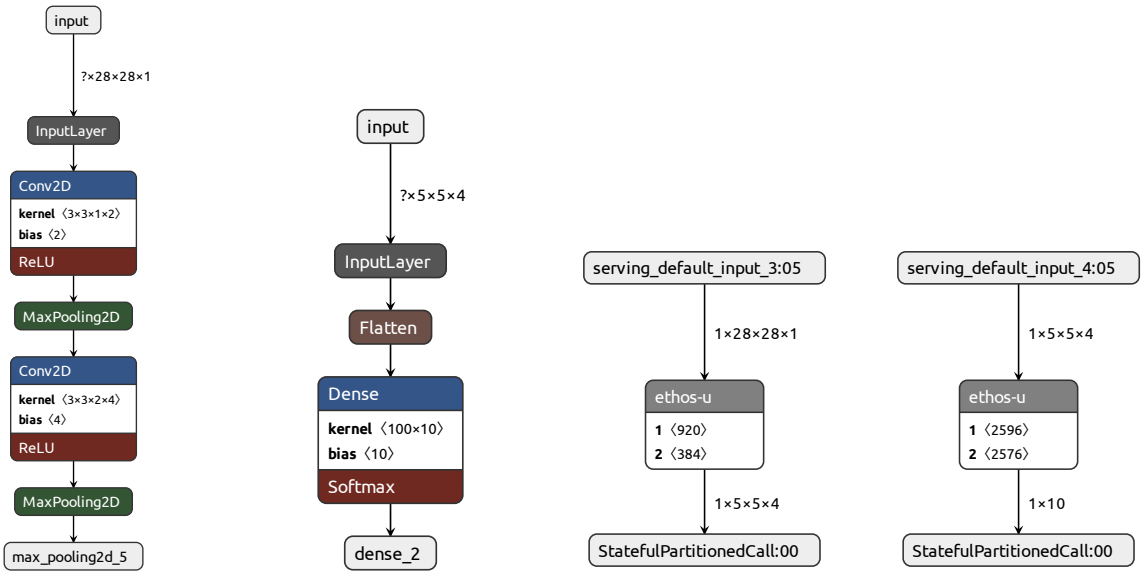


Figure 3.3: **a)** An example with the *swish* activation function (section 2.1.4), where *TFL* converts the function into separate operators. **b)** all operators from the swish activation function are supported by vela and therefore a single Ethos-U operator is generated. **c)** The same network with the *elu* activation function (section 2.1.4, which is not supported by neither *TFL* nor the NPU). **d)** This results after the compilation into two ethos operators and the operators for the Central Processing Unit (CPU)



(a) The small network split after the second *MAX Pool* layer. The output is a 5×5×4 tensor

(b) The second part of the network, the input is the $5 \times 5 \times 4$ tensor from the first part

(c) First part after *vela* compilation

(d) Second part after *vela* compilation

Figure 3.4: The two parts of the splitting idea on the highest level in *Keras* and the *vela* compiler fuses both parts separately into a single *Ethos-U* operator

3.2.2.2 Dummy Layer in JSON

TFL saves the representation of the DNN in a *tflite* file. The files are generated with *FlatBuffers*, a cross-platform serialization library [25]. These files are very structured and the syntax is described in a schema file (*fbs*). And with the *flatc* compiler one can convert a *tflite* file to a *json* file and vice versa [26]. Fig. 3.5 visualizes the layout of the *tflite* file. The most important sections of the *json* file:

- **operator_codes**: All used operators (conv2D, flatten, softmax, max pooling etc.) are stored in this array.
- **subgraphs**
  - **tensors**: All tensors with their properties such as name, shape, data type or quantization parameters
  - **input, output**: Shows which tensor represents the input or the output of the DNN
  - **operators**: Describes the operators with their parameters and which tensor should be taken as input and output.
- **buffers**: The tensor data like the weights and biases

With this regular structure one can easily insert a *dummy layer* into this file. The *CUSTOM* operator defined in *TFL* is used for the *dummy layer* Because the network has already been converted to *TFL* and been quantized, the *dummy layer* will not be automatically removed by

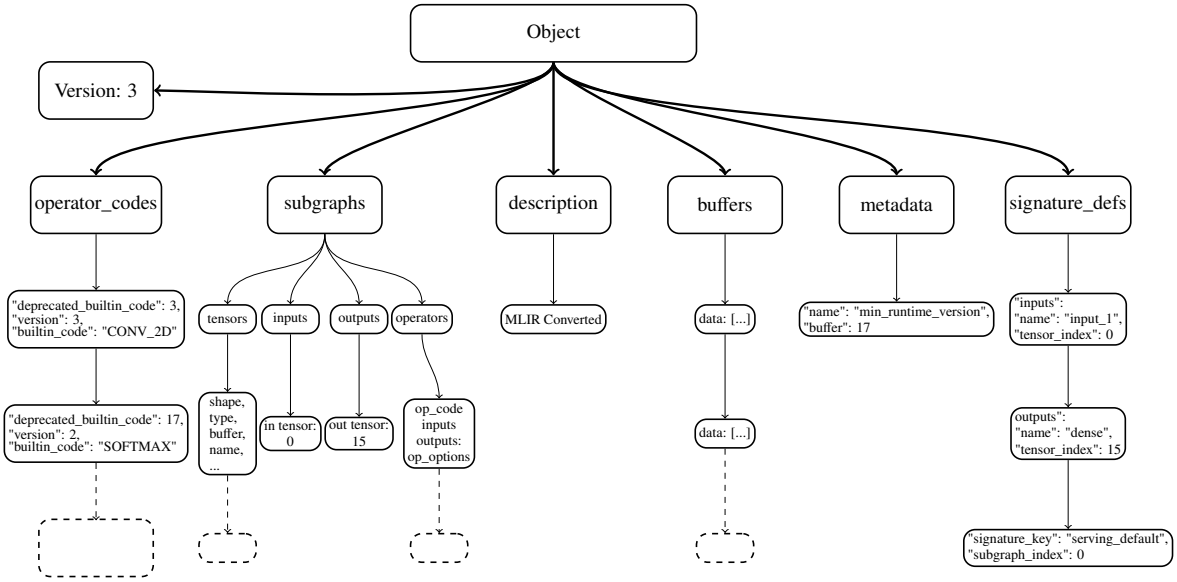


Figure 3.5: The layout of the `JSON` representation of the `tflite`-file. The whole DNN with its biases and weights are represented in this `JSON` structure

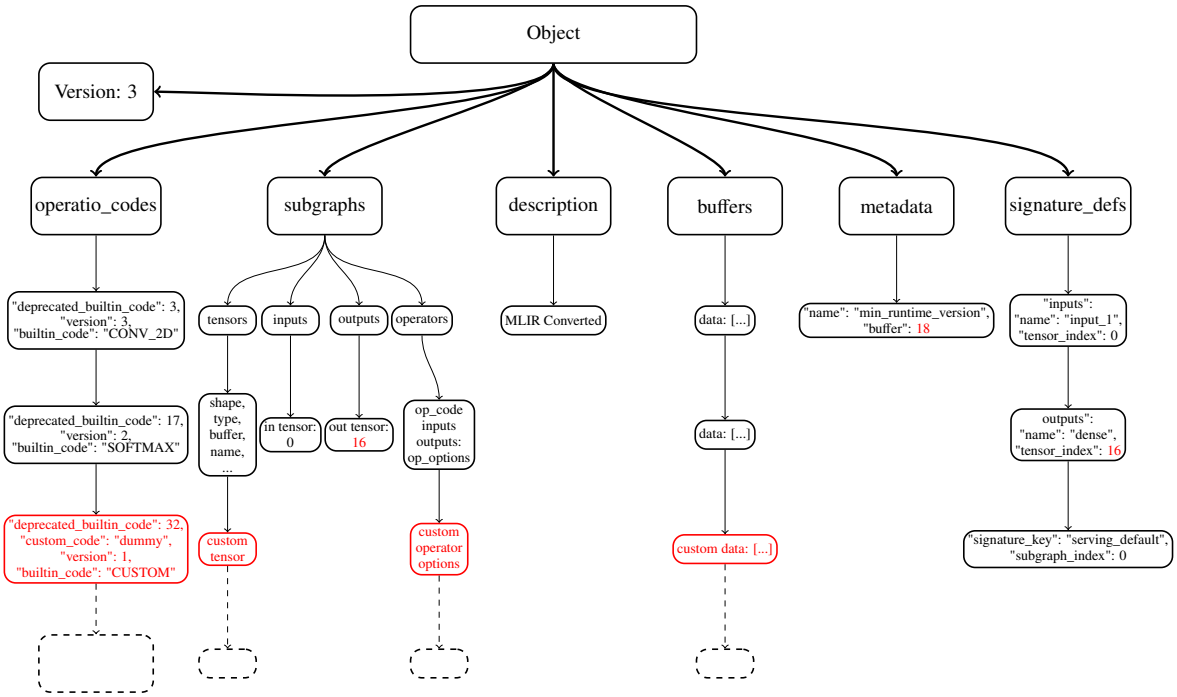
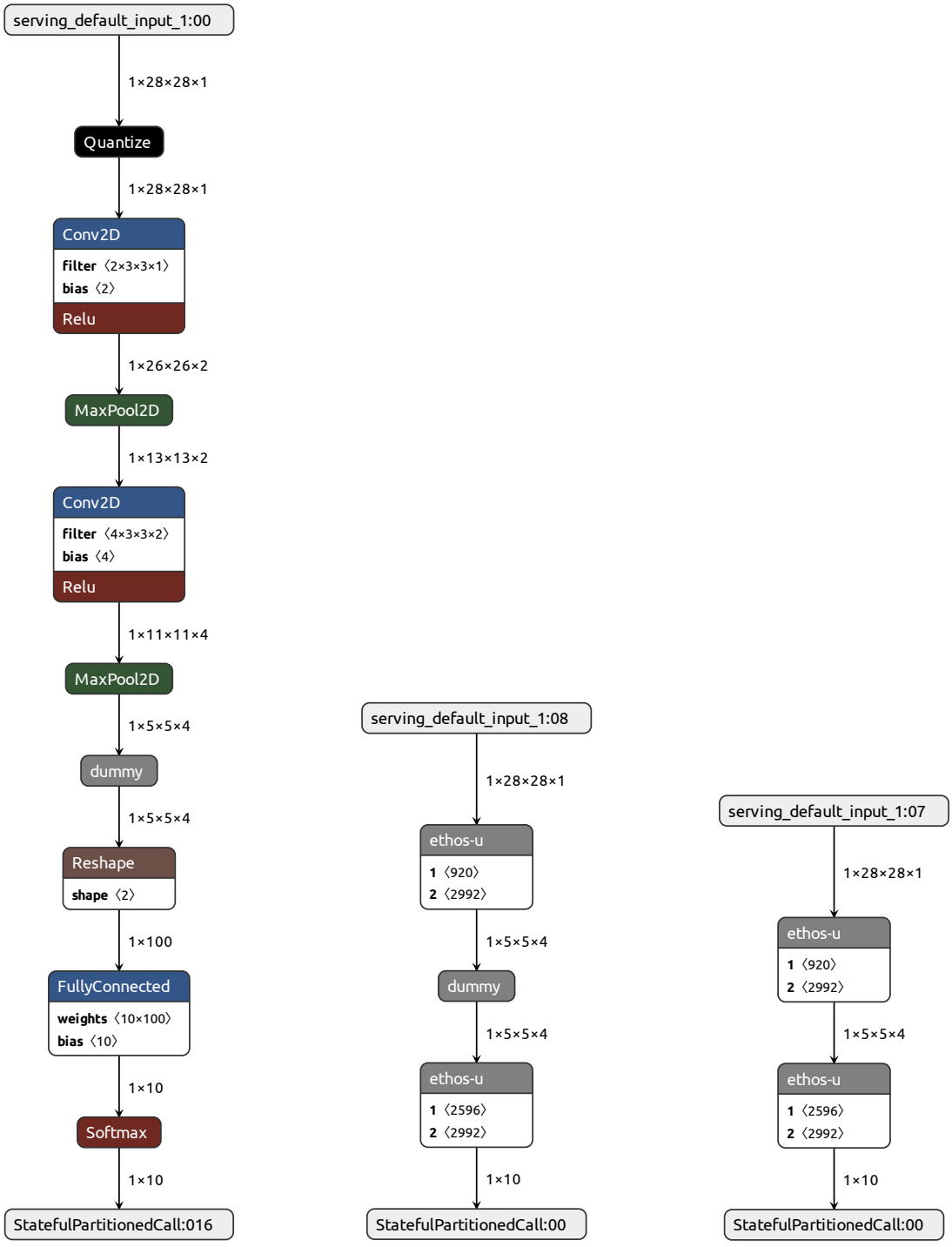


Figure 3.6: A *dummy layer* can be easily inserted into the `json` structure. The red part of the figure represents the modification one needs to perform in order to add a *dummy layer*.

TFL. Fig. 3.6 visualizes how to insert a *dummy layer*. A *Python* script has been written to automate the insertion for the given network structure.

Fig. 3.7 shows the resulting networks after the insertion of a *dummy layer*.



(a) The neural network with the added *dummy layer* in the flatbuffer file

(b) The network after the *vela* compilation

(c) The idea with the removal of the *dummy layer* after the compilation

Figure 3.7

Remove Dummy Layer The insertion of the dummy layer was done so that the *vela* compiler would split the DNN into two consecutive *Ethos-U* operators rather than one. The dummy layer itself performs a memory copy from the input to the output tensor. Since the dummy layer

is only used to trick the Vela compiler, you could simply remove the layer after compilation (Fig. 3.7c) and thus omit the memory copy operation.

Unfortunately, removing a layer is not as easy as inserting the layer into the `json` file. Removing the corresponding tensors, the custom operator and joining the two *Ethos-U* operators together, results in a correct network architecture (Fig. 3.7c). However, when running on the *Ethos-U*, several problems occur.

An important feature of the *vela* compiler is the 16-byte alignment of the data in the buffer space. After aligning the data correctly, the network can be run without an error in the *Ethos-U*, but the resulting data is not correct. Due to time constraints, this issue, and thus the approach to removing the *dummy layer*, was not investigated further.

3.3 Development Setup

In order to have a simple development setup that does not depend on an operating system or a time-consuming installation on each working PC, a *Docker* container was created with all the necessary tools. With this container in combination with the version control system *git*, software development can be moved from one PC to another in no time. The base container has the minimal *Ubuntu 20.04* installation and all the needed packages have been installed manually.

The container can be pulled with the *bash* command shown in Listing 3.3.

```
1    docker pull nunigan/parallel_dnn:latest
```

Listing 3.3: Pull the Dockercontainter

3.3.1 ethos-u Repository

The core component is the *ethos-u* repository which includes all required repositories and libraries and drivers for a system with the *Ethos-U* NPU. It provides them in a tree structure and fetches the correct version with a *Python* script:

- Direcroty
  - core_platfrom
  - core_software
    - applications
    - cmsis
    - cmsis-view
    - core_driver
    - drivers
    - lib
    - rtos
    - tflite_micro
  - linux_driver_stack
  - vela

3.3.2 Arm ML Embedded Evaluation Kit

The purpose of the evaluation kit is to provide an environment with various pre-built and open source applications to get started quickly with an *Ethos-U* system. The main target is the *corstone-300* or on the *MPS3* Field Programmable Gate Array (FPGA).

The engineer can experiment with all the different use cases and include his own neural networks or built an own use case. In all examples, an end-to-end machine learning process is implemented that includes pre- and post-processing of the data. Some use cases are:

- Image classification
- Keyword spotting
- Automated speech recognition
- Noise reduction
- And more

The latest version of the Embedded Evaluation Kit contains a *Dockerfile* with the description to install all the packages. This additional feature was released in the final phase of this project and was therefore not included. It eliminates the need for the custom *Docker* container that was built for this project. The main components are shown in Fig. 3.8.

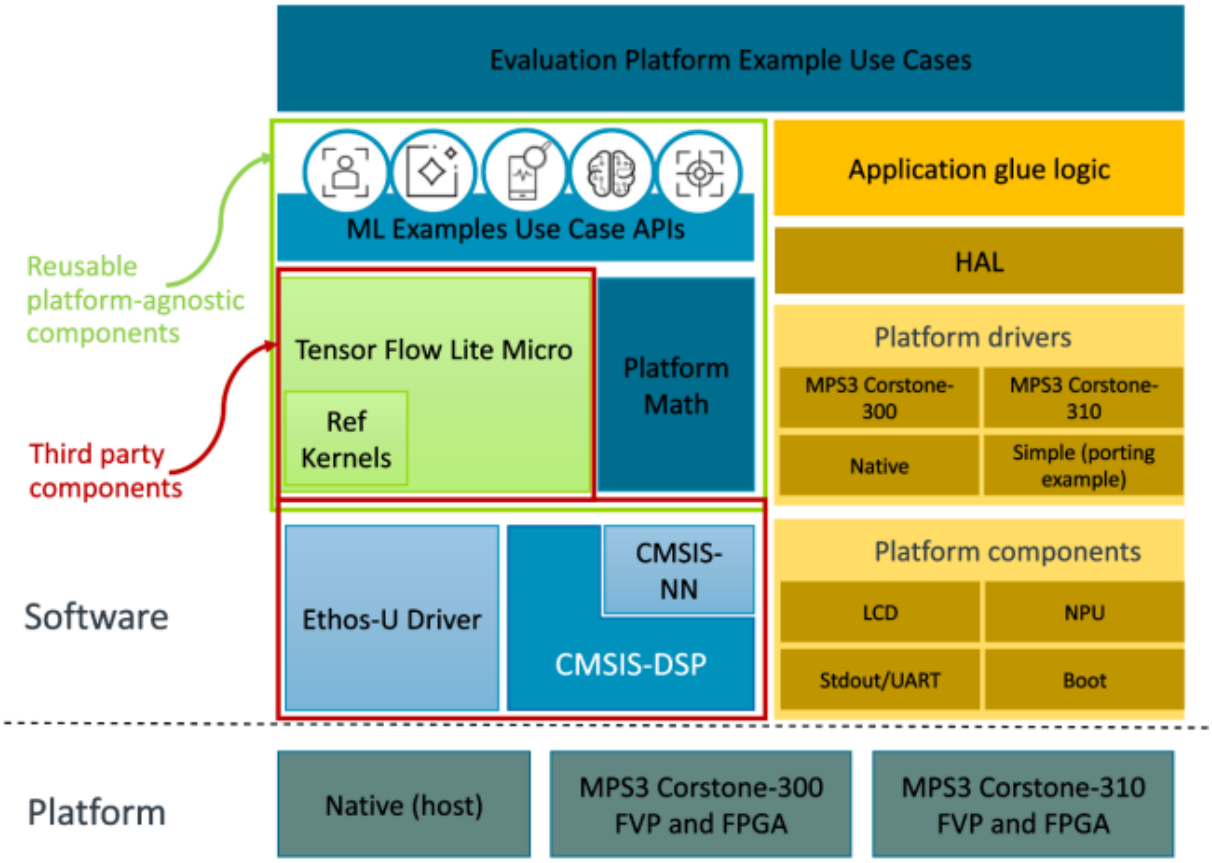


Figure 3.8: The main components of the Arm ML embedded evaluation kit [27]

3.3.2.1 Software Architecture

For the theoretical principles presented in section 3.2 DNNs had to be modified in various ways. In this project the Arm ML embedded evaluation kit has been used to quickly test the DNNs. A `Hello_World` use case was created and the different models have been implemented into the use case.

The following functions are implemented:

- `single_nn()`
  - Simple implementation of running one neural network
- `multi_nn()`
  - Multiple neural networks are run sequential on the same allocated memory space
- `multi_nn_different_alloc()`
  - Multiple networks are run sequential on the different allocated memory space
- `sub_model()`
  - This function implements the method described in section 3.2.1
  - The first part of the model split in *Keras* is executed and then the inference of a second model is carried out before the second part of the split model is run.
  - Important is to copy the output data from the first part and store it during the inference of the second model. When the second model finishes the saved data has to be used as the input of the second part of the first model

3.3.2.2 Custom Operator

The method described in section 3.2.2.2 introduces a custom *dummy layer* into the *TFLM* environment. This layer is to be implemented and because the *dummy layer* shall not do anything it only copies the data from the input to the output. The size of the data to copy is depending on the network structure and the code need adjustment when computing different DNNs.

3.3.3 RTOS

In the *ethos-u* core platform an example of the implementation with the *ethos-u* in combination with a RTOS is given.

The TFLM framework is built to run multiple parallel inferences. Each parallel inference requires a separate tensor arena and a stack. For this a RTOS is required. The example provided in the *core_plotform* is implemented in the application layer. All the RTOS specific implementations in the driver are made with weak functions, which have to be overwritten in the application level. This means any RTOS can theoretically be used. In this project *FreeRTOS* is used. Fig. 3.9 shows the description and visual representation for the system with a RTOS.

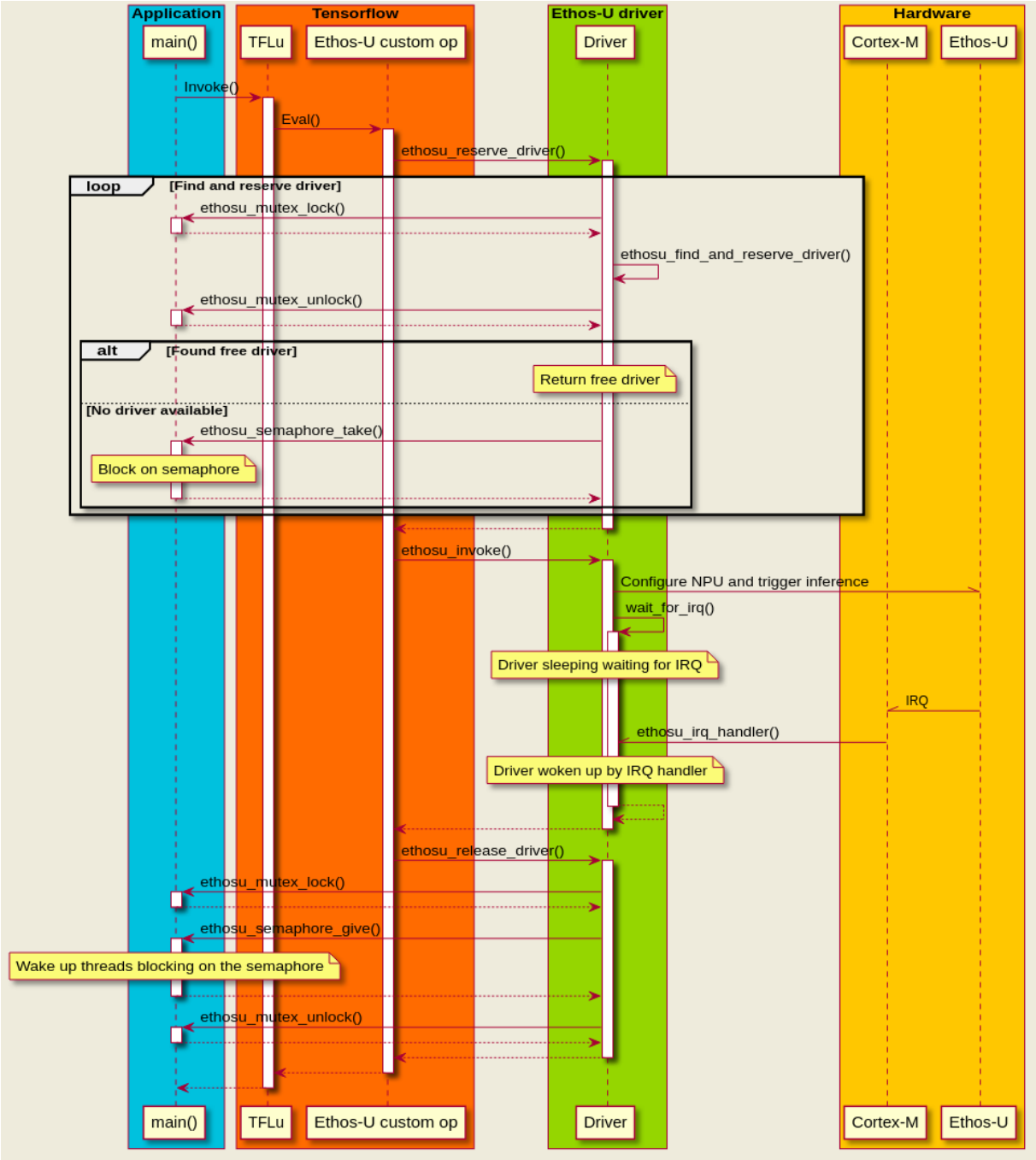


Figure 3.9: The principle of using multiple NPU [28]: The application starts the inference process with the `invoke()` command. The `eval()` method of the custom operator *ethos-u* attempts to reserve an Ethos driver. With the `mutex_lock()` and `mutex_unlock()` functions, the program checks if a driver is available and reserves it, if not, it waits until a driver is available. Then the inference is calculated on the NPU and then the driver is released.

3.3.3.1 Software Architecture

The idea with the network splitting method introduced in section 3.2.2.2 is that the task scheduler of RTOS can perform a context switch from a lower priority task to a higher priority task when necessary. This can be done after the completion or at the position of the dummy layer

within the low priority network. The task scheduler in the RTOS can be operated in two different modes. The cooperative task scheduler does not interrupt a task in progress and generally waits until the task is finished before performing a context switch. Whereas a preemptive task scheduler also interrupts running tasks for a context switch. Because the operations which are computed on the NPU must not be interrupted as *FreeRTOS* is executed with a cooperative task scheduler.

Similar to Fig. 2.22 two tasks are created in the RTOS with different priorities and delay time.

Explicit Suspension Experiments show that the task scheduler has issues to suspend the task at the desired position.

To help the task scheduler of *FreeRTOS*, you can suspend the task with the *dummy level* exactly during the execution of the *dummy level*. *FreeRTOS* provides with the `vTaskSuspend()` function a method to set a task sleeping. With `vTaskResume()` the task can be resumed. This means that the task is interrupted every time the *dummy level* is executed. With an additional task, the low priority task is resumed regularly. Until the task is resumed automatically, the task scheduler has the chance to run the high priority task. The principle is shown in Fig. 3.10.

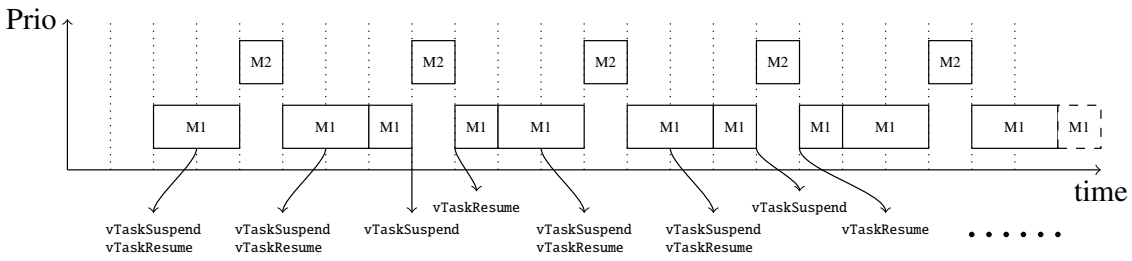


Figure 3.10: Task scheduling with the explicit call of `vTaskResume` and `vTaskSuspend`

Chapter 4

Results

The methods presented in section 3.2 were tested with the simulator corstone-300. Since the networks were modified with several methods, the functionality of a neural network after splitting had to be tested first. For this purpose, the ML evaluation kit described in section 3.3.2 was used.

The concurrent functionalities in FreeRTOS have been tested in the *ethos-u* core platform environment, more specific in the *freertos* application.

4.1 Basic Functionality

The basic functionality of the different neural networks and their performance are tested. Especially when customizing the neural network with a dummy layer, the functionality has to be verified.

Following in Listing 4.1 the output of the *Vela* compiler from the network showed in Fig. 3.1a. Comparing the number of floating points operations of the `Keras` model and the model after the `Vela` compilations shows that the number of operations have been reduced nearly 50 %.

Note from the performance estimation after *vela*: "*This shows the estimated number of MACs in the network per batch. This number includes MACs from convolutions, vector products and pooling operations. It does not include MACs from elementwise or any other type of operation.*" [29]

While *keras_flops* counts all operations. One can still compare the numbers because the fast majority of the operations come from the convolutional layers.

Listing 4.2 shows the log entry after a NPU computation of the small CNN network. All relevant information about the inference itself and also about the model and the TFLM interpreter is displayed in the console.

While numerical correctness was confirmed, other anomalies occurred. In particular, the number of cycles in which the NPU is idle is very high. This should be investigated further. With the prefabricated examples in the *ML Embedded Evaluation Kit*, this idle count is in the range of 500 cycles [30].

The method presented in section 3.2.1 with the splitting in `Keras` also works as expected, but is not presented in more detail in this section.

```
Network summary for mnist
Accelerator configuration                   Ethos_U55_128
System configuration                        Ethos_U55_High_End_Embedded
Memory mode                                 Shared_Sram
Accelerator clock                           500 MHz
Design peak SRAM bandwidth                  4.00 GB/s
Design peak Off-chip Flash bandwidth        0.50 GB/s

Total SRAM used                             22.91 KiB
Total Off-chip Flash used                   6.16 KiB

CPU operators = 0 (0.0%)
NPU operators = 38 (100.0%)

Average SRAM bandwidth                       1.83 GB/s
Input   SRAM bandwidth                       0.05 MB/batch
Weight  SRAM bandwidth                       0.00 MB/batch
Output  SRAM bandwidth                       0.03 MB/batch
Total   SRAM bandwidth                       0.08 MB/batch
Total   SRAM bandwidth            per input  0.08 MB/inference (batch size 1)

Average Off-chip Flash bandwidth             0.07 GB/s
Input   Off-chip Flash bandwidth             0.00 MB/batch
Weight  Off-chip Flash bandwidth             0.00 MB/batch
Output  Off-chip Flash bandwidth             0.00 MB/batch
Total   Off-chip Flash bandwidth             0.00 MB/batch
Total   Off-chip Flash bandwidth  per input  0.00 MB/inference (batch size 1)

Neural network macs                         24436 MACs/batch
Network Tops/s                               0.00 Tops/s

NPU cycles                                  11006 cycles/batch
SRAM Access cycles                          15140 cycles/batch
DRAM Access cycles                              0 cycles/batch
On-chip Flash Access cycles                     0 cycles/batch
Off-chip Flash Access cycles                 3760 cycles/batch
Total cycles                                22001 cycles/batch

Batch Inference time                 0.04 ms, 22725.72 inferences/s (batch size 1)
```

Listing 4.1: Output of the *Vela* compiler from the network showed in Fig. 3.1a

```
INFO - Profile for Inference:
INFO - NPU AXI0_RD_DATA_BEAT_RECEIVED beats: 4234
INFO - NPU AXI0_WR_DATA_BEAT_WRITTEN beats: 3790
INFO - NPU AXI1_RD_DATA_BEAT_RECEIVED beats: 754
INFO - NPU ACTIVE cycles: 21372
INFO - NPU IDLE cycles: 113721
INFO - NPU TOTAL cycles: 135093
0, -128
1, -128
2, 127
3, -128
4, -128
5, -128
6, -128
7, -128
8, -128
9, -128
INFO - ############## model ##########################################################
INFO - Model version: 3
INFO - Model description: Vela Optimised
INFO - Model num operator types: 1
INFO - Model num subgraphs: 1
INFO - Model num tensor buffers 6
INFO - ############## subgraph #######################################################
INFO - Inputs length: 1
INFO - Outputs length: 1
INFO - Input tensors: [1]
INFO - Output tensors: [0]
INFO - Subgraph name: N
INFO - Num operators in the subgraph: 1
INFO - ############## operator type ##################################################
INFO - CUSTOM: 32
INFO - All operators:
INFO - 0 CUSTOM: 32
INFO - ############## flatbuffer metadata ############################################
INFO - Num num_metadata: 2
INFO - Metadata |   Buffer | Name                      |       addr
INFO - ############## flatbuffer buffer ##############################################
INFO - Num buffers: 6
INFO -   Buffer | length |       addr | addr->data
INFO -        0 |      0 | 0x0x70014414 | 0x0x70014420
INFO -        1 |      0 | 0x0x70014408 | 0x0x4
INFO -        2 |   3364 | 0x0x700136d4 | 0x0x700136e0
INFO -        3 |   2928 | 0x0x70012b54 | 0x0x70012b60
INFO -        4 |     16 | 0x0x70012b34 | 0x0x70012b40
INFO -        5 |     36 | 0x0x70012b04 | 0x0x70012b10
INFO - ############## tensor flatbuffer ##############################################
INFO - Num tensors: 6
INFO -   Tensor |   Buffer | isVar |            shape | length
INFO -        0 |        0 |     0 |          [1, 10] |      0
INFO -        1 |        0 |     0 |   [1, 28, 28, 1] |      0
INFO -        2 |        2 |     0 |           [3364] |   3364
INFO -        3 |        3 |     0 |           [2928] |   2928
INFO -        4 |        0 |     0 |          [23456] |      0
INFO -        5 |        0 |     0 |          [23456] |      0
INFO - ############## tensor interpreter ###########################################
INFO - Num tensors: 6
INFO -   Tensor | type | alloc  |isVar|      shape |q.val |q.offset | q.scale |bytes |  addr tens |  addr data
INFO -        0 | Int8 | ArenaRw|    0|    [1, 10] |    1 |    -128 |0.003906 |   10 | 0x0x311ffeac | 0x0x31000000
INFO -        1 | UInt8| ArenaRw|    0|[1,28,28,1] |    1 |       0 |0.003922 |  784 | 0x0x311ffe78 | 0x0x31003100
INFO -        2 | UInt8| MmapRo |    0|     [3364] |    0 |       0 |1.000000 | 3364 | 0x0x311ffe44 | 0x0x700136e0
INFO -        3 | UInt8| MmapRo |    0|     [2928] |    0 |       0 |1.000000 | 2928 | 0x0x311ffe24 | 0x0x70012b60
INFO -        4 | UInt8| ArenaRw|    0|    [23456] |    0 |       0 |1.000000 |23456 | 0x0x311ffe04 | 0x0x31000000
INFO -        5 | UInt8| ArenaRw|    0|    [23456] |    0 |       0 |1.000000 |23456 | 0x0x311ffde4 | 0x0x31000000
INFO - ############## nodes #########################################################
INFO - Num tensors: 6
INFO - Number of operators: 1
INFO - Node | operator             | inputs               | outputs              | addr reg.
INFO -    0 | ethos-u              |      [2, 3, 4, 5, 1] |                  [0] | 0x0x2007fc28
```

Listing 4.2: The output of the *Ethos-U* NPU after the execution of the small neural network. Lines 1 to 17 show the output of NPU and the result of the inference. Then the information of the model and the TFLM interpreter. One can see all the memory locations of the tensors and buffers

Network	Keras FLops	Vela estimation	NPU-cycles
Small CNN (Fig. 3.1a)	47 kFlops/inference	24436 MACs/batch 22'001 cycles/batch	active cycles: 21'372 idle cycles: 113'721 total cycles: 135'093
Small CNN dummy layer (Fig. 3.7a)	47 kFlops/inference	24436 M MACs/batch 22065 cycles/batch	active cycles: 21'967 idle cycles: 230'207 total cycles: 252'174
Big CNN (Fig. 3.1b)	289 MFlops/inference	144 M MACs/batch 2'020'227 cycles/batch	active cycles: 1'604'440 idle cycles: 114'653 total cycles: 1'719'093
Big CNN dummy layer	289 MFlops/inference	144 M MACs/batch 2'020'227 cycles/batch	active cycles: 1'604'965 idle cycles: 227'209 total cycles: 1'832'174

Table 4.1: The inference performance results for different neural networks. If a dummy layer is added some extra cycles for the copy operation have to be done.

4.2 Real Time Application

The methods presented in Fig. 3.10 have been tested. Following Properties can be changed for the RTOS application.

- Preemptive or cooperative task scheduling
- Delay time of the high priority task.
- With or without `vTaskSuspend` and `vTaskResume` in the custom operator
- The priority of the generated tasks

As a test setup, the low priority net is run 6 times as fast as possible. Whereas the high priority net is executed 5 times with a certain delay time.

Small Network First the system was tested with the small network. The low priority model was Fig. 3.7b with the *dummy layer* and the high priority net was the same but without the *dummy layer* Fig. 3.1a.

The system works as expected with following properties:

- Cooperative task scheduling
- With `vTaskSuspend` and `vTaskResume`
- Long delay time

  - If the delay time is too short, all classifications of the network with the low priority will be executed, and only then all classifications of the network with the high priority.

- Short delay time
  - If the delay time is too short, all classifications of the network with the high priority will be executed, and only then all classifications of the network with the low priority.

- With the small networks, a delay time of about 5000 ticks will trigger the task scheduler to schedule both task between each other
  - As expected is the task with the high priority executed when needed. When changing the delay time the task is scheduled either during the *dummy layer* or after the completion of the task.

Big Network Unfortunately, when the large network with the *dummy layer* is used as the network with the low priority, the system does not work as desired. There are two types of errors that are triggered. It is important to know that both networks work, but depending on when the task scheduler executes the context switch, an error is generated. The main problem here is that the task scheduler executes the context switch at points in the code where it is not intended to do so. The context switch should only be executed in the *dummy layer* or after the completion of a network.

- `E: UNSUPPORTED driver_action_command: 134 (ethosu_driver.c:674)`
  - This error occurs when the high priority task is started while the low priority task is still computing. An examination of the log shows that while the high-priority task is executing, the command stream from the low-priority task is loaded into the NPU and an error is generated

- `E: Custom Operator Payload: 805830656 is not correct, expected 31504f43 (ethosu_driver.c:611)`
  - Similar to the other errors, erroneous data is loaded into the NPU and TFLM environments

With both errors, incorrect data is loaded into the system. The reasons for this are not easy to find out. First of all, it was tried that the task scheduler executes the context switch only at the intended places. However, various settings did not lead to the desired result. Especially with the cooperative task scheduling no context switch should be executed at an undesired position.

If it is accepted that the context switch is executed at any point, it must be ensured that the RTOS system around the *Ethos-u* driver is working properly. The concepts of the real-time system as presented in Fig. 3.9 have been thoroughly reviewed and it seems that this implementation does not match the idea of this project perfectly. Due to timing restrictions the *Ethos-U* driver has not been rewritten.

Chapter 5

Conclusion

Principles The whole system with the *Ethos-U* NPU and the development environment with the *corstone-300* could be successfully put into operation. The environment is well-structured and gives the developer a fast and uncomplicated start into the development phase. The workflow via TF to the *vela* compiler and finally to the NPU is uncomplicated and fast.

The theoretical basis for concurrent DNNs execution has been well demonstrated. The different possibilities to run two networks quasi in parallel could be shown. Furthermore, different approaches to modify a DNN to achieve the desired concurrent functionality were given. Not all the shown principles were tested conclusively and completely, in further research projects the shown ideas can be taken up and examined again

RTOS The core of the project, the implementation of concurrent execution of multiple DNNs, could unfortunately not be successfully implemented for all cases. The functionality could be demonstrated with the small network, but not with the large model. The shown approach how a network can be executed in several steps on the *Ethos-U* NPU in order to compute another network in between, is the right one. However, the implementation with the RTOS is still error-prone and needs to be examined more closely. In theory, the task scheduler of the RTOS should be able to execute the system correctly. Even with some additional instructions for the task scheduler with the explicit instructions to pause and resume tasks, the system still does not run as intended. In the end, it turned out that the real-time implementation of the Ethos-U NPU driver might not be suitable for the idea of this project.

A Glimpse to Future Projects The project laid many of the foundations for concurrent inference on the NPU. Many of the obstacles of a newly released development environment took a lot of time, which meant that important aspects of the project could not be sufficiently tested. All the more the potential for further projects is given!

- Network Splitting
  - The removal of the dummy layer after the *Vela* compilation
  - Network splitting by the filters in convolutional layer
  - Network splitting with the input dimension, that only snippets of the input is processed
- RTOS

  - Building a system from scratch, not based on a prefabricated version as in this project. The real time implementation can be specific to the requirements of the project.
  - When a working system is presented, a suitable test environment must be set up to put the system through its paces.

- Use of a neural network with the application in the speech processing area
- Porting the system to the hardware

Summary

This summary is written from the personal perspective of the student.

The project was built with the vision of developing something new that to our knowledge had never been done before. Not only that but also the given target platform, the *Ethos-U* NPU, is a product which is not yet available on the market or has limited availability. One of the most important aspects of such a novel project is the lack of a well-established online community.

This leads to a fascinating project, but one that also brought with it some challenges. With these types of projects, where there is more to it than just implementing something given, it is always very difficult to predict the course of the project. During the progress of this work, unfortunately, there were some stumbling blocks that cost a lot of time. As a result, the project plan had to be adjusted several times and some outstanding and important work had to be removed from it or rescheduled for a later project. Also, exciting approaches could not be fully explored.

Nevertheless, a very deep insight into the whole system could be gained, probably thanks to these many challenges. I have rarely investigated so deeply into a technology and analyzed the code so thoroughly.

In retrospect, I will probably approach some things a little differently. I have already observed in some completed projects that if I spend too long in a predefined development environment and work with prebuilt examples, problems arise over time. The step should be taken early in the project where the system should be designed from the ground up.

Chapter 6

Declaration of Authorship

I hereby declare that I am the sole author of this student research paper and that I have not used any sources other than those listed in the bibliography and identified as reference. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Michael Schmid

Rapperswil, July 1, 2022

Appendices

Appendix A

Task

OST Ostschweizer Fachhochschule

Master Project 2 Spring 2022

Concurrent Deep Learning Inference with Multiple Models on Single Neural Processing Units

Michael Schmid

1. Introduction

Machine learning and particularly deep learning techniques have shown to be exceptionally powerful for certain applications. As a consequence, more and more deep learning algorithms are also deployed on embedded devices. However, embedded systems come with their very own challenges – resources like processing power, memory or power supply are highly limited while requirements like responsiveness or real-time constraints are rather demanding.

An embedded system is often dedicated to one specific application. In the case of deep learning, it would run inference using a single deep neural network (DNN) model that was trained for this exact application. Additionally, the system may be supported by co-processors or hardware accelerators to boost the performance of the use case.

More complex embedded systems, such as hearing aids, may be tasked with more than one application and must serve multiple use cases in parallel. For example, hearing aids sense their surroundings constantly to classify the sound scene, receive and deliver enhanced audio, or listen to the user's inputs, such as voice commands.

The realization of two or more such use cases through deep learning techniques now raises the question as to how the different DNN models can be executed on given hardware concurrently.

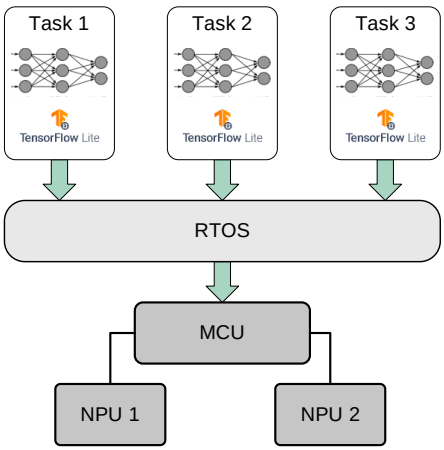


Figure 1: Multitasking for running multiple DNN applications in parallel.

The goal of this project is to investigate this question of running multiple DNN use cases in parallel, making use of the Arm Ethos-U55 neural processing unit (NPU) in combination with a Cortex M-55 or M-33 micro-controller unit (MCU), and possibly further NPUs. Figure 1 visualizes the stated problem of implementing a type of multitasking that serves the tasks of the different use cases by means of a real-time operating system (RTOS) and the available underlying hardware.

2. Task Definition

The work includes the following work packages, clustered into a first and second phase:

- **Phase 1: Simulator**
  - Familiarize yourself with the basic principles and theory.
    1. Arm Cortex M-55 MCU
    2. Arm Ethos U-55 NPU
    3. TensorFlow Lite for Microcontroller (TFLu)
    4. Ethos-u ML Evaluation Kit software
  - Set up the work and development environment and get everything running.
    1. GNU Arm embedded toolchain 10.2.1. (or higher)
    2. Corstone-300 MPS3 based Fixed Virtual Platform (FVP)
    3. Build and run first examples provided with Ethos-u ML Evaluation Kit on the FVP simulator
  - Develop and evaluate methods to support two concurrent DNN use cases on a single NPU (Arm Ethos U-55).
    1. Start with the two DNN use cases running on the same inference rate.

       How to handle TFLu interpreter and tensor arena? RTOS support needed?
    2. Now have two DNN use cases running on different inference rates.

       For example, high rate (690 Hz) for high computational complexity use case like speech enhancement, low rate (10 Hz) for low computational complexity use case like classification.
    3. Develop methods to spread the load of the execution for the low rate and low priority use case over several inferences for the high rate, higher priority use case.
  - Implement and test a solution that supports the concurrent DNN use cases.
    1. Select the most promising method and implement it on the Corstone-300 MPS3 based FVP.
    2. Verify the implementation and evaluate the performance.

- **Phase 2: Sonova platform**
  - Port the developed solution onto Sonova's custom hearing aid platform.
  - *Optional*: Use Sonova's custom simulator to include support for an additional NPU. How can the previously developed solution be extended?
  - *Optional*: Realize a full demonstration of the parallel DNN use cases (which requires additional development of necessary feature extraction capabilities).

OST Ostschweizer Fachhochschule

3. Goals

The following objectives shall be met:

- Gained a deeper understanding of the problem and its solutions, of the setup and embedded platforms (including Arm Cortex M-55 and Ethos-U55, Corstone-300 MPS3 based FVP and TFLu frameworks, Sonova's custom hearing aid platform and tools), of DNN models and their characteristics (including DNNs for classification or speech enhancement applications).
- Developed and evaluated one or several methods for running at least two different DNN models in parallel on the given embedded platforms.
- Implemented and tested at least one solution based on the found methods and given platforms.
- Analyzed the implemented solution(s) in depth for the specific use cases of Sonova.
- Documented learnings and recommendations for mapping two or more DNN models to the embedded platforms comprehensively, with particular focus on Sonova's custom hearing aid platform (for example, pros and cons of methods/solutions with respect to the specific use cases).

4. Project Schedule

Plan a total of 360 working hours (12 ECTS x 30h / ECTS) per student.

At the beginning of the project, a project plan and specifications must be prepared and discussed with the advisor within the first two weeks. In the course of the project, regular status review meetings are to be scheduled to check important work steps. The status review meetings shall be documented by written meeting minutes.

We recommend that you document the project from the beginning and keep a lab journal. The lab journal can help you record your thoughts and allows to use them for later reference. The lab journal can be viewed by the advisor during meetings. However, the lab journal is not part of the final report.

5. Documentation

The project must be documented in a final report, which must contain all considerations, clarifications, calculations and investigations in detail in text and figures. The report should be written in a legible and clearly structured way. An external person with appropriate expertise shall be able to follow the report.

The report shall include the following sections:

- Table of content
- Abstract of 0.5 - 1 page length
- Original text of the task definition
- Introduction
- Main section including the chosen solution approach, evaluation of other considered solutions, solution description and results (for example, findings from tests, measurements, experiments)
- Summary of 2 - 4 pages length, including a self-critical assessment

- Appendix with all relevant data and detailed information, such as tools used or project plans created, including a comparison of the original project plan (target) and its final realization (actual)
- Bibliography

The report (without appendix and bibliography) should not exceed 60 pages.
The report must include a signed non-plagiarism declaration (declaration of independence).
Further guidelines for writing technical or scientific reports can be found, for example, in [1] and [2].

A printed copy of the report must be submitted to the advisor. In addition, the report as well as all data relevant for the continuation of the work shall be submitted in electronic form.

6. Assessment

Your project will be evaluated according to the following criteria:

- Organization: initiative and commitment, teamwork, independence, planning
- Working methods and project implementation: analysis, design and methodology
- Quality of results: degree of performance, ideas, approaches, originality, innovation, maturity of the solution
- Documentation: structure and presentation, text quality, clarity, completeness
- Final presentation/demo: presentation style and expression skills, competence in topic

7. Important Dates

Project start:	Week 8 – as from Monday, February 21, 2022
Final presentation/demo:	date by arrangement
Submission of report:	Week 26 – by Friday, July 1, 2022, before 3 pm

8. Administrative Information

Advisor:	Prof. Andreas Breitenmoser (+41 58 257 46 56, andreas.breitenmoser@ost.ch)
Industry partner:	Sonova AG, Laubisrütistrasse 28, 8712 Stäfa
Co-advisor Sonova:	Sebastian Stenzel
Meetings:	weekly, by arrangement
Workplace/lab:	rooms 8.221 and 6.007 at OST

9. References

[1] A. Verhein und A. Simeon, Werkzeugkasten Technische Berichte 1, 2013.

[2] P. Mayer, 77 mal wissenschaftliches Schreiben - eine Anleitung, Basel: Advanced Study Center, Universität Basel, 2010.

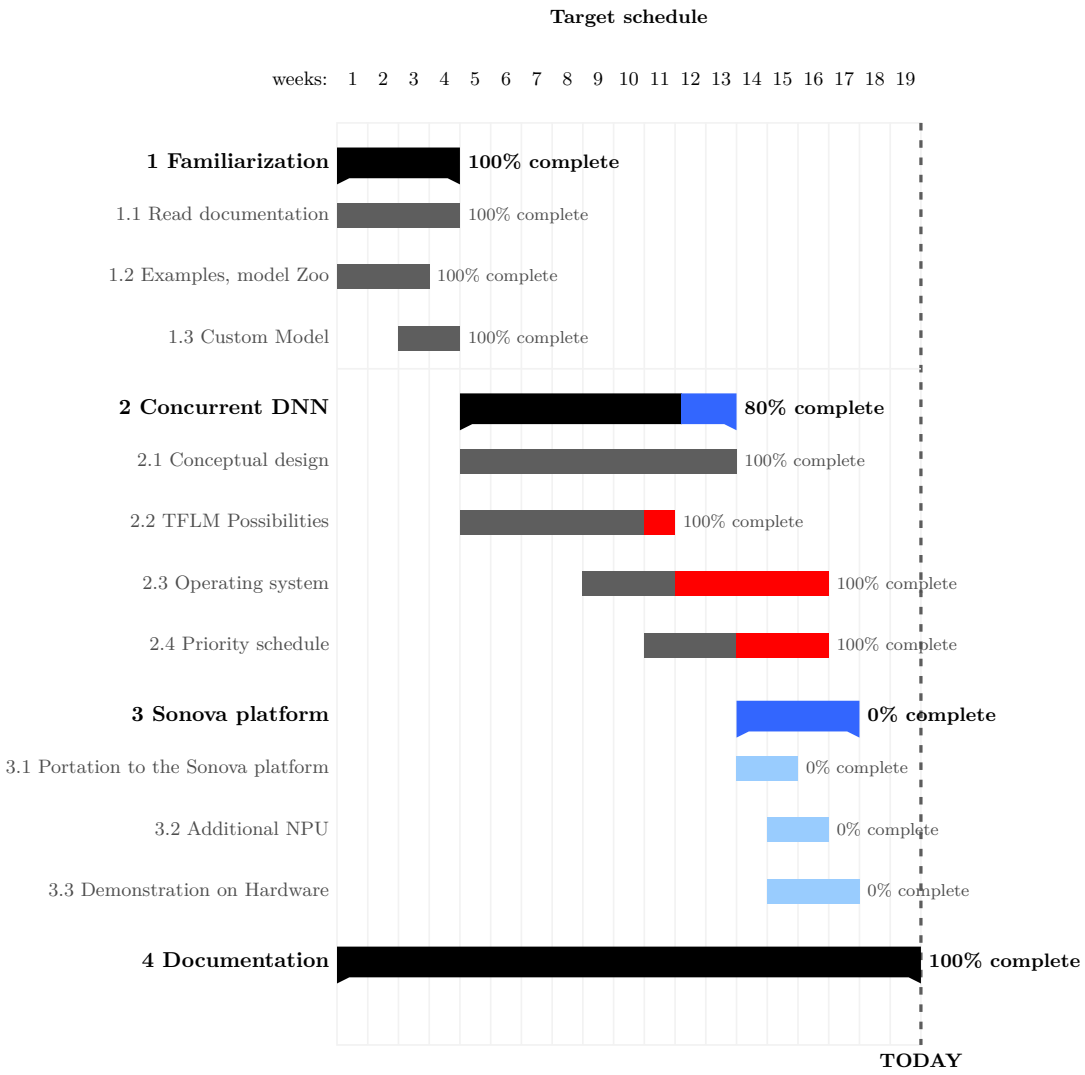
I wish you good luck with your project – have fun!

Rapperswil, February 2022

Andreas Breitenmoser

Appendix B

Schedule



Appendix C

Additional Informations

C.1 Training Artificial Neural Networks with Backpropagation

In order to use such an architecture of a neural network, the weights, possible millions of them, needs to be computed. In deep learning, one calls this process learning, because of its iterative algorithms.

Error Function Firstly, one needs to know how well the network performed for a given input when the desired output is known. The error of a neuron in the output layer can be defined as

$$E_j = \frac{1}{2}\left(r_j - a_j(L)\right)^2, \tag{C.1}$$

where j is the neuron, r_j the desired and a_j the actual response. For all Neurons of the output layer, the error

$$E = \sum_{j=1}^{n_L} E_j = \frac{1}{2}\sum_{j=1}^{n_L}\left(r_j - a_j\left(L\right)\right)^2 = \frac{1}{2}\left\|\mathbf{r} - \mathbf{a}(L)\right\|^2 \tag{C.2}$$

can be written as the euclidean vector norm.

Other Error Functions Different error functions are used to train neural networks, two common examples:

- The mean squared error

$$\text{MSE} = \frac{\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2}{n} \tag{C.3}$$

- Cross Entropy Loss/Negative Log Likelihood

$$\text{CE} = -\left(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)\right) \tag{C.4}$$

Gradient Decent The main goal of the training is to find the weights and biases to minimize the error function. With gradient decent one calculates the gradient of the error function with respect to the weights and biases. Knowing the steepest direction of the error function, one can change the weights and biases a little so that the error function is slowly minimized.

To make the notation a bit more readable the simplification of the gradient of neuron j in the last layer L

$$\delta_j(L) = \frac{\partial E}{\partial z_j(L)} \tag{C.5}$$

is used.

The gradient of the last layer L in terms of the output $a_j(L)$ with the activation function σ is given by

$$\begin{aligned}
\delta_j(L) &= \frac{\partial E}{\partial z_j(L)} \\
&= \frac{\partial E}{\partial a_j(L)} \frac{\partial a_j(L)}{\partial z_j(L)} \\
&= \frac{\partial E}{\partial a_j(L)} \frac{\partial \sigma\left(z_j(L)\right)}{\partial z_j(L)} \\
&= \frac{\partial E}{\partial a_j(L)} \sigma'\left(z_j(L)\right).
\end{aligned} \tag{C.6}$$

Now one wants to know the gradient of every node in the network

$$\delta_j(l) = \frac{\partial E}{\partial z_j(l)}. \tag{C.7}$$

$\delta_j(l)$ is to be expressed with $\delta_j(l+1)$. Due to the backpropagation from the output, one starts with $\delta_j(L)$ and then computes $\delta_j(L-1)$ and with that result $\delta_j(L-2)$ and so on.

Equation (C.7) show how E changes with respect to a small change to any neuron in the network.

$$\begin{aligned}
\delta_j(l) = \frac{\partial E}{\partial z_j(l)} &= \sum_i \frac{\partial E}{\partial z_i(l+1)} \frac{\partial z_i(l+1)}{\partial a_j(l)} \frac{\partial a_j(l)}{\partial z_j(l)} \\
&= \sum_i \delta_i(l+1) \frac{\partial z_i(l+1)}{\partial a_j(l)} \sigma'\left(z_j(l)\right) \\
&= \sigma'\left(z_j(l)\right) \sum_i w_{ij}(l+1)\delta_i(l+1)
\end{aligned} \tag{C.8}$$

$\delta_j(l)$ is computed by all the $\delta_i(l+1)$ from the previous layer. $\frac{\partial a_j(l)}{\partial z_j(l)}$ is replaced with the derivative of the activation function. $\frac{\partial z_i(l+1)}{\partial a_j(l)}$ can be replaced with $w_{ij}(l+1)$ because $z_i(l+1) = w_{ij}(l+1)a_j(l)$

In the end we need to know how much the error changes with respect to a change to the weights.

$$\begin{aligned}
\frac{\partial E}{\partial w_{ij}(l)} &= \frac{\partial E}{\partial z_j(l)} \frac{\partial z_j(l)}{\partial w_{ij}(l)} \\
&= \delta_i(l) \frac{\partial z_i(l)}{\partial w_{ij}(l)} \\
&= a_j(l-1)\delta_i(l)
\end{aligned} \tag{C.9}$$

Similar to the weights the derivation of the error in respect to the bias term

$$\frac{\partial E}{\partial b_i(l)} = \delta_i(l) \tag{C.10}$$

The weights can now be updated using the Gradient Decent algorithm

$$\begin{aligned} w_{ij}(l) &= w_{ij}(l) - \alpha \frac{\partial E(l)}{\partial w_{ij}(l)} \\ &= w_{ij}(l) - \alpha \delta_i(l) a_j(l-1), \end{aligned} \tag{C.11}$$

similar for the bias terms

$$\begin{aligned} b_i(l) &= b_i(l) - \alpha \frac{\partial E(l)}{\partial b_i(l)} \\ &= b_i(l) - \alpha \delta_i(l) a_j(l), \end{aligned} \tag{C.12}$$

where α stands for the step size towards the minimum of the error function.

Matrix Notation Following the matrix notation of the previous shown equations.

Equation (C.6) translates to

$$\boldsymbol{\delta}(L) = \begin{bmatrix} \delta_1(L) \\ \delta_2(L) \\ \vdots \\ \delta_{n_L}(L) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(L)} \sigma'(z_1(L)) \\ \frac{\partial E}{\partial a_2(L)} \sigma'(z_2(L)) \\ \vdots \\ \frac{\partial E}{\partial a_{n_L}(L)} \sigma'(z_{n_L}(L)) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(L)} \\ \frac{\partial E}{\partial a_2(L)} \\ \vdots \\ \frac{\partial E}{\partial a_{n_L}(L)} \end{bmatrix} \odot \begin{bmatrix} \sigma'(z_1(L)) \\ \sigma'(z_2(L)) \\ \vdots \\ \sigma'(z_{n_L}(L)) \end{bmatrix} \tag{C.13}$$

where $\odot$ represents the element wise multiplication between two vectors or matrices.

$$\boldsymbol{\delta}(L) = \frac{\partial E}{\partial \mathbf{a}(L)} \odot \sigma'(\mathbf{z}(L)) \tag{C.14}$$

The vector $\boldsymbol{\delta}(L)$ accounts for one given input vector. The matrix $\mathbf{D}(L)$ accounts for all input vector one wants to use for the training algorithm.

$$\mathbf{D}(L) = \frac{\partial E}{\partial \mathbf{A}(L)} \odot \sigma'(\mathbf{Z}(L)) \tag{C.15}$$

Equation (C.8) translates to

$$\mathbf{D}(l) = \left(\mathbf{W}^T(l+1)\mathbf{D}(l+1)\right) \odot \sigma'(\mathbf{Z}(l)) \tag{C.16}$$

and the two gradient decent algorithms in matrix notation can be written as

$$\mathbf{W}(l) = \mathbf{W}(l) - \alpha \mathbf{D}(l)\mathbf{A}^T(l-1) \tag{C.17}$$

respectively,

$$\mathbf{b}(l) = \mathbf{b}(l) - \alpha \sum_{k=1}^{n_p} \boldsymbol{\delta}_k(l). \tag{C.18}$$

Bibliography

[1] A. Breitenmoser, "Concurrent Deep Learning Inference with Multiple Models on Single Neural Processing Units." Task Project Thesis.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] M. Schmid, "Hardware Accelerated Pose Tracking." Project Thesis AS2021.

[4] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, no. 6, p. 386–408, 1958.

[5] R. C. Gonzalez and R. E. Woods, *Digital image processing*. Pearson, 2018.

[6] "tf.keras.layers.Concatenate." https://www.tensorflow.org/api_docs/python/tf/keras/layers/Concatenate, January 2022.

[7] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017.

[8] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for mobilenetv3," 2019.

[9] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," Feb 2016.

[10] "Weight Pruning." https://www.tensorflow.org/model_optimization/guide/pruning, June 2022.

[11] "Weight Clustering." https://www.tensorflow.org/model_optimization/guide/clustering, June 2022.

[12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," Feb 2016.

[13] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," Dec 2017.

[14] "Arm Cortex-M55 Processor Product Brief." https://armkeil.blob.core.windows.net/developer/Files/pdf/AI/Arm-Cortex-M55-Product-Brief.pdf, June 2022.

[15] "CMSIS NN." https://github.com/ARM-software/CMSIS_5/tree/develop/CMSIS/NN, June 2022.

[16] "Arm Ethos-U55 microNPU Product Brief." https://armkeil.blob.core.windows.net/developer/Files/pdf/product-brief/arm-ethos-u55-product-brief.pdf, June 2022.

[17] "Arm ® Ethos™-U55 NPU Technical reference manual, Issue 02 102420_0200_02_en." https://documentation-service.arm.com/static/60b5e972e022752339b44b4c?token=, June 2022.

[18] "Vela Complier: The first step to deploy your NN model on the Arm Ethos-U microNPU." https://community.arm.com/arm-community-blogs/b/ai-and-ml-blog/posts/vela-complier-deploy-your-nn-model-on-the-arm-ethos-u-micronpu, June 2022.

[19] "Vela Supported Ops." https://git.mlplatform.org/ml/ethos-u/ethos-u-vela.git/tree/SUPPORTED_OPS.md, June 2022.

[20] "Arm® Ethos™-U NPU Application development overview, Issue 101888_0500_05_en." https://documentation-service.arm.com/static/5fae5cefca04df4095c1ca31?token=, June 2022.

[21] "Corstone-300." https://developer.arm.com/Processors/Corstone-300, June 2022.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "[pdf] gradient-based learning applied to document recognition: Semantic scholar," Jan 1998.

[23] "FLOPs calculator for neural networks." https://github.com/tokusumi/keras-flops, June 2022.

[24] "Custom Layers." https://www.tensorflow.org/tutorials/customization/custom_layers, June 2022.

[25] "Custom Layers." https://google.github.io/flatbuffers/, June 2022.

[26] "Studying Flatbuffers to play with TFLite models." https://ricardodeazambuja.com/deep_learning/2021/05/23/tflite-flatbuffers/, June 2022.

[27] "Arm® ML embedded evaluation kit." https://review.mlplatform.org/plugins/gitiles/ml/ethos-u/ml-embedded-evaluation-kit/, June 2022.

[28] "ethos-u-core-platform." https://git.mlplatform.org/ml/ethos-u/ethos-u-core-platform.git/about/, June 2022.

[29] "Vela Performance Estimation Summary." https://git.mlplatform.org/ml/ethos-u/ethos-u-vela.git/tree/PERFORMANCE.md, June 2022.

[30] “Image Classification Code Sample.” https://review.mlplatform.org/plugins/gitiles/ml/ethos-u/ml-embedded-evaluation-kit/+/HEAD/docs/use_cases/img_class.md, June 2022.

List of Figures

List of Tables

Listings